



# User Manual Version 4.1

"Alpha Version of Documentation"

## #39 Feb 2015

# FRONTMATTER



{ }open**Euphoria User Manual**

version 4.1 *work in progress*

1. 39 FEB 2015

Euphoria Programming Language v4.1  
for Windows, Linux, FreeBSD  
and other unix versions available.  
Simple, flexible, and easy to learn.  
Outperforms all popular interpreted  
languages. Complete Reference Manual  
included. Lots of free example programs  
with full source. Makes .EXE files. Great  
for games, utilities, CGI, and GUI programs.  
See docs/ and/or <http://openeuphoria.org>

(c) 2007-2015 by OpenEuphoria Group  
(c) 1993-2006 by Rapid Development Software (RDS)

# Contents



{ }open**Euphoria** Group

- **License**: open source license.
- **Credits**: authors and contributors.

Showcasing Euphoria:

- **Friendly, Flexible, and Fast**; yes, it's true.

Introducing Euphoria:

- **Hello World**; a one line program.
- **Sorting**; a real program.
- **GUI**; simple GUI demo.
- **Mighty Object**; understanding Euphoria.

Start programming:

- **Install**; set up, edit, run.
- **Mini Manual**; start programming now.
- **ThinkEuphoria**; intro to programming.

Language essentials:

1. **Value**; atom and sequence; numbers and text.
2. **Name**; variables, subroutines, namespace.
3. **Express**; relations, logic, arithmetic, sequence.
4. **Control**; branch (break), repeat (exit), goto, return.
5. **Organize**; procedure, function, type, include, scope.
6. **Send**; input, output, database
7. **Execute**; interpret, shroud, bind, compile.

Using Euphoria:

- **Configure**; configuration file.
- **Options**; command-line choices.

- [Optimize](#); execute faster.

## Features:

- [Preprocess](#); extending Euphoria.
- [Multi-Task](#)
- [Debug](#)
- [eutest](#)
- [eudis](#)

## Windows GUI:

- [win32libex](#); complete *windows* GUI.
- [tinewg](#); easy and small *windows* GUI.

## Unix GUI:

- [euGTK](#); complete *unix* GUI.

## Multi-platform GUI:

- [wxWidgets](#); complete *multi-platform* GUI.
- [euIUP](#); easy and small *multi-platform* GUI.

## More GUI choices:

- [Arwen](#) | [Fluid](#) | [Japi](#) | [tcl](#)

## Mini-Guides:

- [Archive](#); RDS resource.
- [Tutor](#); tutorial programs.
- [Documenting](#); eudoc, creole.
- [Regular expressions](#); regex, native.
- [Bnf](#); language description.
- [Internals](#); language design.
- [Eds](#); database.
- [Edx](#); demo editor.
- [Benchmark](#); fast, very fast.
- [Mongoose](#); Rudyard Kipling.
- [Include](#); include, scope.

## The Euphoria API:

- [Built-in](#)
- [Standard Library](#)

## Appendix:

- [Small](#); easy to learn.
- [Coherent](#); easy to read.
- [Simple](#); easy to write.
- [Powerful](#); easy to write.
- [Fast](#); develop rapidly and execute faster.

## Index:

- [Glossary](#)
- [Keyword and Api](#)

# Licensing

This product is free and open source, and has benefited from the contributions of many people. You have complete royalty-free rights to distribute any Euphoria programs that you develop. You are also free to distribute the eui interpreter, eushroud shrouder, eub backend, eubind binder, and even the euc compiler. You can shroud, bind, or compile your program and distribute the resulting files royalty-free.

You may incorporate any Euphoria source files from this package into your program, either *as is* or with your modifications. (You will probably need at least a few of the standard `std\include` files in any large program).

We would appreciate it if you told people that your program was developed using Euphoria, and gave them the address <http://www.openeuphoria.org/> for our Web page, but we do not require any such acknowledgment.

The Euphoria wordmark, Euphoria claws symbol, and Euphoria mongoose icon are trademarks of the OpenEuphoria group.

Icon files, such as `euphoria.ico` in `euphoria\bin`, may be distributed with or without your changes.

The high-speed version of the Euphoria Interpreter back-end is written in ANSI C, and can be compiled with many different C compilers. The complete source code is in `euphoria\source`, along with `execute.e`, the alternate, Euphoria-coded back-end. The generous Open Source License allows both personal and commercial use, and unlike many other open source licenses, your changes do not have to be made open source.

Some additional 3rd-party legal restrictions might apply when you use the euc Compiler.

Copyright (c) 2007-2014 by OpenEuphoria Group  
Copyright (c) 1993-2006 by Rapid Deployment Software (RDS)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The copyright holders request, but do not require, that you:

1. Acknowledge RDS and others who contributed to this software.
2. Provide a link to [www.RapidEuphoria.com](http://www.RapidEuphoria.com), if possible, from your Web site.

# Credits

Euphoria has been continuously developed since it was started in 1993 by Robert Craig. In 2006 Euphoria 3.0 was released as open source followed by various releases in the 3.x series. In the fourth quarter of 2010 Euphoria 4.0 introduced the largest update ever. Currently the the Euphoria 4.x series is being actively developed.

It has taken quite a few people to get this far and we would like to recognize them here. Authors and contributors are listed in alphabetical order by their last name.

For an up-to-date listing, see the Euphoria Contributors page at the [OpenEuphoria Wiki](#).

## Current Authors

- Jim Brown
- Tom Cipliauskas
- Jeremy Cowgar
- C. K. Lester
- Matthew Lewis
- Derek Parnell
- Shawn Pringle

## Past Authors

- Robert Craig
- Chris Cuvier
- Junko Miura

## Contributors

- Jiri Babor
- Chris Bensler
- CoJaBo
- Jason Gade
- Ryan Johnson
- Lonny Nettnay
- Marco Antonio Achury Palma
- Michael Sabal
- Dave Smith - Graphics
- Kathy Smith
- Randy Sugianto

# SHOWCASE \_\_\_\_\_



- showcase
- helloworld
- sorting
- gui



# Showcasing OpenEuphoria



**Euphoria is...**

Friendly ... Flexible ... Fast

**...the first choice in programming.**

## Euphoria is Friendly

Free and open source, Euphoria is friendly because of its great features.

Euphoria is small; there is less to learn. Euphoria is coherent; what you learn always works the same way. Euphoria is simple; write readable programs without fuss or complications. Euphoria is powerful; write solid programs without OOP or exotic constructs. Euphoria is rapid; fix bugs quickly; create apps that outperform other interpreted languages.

## Euphoria is Flexible

Your source-code can be both interpreted and compiled. Programs are multi-platform.

Euphoria data-types adapt to any situation; an atom is *any* one value; a sequence is *any* collection of values. The same operators and routines work equally well on numbers and on strings.

## Euphoria is Fast

A fast interpreter makes program development easier. Euphoria has the fastest interpreter.

Evaluate Euphoria performance yourself. Benchmark programs for Euphoria, Python, Ruby, Perl are provided; its best to believe your own results.

You can go even quicker. Euphoria programs can be compiled into applications that gain even more speed.

## Linux Format Review

Linux Format has reviewed Euphoria and discovered:

# Joyous coding with OpenEuphoria



**Juliet Kemp** dives into Euphoria and discovers it's fast to read, fast to pick up and fast to run.



Visit Linux Format to download their review and tutorial.

## Yes, It's True

Euphoria may be your first choice when you start a fun programming project.

Every language introduction starts with an enthusiastic description; we finish by providing proof with benchmarks, sample-code, and comparisons. See the appendix.

### See Also:

Small | Coherent | Simple | Powerful | Fast |

Linux Format: [http://www.linuxformat.com/includes/download.php?PDF=LXF176.code\\_oeuphoria.pdf](http://www.linuxformat.com/includes/download.php?PDF=LXF176.code_oeuphoria.pdf)

# Hello Euphoria

A few sample programs to give you a sense of Euphoria source-code:

## Hello Euphoria

The traditional *Hello* program is a one-liner in Euphoria.

```
puts(1, "Hello, Euphoria\n")
```

The built-in procedure `puts` does the job of displaying text on a screen. It requires two arguments. The first argument, `1`, directs the output to STDOUT or the console. The second argument, `"Hello, Euphoria\n"`, is a string of text that will be output.

The result is:

```
Hello, Euphoria
```

## Sorting

The following is an example of a more useful Euphoria program.

```
include std/console.e
sequence original_list

function merge_sort(sequence x)
-- put x into ascending order using a recursive merge sort
integer n, mid
sequence merged, a, b

n = length(x)
if n = 0 or n = 1 then
return x -- trivial case
end if

mid = floor(n/2)
a = merge_sort(x[1..mid])      -- sort first half of x
b = merge_sort(x[mid+1..n])    -- sort second half of x

-- merge the two sorted halves into one
merged = {}
while length(a) > 0 and length(b) > 0 do
if compare(a[1], b[1]) < 0 then
merged = append(merged, a[1])
a = a[2..length(a)]
else
merged = append(merged, b[1])
b = b[2..length(b)]
end if
end while
return merged & a & b -- merged data plus leftovers
end function

procedure print_sorted_list()
-- generate sorted_list from original_list
sequence sorted_list

original_list = {19, 10, 23, 41, 84, 55, 98, 67, 76, 32}
sorted_list = merge_sort(original_list)
for i = 1 to length(sorted_list) do
display("Number [] was at position [:2], now at [:2]",
{sorted_list[i], find(sorted_list[i], original_list), i}
)
end for
end procedure

print_sorted_list() -- this command starts the program
```

The above example contains a number of statements that are processed in order.

include std/console.e

This tells Euphoria that this application needs access to the public symbols declared in the `std/console.e` file. This is referred to as a *library* file. In our case here, the application will be using the `display` procedure by including the

sequence original\_list

This declares a variable that is not public but is accessible from anywhere in this file. The data-type for the variable is a sequence, which is a variable-length "array," and whose symbol name is `original_list`.

function merge\_sort(sequence x) ... end function

This declares and defines a function subroutine. Functions return values when called. This function must be passed a single argument when called--a sequence.

```
procedure print_sorted_list() ... end procedure
```

This declares and defines a procedure subroutine. Procedures never return values when called. This procedure must not be passed any parameters when called.

```
print_sorted_list
```

This calls the subroutine called `print_sorted_list`.

The output from the program will be:

```
Number 10 was at position 2, now at 1
Number 19 was at position 1, now at 2
Number 23 was at position 3, now at 3
Number 32 was at position 10, now at 4
Number 41 was at position 4, now at 5
Number 55 was at position 6, now at 6
Number 67 was at position 8, now at 7
Number 76 was at position 9, now at 8
Number 84 was at position 5, now at 9
Number 98 was at position 7, now at 10
```

Note that `merge_sort` will just as easily sort any list of data items:

```
{1.5, -9, 1e6, 100}
{"oranges", "apples", "bananas"}
```

This example is stored as `euphoria\tutorial\example.ex`. This is not the fastest way to sort in Euphoria. Go to the `euphoria\demo` directory and type

```
eui allsorts
```

to compare timings on several different sorting algorithms for increasing numbers of objects.

You can compile `allsorts.ex` at the command line by typing

```
euc allsorts
```

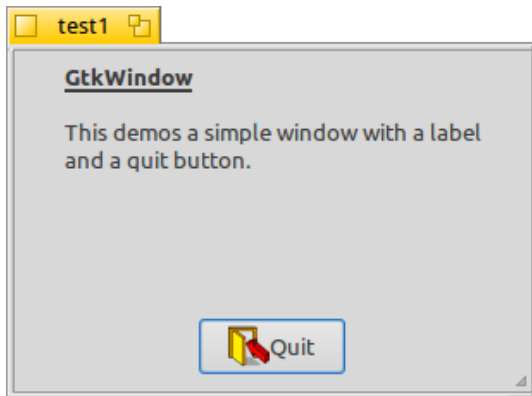
When you run the compiled application the timings increase dramatically.

For a quick example of Euphoria programming see `euphoria\demo\bench\filesort.ex`.

## GUI Programs

Euphoria has several GUI toolkits to choose from including win32libex for *windows* and wxWidgets for all platforms.

This example illustrates euGTK on the *unix* platform:



```
include GtkEngine.e

constant docs = `<u>GtkWindow</u>
</b>

This demos a simple window with a label
and a quit button.
`

constant win = create(GtkWindow)
connect(win,"destroy","Quit")
set(win,"border width",10)
set(win,"default size", 300,200)
set(win,"position",GTK_WIN_POS_CENTER)
set(win,"icon","face-laugh")

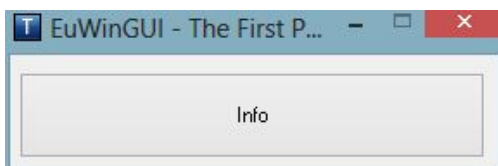
constant panel = create(GtkBox,VERTICAL)
add(win,panel)

constant lbl1 = create(GtkLabel)
set(lbl1,"markup",docs)
add(panel, lbl1)

constant
btn = create(GtkButton,"gtk-quit","Quit"),
box = create(GtkButtonBox)
add(box,btn)
pack(panel,-box)

show_all(win)
main()
```

For simple programming on *windows* you can try the tinEWG which includes an IDE to make development easier:



```
include tinewg.exw
Window ("EuWinGUI - The First Program",100,100,290,95)
constant button1=Control(Button,"Info",5,10,270,50)
SetIcon("T.ICO")

procedure clickbutton()
InfoMsg("Programmed in Euphoria with the EuWinGUI Library!!","EuWinGUI")
```

```
end procedure
```

```
SetHandler (button1,Click,routine_id("clickbutton"))
```

```
WinMain ()
```

# Euphoria in Context

## "Obvious" languages to learn.

Pick a language based on what some large corporation may be using; an unimaginative approach to learning programming.

Sponsor	Languages
Red Hat	C, C++, Ruby, Python, Perl, Javascript
Microsoft, Apple	purchase their current language
Google	Java, Go

## "Distinctive" languages to understand.

Pick a language by understanding how it is designed and how it can satisfy your programming needs.

Characteristic	Example	Features
Object	Euphoria	... atom and sequence
Near Machine	Basic, C	... bit based data-types
Structured	Pascal	... control
Lambda Calculus	Lisp	... recursive
Pattern	Snobol Perl	... goal directed style ... regex style
Functional	Haskel	... higher level math
Entity	Python, Ruby	... class organization

## "Choice" in language design.

Get personal and pick a language by its look and feel.

Characteristic	Example
Plain English	Euphoria Basic
Pseudocode	Euphoria some textbooks
Punchcard	Fortran Python
Mathematical	C Lisp Haskel
Regular Expression	Perl

If you "mix and match" requirements you end up with hundreds (yeah thousands) of languages. For example C became C++ (expanded C) became Java (alternate C++) became C# (alternate Java) became ... If you "want it all" you will have to learn several languages.

Because the atom-sequence design is so great the Euphoria language is being pulled into several directions! It is easy to imagine several variations



of Euphoria: minimalist, featurefull, class oriented, ... Today, we have the resources to make just **one** version of Euphoria.

If you look at all possible languages, all possible variations, mix in some personal bias, and then you ask for... **Friendly, Flexible, and Fast**...you end up with... `{}`*open***Euphoria**.

## OpenEuphoria

The first choice in programming.

`_tom`

# INSTALL



- internet
- download
- EuphoriaPorteus
- install
- scm

# Typography

Special content in this manual gets a particular style.

Style		Meaning
<b>bold</b>	Words	technical vocabulary defined in this manual
<code>display("hello")</code>	Euphoria	source-code sample
( )	Parenthesis	grouping operations, subroutine argument lists
[ ]	Brackets	subscripting sequences
{ }	Braces	creating sequences

## Internet Links

The **OpenEuphoria** webpage is [www.openeuphoria.org](http://www.openeuphoria.org) where you will find the Forum, the Wiki, documentation, download links, SCM link, and related Euphoria websites.

The **SCM** is a "*source code management* system using Mercurial software for the development of Euphoria source-code." To download Mercurial software visit [mercurial.selenic.com](http://mercurial.selenic.com). You can use Mercurial to mirror the Euphoria SCM site or to develop your own software.

**RDS** or Rapid Development Software is the original author of Euphoria. They maintain a website [www.rapideuphoria.com](http://www.rapideuphoria.com) where you can download Euphoria, along with applications, and libraries written in Euphoria.

The **Archive** is "a repository of programs written in Euphoria." The Archive is located at [rapideuphoria.com/archive.htm](http://rapideuphoria.com/archive.htm)

**SourceForge** is "the website where Open Source software is published." Euphoria downloads are hosted at SourceForge [sourceforge.net/projects/rapideuphoria/](http://sourceforge.net/projects/rapideuphoria/) where you can find the latest stable packages.

A **eubin** is "a compiled Euphoria package shared between developers used for testing." A eubin may be unstable, incomplete, and produced on an unpredictable schedule. You are free to download a eubin from <http://openeuphoria.org/eubins/> if you are curious.

# DOWNLOAD

Euphoria documentation is available for reading at the OpenEuphoria website. Documentation is included as part of every Euphoria distribution. You can also download just this documentation in HTML or PDF formats from Source-Forge: [link](#).

A **package** is "a complete set of files containing everything you need to program with OpenEuphoria."

A **binary** package "contains a compiled interpreter (and other compiled products), library files, documentation, demos, tutorials, and source-code."

Windows binary packages come with an installer program and are available in 32-bit and 64-bit versions. Windows binary distributions bundled with a C compiler are also available--choose this installer if you want to compile applications.

Linux binary packages are available as *.deb Debian packages* in 32-bit and 64-bit versions. These packages install on Debian, Ubuntu, Mint, and similar Linux distributions.

Linux binary packages are also available as *tar archives* for Linux distributions that do not use *.deb* packages.

A **source** package "contains source-code, library files, and documentation." Use a C compiler to make an interpreter (and other products) when a binary package is not available for your system.

## GUI toolkits

Windows, Linux, and multi-platform toolkits for writing GUI programs are the products of individual authors and they choose where to host their files.

Links to GUI toolkits are found at the OpenEuphoria wiki. A few of these toolkits are described in this documentation.

## Euphoria & Porteus

(idea under development!)

The developers of **Porteus** Linux describe it as:

"Porteus is a complete linux operating system that is optimized to run from CD, USB flash drive, hard drive, or other bootable storage media. It's small (under 300Mb) and insanely fast which allows you to start up and get online while most other operating systems are left spitting dust. Porteus comes in both 32 & 64 bit and aims to keep on the bleeding edge. It also supports several different languages and the user forum has language sections. Join the community now!"

You can find Porteus Linux at [www.porteus.org](http://www.porteus.org) and download it at no cost. Porteus has a utility that changes a euphoria.deb package into a euphoria.xzm. Activate the .xzm package to install Euphoria.

**E&P (Euphoria&Porteus)** is a "remix that adds Euphoria, documentation, examples, and gui software." E&P will run on any computer (Windows or Linux) requiring no changes to your computer--no installation needed. Trying Euphoria can not get easier.

Refer to the Porteus website for installation instructions and additional software.

(Not available yet) Download the E&P ISO file from [source](#)

## CD Live

Burn a CD from the ISO file and run E&P:

- Burn Porteus ISO to CD using your installed burning software.
- Set the bios to boot from the CD.
- Boot Porteus, explore Euphoria.

From your live CD running Porteus you can install everything to a USB drive or a hard drive.

## USB Drive

You can install the contents of the E&P ISO file directly to a USB drive. Running Porteus from a USB drive has the advantage that any changes you make can be saved. The Porteus website describes several ways to install the ISO download to a USB drive.

The utility **UNetbootin** is "a way to create Live USB drives from an ISO file without the need to burn a CD." UNetbootin can be downloaded from [unetbootin.sourceforge.net](#) and is available for Windows, Linux, and OSX.

- Download UNnetbootin USB installer
- install the Porteus ISO to the USB drive
- set bios to boot from USB
- Boot Porteus, explore Euphoria.

## Euphoria-Porteus

( Idea under development...so far.)

**Porteus** is "a small Linux distribution that can run live from a CD or a USB pendrive." You can run Porteus on any computer (Windows or Linux) and have a capable operating system that requires no installation or changes to your computer.

**Euphoria-Porteus** is "a remix distribution with Euphoria pre-installed." This could be the easiest way to get started with Euphoria programming. You can run text and gui programs and explore what Euphoria has to offer.

## Installation

- Basic Porteus operating system
- Firefox
- OpenEuphoria
- customize Geany
- euGTK
- diffmerge

# Installation

To install Euphoria, consult the instructions below for your particular operating system.

## Windows

All versions other than Windows 95 work without problems. To use Windows 95, it must have Internet Explorer version 4 or higher installed (included in service pack 2.5). To use the new socket functions you will also need Windows 2000 or later. To use all of the new standard library functions you will need at least Windows XP or later.

Euphoria is frequently tested on Windows versions: XP, Vista, 7.

To install Euphoria on *Windows*, visit the following URL:

<http://openeuphoria.org/wiki/view/DownloadEuphoria.wc>

The **Standard** package is a complete Euphoria installation, with Interpreter, Binder, Translator. Included are demo programs and documentation.

The **Open Watcom** package has the contents of the "Standard" version, plus a bundled compiler. This is a convenient way of producing compiled executables from Euphoria programs.

Download the latest *Windows* installer found under the Binary Releases heading of the Current version of Euphoria. Run the program and follow the prompts to get Euphoria installed.

The installer copies the required files; adds the binary subdirectory to your path, if you leave 'update environment' checked; and if you leave 'Associate file extensions' checked, it associates icons and various actions to Euphoria file extensions. Please do not open 'Euphoria Console Files' from Explorer; they are meant to be run from the command line.

The installer does not set the environment variable **EUDIR** to the Euphoria directory even though many third-party programs expect that to be set. This is so an older version of EUPHORIA can also still work on the same system. To set this variable please see the section "[How to manually edit your environment in Windows](#)" below.

## Possible Problems

- On Windows XP/2000, be careful that your PATH and EUDIR do not conflict with autoexec.nt, which can also be used to set environment variables.
- On WinME/98/95 if the install program fails to edit your autoexec.bat file, you will have to do it yourself. Follow the manual procedure described below.
- Euphoria cannot be run under Windows 3.1 and some unpatched versions of Windows 95 will not be able to run EUPHORIA 4.0.
- You have two EUPHORIA installs and you want to change the environment to use another EUPHORIA.

## Manually Edit the Windows Environment

Your Euphoria installation directory by default will be C:\Euphoria. It is possible to install to %PROGRAMFILES%\Euphoria, or anywhere you wish. Careful when using the %ProgramFiles% special location (C:\Program Files on most systems in English). The %ProgramFiles% directory invariably contains spaces by default. It is a good idea to use the short 8.3 version of the name, or surround with double quotes. Of course, you will just have to substitute your real installation directory for the C:\EUPHORIA examples below.

## Environment in Windows NT/2000/XP



Applies to .

On Windows XP select:

Start Menu -> Control Panel -> Performance & Maintenance -> System -> Advanced then click the "Environment Variables" button. Click the top "New..." button then enter EUDIR as the Variable Name and c:\euphoria (or whatever is correct) for the value, then click OK. Find PATH in the list of your variables, select it, then click "Edit...". Add ;c:\euphoria\bin at the end and click OK.

On Windows Vista: Other versions of Windows will have the environment variables somewhere in the control panel.

## Environment in Windows ME/98/95/3.1

1. In the file c:\autoexec.bat add C:\EUPHORIA\BIN to the list of directories in your PATH command. You might use the MS-DOS Edit command, Windows Notepad or any other text editor to do this.

You can also go to the Start Menu, select Run, type in sysedit and press Enter. autoexec.bat should appear as one of the system files that you can edit and save.

2. In the same autoexec.bat file add a new line:  
SET EUDIR=C:\EUPHORIA  
The EUDIR environment variable indicates the full path to the main Euphoria directory.
3. Reboot (restart) your machine. This will define your new PATH and EUDIR environment variables.

Some systems, such as Windows ME, have an autoexec.bat file, but it's a hidden file that might not show up in a directory listing. Nevertheless it's there, and you can view it and edit it if necessary by typing, for example: notepad c:\autoexec.bat in a DOS window.

## More on editing environment variables

- set EUDIR to the location of your Euphoria installation directory.
- In PATH you need to include %EUDIR%\BIN.
- There is another, optional, environment variable used by some experienced users of Euphoria. It is called EUINC (see the [include statement](#)). It determines a search path for included files and this variable is used by new and older versions of EUPHORIA. However, for 4.0 and above we now have a ["configuration file"](#) for adding include paths and other settings.

## Modifying the Registry

Updating the environment is not enough, your old installation will still be called when you open a Euphoria program in explorer or invoke the Euphoria program on the command line without typing in the interpreter (eui euiw). Do not type in the single quotes.

You can set these in regedit (replace C:\EUPHORIA with your Euphoria installation directory):

```
HKEY_CLASSES_ROOT\.exw\(\Default)
=> 'EUWinApp'
HKEY_CLASSES_ROOT\EuWinApp\(\Default)
=> 'Euphoria Windows App'
HKEY_CLASSES_ROOT\EUWinApp\shell\open\command\(\Default)
=> 'C:\EUPHORIA\BIN\euw.exe "%1"'
HKEY_CLASSES_ROOT\EUWinApp\shell\translate\command\(\Default)
=> 'C:\EUPHORIA\BIN\euc.exe "%1"'

HKEY_CLASSES_ROOT\.ex\(\Default)
=> 'EUConsoleApp'
HKEY_CLASSES_ROOT\EUConsoleApp\(\Default)
=> 'Euphoria Console App'
HKEY_CLASSES_ROOT\EUConsoleApp\shell\open\command\(\Default)
=> 'C:\EUPHORIA\BIN\eui.exe "%1"'
```

```

HKEY_CLASSES_ROOT\EUConsoleApp\shell\translate\command\(\Default)
=> 'C:\EUPHORIA\BIN\euc.exe -con "%1"'

HKEY_CLASSES_ROOT\.e\(\Default) => 'EUInc'
HKEY_CLASSES_ROOT\EUInc\(\Default) => 'Euphoria Include File'
HKEY_CLASSES_ROOT\.ew\(\Default) => 'EUInc'

```

You can also set an editor for your EUPHORIA programs this way:

```

HKEY_CLASSES_ROOT\EUWinApp\shell\edit\command\(\Default)
=> 'C:\EUPHORIA\BIN\euw.exe C:\EUPHORIA\BIN\edx.exe "%1"'
HKEY_CLASSES_ROOT\EUConsoleApp\shell\edit\command\(\Default)
=> 'C:\EUPHORIA\BIN\euw.exe C:\EUPHORIA\BIN\edx.exe "%1"'
HKEY_CLASSES_ROOT\EUInc\shell\edit\command\(\Default)
=> 'C:\EUPHORIA\BIN\euw.exe C:\EUPHORIA\BIN\edx.exe "%1"'

```

You can setup to allow the supplied editor program open to the line where the last failure occurred in ex.err files:

```

HKEY_CLASSES_ROOT\.err\(\Default) => 'EUError'
HKEY_CLASSES_ROOT\EUError\(\Default) => 'Error File'
HKEY_CLASSES_ROOT\EUError\shell\debug
=> 'Debug what created this error file'
HKEY_CLASSES_ROOT\EUError\shell\debug\command\(\Default)
=> 'C:\EUPHORIA\BIN\euw.exe C:\EUPHORIA\BIN\edx.exe'
HKEY_CLASSES_ROOT\EUError\DefaultIcon\(\Default)
=> 'C:\Windows\system32\shell32.dll,78'

```

## Linux and FreeBSD

Euphoria may be installed using either a *Unix* archive ( .tar.gz or .tar.bz2 ) or, a distribution specific package, if available.

<http://openeuphoria.org/wiki/view/DownloadEuphoria.wc>

The *Unix* tarball "Archive" is laid out similarly to the *Windows* directory structure. This may be convenient if working cross-platform between *Windows* and *Unix*. The files at ~SourceForge are also in this form, making it convenient if you wish to use updates directly from the SVN depository.

To install this version you must manually unarchive the tarball. Then copy the files to a suitable directory.

You'll need to manually edit:

- /etc/profile so the PATH contains
  - euphoria/bin, and either create
    - an eu.cfg file or
    - set up EUDIR and EUINC. See the [include statement](#).

The "Packaged" version installs Euphoria in a more *Unix*-like way, putting the executables into

- /usr/bin,
- /usr/share/euphoria and
- /usr/share/doc/euphoria.
- Man pages for eui, euc, eub, shroud and bind are also installed.
- It will also create /etc/euphoria/eu.cfg, which will point to the standard euphoria include directory in /usr/share/euphoria/include.

Other *Unix* based installations can be compiled from "Source Releases".

## OSX

Look for an installation package for Apple installations.

## DOS

There is DOS support only up to Euphoria 3.1. DOS developers are invited to contribute their skills.

## Post Install

The directory layouts will help you locate the Euphoria executables, documentation, and sample programs.

Operating system specific files are not included:

- The *Linux* subdirectory is not included in the *Windows* distribution,
- The *win32* subdirectories are not included in the *unix* distribution.

The `../include`, `../demo` and `../tutorial` *directories are the same in windows and unix*.

In this manual directory names are sometimes shown using a `\` backslash. *Unix* users should substitute the `/` forward slash.

## Windows Installation

This is the standard Windows directory arrangement. This is also the standard directory arrangement source-code files at the SCM repository. It is possible to use the same arrangement when installing Euphoria on a *unix* system.

```

|
|__ euphoria
|   file_id.diz
|   License.txt
|
|__ bin
|   Interpreter (eui.exe and euiw.exe,  if on Windows)
|               (eui, if on Unix)
|   Binder      (eubind, with eub)
|   Translator  (euc.exe, if on Windows)
|               (euc, if on Unix)
|   Utilities   (bugreport.ex, bench.ex, edx.ex, ...)
|
|__ include
|   |
|   |           (original include files)
|   |__ std      (standard Euphoria library: io.e, sequence.e, ...)
|   |__ euphoria (Euphoria specific)
|
|__ docs          (html and pdf documentation files)
|__ tutorial      (small tutorial programs to help you learn Euphoria)
|__ demo         (generic demo programs that run on all platforms)
|   |__ win32     (Windows specific demo programs (optional) )
|   |__ unix      (Linux/FreeBSD/OS X specific demo programs (optional))
|   |__ langwar   (language war game for Linux/FreeBSD/OS X )
|   |__ bench     (benchmark program )
|
|__ source        (the complete source code for: interpreter, translator)
|__ tests         (unit tests for Euphoria)
|__ packaging     (software for making installation packages)

```

## Debian Installaion

A `.deb` or *Debian* package applies to Debian, Ubuntu, Mint and other similar Linux distributions. The standard Debian installation places files their traditional *unix* locations.

```
|
|__ /usr/bin                (executables: eui, euc, ... )
|
|__ /usr/share/euphoria
|
|   |__ bin                (utility programs)
|   |__ demo              (general demonstration programs)
|   |__ include           (standard library)
|   |__ source            (source-code for Euphoria)
|   |__ tutorial          (tutorial programs for learning Euphoria)
|
|__ /usr/share/doc/euphoria (html and pdf documentation)
|
|__ /etc/euphoria           ( eu.cfg )
```

## Source Installation

A standard installion from source-code places files in `../usr/local` (instead of `../usr`). Otherwise, the files are placed in the same pattern as a Debian installation package.

When installing from source-code you can select your own destination directory:

```
$ ./configure --prefix /some/other/location
```

### See Also:

[SCM | source-code installation](#)

# SCM

Euphoria source-code is hosted at <http://scm.openeuphoria.org/hg/euphoria/> and is free for anyone to download.

- Install Mercurial and you can mirror the entire SCM site. This provides you with *all* historical changes to Euphoria.
- At any time you can download just the current source archived as either a .zip or .bz2 file.

## Required

- GCC C compiler.
- Current Euphoria installation.
- Administrative privileges for final installation.

## Compiling

- Download the current source from the SCM and unarchive.
  - Navigate to /euphoria/source directory
  - Open terminal and at the \$ prompt:
    - \$ ./configure
    - \$ make
  - Using administrator privileges:
    - \$ make install
- Euphoria will be installed to the /usr/local directory tree.
- Test:
    - \$ eui

```
Euphoria Interpreter v4.1.0 development
**-bit Linux, Using System Memory
Revision Date: unknown, Id: 0:unknown
```

```
ERROR: Must specify the file to be interpreted on the command line
```

Depending on your system *\*\*-bit* will be either 32-bit or 64-bit. There is no revision date since this is an SCM installation.

# Editors

Euphoria programs are plain text files. That means *any* text editor may be used to write Euphoria programs.

Immediately after installing Euphoria you have some simple choices:

- Try the `ed.ex` editor found in `.../euphoria/bin`. This is a text based demo program that has syntax coloring and will execute the demo programs included with Euphoria.
- Every operating system will have a default text editor that will edit programs. A *windows* system will have ~Notepad and a *unix* system will have an editor like nano, vi, and a simple gui editor.

The ~RapidEuphoria Archive has many [editors](#) listed. You will find editors written in Euphoria and syntax files for a variety of popular editors.

There are third-party editors that support Euphoria by providing syntax highlighting and other useful features.

## Code Editors

A **code editor** is a "plain text editor with features that make writing source-code easier." As a minimum a code editor will provide syntax color to make reading Euphoria source-code easier.

### Edx Demo Editor

The **edx** editor is included in every Euphoria package. This is a simple text editor, written in Euphoria, that can get you started if you can not decide on your own favorite editor. Edx is provided as an example of Euphoria programming rather than as an editor for daily use.

Kenneth Rhodes has taken edx and produced **mdx** which has *more* features; download mdx from The Archive [<http://www.rapideuphoria.com/mdx-1.2.tar.gz> ].

## Windows and Unix

Text	Gui	Win	Linux	OSX	Indent	Ex.err	Free	Open	Source	Syntax
	CodeLite	win	unix						c++	<a href="#">eu-editor project</a>
<b>eFTE</b>	<b>eFTE</b>	win	unix						c++	ready
	E-TextEditor	win	?		yes	yes	30 day	closed		<a href="#">eu-editor project</a>
	Geany	win	unix						c	<a href="#">RDS archive</a>
	Gedit	win	unix						c	<a href="#">RDS archive</a>
	jEdit	win	unix				free	open	java	<a href="#">eu-editor project</a>
	SciTE	win	unix		partial				c	<a href="#">RDS archive</a>
<b>Vim</b>	<b>Vim</b>	win	unix		partial				c	<a href="#">eu-editor project</a>

	<b>wxEditor</b>	win	unix						<b>Euphoria</b>	ready (incomplete)
--	-----------------	-----	------	--	--	--	--	--	-----------------	-----------------------

## Windows Only

Text	Gui	Win	Linux	OSX	Indent	Ex.err	Free	Open	Source	Syntax
	Context	win			?				pascal	3.x
	Crimson Editor	win			partial				c++	rules keywords
	<b>Edita</b>	win			yes	yes			<b>Euphoria</b>	ready
	~EditPlus	win			yes		30 day	closed		editplus
<b>eFTE</b>	<b>eFTE</b>	win	unix						c++	ready
	E-TextEditor	win	?		yes	yes	30 day	closed		eu-editor project
	HippoEDIT	win					30 day	closed		eu-editor project
	jEdit	win	unix				free	open	java	eu-editor project
MicroEmacs	MicroEmacs	*	unix		yes				c++	eu-editor project
<b>Minimum Profit</b>	<b>Minimum Profit</b>	win	unix						c++	ready
Nano			unix						c	eu-editor project
	Notepad++	win							c++	eu_editor project
	PSPad	win			yes		free	closed		eu-editor project
	SynWrite	win					free	closed		available
	TextMate			osx	yes	yes	30 day	closed		eu-editor project
	TextPad	win			?		evaluate	closed		?

## Unix Only

Text	Gui	Win	Linux	OSX	Indent	Ex.err	Free	Open	Source	Syntax
<b>eFTE</b>	<b>eFTE</b>	win	unix						c++	ready
	Geany	win	unix						c	RDS archive
Joe		*	unix						c	eu- editor project
	<b>Kate</b>	*	unix		yes				c++	ready eu- editor- project
	<b>Kwrite</b>	*	unix		yes				c++	ready (needs update)
MicroEmacs	MicroEmacs	*	unix		yes				c++	eu- editor project
<b>Minimum Profit</b>	<b>Minimum Profit</b>	win	unix						c++	ready

Nano			unix						c	eu-editor project
	SciTE	win	unix		partial				c	RDS archive
	Scribes		unix		yes				python	same as gedit
	TextMate			osx	yes	yes	30 day	closed		eu-editor project
Vim	Vim	win	unix		partial				c	eu-editor project

- **Indent** = Knows the language syntax and will indent structures such as if/end if automatically
- **Ex.err** = Reads ex.err file and jumps to error in source file.
- **Syntax** = Syntax Coloring
  - ready = Syntax file for Euphoria already is installed.

## Integrated Development Environments

Among the editors you will find Judith's IDE and wxIDE: Integrated Development Environments.

## Special Features

### ex.err handling

The editor can read ex.err and jump to errors in the source file

- Edita
- e-~TextEditor
- ~TextMate

### Highlighting EuDoc markup w/in comments

- e-~TextEditor
- ~MicroEmacs
- ~TextMate
- Vim

### Euphoria Template Markup Language

(Euphoria inside of HTML)

- e-~TextEditor
- ~MicroEmacs
- ~TextMate
- Vim



## CodeLite

Multi-platform. Open. Features wxWidgets.

Linux installation.

1. `sudo apt-key adv`
2. `sudo apt-add-repository 'deb http://repos.codelite.org/ubuntu/ trusty universe'`
3. `sudo apt-get update`

The latest version of CodeLite is now listed in the Synaptic Package Manager

- Install with Synaptic
- Or, install from terminal: `sudo apt-get install codelite wxcrafter`

# TUTORIAL



- `mightyobject`
- `thinkeuphoria`
- `tutorial`
- `demo`

# The All Mighty Object

Here's a interesting way to write programs: how would you like to write entire programs using just a single data-type (hint: Euphoria lets you do this), and understanding how this is possible will help you program with the other built-in data-types: atoms, integers, and sequences.

Some background first. A computer is a just a machine that, when it comes down to it, works only with numbers; Euphoria is a language that lets you direct how the computer is to work with those numbers. So when you think about it, only one data-type is really needed. In Euphoria that is called an **object** (some other languages call these variants). Any Euphoria object contains either a single number or a collection of zero or more other objects--that's it--really! The object data-type does it all.

A variable is a name given to an object; the name then stands for the value of that object. You create a variable by **declaring** an identifier (a unique name) to belong to a data-type. **Assign** a value to your identifier an you can use it for any purpose.

Declare a variable, named `x` , as an object:

```
object x
```

Assign a value to the variable:

```
x = 5
```

Hint: You can declare and assign in a single statement one or more variables:

```
object x = 5
```

```
object x = 3, y = 2
```

Hint: The `--` double dash is the Euphoria **line comment**; anything following a double dash to the end of the line is ignored by Euphoria. Comments are often colored red.

The value you give an object is anything you can imagine and is written in a style most convenient for you.

```
object z
```

```
-- any individual value in choice of format
-- these are Euphoria "atom" examples
```

```
z = 4           -- an integer
z = -5          -- a negative integer
z = 3.14159     -- a real number
z = 1_000_000   -- a million using underscores for spacing
z = 1e6         -- a million in scientific notation
z = 'a'         -- the character 'a'
```

```
-- any collection of values in choice of format
-- these are Euphoria "sequence" examples
-- use { } for grouping items as a list of objects
```

```
z = {}          -- empty
z = { 1, 2, 3, 4, 5 } -- a flat list of numbers
z = { {1,2}, {3,4}, {5,6} } -- a 2x3 matrix of numbers
z = "Hello Euphoria!" -- a character string
z = { "one", "two", "three" } -- strings inside a sequence
z = { {1,"one"}, {2,"two"}, {3,"three"} } -- mix values together
z =
`
```

```
Write anything
you want.
```

```
-- free-form raw text
```

An object variable may be huge and contain items that are deeply nested collections of other objects; an object may also be just a single number.

You can *do anything* with an object in your program.

Any Euphoria object is *very dynamic*: create, delete, extend, alter, reshape, change items, remove items, search, sort, ..., lots and lots of possibilities. It doesn't matter if your object contains numbers or text; everything works the same.

Objects are *very efficient*. You can make a copy of a monster object or use that monster object as an argument to a subroutine without cost! Internally, copies and arguments are just pointers to the original object. Euphoria doesn't change anything, make actual copies, or use more memory until it is absolutely required.

This code "looks" brutal:

```
object huge = repeat( "hello", 1_000_000_000 )
--> that is an object containing 1_000_000_000 copies of "hello"
huge = 0
--> the 1_000_000_000 items just vanished
```

First a monster object is created and in the next line it is reset to zero. The actual cost of executing this code is minimal. Very little memory is consumed, the copies of "hello" are virtual, and the code is executed quickly.

You have *great freedom* when writing programs. Euphoria delivers efficiency without demanding anything from you.

```
--> It is possible to write a program of any size using just the single object data-type.
```

Aside: A **real** number is limitless. Euphoria uses industry standard **IEEE values** for atoms. That means atoms have fifteen digits of accuracy and are limited in size. Sometimes calculation results are slightly off or even strangely wrong in special cases--but that's true for all programming languages.

## Displaying Objects

Euphoria converts all values into numbers that a computer can process; only numbers remain. At any time you can use `print` to display an object as *numbers*.

Hint: The `?` question mark is a Euphoria shortcut for the `print` procedure:

```
-- both ? and print have the same output
? { 3, 3.333 }
--> { 3, 3.33 }
print(1, { 3, 3.333 } )
--> { 3, 3.333 }
```

Using `print` to output a variety of objects:

```
-- everything is a number
print(1, {1,2,3,4} )
--> {1,2,3,4}
print(1, 1_000_000 )
--> 1000000
print(1, 'a')
--> 97
print(1, "Hello Euphoria")
--> {72,101,108,108,111,32,69,117,112,104,111,114,105,97}
```

Notice that all of the features that make data entry convenient will vanish: spacers in numbers are removed, quotation marks are removed, and individual characters become

integers. What remains is either a single number or a collection of numbers.

But, all is not lost! If you use puts you can display characters and character strings:

```
-- characters and character strings output as expected

puts(1, 'a')
--> a
puts(1, 97 )
--> a
puts(1, "Hello Euphoria")
--> Hello Euphoria
puts(1, {72,101,108,108,111,32,69,117,112,104,111,114,105,97} )
--> Hello Euphoria
```

The first 127 characters of Unicode are the traditional ASCII characters known as **plain text**. Euphoria programs are written in plain text using any editor. Each Unicode character has a number value and a **glyph** that can be displayed as text.

If a number has a "sensible value" then the puts procedure ( short for put string) will display text instead of numbers. Numbers that do not have a corresponding glyph will display as noise.

Note: A **string** is a "sequence of characters." A string object is therefore **a flat list of items**; we say that a string has a simple one-dimensional shape. You can put strings as items inside a larger object; the object is now multi-dimensional; this object contains text data but is no longer a character string. The puts procedure will only display a character string:

```
-- puts will display a character string
object str = "A string is flat."

-- puts will NOT display arbitrary text
object txt = { "An object ", "can contain text ",
               "but that object is ", "no longer a string." }
```

The output for puts:

```
puts(1, str)
--> A string is flat.

puts(1, txt)
--> error
-- sequence found inside character string
```

Hint: Using puts to display a elaborate object (even if it contains text) is common mistake.

Again, all is not lost! The display procedure is used to output complicated objects:

```
include std/console.e

display( txt )
-- {
-- "An object ",
-- "can contain text ",
-- "but that object is ",
-- "no longer a string."
-- }
```

The display procedure examines the items inside an object, makes an "educated guess", and displays the object in the most readable form it can.

Note: Before using display you have to write include std/console.e because this procedure is read from a standard library file.

Use `pretty_print`, another library procedure, to display an object in various ways:

```
include std/pretty.e

pretty_print(1, txt, {1} )
-- {
--   {65'A',110'n',32' ' ,111'o',98'b',106'j',101'e',99'c',116't',32' '},
--   {99'c',97'a',110'n',32' ' ,99'c',111'o',110'n',116't',97'a',105'i',110'n',
--   32' ' ,116't',101'e',120'x',116't',32' '},
--   {98'b',117'u',116't',32' ' ,116't',104'h',97'a',116't',32' ' ,111'o',98'b',
--   106'j',101'e',99'c',116't',32' ' ,105'i',115's',32' '},
--   {110'n',111'o',32' ' ,108'l',111'o',110'n',103'g',101'e',114'r',32' ' ,97'a',
--   32' ' ,115's',116't',114'r',105'i',110'n',103'g',46'.'}
-- }

pretty_print(1, txt, {2} )
-- {
--   "An object ",
--   "can contain text ",
--   "but that object is ",
--   "no longer a string."
-- }
```

When doing something fancy, it is assumed that you know what you want. You can specify output exactly with the `printf` procedure:

```
printf(1, "%s%s%s%s", txt)
--> An object can contain text but that object is no longer a string.
```

The second argument, `"%s%s%s%s"`, is a **format string** that determines the appearance of text when it is output. The codes `printf` uses are similar to those found in most languages.

## Data Object, Entity Object

Euphoria uses **data objects** which are pure data values with no limitations. All operators and subroutines work with these objects. When you learn something you can use that knowledge everywhere--Euphoria is simple.

OOP languages use **entity objects** where an identifier references data (called members) and subroutines (called methods). Every entity class behaves differently. You must put some effort into defining entity classes and then use inheritance and polymorphism to claw back some flexibility--OOP is not simple.

Like a truck, an OOP language is intrinsically bigger, harder to learn, and slower. OOP languages let you write big complicated applications and programmers enjoy using dot notation. The art of programming is knowing when you can use a language, Euphoria, that is simple, fast, and fun to program.

## Object, Atom, Integer, Sequence, User-Defined

Using only the object data-type is possible, but practical Euphoria programs are designed around **atom** and **sequence** data-types. It is best to view object and integer as **helper data-types**; they are used for special purposes.

The atom and sequence data-types are used for most programming. The object is the root data-type that makes Euphoria operate. Using an object for everything is overkill; we use object data-types sparingly and even call it one of the "helper" data-types. The **integer** is a branch of the atom type; it is also used as a "helper" data-type. When defining a **UDT** (user-defined data-type) you write a function that describes your custom data-type perfectly.

Euphoria data-types make writing programs simpler. You can choose between the limitless flexibility of an object and the rigour of custom data-types. Finally, for extra speed, you can turn off data-type checking for working programs.

- If a variable is to be used with only individual numbers then the atom is the best choice.
- If your data is always going to be a collection of values then the sequence data-type is the best choice.
- The integer is a subset of the atom data-type; use an integer for counting and indexing.
- The object data-type has special uses when you can not know in advance if a variable will be used for atom values or sequence values. For example a variable that reads a file normally gets a character string and is thus a sequence, but when the file is empty or end-of-file an atom is returned--the object data-type is the only way to read either a sequence or an atom.
- It is possible to define a user-defined data-type to exactly specify what values are permitted for a variable. These data-types are useful when developing a program since they catch errors.

Real programs are written with atom and sequence data-types and use object and integer as "helper" data-types. Larger programs can use user-defined data-types to advantage.

# Think Euphoria

**Think Euphoria** is "an introduction to programming featuring the Euphoria programming language." It is based on the work by Downey who wrote *Think Python*.

*Think Euphoria* is included as a separate PDF document.

If you are new to programming then *Think Euphoria* will introduce to the essential concepts needed to become a programmer.



## Sequence Anatomy

A **sequence** is "a list of items." An **item** is "any data-object: object, atom, integer, or sequence."

A typical sequence looks like:

```
{ a, b, c, d }
```

The { } braces delimit a list of items and commas are used to separate items: a, b, c, d. You may call an item an *element* if you wish to use a longer word.

Names like s, s1, s2 are used in the documentation as sequence identifiers used in examples. The keyword sequence is used to declare a variable of the sequence data-type.

```
sequence s
```

### Here is some Euphoria jargon used to talk about sequences.

The **first item** is "the one at the beginning of a list." The **last item** is "the one at the end of a list."

```
--      first      last
--      ↓          ↓
--      {  a,   b,   c,   d   }
```

The **length** of a sequence is "the count of items in the list." An item may be a sequence; even if a sequence contains other sequences it is still considered to be just one item.

The **head** is "the beginning direction of a sequence." A **tail** is "the ending direction of a sequence." Head and tail meanings are usually clear when used to describe how a sequence is used.

```
--      {  a,   b,   c,   d   }
--      ↑ ↑           ↑ ↑
--      head         tail
```

**Subscript** notation is "used to identify a specific item in a sequence." Subscript notation uses the identifier name, [ ] brackets, and index value to select an item of a sequence:

```
sequence      s = { a,   b,   c,   d   }
--              s[1] s[2] s[3] s[4]
--              first      last
```

**One based indexing** means "that the first item is the number one; the remaining items are identified by counting."

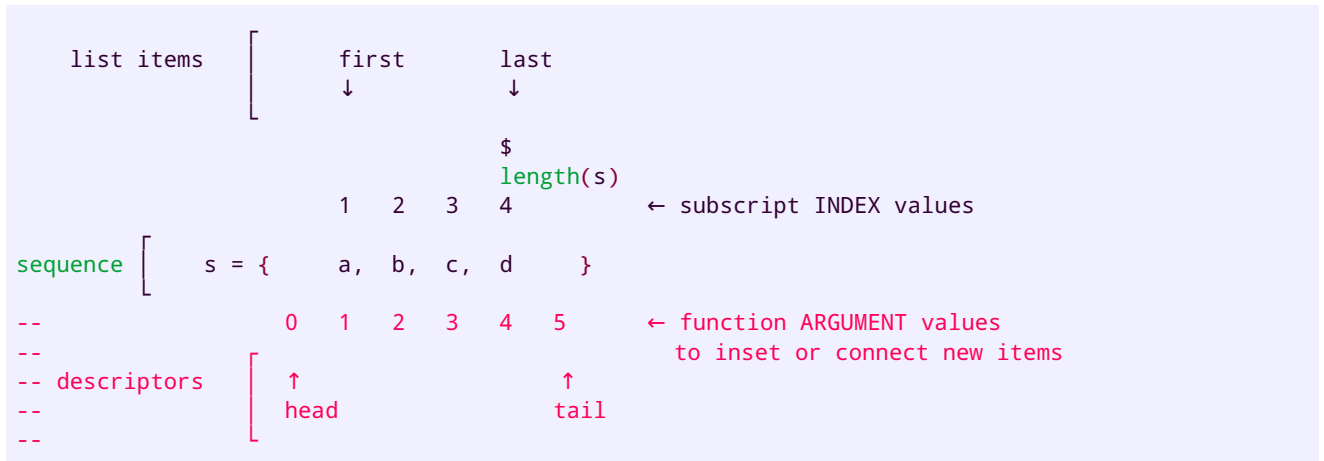
```
--      1 2 3 4   ← integer index values start at one
sequence s = { a, b, c, d }
```

The last item often gets special treatment when a sequence is used. The last item may be indexed as s[length(s)] or as s[\$] where the \$ dollar sign is used as a special index value:

```
sequence s = { a,   b,   c,   d   }
--              ↑
--              s[length(s)]  -- last item
--              s[$]          -- shorthand last item
```

The Interpreter enforces very strict control over index values. They can not be negative, smaller than one, or larger than the length of the list. Index values are always integer values; if you use a fractional value it will be truncated to an integer value before being

## Sequence Jargon Summarized



An **index** is "an integer ranging from one to the length of the sequence; Euphoria uses one-based indexing." The **first item** is "indexed as s[1]." The **last item** is "indexed as s[\$] which is the same as s[length(s)] ."

The **head** of a sequence is "the empty space before the first item s[1] ."

The **tail** of a sequence is "the empty space after the last item s[\$] ."

The **length** "for a sequence is the count of the *top-level* items in a sequence; an empty sequence has 0 zero length." Use the length function to get the length of any data-object. The length of an atom is 1 one.

## Insert: prepend, insert, append

The insert function has three arguments: who, what, where.

- Who: the sequence; the target of the renovation.
- What: item; data-object to be inset.
- Where: index; future location for the new item.

The insert function insets *one* new item just before the *where argument* of the target sequence.

If it helps, you can pretend that the head and tail are "invisible" items in a sequence list. That explains why you can use function argument values, like 0 zero, that are outside the values allowed for item indexing.



The functions prepend (insert before first item) and append (insert after last item) are special cases of the the insert function (insert before any item). All inserting functions increase length by one.

- prepend inserts *one* item at the head
- insert inserts *one* item anywhere in a sequence
- append inserts *one* item at the tail

INSERT	Head	Anywhere	Tail
	◀	▼	▶
prepend	prepend(s,x)		
insert	insert(s,x, 1)	insert(s,x,i)	insert(s,x, length(s)+1)
append			append(s,x)

The append function is probably the most commonly used function. Each time you use append you add one item to the end of your sequence list.

### Example 1:

```
-- the possible insert locations

sequence s = {1,2,3}
for i=0 to length(s)+1 do
  ? insert(si, 99, i )
end for

-- the output is

-- {99,1,2,3}
-- {99,1,2,3}      -- same as prepend(s,99)
-- {1,99,2,3}
-- {1,2,99,3}
-- {1,2,3,99}     -- same as append(s,99)
```

### Example 2:

```
-- prepend
-- inset at "head" before first item

s = { 'a', 'b', 'c', 'd' }
s = prepend(s, 100)
--> s is { 100, 'a', 'b', 'c', 'd' }

-- same as prepend
S = { 'a', 'b', 'c', 'd' }
S = insert(s, {"cat","dog"}, 1)
--> S is { {"cat","dog"}, 'a', 'b', 'c', 'd' }
```

### Example 3:

```
-- append
-- inset at "tail" after last item

s = { 'a', 'b', 'c', 'd' }
s = append(s, 100)
--> s is { 'a', 'b', 'c', 'd', 100 }

-- same as append
S = { 'a', 'b', 'c', 'd' }
S = insert(s, {"cat","dog"}, length(S)+1 )
--> S is { 'a', 'b', 'c', 'd', {"cat","dog"} }
```

A common use for append is to make a *list of lists*. It helps if you start with an empty list:

### Example 4:

```
sequence mylists = {}
sequence list1 = { "12", "23" }
sequence list2 = { "34", "45" }

mylists = append(mylists, list1 )
--> mylists is { { "12", "23" } }

mylists = append(mylists, list2 )
--> mylists is {
--           { "12", "23" },
--           { "34", "45" }
--           }
```

### Example 5:

A commonly used Euphoria idiom for making a list of lists is to write:

```
sequence mylists
sequence list1 = { "12", "23" }
sequence list2 = { "34", "45" }

mylists = append( { list1 }, list2 )
```

The result is the same as Example 4.

Possible Future enhancement

The prepend and append functions are especially flexible when used with atoms. If the first argument is an atom it will be promoted to a one item sequence; then the second argument is inset as one more item.

### Example 6:

```
/*
? prepend(3,12)      -- atom 3 is not promoted to sequence {3}
--> error
-- first argument of prepend must be a sequence
*/

? prepend( {3}, {12} )
--> {{12},3}

? append( {}, {'a','b'} ) -- object inset into the tail of empty sequence
--> { {'a','b'} }

? append( {'a','b'}, {} ) -- empty sequence inset into the tail of a sequence
--> {'a','b',{}}
```

## Splice: &, splice, &

**Splicing** "adds two objects *seamlessly* together to make a longer object." The final length is the sum of the two original lengths.

SPLICE	Head	Anywhere	Tail
	◀	▼	▶
<b>&amp;</b>	x & s		
<b>splice</b>	splice(s,x, 1)	splice(s,x,i)	splice(s,x, length(s)+1)
<b>&amp;</b>			s & x

## Splicing with &

**Concatenation** is "a binary operation using the & concatenation operator that joins two objects *seamlessly* together." When two atoms are concatenated they make a two item sequence. When an atom is concatenated to a sequence the atom is added as one item. Sequences are connected *tail to head*.

You can join a object x to the head of object s by writing x & s, or you can join object x to the tail of object s by writing s & x.

While we do not consider a number to have length it is convenient to think of an atom as having a length of one. This is because one atom concatenated to a sequence increases length by one; two atoms concatenated together make a sequence of length two.

Any data-object is just one item in a sequence; an atom, sequence, string, or nested sequence are all one item; each has a length of one.

### Example 1:

```

sequence s = 2 & 8
--> s is { 2, 8 } -- length 1 + 1 is two

sequence S = { 'a', 'b', 'c', 'd' }
S = S & 100
--> S is { 'a', 'b', 'c', 'd', 100 } -- length 4+1 is five
--> S is "abcd100" -- 'd' is ASCII 100

sequence g = "Hello"
g = g & " Euphoria"
--> g is "Hello Euphoria" -- length 5+9 is fourteen

sequence x = { "hello", 10, -2 }
x = { "open", "Euphoria" } & x
--> x is { "open", "Euphoria", "hello", 10, -2 } -- length 2+3 is five

```

## Connecting with splice

The splice function will "connect an object *seamlessly* anywhere in a sequence list; the new object appears *before* the 'index' value used as an argument."

The & concatenation operator and splice function both increase the length of a sequence by the length of object being connected.

Concatenating x & s is the same as splice(s,x,1) where the new object is connected before the first item.

Concatenating s & x is the same as splice(s,x,length(s)+1) where the new object is connected after the last item.

## Example 2

```

-- the possible splice locations

sequence si = { 1, 2, 3 }

for i=0 to length(si)+1 do
  ? splice(si, {99,99,99}, i )
end for

-- the output is

-- {99,99,99,1,2,3}
-- {99,99,99,1,2,3} -- same as {99,99,99} & {1,2,3}
-- {1,99,99,99,2,3}
-- {1,2,99,99,99,3}
-- {1,2,3,99,99,99} -- same as {1,2,3} & {99,99,99}

```

## Insert Contrasted with Splice

When the new object is an atom there is one case where an insert and a splice have the same result. This causes confusion to some new users of Euphoria. The thing to remember is that after you learn a syntax rule it will *always* work the same way.

## Example 1:

The append function and the & concatenation operator produce identical results when the item added is an atom:

```

sequence s = { 'a', 'b', 'c' }
object a = 10

? append(s, a)
-- "inserting"
--> s is { 'a','b','c', 10 }

```

```
? s & a
-- "splicing"
--> s is { 'a','b','c', 10 }
```

## Example 2:

The & concatenation operator and the append function produce different results when the item added is a sequence:

```
sequence s = { 'a', 'b', 'c' }
object a = {10}

? append(s, a)
-- "inserting"
--> s is { 'a','b','c', {10} }

? s & a
-- "splicing"
--> s is { 'a','b','c', 10 }
```

# Tutorial

Several small tutorial programs are included in the Euphoria installation.

These programs work with the Euphoria tracing facility. You can observe how Euphoria works line by line.

## Demos

Several demonstration programs are also included in Euphoria installation. These are short programs that illustrate more advanced ideas in Euphoria programming.

The `edx.ex` editor is also included as a demonstration program. This is a small text based editor. It features Euphoria syntax coloring, easy execution of source-code, and integration with Euphoria `ex.err` error files, and illustrates a range of programming concepts. The `edx` is useful for testing small Euphoria programs.



# LANGUAGE \_\_\_\_\_



- basics
- code
- name
- value
- express
- control
- organize
- send
- execute
- scope

# Principles

The look and feel of Euphoria syntax has been carefully crafted. The makes learning and using Euphoria easier.

## Comments

Single Line	Multi Line	Note
<pre>-- a comment</pre>	<pre>/* multi line comment */</pre>	Comments are ignored by the interpreter; comments have no impact program execution or performance.

## Documentation

You are encouraged to write documentation as part of your source-code. A -- line comment that has \*\* or tags marks the beginning of a section of comments that is used by the eudoc utility to produce an external documentation file.

Tagged Line Comment	Eudoc Will
<pre>--**</pre>	Extract description and signature of a subroutine.
<pre>--****</pre>	Extract generic text for documentation.

You can use *creole* wiki style text formatting to enhance your documentation.

## Syntax is Freeform, Uniform, and Obvious.

You can write code in any layout; you can arrange code to maximize its meaning to you.

Writing Freedom	Comments
<pre>for i=1 to 10 do display( i ) end for</pre>	Pretty code formatters are available.
<pre>for i=1 to 10 do display( i ) end for</pre>	

All structures, uniformly, start with a *keyword* and finish with *end keyword*.

Uniform Syntax	Comments
<pre>for -- -- end for</pre>	No invisible syntax. No extra punctuation.
<pre>if -- -- end if</pre>	
<pre>function -- -- end function</pre>	

When you nest structures you can easily untangle where each starts and finishes.

You can make your intended action obvious by adding a label:

Labeled Exit	Comments
--------------	----------

```
while 1 label "important" do
-- --
exit "important"
-- --
end while
```

Useful as nesting level increases.

## Conservation

All variables are declared, all subroutines are defined. No data-objects are spontaneously created or destroyed. All variables must be assigned a value before they may be used; there are no defaults for unassigned data-objects.

A variable does not spontaneously appear by just typing a name:

Declarations	Comments
<pre>atom x = 3 y = 3 -- error -- y is undeclared display( x+y )</pre>	<p>Typos are caught this way.</p>

## Explicit Assignments

Variables only change values after an assignment statement. The arguments to a subroutines are copies; the originals are never altered by a subroutine. The return value of a function has to be assigned to a variable to alter that variable.

Assignments	Comments
<pre>atom x = 2 function double( atom z ) return 2*z end function -- "z" is internal to function "double" -- it does not change external values double(x) -- variable "x" remains unchanged x = double(x) -- only now does the value of "x" change</pre>	<p>Internally, "copies" actually point to the original until a real copy must be made. Euphoria is fast and efficient.</p>

## Atom and Sequence

Programming simpler because an atom is *any* individual data-object (number or character) and a sequence is *any* list of data-objects.

However, you can define your own data-types if it makes your code easier to read.

## Object

An object can be *either* an atom or sequence. Use the object data-type for maximum flexibility and when you can not anticipate the required data-type.

## Count From One

The first item of sequence is number *one*.

One Based	Comments
<pre>sequence s = { 10, 200, 3000 } ? s[1] --&gt; is 10 ? s[3] --&gt; is 3000</pre>	<p>This is natural indexing.</p>

One-based-indexing is easier to read and creates fewer *off by one* errors than zero-based-indexing.

## Postive Indexing

The index value of sequence is always from one to the maximum which is the length of the sequence. This simplifies checking the correctness of index values.

## Accessible

You can always read and write to items in a sequence. There are no extra data-types with unique syntax rules resulting in mutable and immutable variables.

Polymorphic

Operators and subroutines often apply to all data-objects:

For example a math operator applies to both atoms and sequences:

Atom	Sequence	Comments
<pre>atom x=2 ? x*2 --&gt; x is 4</pre>	<pre>sequence s = { 0, -1, 3, 5 } ? s*2 --&gt; s is {0,-1,6,10}</pre>	Things work the same.

The same operators and functions work on all data regardless if it is numeric or text based. You can even intermix numbers and text items in a sequence.

Numbers	Text	Comment
<pre>splice( {1,2,3}, {10,20}, 3 ) --&gt; {1,2,10,20,3}</pre>	<pre>splice( "hello Euphoria", "open", 7 ) --&gt; hello openEuphoria</pre>	Things work the same.

Choose From Two

When learning and then when writing code remember that Euphoria syntax is divided into two choices: atom viewpoint and sequence viewpoint. If you choose wrong you only have one other option; this makes debugging Euphoria programs easier.

Choose Shallow or Deep

There are two views of a sequence. The shallow view sees the sequence as one complete unit. The deep viewpoint sees all of the items (and nested items) that make up a sequence.

T and F Comparisons

The = equals sign is used for comparisons *between atoms only*; the = equal sign results in a deep comparison of every item in a sequence down to the atom level. A comparison test for a branching structure must return a 1 one (true) or 0 zero (false) result; a comparison test needs the shallow comparison you get with a comparison function.

Shallow Comparison	Deep Comparison	Comment
<pre>? equal( "cat", "dog" ) --&gt; 0 -- that is: false</pre>	<pre>? "cat" = "dog" --&gt; {0,0,0} -- that is:  false, false, false</pre>	Works for all sequences.

Function Arguments

searching

fcnname( needle, haystack, options )

generic

fcnname(object, options)

No String Data-Type

Euphoria does not have a string data-type. The integer (or atom) data-type serve to represent any ASCII character up to 254 or Unicode code-point upto 127; a *string* sequence is then a list of integer ASCII values. The atom data-type can represent any UTF-32 code point; a *string* sequence is then a list of UTF-32 code points. A *string* sequence can also be a list of UTF-8 multi-byte characters. Subroutines will vary in their capability to work with these different encoding schemes.

? change this to 'core language'

# Preliminaries

## Comments

**Comments** are "text that is ignored by Euphoria during interpretation or compilation." Comments have no effect on execution speed. In this manual comments are shown in red.

There are three forms of comments:

- Single Line
- Multi Line
- #!

### Single Line

The **single line comment** starts with - - two dashes and extends to the end of the current line.

```
-- This is a comment which extends to the end of this line only.
```

Single line comments are used for short annotations.

Single line comments that begin with --\*\* or --\*\*\*\* are use by the eudoc.ex application to produce documentation like this manual. See the [miniguide](#) for more information.

### Multi Line

The **multi-line comment** starts with /\* and extends to the next occurrence of \*/. A multi-line comment can be very short or enclose many lines code.

```
/* This is a comment which  
   extends over a number  
   of text lines.  
*/
```

Multi-line comments are often used to *block out* sections of code during program debugging.

The **unix comment**, written *only* on the first line only of your program, begins with the two characters #! (sometimes called a **shebang**).

```
#!/usr/share/euphoria/bin/eui
```

Unix comments (on a Linux system) can be used to make source-code execute in a terminal without invoking the interpreter directly.

### #! Execution

On a unix system you can use the #! *unix comment* to make source-code execute as if it were an application.

The first line of your source-code must contain the path to the Euphoria interpreter such as: `#!/usr/share/euphoria/bin/eui` or possibly `#!/home/oe/euphoria/bin/eui`; only one `#!` comment is allowed.

- Write a file named `greetings` which contains

```
#!/usr/share/euphoria/bin/eui  
  
puts(1, "hello")
```

- Make the text file `greetings` executable. At the `$` of a terminal:

```
$ chmod +x greetings
```

- If you type `$ ./greetings` in a terminal, then this source-code will execute as if it were an application.

```
$ ./greetings  
hello
```

the same as if you had typed `$ eui greetings` instead.

## #! Shrouded Execution

The `#!` comment line should now contain the path to `eub` (instead of `eui`).

Typing `$ eushroud greetings` will produce a new file named `greetings.il` which may be executed as `$ ./greetings.il` from a terminal.

The result is the same as typing `$ eub greetings.il` directly.

## #! on Windows

On a *windows* system the `#!` comment is treated as an ordinary comment line; no source-code will be executed.

On some *windows* systems with the Apache Web server installed the `#!` will be recognized.

## Alphabet

Euphoria source-code is written as plain text.

Each character is a single integer value. Characters ranging from 1 to 127 are traditional ASCII characters; they correspond to the first 127 characters of the current UTF8 encoding system. On some systems extended ASCII characters that range up to 254 may also be used.

## Euphoria Code

Euphoria **source-code** is "a plain text file."

characters are integer values from 0 to 255

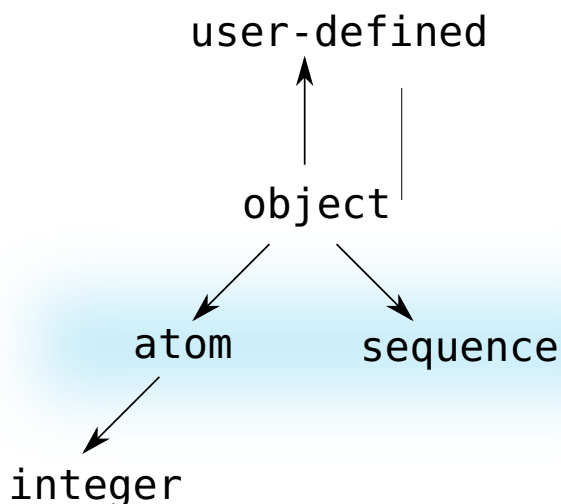
character values

*windows* 0 to 255 is possible

*unix* with typical UTF8 encoding use 0 to 127 (characters above 127 are multi-byte in UTF8 encoding)

*unix* without UTF8, then 0 to 255 is possible by setting a code page

# Value



## integer

A Euphoria integer is a mathematical integer restricted to the range  $-1,073,741,824$  to  $+1,073,741,823$ . As a result, a variable of the integer type, while allowing computations as fast as possible, cannot hold 32-bit machine addresses, even though the latter are mathematical integers. You must use the `atom` type for this purpose. Also, even though the product of two integers is a mathematical integer, it may not fit into an integer, and should be kept in an atom instead.

## atom

An atom can hold three kinds of data:

- Mathematical integers in the range  $-\text{power}(2,53)$  to  $+\text{power}(2,53)$
- Floating point numbers, in the range  $-\text{power}(2,1024)+1$  to  $+\text{power}(2,1024)-1$
- Large mathematical integers in the same range, but with a fuzz that grows with the magnitude of the integer.

$\text{power}(2,53)$  is slightly above  $9 \cdot 10^{15}$ ,  $\text{power}(2,1024)$  is in the  $10^{308}$  range.

Because of these constraints, which arise in part from common hardware limitations, some care is needed for specific purposes:

- The sum or product of two integers is an atom, but may not be an integer.
- Memory addresses, or handles acquired from anything non Euphoria, including the operating system, **must** be stored as an atom.
- For large numbers, usual operations may yield strange results:

```

integer n = power(2, 27) -- ok
integer n_plus = n + 1, n_minus = n - 1 -- ok
atom a = n * n -- ok
atom a1 = n_plus * n_minus -- still ok
? a - a1 -- prints 0, should be 1 mathematically
  
```

*This is not an Euphoria bug.* The IEEE 754 standard for floating point numbers provides for 53 bits of precision for any real number, and an accurate computation of `a-a1` would require 54 of them. Intel FPU chips do have 64 bit precision registers, but the low order 16 bits are only used internally, and Intel recommends against using them for high precision arithmetic. Their SIMD machine instruction set only uses the IEEE 754 defined format.

## sequence



A sequence is a type that is a *container*. A sequence has *elements* which can be accessed through their *index*, like in `my_sequence[3]`. sequences are so generic as being able to store all sorts of data structures: strings, trees, lists, anything. Accesses to sequences are always bound checked, so that you cannot read or write an element that does not exist, ever. A large amount of extraction and shape change operations on sequences is available, both as built-in operations and library routines. The elements of a sequence can have any type.

sequences are implemented very efficiently. Programmers used to pointers will soon notice that they can get most usual pointer operations done using sequence indexes. The loss in efficiency is usually hard to notice, and the gain in code safety and bug prevention far outweighs it.

## object

This type can hold any data Euphoria can handle, both atoms and sequences.

The object type returns 0 if a variable is not initialized, else 1.

## constants

These are variables that are assigned an initial value that can never change e.g.

```
constant MAX = 100
constant Upper = MAX - 10, Lower = 5
constant name_list = {"Fred", "George", "Larry"}
```

The result of any expression can be assigned to a constant, even one involving calls to previously defined functions, but once the assignment is made, the value of the constant variable is "locked in".

Constants may not be declared inside a subroutine.

## Specifying the type of a variable

So far you've already seen some examples of variable types but now we will define types more precisely.

Variable declarations have a type name followed by a list of the variables being declared. For example,

```
object a
global integer x, y, z
procedure fred(sequence q, sequence r)
```

The types: **object**, **sequence**, **atom** and **integer** are **predefined**. Variables of type **object** may take on *any* value. Those declared with type **sequence** must always be sequences. Those declared with type **atom** must always be atoms.

Variables declared with type **integer** must be atoms with integer values from -1073741824 to +1073741823 inclusive. You can perform exact calculations on larger integer values, up to about 15 decimal digits, but declare them as **atom**, rather than integer.

### Note:

In a procedure or function parameter list like the one for `fred` above, a type name may only be followed by a single parameter name.

### Performance Note:

Calculations using variables declared as integer will usually be somewhat faster than calculations involving variables declared as atom. If your machine has floating-point hardware, Euphoria will use it to manipulate atoms that are not integers. If your machine doesn't

have floating-point hardware (this may happen on old 386 or 486 PCs), Euphoria will call software floating-point arithmetic routines contained in **euid.exe** (or in *Windows*). You can force eui.exe to bypass any floating-point hardware, by setting an environment variable:

```
SET NO87=1
```

The slower software routines will be used, but this could be of some advantage if you are worried about the floating-point bug in some early Pentium chips.

## Values

All data **objects** in Euphoria are either **atoms** or **sequences**. An **atom** is a single numeric value. A **sequence** is a collection of objects, either atoms or sequences themselves. A sequence can contain any mixture of atom and sequences; a sequence does not have to contain all the same data type. Because the **objects** contained in a sequence can be an arbitrary mix of atoms or sequences, it is an extremely versatile data structure, capable of representing any sort of data.

A sequence is represented by a list of objects in brace brackets **{ }**, separated by commas with an optional sequence terminator, **\$**. Atoms can have any integer or double-precision floating point value. They can range from approximately -1e300 (minus one times 10 to the power 300) to +1e300 with 15 decimal digits of accuracy. Here are some Euphoria objects:

```
-- examples of atoms:
0
1000
98.6
-1e6
23_100_000
x
$

-- examples of sequences:
{2, 3, 5, 7, 11, 13, 17, 19}
{1, 2, {3, 3, 3}, 4, {5, {6}}}
{"jon", "smith", 52389, 97.25}
{} -- the 0-element sequence
```

By default, number literals use *base 10*, but you can have integer literals written in other bases, namely binary (*base 2*), octal (*base 8*), and hexadecimal (*base 16*). To do this, the number is prefixed by a 2-character code that lets Euphoria know which base to use.

Code	Base
0b	2 = <b>B</b> inary
0t	8 = <b>O</b> ctal
0d	10 = <b>D</b> ecimal
0x	16 = <b>H</b> exadecimal

For example:

```
0b101 --> decimal 5
0t101 --> decimal 65
0d101 --> decimal 101
0x101 --> decimal 257
```

Additionally, hexadecimal integers can also be written by prefixing the number with the '#' character.

For example:

```
#FE          -- 254
#A000        -- 40960
#FFFF00008   -- 68718428168
-#10         -- -16
```

Only digits and the letters A, B, C, D, E, F, in either uppercase or lowercase, are allowed in hexadecimal numbers. Hexadecimal numbers are always positive, unless you add a minus sign in front of the # character. So for instance

1. FFFFFFFF is a huge positive number (4294967295), **not** -1, as some machine-language programmers might expect.

Sometimes, and especially with large numbers, it can make reading numeric literals easier when they have embedded grouping characters. We are familiar with using commas (periods in Europe) to group large numbers by three-digit subgroups. In Euphoria we use the underscore character to achieve the same thing, and we can group them anyway that is useful to us.

```
atom big = 32_873_787    -- Set 'big' to the value 32873787
atom salary = 56_110.66 -- Set salary to the value 56110.66
integer defflags = #0323_F3CD
object phone = 61_3_5536_7733
integer bits = 0b11_00010_1
```

**Sequences** can be nested to any depth, i.e. you can have sequences within sequences within sequences and so on to any depth (until you run out of memory). Brace brackets are used to construct sequences out of a list of expressions. These expressions can be constant or evaluated at run-time. e.g.

```
{ x+6, 9, y*w+2, sin(0.5) }
```

All sequences can include a special *end of sequence* marker which is the \$ character. This is for convenience of editing lists that may change often as development proceeds.

```
sequence seq_1 = { 10, 20, 30, $ }
sequence seq_2 = { 10, 20, 30 }

equal(seq_1, seq_2) -- TRUE
```

The "**Hierarchical Objects**" part of the Euphoria acronym comes from the hierarchical nature of nested sequences. This should not be confused with the class hierarchies of certain object-oriented languages.

Why do we call them atoms? Why not just "numbers"? Well, an atom *is* just a number, but we wanted to have a distinctive term that emphasizes that they are indivisible (that's what "atom" means in Greek). In the world of physics you can 'split' an atom into smaller parts, but you no longer have an atom--only various particles. You can 'split' a number into smaller parts, but you no longer have a number--only various digits.

Atoms are the basic building blocks of all the data that a Euphoria program can manipulate. With this analogy, **sequences** might be thought of as "molecules", made from atoms and other molecules. A better analogy would be that sequences are like directories, and atoms are like files. Just as a directory on your computer can contain both files and other directories, a sequence can contain both atoms and other sequences (and *those* sequences can contain atoms and sequences and so on).

```
.      object
.      /  \
.      /   \
.      atom sequence
```

As you will soon discover, sequences make Euphoria very simple *and* very powerful.  
**Understanding atoms and sequences is the key to understanding Euphoria.**

## Character Strings and Individual Characters

A **character string** is just a sequence of characters. It may be entered in a number of ways ...

- Using double-quotes e.g.

```
"ABCDEFGH"
```

- Using raw string notation e.g.

```
-- Using back-quotes  
`ABCDEFGH`
```

or

```
-- Using three double-quotes  
"""ABCDEFGH"""
```

- Using binary strings e.g.

```
b"1001 00110110 0110_0111 1_0101_1010" -- ==> {#9,#36,#67,#15A}
```

- Using hexadecimal byte strings e.g.

```
x"65 66 67 AE" -- ==> {#65,#66,#67,#AE}
```

When you put too many hex characters together they are split up appropriately for you:

```
x"656667AE" -- 8-bit ==> {#65,#66,#67,#AE}
```

### The rules for double-quote strings are:

1. They begin and end with a double-quote character
2. They cannot contain a double-quote
3. They must be only on a single line
4. They cannot contain the TAB character
5. If they contain the back-slash '\' character, that character must immediately be followed by one of the special *escape* codes. The back-slash and escape code will be replaced by the appropriate single character equivalent. If you need to include double-quote, end-of-line, back-slash, or TAB characters inside a double-quoted string, you need to enter them in a special manner.

e.g.

```
"Bill said\n\t\"This is a back-slash \\ character\".\n"
```

Which, when displayed should look like ...

```
Bill said  
  "This is a back-slash \ character".
```

### The rules for raw strings are:

1. Enclose with three double-quotes """"..."""" or back-quote. `...`
2. The resulting string will never have any carriage-return characters in it.
3. If the resulting string begins with a new-line, the initial new-line is removed and any trailing new-line is also removed.
4. A special form is used to automatically remove leading whitespace from the source code text. You might code this form to align the source text for ease of reading. If the first line after the raw string start token begins with one or more underscore characters, the number of consecutive underscores signifies the maximum number of whitespace

characters that will be removed from each line of the raw string text. The underscores represent an assumed left margin width. **Note**, these leading underscores do not form part of the raw string text.

e.g.

```
-- No leading underscores and no leading whitespace
\
Bill said
    "This is a back-slash \ character".
\
```

Which, when displayed should look like ...

```
Bill said
    "This is a back-slash \ character".
```

```
-- No leading underscores and but leading whitespace
\
    Bill said
        "This is a back-slash \ character".
\
```

Which, when displayed should look like ...

```
    Bill said
        "This is a back-slash \ character".
```

```
-- Leading underscores and leading whitespace
\
____Bill said
        "This is a back-slash \ character".
\
```

Which, when displayed should look like ...

```
Bill said
    "This is a back-slash \ character".
```

Extended string literals are useful when the string contains new-lines, tabs, or back-slash characters because they do not have to be entered in the special manner. The back-quote form can be used when the string literal contains a set of three double-quote characters, and the triple quote form can be used when the text literal contains back-quote characters. If a literal contains both a back quote and a set of three double-quotes, you will need to concatenate two literals.

```
object TQ, BQ, QQ
TQ = `This text contains "" for some reason.`
BQ = """"This text contains a back quote ` for some reason.""
QQ = """"This text contains a back quote ` "" & `and "" for some reason.`
```

### The rules for binary strings are...

1. they begin with the pair `b"` and end with a double-quote `"` character
2. they can only contain binary digits (0-1), and space, underscore, tab, newline, carriage-return. Anything else is invalid.
3. an underscore is simply ignored, as if it was never there. It is used to aid readability.
4. each set of contiguous binary digits represents a single sequence element
5. they can span multiple lines
6. The non-digits are treated as punctuation and used to delimit individual values.

```
b"1 10 11_0100 01010110_01111000" == {0x01, 0x02, 0x34, 0x5678}
```

### The rules for hexadecimal strings are:

1. They begin with the pair `x"` and end with a double-quote `"` character
2. They can only contain hexadecimal digits (0-9 A-F a-f), and space, underscore, tab, newline, carriage-return. Anything else is invalid.
3. An underscore is simply ignored, as if it was never there. It is used to aid readability.
4. Each pair of contiguous hex digits represents a single sequence element with a value from 0 to 255
5. They can span multiple lines
6. The non-digits are treated as punctuation and used to delimit individual values.

```
x"1 2 34 5678_AbC" == {0x01, 0x02, 0x34, 0x56, 0x78, 0xAB, 0x0C}
```

Character strings may be manipulated and operated upon just like any other sequences. For example the string we first looked at "ABCDEFGH" is entirely equivalent to the sequence:

```
{65, 66, 67, 68, 69, 70, 71}
```

which contains the corresponding ASCII codes. The Euphoria compiler will immediately convert "ABCDEFGH" to the above sequence of numbers. In a sense, there are no "strings" in Euphoria, only sequences of numbers. A quoted string is really just a convenient notation that saves you from having to type in all the ASCII codes. It follows that "" is equivalent to {}. Both represent the sequence of zero length, also known as the **empty sequence**. As a matter of programming style, it is natural to use "" to suggest a zero length sequence of characters, and {} to suggest some other kind of sequence. An **individual character** is an **atom**. It must be entered using single quotes. There is a difference between an individual character (which is an atom), and a character string of length 1 (which is a sequence). e.g.

```
'B'  -- equivalent to the atom 66 - the ASCII code for B
"B"  -- equivalent to the sequence {66}
```

Again, 'B' is just a notation that is equivalent to typing 66. There are no "characters" in Euphoria, just numbers (atoms). However, it is possible to use characters without ever having to use their numerical representation.

Keep in mind that an atom is *not* equivalent to a one-element sequence containing the same value, although there are a few built-in routines that choose to treat them similarly.

## Escaped Characters

Special characters may be entered using a back-slash:

Code	Meaning
\n	newline
\r	carriage return
\t	tab
\\	backslash
\"	double quote
\'	single quote
\0	null
\e	escape
\E	escape
\b/d..d/	A binary coded value, the \b is followed by 1 or more binary digits. Inside strings, use the space character to delimit or end a binary value.
\x/hh/	A 2-hex-digit value, e.g. "\x5F" ==> {95}
\u/hhhh/	A 4-hex-digit value, e.g. "\u2A7C" ==> {10876}
\U/hhhhhhhh/	An 8-hex-digit value, e.g. "\U8123FEDC" ==> {2166619868}

For example, "Hello, World!\n", or "\n". The demonstration editor edx displays character strings in green.

Note that you can use the underscore character '\_' inside the \b, \x, \u, and \U values to aid readability, e.g. "\U8123\_FEDC" ==> {2166619868}

## Name

### Identifiers

An identifier is just the name you give something in your program. This can be a variable, constant, function, procedure, parameter, or namespace. An identifier must begin with either a letter or an underscore, then followed by zero or more letters, digits or underscore characters. There is no theoretical limit to how large an identifier can be but in practice it should be no more than about 30 characters.

Identifiers are **case-sensitive**. This means that "Name" is a different identifier from "name", or "NAME", etc...

Examples of valid identifiers:

```
n
color26
ShellSort
quick_sort
a_very_long_identifier_that_is_really_too_long_for_its_own_good
_alpha
```

Examples of invalid identifiers:

```
0n          -- must not start with a digit
^color26    -- must not start with a punctuation character
Shell Sort  -- Cannot have spaces in identifiers.
quick-sort  -- must only consist of letters, digits or underscore.
```

### Identifiers

**Identifiers**, which encompass all explicitly declared variable, constant or routine names, may be of any length. Upper and lower case are distinct. Identifiers must start with a letter or underscore and then be followed by any combination of letters, digits and underscores. The following **reserved words** have special meaning in Euphoria and cannot be used as identifiers:

and	export	public
as	fallthru	retry
break	for	return
by	function	routine
case	global	switch
constant	goto	then
continue	if	to
do	ifdef	type
else	include	until
elseif	label	while
elseifdef	loop	with
endif	namespace	without
end	not	xor
entry	or	
enum	override	
exit	procedure	

For example, the edx editor displays these words in blue.

The following are Euphoria built-in routines. It is best if you do not use these for your own identifiers:

abort	getenv	peek4s	system
and_bits	gets	peek4u	system_exec

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

Identifiers can be used in naming the following:

- procedures
- functions
- types
- variables
- constants
- enums



# Express

A **value** is "a number or numbers representing a data-object." The form of a number could be digital, binary, or even a printable character. The number 2324 is a value and so is 'a' the first letter of the alphabet.

An **expression** is "code that evaluates to a data-object." The simplest expression is just a value which evaluates to itself. Other expressions look like formulas which, after evaluation, give you an answer which is a number or possibly a collection of numbers.

An **operator** "represents some action on a value or several values producing a new data-object."

Programs run because expressions get evaluated and values get assigned to variables.

## Precedence Chart

When two or more operators follow one another in an expression, there must be rules to tell in which order they should be evaluated, as different orders usually lead to different results. It is common and convenient to use a **precedence order** on operators. Operators with the highest degree of precedence are evaluated first, then those with highest precedence among what remains, and so on.

The precedence of operators in expressions is as follows:

highest precedence

calls	function and type
unary- unary+ not	one argument expression
* /	multiplication division
+ -	addition subtraction
&	concatenation
< > <= >= = !=	comparisons
and or xor	logic
{ , , , }	sequence formation

lowest precedence

Thus  $2+6*3$  means  $2+(6*3)$  rather than  $(2+6)*3$ . Operators on the same line above have equal precedence and are evaluated left to right. You can force any order of operations by placing round brackets ( ) around an expression. For instance,  $6/3*5$  is  $2*5$ , not  $6/15$ .

Different languages or contexts may have slightly different precedence rules. You should be careful when translating a formula from a language to another; Euphoria is no exception. Adding superfluous parentheses to explicitly denote the exact order of evaluation does not cost much, and may help either readers used to some other precedence chart or translating to or from another context with slightly different rules.

Watch out for:

- and or
- \* /

The '=' equals symbol used in an **assignment statement** is not an operator; it is just part of the syntax of the language.

## Expressions

Euphoria lets you calculate results by forming expressions. A feature of Euphoria is calculation on entire sequences of data with one expression (in most other languages

you would have to construct a loop.) In Euphoria you can handle a sequence much as you would a single number. It can be copied, passed to a subroutine, or calculated upon as a unit. For example,

```
{1,2,3} + 5
```

is an expression that adds the sequence {1,2,3} and the atom 5 to get the resulting sequence {6,7,8}.

We will see more examples later.

## Relational Operators

The relational operators `<` `>` `<=` `>=` `=` `!=` each produce a 1 (true) or a 0 (false) result.

```
8.8 < 8.7    -- 8.8 less than 8.7 (false)
-4.4 > -4.3   -- -4.4 greater than -4.3 (false)
8 <= 7       -- 8 less than or equal to 7 (false)
4 >= 4       -- 4 greater than or equal to 4 (true)
1 = 10      -- 1 equal to 10 (false)
8.7 != 8.8   -- 8.7 not equal to 8.8 (true)
```

As we will soon see you can also apply these operators to sequences.

## Logical Operators

The logical operators `and`, `or`, `xor`, and `not` are used to determine the "truth" of an expression. e.g.

```
1 and 1      -- 1 (true)
1 and 0      -- 0 (false)
0 and 1      -- 0 (false)
0 and 0      -- 0 (false)

1 or 1       -- 1 (true)
1 or 0       -- 1 (true)
0 or 1       -- 1 (true)
0 or 0       -- 0 (false)

1 xor 1      -- 0 (false)
1 xor 0      -- 1 (true)
0 xor 1      -- 1 (true)
0 xor 0      -- 0 (false)

not 1        -- 0 (false)
not 0        -- 1 (true)
```

You can also apply these operators to numbers other than 1 or 0. The rule is: zero means false and non-zero means true. So for instance:

```
5 and -4     -- 1 (true)
not 6        -- 0 (false)
```

These operators can also be applied to sequences. See below.

In some cases **short\_circuit** evaluation will be used for expressions containing `and` or `or`. Specifically, short circuiting applies inside decision making expressions. These are found in the **if statement**, **while statement** and the **loop until statement**. More on this later.

## Arithmetic Operators

The usual arithmetic operators are available: add, subtract, multiply, divide, unary minus, unary plus.

```
3.5 + 3      -- 6.5
```

```
3 - 5    -- -2
6 * 2    -- 12
7 / 2    -- 3.5
-8.1     -- -8.1
+8       -- +8
```

Calculations always result in the correct calculation. For example if division between two integers results in a fraction then a decimal result will be produced:

```
? 10 / 4
--> 2.5
```

If a variable is declared to be an integer then you can not assign an integer value to it:

```
integer k = 1 / 4
-- error
```

You can assign any number to an atom variable.

Some languages provide an *integer division operator* and a *% modulo operator* where:

~~{{{ (b \* (a b) + a % b) is equal to a }}}}~~

In Euphoria this is expressed as:

```
b * floor(a/b) + mod(a,b)
```

is equal to a </eucode>

The floor and mod functions apply to both atoms and sequences

?? What is the efficiency difference between real 4.0 and integer 4 (no decimal)

?? Is it better to write (3.5 \* 4.0) or (3.5 \* 4)

Computing a result that is too big (i.e. outside of -1e300 to +1e300) will result in one of the special atoms **+infinity** or **-infinity**. These appear as inf or -inf when you print them out. It is also possible to generate nan or -nan. "nan" means "not a number", i.e. an undefined value (such as inf divided by inf). These values are defined in the IEEE floating-point standard. If you see one of these special values in your output, it usually indicates an error in your program logic, although generating inf as an intermediate result may be acceptable in some cases. For instance, 1/inf is 0, which may be the "right" answer for your algorithm.

If a calculation is going to produce a meaningless mathematical result then Euphoria issues an error message and the execution is aborted. Examples resulting in an error: division by zero, square root of a negative number, log of a negative number.

The unary + operator has does not change the value of a data-object and the interpreter does not calculate anything as a consequence. The only practical value for writing unary + is to emphasize, to the reader, that a value is positive.

See Also:

[[:trunc | remainder | mod

## Operations on Sequences

All of the relational, logical and arithmetic operators described above, as well as the math routines described in [Language Reference](#), can be applied to sequences as well as to single numbers (atoms).

When applied to a sequence, a unary (one operand) operator is actually applied to each element in the sequence to yield a sequence of results of the same length. If one of these elements is itself a sequence then the same rule is applied again recursively. e.g.

```
x = {-1, 2, 3, {4, 5}} -- x is {-1, -2, -3, {-4, -5}}
```

If a binary (two-operand) operator has operands which are both sequences then the two sequences must be of the same length. The binary operation is then applied to corresponding elements taken from the two sequences to get a sequence of results. e.g.

```
x = {5, 6, 7, 8} + {10, 10, 20, 100}
-- x is {15, 16, 27, 108}
x = {{1, 2, 3}, {4, 5, 6}} + {-1, 0, 1} -- ERROR: 2 != 3
-- but
x = {{1, 2, 3} + {-1, 0, 1}, {4, 5, 6} + {-1, 0, 1}} -- CORRECT
-- x is {{0, 2, 4}, {3, 5, 7}}
```

If a binary operator has one operand which is a sequence while the other is a single number (atom) then the single number is effectively repeated to form a sequence of equal length to the sequence operand. The rules for operating on two sequences then apply. Some examples:

```
y = {4, 5, 6}
w = 5 * y -- w is {20, 25, 30}

x = {1, 2, 3}
z = x + y -- z is {5, 7, 9}
z = x < y -- z is {1, 1, 1}

w = {{1, 2}, {3, 4}, {5}}
w = w * y -- w is {{4, 8}, {15, 20}, {30}}

w = {1, 0, 0, 1} and {1, 1, 1, 0} -- {1, 0, 0, 0}
w = not {1, 5, -2, 0, 0} -- w is {0, 0, 0, 1, 1}

w = {1, 2, 3} = {1, 2, 4} -- w is {1, 1, 0}

-- note that the first '=' is assignment, and the
-- second '=' is a relational operator that tests
-- equality
```

**Note:** When you wish to compare two strings (or other sequences), you should **not** (as in some other languages) use the '=' operator:

```
if "APPLE" = "ORANGE" then -- ERROR!
```

'=' is treated as an operator, just like '+', '\*' etc., so it is applied to corresponding sequence elements, and the sequences must be the same length. When they are equal length, the result is a sequence of ones and zeros. When they are not equal length, the result is an error. Either way you'll get an error, since an if-condition must be an atom, not a sequence. Instead you should use the equal built-in routine:

```
if equal("APPLE", "ORANGE") then -- CORRECT
```

In general, you can do relational comparisons using the compare built-in routine:

```
if compare("APPLE", "ORANGE") = 0 then -- CORRECT
```

You can use compare for other comparisons as well:

```
if compare("APPLE", "ORANGE") < 0 then -- CORRECT
-- enter here if "APPLE" is less than "ORANGE" (TRUE)
```

Especially useful is the idiom `compare(x, "") = 1` to determine whether `x` is a non empty sequence. `compare(x, "") = -1` would test for `x` being an atom, but `atom(x) = 1` does the same faster and is clearer to read.

## Subscripting of Sequences

**Subscript notation** is "used to select one item from a sequence list."

*Euphoria subscript*  
`x[2]`  

1	2	3	4	5	Index
---	---	---	---	---	-------

  
`x={ 5, 7.2, 9, 0.5, 13 }`  
 $x_2$   
*Mathematical subscript*

An **item** is "one element from a sequence list." A single item of a sequence may be selected by giving the index number inside [ ] brackets. Item numbers start at one. A fractional real value or an expression is permitted for indexing; all values are truncated down to an integer before being applied.

For example, if `x` contains {5, 7.2, 9, 0.5, 13} then `x[2]` is 7.2. Suppose we assign something different to `x[2]`:

```
x[2] = {11,22,33}
```

Then `x` becomes: {5, {11,22,33}, 9, 0.5, 13}. Now if we ask for `x[2]` we get {11,22,33} and if we ask for `x[2][3]` we get the atom 33. If you try to subscript with a number that is outside of the range 1 to the number of elements, you will get a subscript error. For example `x[0]`, `x[-99]` or `x[6]` will cause errors. So will `x[1][3]` since `x[1]` is not a sequence. There is no limit to the number of subscripts that may follow a variable, but the variable must contain sequences that are nested deeply enough. The two dimensional array, common in other languages, can be easily represented with a sequence of sequences:

```
x = {
  {5, 6, 7, 8, 9},      -- x[1]
  {1, 2, 3, 4, 5},      -- x[2]
  {0, 1, 0, 1, 0}       -- x[3]
}
```

where we have written the numbers in a way that makes the structure clearer. An expression of the form `x[i][j]` can be used to access any element.

The two dimensions are not symmetric however, since an entire "row" can be selected with `x[i]`, but you need to use `vslice` in the Standard Library to select an entire column. Other logical structures, such as n-dimensional arrays, arrays of strings, structures, arrays of structures etc. can also be handled easily and flexibly:

3-D array:

```
y = {
  {{1,1}, {3,3}, {5,5}},
  {{0,0}, {0,1}, {9,1}},
  {{-1,9}, {1,1}, {2,2}}
}

-- y[2][3][1] is 9
```

Array of strings:

```
s = {"Hello", "World", "Euphoria", "", "Last One"}

-- s[3] is "Euphoria"
-- s[3][1] is 'E'
```

A Structure:

```
employee = {
  {"John", "Smith"},
  45000,
  27,
  185.5
}
```

To access "fields" or elements within a structure it is good programming style to make up an enum that names the various fields. This will make your program easier to read. For the example above you might have:

```
enum NAME, SALARY, AGE, WEIGHT
enum FIRST_NAME, LAST_NAME

employees = {
  {"John","Smith"}, 45000, 27, 185.5}, -- a[1]
  {"Bill","Jones"}, 57000, 48, 177.2}, -- a[2]
  -- .... etc.
}

-- employees[2][SALARY] would be 57000.
```

The length built-in function will tell you how many elements are in a sequence. So the last element of a sequence *s*, is:

```
s[length(s)]
```

A short-hand for this is:

```
s[$]
```

Similarly,

```
s[length(s)-1]
```

can be simplified to:

```
s[$-1]
```

The *\$* may only appear between square braces and it equals the length of the sequence that is being subscripted. Where there's nesting, e.g.:

```
s[$ - t[$-1] + 1]
```

The first *\$* above refers to the length of *s*, while the second *\$* refers to the length of *t* (as you'd probably expect). An example where *\$* can save a lot of typing, make your code clearer, and probably even faster is:

```
longname[$][$] -- last element of the last element
```

Compare that with the equivalent:

```
longname[length(longname)][length(longname[length(longname)])]
```

### Subscripting and function side-effects:

In an assignment statement, with left-hand-side subscripts:

```
lhs_var[lhs_expr1][lhs_expr2]... = rhs_expr
```

The expressions are evaluated, and any subscripting is performed, from left to right. It is possible to have function calls in the right-hand-side expression, or in any of the left-hand-side expressions. If a function call has the side-effect of modifying the *lhs\_var*, it is not defined whether those changes will appear in the final value of the *lhs\_var*, once the assignment has been completed. To be sure about what is going to happen, perform the function call in a separate statement, i.e. do not try to modify the *lhs\_var* in two different ways in the same statement. Where there are no left-hand-side subscripts, you can always assume that the final value of the *lhs\_var* will be the value of *rhs\_expr*, regardless of any side-effects that may have changed *lhs\_var*.

## Euphoria data structures are almost infinitely flexible.

Arrays in many languages are constrained to have a fixed number of elements, and those elements must all be of the same type. Euphoria eliminates both of those restrictions by defining all arrays (sequences) as a list of zero or more Euphoria objects whose element count can be changed at any time. You can easily add a new structure to the employee sequence above, or store an unusually long name in the NAME field and Euphoria will take care of it for you. If you wish, you can store a variety of different employee "structures", with different sizes, all in one sequence. However, when you retrieve a sequence element, it is not guaranteed to be of any type. You, as a programmer, need to check that the retrieved data is of the type you'd expect, Euphoria will not. The only thing it will check is whether an assignment is legal. For example, if you try to assign a sequence to an integer variable, Euphoria will complain at the time your code does the assignment.

Not only can a Euphoria program represent all conventional data structures but you can create very useful, flexible structures that would be hard to declare in many other languages.

Note that expressions in general may not be subscripted, just variables. For example: `{5+2,6-1,7*8,8+1}[3]` is *not* supported, nor is something like: `date()[MONTH]`. You have to assign the sequence returned by `date` to a variable, then subscript the variable to get the month.

## Slicing of Sequences

A sequence of consecutive elements may be selected by giving the starting and ending element numbers. For example if `x` is `{1, 1, 2, 2, 2, 1, 1, 1}` then `x[3..5]` is the sequence `{2, 2, 2}`. `x[3..3]` is the sequence `{2}`. `x[3..2]` is also allowed. It evaluates to the zero length sequence `{}`. If `y` has the value: `{"fred", "george", "mary"}` then `y[1..2]` is `{"fred", "george"}`.

We can also use slices for overwriting portions of variables. After `x[3..5] = {9, 9, 9}` `x` would be `{1, 1, 9, 9, 9, 1, 1, 1}` We could also have said `x[3..5] = 9` with the same effect. Suppose `y` is `{0, "Euphoria", 1, 1}`. Then `y[2][1..4]` is "Euph". If we say `y[2][1..4] = "ABCD"` then `y` will become `{0, "ABCDoria", 1, 1}`.

In general, a variable name can be followed by 0 or more subscripts, followed in turn by 0 or 1 slices. Only variables may be subscripted or sliced, not expressions.

We need to be a bit more precise in defining the rules for **empty slices**. Consider a slice `s[i..j]` where `s` is of length `n`. A slice from `i` to `j`, where `j = i - 1` and `i >= 1` produces the **empty sequence**, even if `i = n + 1`. Thus `1..0` and `n + 1..n` and everything in between are legal **(empty) slices**. Empty slices are quite useful in many algorithms. A slice from `i` to `j` where `j < i - 1` is illegal, i.e. "reverse" slices such as `s[5..3]` are not allowed.

We can also use the `$` shorthand with slices, e.g.

```
s[2..$]
s[5..$-2]
s[$-5..$]
s[$][1..floor($/2)] -- first half of the last element of s
```

## Concatenation of Sequences and Atoms - The '&' Operator

Any two objects may be concatenated using the `&` operator. The result is a sequence with a length equal to the sum of the lengths of the concatenated objects. e.g.

```
{1, 2, 3} & 4           -- {1, 2, 3, 4}
4 & 5                   -- {4, 5}
{{1, 1}, 2, 3} & {4, 5} -- {{1, 1}, 2, 3, 4, 5}
x = {}
y = {1, 2}
```

```
y = y & x           -- y is still {1, 2}
```

You can delete element *i* of any sequence *s* by concatenating the parts of the sequence before and after *i*:

```
s = s[1..i-1] & s[i+1..length(s)]
```

This works even when *i* is 1 or `length(s)`, since `s[1..0]` is a legal empty slice, and so is `s[length(s)+1..length(s)]`.

## Sequence-Formation

Finally, sequence-formation, using braces and commas:

```
{a, b, c, ... }
```

is also an operator. It takes *n* operands, where *n* is 0 or more, and makes an *n*-element sequence from their values.

```
x = {apple, orange*2, {1,2,3}, 99/4+foobar}
```

The sequence-formation operator is listed at the bottom of the a [precedence chart](#).

## Multiple Assignment

Special sequence notation on the left hand side of an assignment allows assignment to multiple variables in a single statement.

```
'{ <variable> [, <variable> ] }' = { <expression> [, <expression> ] }'
```

- RHS items are a unique list of variables (no duplicates).
- RHS does not allow sequence items or slices.
- RHS can use a ? question mark placeholder for assignments that will be skipped.
- LHS must have sufficient values to be assigned to the RHS
- LHS can have surplus values; they will be ignored.

### Example 1

Multiple assignment is useful with functions that return multiple values in a sequence; such as the `value` function:

```
? value( "100" )
--> { GET_SUCCESS, 100 }

atom success, val
{ success, val } = value( "100" )
--> success is GET_SUCCESS
--> val is 100
```

### Example 2

It is also possible to ignore some of the values in the right hand side. Any items beyond the number supplied on the left hand side are ignored. Other values can also be ignored by using a ? question mark instead of a variable name:

```
{ ?, val } = value( "100" )
```

### Example 3

Variables may only appear once on the left hand side, however, they may appear on both the left and right hand side. For instance, to swap the values of two variables:

```
{ a, b } = { b, a }
```



## Example 4

You can not have a sequence item or a slice in a multiple assignment:

```
-- error
{ s[1], b } = { 10, 15 }
```

## Other Operations on Sequences

Some other important operations that you can perform on sequences have English names, rather than special characters. These operations are built-in to **eui.exe/euiw.exe**, so they'll always be there, and so they'll be fast. They are described in detail in the [Language Reference](#), but are important enough to Euphoria programming that we should mention them here before proceeding. You call these operations as if they were subroutines, although they are actually implemented much more efficiently than that.

### length(sequence s)

Returns the length of a sequence s.

This is the number of elements in s. Some of these elements may be sequences that contain elements of their own, but length just gives you the "top-level" count. Note however that the length of an atom is always 1.

```
length({5,6,7})      -- 3
length({1, {5,5,5}, 2, 3}) -- 4 (not 6!)
length({})           -- 0
length(5)            -- 1
```

### repeat(object o1, integer count)

Returns a sequence that consists of an item repeated count times.

```
repeat(0, 100)      -- {0,0,0,...,0}   i.e. 100 zeros
repeat("Hello", 3)  -- {"Hello", "Hello", "Hello"}
repeat(99,0)        -- {}
```

The item to be repeated can be any atom or sequence.

### append(sequence s1, object o1)

Returns a sequence by adding an object o1 to the end of a sequence s1.

```
append({1,2,3}, 4)      -- {1,2,3,4}
append({1,2,3}, {5,5,5}) -- {1,2,3,{5,5,5}}
append({}, 9)           -- {9}
```

The length of the new sequence is always 1 greater than the length of the original sequence. The item to be added to the sequence can be any atom or sequence.

### prepend(sequence s1, object o1)

Returns a new sequence by adding an element to the beginning of a sequence s. e.g.

```
append({1,2,3}, 4)      -- {1,2,3,4}
prepend({1,2,3}, 4)     -- {4,1,2,3}

append({1,2,3}, {5,5,5}) -- {1,2,3,{5,5,5}}
prepend({}, 9)           -- {9}
append({}, 9)           -- {9}
```

The length of the new sequence is always one greater than the length of the original sequence. The item to be added to the sequence can be any atom or sequence.

These two built-in functions, `append` and `prepend`, have some similarities to the concatenate operator, `&`, but there are clear differences. e.g.

```
-- appending a sequence is different
append({1,2,3}, {5,5,5}) -- {1,2,3,{5,5,5}}
{1,2,3} & {5,5,5}        -- {1,2,3,5,5,5}

-- appending an atom is the same
append({1,2,3}, 5)       -- {1,2,3,5}
{1,2,3} & 5               -- {1,2,3,5}
```

## `insert(sequence in_what, object what, atom position)`

This function takes a target sequence, `in_what`, shifts its tail one notch and plugs the object `what` in the hole just created. The modified sequence is returned. For instance:

```
s = insert("Joe", 'h', 3)      -- s is "Johe", another string
s = insert("Joe", "h", 3)     -- s is {'J','o',{'h'},'e'}, not a string
s = insert({1,2,3}, 4, -0.5)  -- s is {4,1,2,3}, like prepend()
s = insert({1,2,3}, 4, 8.5)   -- s is {1,2,3,4}, like append()
```

The length of the returned sequence is one more than the one of `in_what`. This is the same rule as for `append` and `prepend` above, which are actually special cases of `insert`.

## `splice(sequence in_what, object what, atom position)`

If `what` is an atom, this is the same as `insert`. But if `what` is a sequence, that sequence is inserted as successive elements into `in_what` at position. Example:

```
s = splice("Joe", 'h', 3)
  -- s is "Johe", like insert()
s = splice("Joe", "hn Do", 3)
  -- s is "John Doe", another string
s = splice("Joh", "n Doe", 9.3)
  -- s is "John Doe", like with the & operator
s = splice({1,2,3}, 4, -2)
  -- s is {4,1,2,3}, like with the & operator in reversed order
```

The length of `splice(in_what, what, position)` always is `length(in_what) + length(what)`, like for concatenation using `&`.

= Control

## Branching Statements

### if statement

An **if statement** tests a condition to see whether it is true or false, and then depending on the result of that test, executes the appropriate set of statements.

The syntax of if is

```
IFSTMT ==: IFTEST [ ELSIF ... ] [ELSE] ENDIF
IFTEST ==: if ATOMEXPR [ LABEL ] then [ STMTBLOCK ]
ELSIF  ==: elsif ATOMEXPR then [ STMTBLOCK ]
ELSE   ==: else [ STMTBLOCK ]
ENDIF  ==: end if
```

### Description of syntax

- An *if statement* consists of the keyword **if**, followed by an *expression* that evaluates to an atom, optionally followed by a *label* clause, followed by the keyword **then**. Next is a set of zero or more statements. This is followed by zero or more *elsif* clauses. Next is an optional *else* clause and finally there is the keyword **end** followed by the keyword **if**.
- An *elsif* clause consists of the key word **elsif**, followed by an *expression* that evaluates to an atom, followed by the keyword **then**. Next is a set of zero or more statements.
- An *else* clause consists of the keyword **else** followed by a set of zero or more statements.

In Euphoria, *false* is represented by an atom whose value is zero and *true* is represented by an atom that has any non-zero value.

- When an *expression* being tested is true, Euphoria executes the statements immediately following the **then** keyword after the *expression*, up to the corresponding **elsif** or **else**, whichever comes next, then skips down to the corresponding **end if**.
- When an *expression* is false, Euphoria skips over any statements until it comes to the next corresponding **elsif** or **else**, whichever comes next. If this is an **elsif** then its *expression* is tested otherwise any statements following the **else** are executed.

For example:

```
if a < b then
  x = 1
end if

if a = 9 and find(0, s) then
  x = 4
  y = 5
else
  z = 8
end if

if char = 'a' then
  x = 1
elsif char = 'b' or char = 'B' then
  x = 2
elsif char = 'c' then
  x = 3
else
  x = -1
end if
```

Notice that **elsif** is a contraction of *else if*, but it's cleaner because it does not require an **end if** to go with it. There is just one **end if** for the entire *if statement*, even when there are many **elsif** clauses contained in it.

The **if** and **elsif** expressions are tested using **short\_circuit** evaluation.

An *if statement* can have a *label clause* just before the first **then** keyword. See the section on [Header Labels](#). Note that an *elsif clause* can not have a label.

## tom 'switch' wishlist

switch day with break do ~~same as: switch day do~~ ~~switch day with fallthru do~~

allow “break” keyword folowing “with” (even if redundant)

fix error message when variable name is used ( not just “atom” but literal atom, literal string, ...

? is else fallthru meaningless ?

## switch Statement

The switch statement is an alternative to an if—elsif—else statement when branching is based on the value of *one* evaluated expression. The resulting control:

- Picks a case branch based on a literal or constant value.
- Is compact and easy to read.
- Is highly optimized for speed.

The syntax of a switch statement:

```
<val> ::= literal_atom | literal_string | constant | enum

switch <expr> [with fallthru] [label "<label name>" ] do

    case <val>[, <val2>, ...] then
        [code block]
        [ break [label] ] | [ fallthru ]

    case <val>[, <val2>, ...] then
        [code block]
        [ break [label] ] | [ fallthru ]

    case <val>[, <val2>, ...] then
        [code block]
        [ break [label] ] | [ fallthru ]

    ...

    [case else]
        [code block]
        [ break [label] ] | [ fallthru ]

end switch
```

A <val> in a case ... then must be either an literal atom, literal string, constant or enum. No variables are permitted.

A case ... then branch may contain one value or several values separated by commas.

A case ... then branch is selected based a value not found in any other branch:

- *Within* a branch duplicated values are quietly ignored.
- *Between* branches duplicated values result in an error.

### Example 1

```
include std/console.e
```

```
integer day = prompt_number("Enter day number: ", {1,7} )

switch day do
  case 1 then
    display( "Monday" )
  case 2 then
    display( "Tuesday" )
  case 3 then
    display( "Wednesday" )
  case else
    display( "invalid day number" )
end switch

-- sample run
--
-- Enter day number: 3
--> Wednesday

-- Enter day number: 4
--> invalid day number
```

## Example 2

The code in Example 1 written using an if statement.

```
include std/console.e

integer day = prompt_number("Enter day number: ", {1,7} )

if day = 1 then
  display( "Monday" )
elseif day = 2 then
  display( "Tuesday" )
elseif day = 3 then
  display( "Wednesday" )
else
  display( "invalid day number" )
end if
```

## break and fallthru

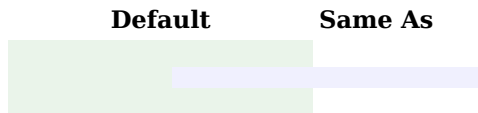
The switch statement is used to run a specific set of statements, depending on the value of an expression. It often replaces a set of if-elsif statements due to its ability to be highly optimized, thus much greater performance. There are some key differences, however. A switch statement operates upon the value of a single expression, and the program flow continues based upon defined cases.

Multiple values for a single case can be specified by separating the values by commas. The same symbol (or literal) may not be used multiple times as a case for the same switch.

If two different symbols used as case values happen to have the same value, they must be in the same case...then statement, or an error will occur. If the parser can determine all values when the switch is parsed, then a compile time error will be thrown. Otherwise, the error will occur the first time that the switch is encountered. Likewise, when translating code, if the parser cannot determine all values at the time when the case values are parsed, the compilation will fail due to multiple case values in the emitted C code (it is assumed that the programmer should work out this sort of bug in interpreted mode).

## switch Features

Default break implied:



### Example 3

A featurefull switch statement could be written with if statements like this:

```
object temp = expression
object breaking = false

if equal(temp, val1) then
  [code block 1]
  [breaking = true]
end if
if not breaking and equal(temp, val2) then
  [code block 2]
  [breaking = true]
end if
if not breaking and equal(temp, val3) then
  [code block 3]
  [breaking = true]
end if
...

if not breaking then
  [code block 4]
  [breaking = true]
end if
```

By default, control flows to the end of the switch block when the next case is encountered. The default behavior can be modified in two ways. The default for a particular switch block can be changed so that control passes to the next executable statement whenever a new case is encountered by using with fallthru in the switch statement:

### Example

```
switch x with fallthru do
  case 1 then
    ? 1
  case 2 then
    ? 2
    break
  case else
    ? 0
end switch
```

Note that when with fallthru is used, the break statement can be used to jump out of the switch block. The behavior of individual cases can be changed by using the fallthru statement:

```
switch x do
  case 1 then
    ? 1
    fallthru
  case 2 then
    ? 2
  case else
```

```

    ? 0
end switch

```

Note that the break statement before case else was omitted, because the equivalent action is taken automatically by default.

```

switch length(x) do
  case 1 then
    -- do something
    fallthru
  case 2 then
    -- do something extra
  case 3 then
    -- do something usual

  case else
    -- do something else
end switch

```

The label "name" is optional and if used it gives a name to the switch block. This name can be used in nested switch break statements to break out of an enclosing switch rather than just the owning switch.

Example:

```

switch opt label "LBLa" do
  case 1, 5, 8 then
    FuncA()

  case 4, 2, 7 then
    FuncB()
    switch alt label "LBLb" do
      case "X" then
        FuncC()
        break "LBLa"

      case "Y" then
        FuncD()

      case else
        FuncE()
    end switch
    FuncF()

  case 3 then
    FuncG()
    break

  case else
    FuncH()
end switch
FuncM()

```

In the above, if opt is 2 and alt is "X" then it runs...

FuncB() FuncC() FuncM()

But if opt is 2 and alt is "Y" then it runs ...

FuncB() FuncD() FuncF() FuncG() FuncM()

In other words, the break "LBLa" skips to the end of the switch called "LBLa" rather than the switch called "LBLb".

## ifdef statement

The `ifdef` statement has a similar syntax to the `if` statement.

```
ifdef SOME_WORD then
  --... zero or more statements
elsifdef SOME_OTHER_WORD then
  --... zero or more statements
elsedef
  --... zero or more statements
end ifdef
```

Of course, the `elsifdef` and `elsedef` clauses are optional, just like `elsif` and `else` are optional in an `if` statement.

The major differences between `if` and `ifdef` statement are that `ifdef` is executed at parse time not runtime, and `ifdef` can only test for the existence of a defined word whereas `if` can test any boolean expression.

**Note** that since the `ifdef` statement executes at parse time, run-time values cannot be checked, only words defined by the `-D` command line switch, or by the `with define` directive, or one of the special predefined words.

The purpose of `ifdef` is to allow you to change the way your program operates in a very efficient manner. Rather than testing for a specific condition repeatedly during the running of a program, `ifdef` tests for it once during parsing and then generates the precise IL code to handle the condition.

For example, assume you have some debugging code in your application that displays information to the screen. Normally you would not want to see this display so you set a condition so it only displays during a 'debug' session. The first example below shows how you could do this just using the `if` statement, and the second example shows the same thing but using the `ifdef` statement.

```
-- Example 1. --
if find("-DEBUG", command_line()) then
  writeln("Debug x=[], y=[]", {x,y})
end if
```

```
-- Example 1. --
ifdef DEBUG then
  writeln("Debug x=[], y=[]", {x,y})
end ifdef
```

As you can see, they are almost identical. However, in the first example, everytime the program gets to this point in the code, it tests the command line for the `-DEBUG` switch before deciding to display the information or not. But in the second example, the existence of `DEBUG` is tested *once* at parse time, and if it exists then, Euphoria generates the IL code to do the display. Thus when the program is running then everytime it gets to this point in the code, it does **not** check that `DEBUG` exists, instead it already knows it does so it just does the display. If however, `DEBUG` did not exist at parse time, then the IL code for the display would simply be omitted, meaning that during the running of the program, when it gets to this point in the code, it does not recheck for `DEBUG`, instead it already knows it doesn't exist and the IL code to do the display also doesn't exist so nothing is displayed. This can be a much needed performance boost for a program.

Euphoria predefines some words itself:

## Euphoria Version Definitions

- **EU4** - Major Euphoria Version
- **EU4\_1** - Major and Minor Euphoria Version
- **EU4\_1\_0** - Major, Minor and Release Euphoria Version



Euphoria is released with the common version scheme of Major, Minor and Release version identifiers in the form of major.minor.release. When 4.1.1 is released, EU4\_1\_1 will be defined and EU4\_1 will still be defined, but EU4\_1\_0 will no longer be defined. When 4.2 is released, EU4\_1 will no longer be defined, but EU4\_2 will be defined. Finally, when 5.0 is released, EU4 will no longer be defined, but EU5 will be defined.

## Platform Definitions

- **CONSOLE** - Euphoria is being executed with the Console version of the interpreter (on windows, eui.exe, others are eui)
- **GUI** - Platform is Windows and is being executed with the GUI version of the interpreter (euiw.exe)
- **WINDOWS** - Platform is Windows (GUI or Console)
- **LINUX** - Platform is Linux
- **OSX** - Platform is Mac OS X
- **FREEBSD** - Platform is FreeBSD
- **OPENBSD** - Platform is OpenBSD
- **NETBSD** - Platform is NetBSD
- **BSD** - Platform is a BSD variant (FreeBSD, OpenBSD, NetBSD and OS X)
- **UNIX** - Platform is any Unix

## Architecture Definitions

Chip architecture:

- **X86**
- **X86\_64**
- **ARM**

Size of pointers and euphoria objects. This information can be derived from the chip architecture, but is provided for convenience.

- **BITS32**
- **BITS64**

Size of long integers. On Windows, long integers are always 32 bits. On other platforms, long integers are the same size as pointers. This information can also be derived from a combination of other architecture and platform ifdefs, but is provided for convenience.

- **LONG32**
- **LONG64**

## Application Definitions

- **EUI** - Application is being interpreted by eui.
- **EUC** - Application is being translated by euc.
- **EUC\_DLL** - Application is being translated by euc into a *DLL* file.
- **EUB** - Application is being converted to a bound program by eub.
- **EUB\_SHROUD** - Application is being converted to a shrouded program by eub.
- **CONSOLE** - Application is being translated, or converted to a bound *console* program by euc or eub, respectively.
- **GUI** - Application is being converted to a bound *Windows GUI* program by eub.

## Library Definitions

- **DATA\_EXECUTE** - Application will always get executable memory from allocate even when the system has Data Execute Protection enabled for the Euphoria Interpreter.

- **SAFE** - Enables safe runtime checks for operations for routines found in machine.e and dll.e
- **UCSTYPE\_DEBUG** - Found in include/std/ucstypes.e
- **CRASH** - Found in include/std/unittest.e

More examples

```
-- file: myproj.ex
puts(1, "Hello, I am ")
ifdef EUC then
    puts(1, "a translated")
end ifdef
ifdef EUI then
    puts(1, "an interpreted")
end ifdef
ifdef EUB then
    puts(1, "a bound")
end ifdef
ifdef EUB_SHROUD then
    puts(1, "a shrouded")
end ifdef
puts(1, " program.\n")
```

```
C:\myproj> eui myproj.ex
Hello, I am an interpreted program.
C:\myproj> euc -con myprog.ex
... translating ...
... compiling ...
C:\myproj> myprog.exe
Hello, I am a translated program.
C:\myproj> bind myprog.ex
...
C:\myproj> myprog.exe
Hello, I am a bound program.
C:\myproj> shroud myprog.ex
...
C:\myproj> eub myprog.il
Hello, I am a bound, shrouded program.
```

It is possible for one or more of the above definitions to be true at the same time. For instance, EUC and EUC\_DLL will both be true when the source file has been translated to a DLL. If you wish to know if your source file is translated and not a DLL, then you can

```
ifdef EUC and not EUC_DLL then
    -- translated to an application
end ifdef
```

## Using ifdef

You can define your own words either in source:

```
with define MY_WORD      -- defines
without define OTHER_WORD -- undefines
```

or by command line:

```
eui -D MY_WORD myprog.ex
```

This can handle many tasks such as change the behavior of your application when running on *Linux* vs. *Windows*, enable or disable debug style code or possibly work differently in demo/shareware applications vs. registered applications.

You should surround code that is not portable with ifdef like:

```
ifdef WINDOWS then
```

```

-- Windows specific code.
elseif
  include std/error.e
  crash("This program must be run with the Windows interpreter.")
end ifdef

```

When writing **include files** that you cannot run on some platform, issue a crash call in the **include file**. **Yet** make sure that public constants and procedures are defined for the unsupported platform as well.

```

ifndef UNIX then
  include std/bash.e
end ifdef

-- define exported and public constants and procedures for
-- OSX as well
ifndef WINDOWS or OSX then
  -- OSX is not supported but we define public symbols for it anyhow.

```

The reason for doing this is so that the user that includes your include file sees an "OS not supported" message instead of an "undefined reference" message.

Defined words must follow the same character set of an identifier, that is, it must start with either a letter or underscore and contain any mixture of letters, numbers and underscores. It is common for defined words to be in all upper case, however, it is not required.

A few examples:

```

for a = 1 to length(lines) do
  ifdef DEBUG then
    printf(1, "Line %i is %i characters long\n", {a, length(lines[a])})
  end ifdef
end for

sequence os_name
ifndef UNIX then
  include unix ftp.e
elseifdef WINDOWS then
  include win32 ftp.e
elseifdef
  crash("Operating system is not supported")
end ifdef

ifndef SHAREWARE then
  if record_count > 100 then
    message("Shareware version can only contain 100 records. Please register")
    abort(1)
  end if
end ifdef

```

The ifdef statement is very efficient in that it makes the decision only once during parse time and only emits the TRUE portions of code to the resulting interpreter. Thus, in loops that are iterated many times there is zero performance hit when making the decision. Example:

```

while 1 do
  ifdef DEBUG then
    puts(1, "Hello, I am a debug message\n")
  end ifdef
  -- more code
end while

```

If DEBUG is defined, then the interpreter/translator actually sees the code as being:

```

while 1 do
  puts(1, "Hello, I am a debug message\n")

```

```
-- more code  
end while
```

Now, if `DEBUG` is not defined, then the code the interpreter/translator sees is:

```
while 1 do  
  -- more code  
end while
```

Do be careful to put the numbers after the platform names for *Windows*:

```
-- This puts() routine will never be called  
-- even when run by the Windows interpreter!  
ifdef WINDOWS then  
  puts(1,"I am on Windows\n")  
end ifdef
```

# Organize

There are three kinds of subroutines in Euphoria: procedures, functions, and types.

## Procedures

These perform some computation and may contain a list of parameters, e.g.

```
procedure empty()
end procedure

procedure plot(integer x, integer y)
  position(x, y)
  puts(1, '*')
end procedure
```

There are a fixed number of named parameters, but this is not restrictive since any parameter could be a variable-length sequence of arbitrary objects. In many languages variable-length parameter lists are impossible. In C, you must set up strange mechanisms that are complex enough that the average programmer cannot do it without consulting a manual or a local guru.

A copy of the value of each argument is passed in. The formal parameter variables may be modified inside the procedure but this does not affect the value of the arguments. Pass by reference can be achieved using indexes into some fixed sequence.

### Performance Note:

The interpreter does not actually copy sequences or floating-point numbers unless it becomes necessary. For example,

```
y = {1,2,3,4,5,6,7,8.5,"ABC"}
x = y
```

The statement `x = y` does not actually cause a new copy of `y` to be created. Both `x` and `y` will simply "point" to the same sequence. If we later perform `x[3] = 9`, then a separate sequence will be created for `x` in memory (although there will still be just one shared copy of `8.5` and `"ABC"`). The same thing applies to "copies" of arguments passed in to subroutines.

For a number of procedures or functions--see below--some parameters may have the same value in many cases. The most expected value for any parameter may be given a default value. To pass the default value, use a question mark `?`, or omit the value. When the parameter is not the last in the list to the routine, you should use the `?` for clarity, rather than simply omitting the parameter, and having consecutive commas.

```
procedure foo(sequence s, integer n=1)
  ? n + length(s)
end procedure

foo("abc")      -- prints out 4 = 3 + 1. n was not specified, so was set to 1.
foo("abc", ? )  -- prints out 4 = 3 + 1. n was not specified, so was set to 1.
foo("abc", 3)   -- prints out 6 = 3 + 3
```

This is not limited to the last parameter(s):

```
procedure bar(sequence s="abc", integer n, integer p=1)
  ? length(s)+n+p
end procedure
```

```

bar(?, 2)      -- prints out 6 = 3 + 2 + 1
bar(, 2)       -- prints out 6 = 3 + 2 + 1.  Legal, but considered bad form.
bar(2)         -- errors out, as 2 is not a sequence
bar(?, 2, ?)   -- same as bar(,2)
bar(?, 2, 3)   -- prints out 8 = 3 + 22 + 3
bar({}, 2, ?)  -- prints out 3 = 0 + 2 + 1
bar()          -- errors out, second parameter is omitted,
               -- but doesn't have a default value

```

Any expression may be used in a default value. Parameters that have been already mentioned may even be part of the expression:

```

procedure baz(sequence s, integer n=length(s))
    ? n
end procedure

baz("abcd") -- prints out 4

```

## functions

These are just like procedures, but they return a value, and can be used in an expression, e.g.

```

function max(atom a, atom b)
    if a >= b then
        return a
    else
        return b
    end if
end function

```

## return statement

Any Euphoria object can be returned. You can, in effect, have multiple return values, by returning a sequence of objects. e.g.

```

return {x_pos, y_pos}

```

However, Euphoria does not have variable lists. When you return a sequence, you still have to dispatch its contents to variables as needed. And you cannot pass a sequence of parameters to a routine, unless using `call_func` or `call_proc`, which carries a performance penalty.

We will use the general term "subroutine", or simply "routine" when a remark is applicable to both procedures and functions.

Defaulted parameters can be used in functions exactly as they are in procedures. See the section above for a few examples.

## types

These are special functions that may be used in declaring the allowed values for a variable. A type must have exactly one parameter and should return an atom that is either true (non-zero) or false (zero). Types can also be called just like other functions. See [Specifying the Type of a variable](#).

Although there are no restrictions to using defaulted parameters with types, their use is so much constrained by a type having exactly one parameter that they are of little practical help there.

You cannot use a type to perform any adjustment to the value being checked, if only because this value may be the temporary result of an expression, not an actual variable.

## variables

These may be assigned values during execution e.g.

```
-- x may only be assigned integer values
integer x
x = 25

-- a, b and c may be assigned *any* value
object a, b, c
a = {}
b = a
c = 0
```

When you declare a variable you name the variable (which protects you against making spelling mistakes later on) and you define which sort of values may legally be assigned to the variable during execution of your program.

The simple act of declaring a variable does not assign any value to it. If you attempt to read it before assigning any value to it, Euphoria will issue a run-time error as "variable xyz has never been assigned a value".

To guard against forgetting to initialize a variable, and also because it may make the code clearer to read, you can combine declaration and assignment:

```
integer n = 5
```

This is equivalent to

```
integer n
n = 5
```

It is not infrequent that one defines a private variable that bears the same name as one already in scope. You can reuse the value of that variable when performing an initialization on declare by using a default namespace for the current [file](#):

```
namespace app

integer n
n=5

procedure foo()
    integer n = app:n + 2
    ? n
end procedure

foo() -- prints out 7
```

## Scope

### Why scopes, and what are they?

The *scope* of an identifier is the portion of the program where its declaration is in effect, i.e. where that identifier is *visible*.

Euphoria has many pre-defined procedures, functions and types. These are defined automatically at the start of any program. For example, the edx editor shows them in magenta. These pre-defined names are not reserved. You can override them with your own variables or routines.

It is possible to use a user-defined identifier before it has been declared, provided that it will be declared at some point later in the program.

For example, procedures, functions and types can call themselves or one another *recursively*. Mutual recursion, where routine A calls routine B which directly or

indirectly calls routine A, implies one of A or B being called before it is defined. This was traditionally the most frequent situation which required using the `routine_id` mechanism, but is now supported directly. See [Indirect Routine Calling](#) for more details on the `routine_id` mechanism.

## Defining the scope of an identifier

The scope of an identifier is a description of what code can 'access' it. Code in the same scope of an identifier can access that identifier and code not in the same scope cannot access it.

The scope of a **variable** depends upon where and how it is declared.

- If it is declared within a **for**, **while**, **loop** or **switch**, its scope starts at the declaration and ends at the respective **end** statement.
- In an **if** statement, the scope starts at the declaration and ends either at the next **else**, **elsif** or **end if** statement.
- If a variable is declared within a routine (known as a private variable) and outside one of the structures listed above, the scope of the variable starts at the declaration and ends at the routine's **end** statement.
- If a variable is declared outside of a routine (known as a module variable), and does not have a scope modifier, its scope starts at the declaration and ends at the end of the file it is declared in.

The scope of a **constant** that does not have a scope modifier, starts at the declaration and ends at the end of the file it is declared in.

The scope of a **enum** that does not have a scope modifier, starts at the declaration and ends at the end of the file it is declared in.

The scope of all **procedures**, **functions** and **types**, which do not have a scope modifier, starts at the beginning of the source file and ends at the end of the source file in which they are declared. In other words, these can be accessed by any code in the same file.

Constants, enums, module variables, procedures, functions and types, which do not have a scope modifier are referred to as **local**. However, these identifiers can have a scope modifier preceding their declaration, which causes their scope to extend beyond the file they are declared in.

- If the keyword **global** precedes the declaration, the scope of these identifiers extends to the whole application. They can be accessed by code anywhere in the application files.
- If the keyword **public** precedes the declaration, the scope extends to any file that explicitly includes the file in which the identifier is declared, or to any file that includes a file that in turn public includes the file containing the public declaration.
- If the keyword **export** precedes the declaration, the scope only extends to any file that directly includes the file in which the identifier is declared.

When you **include** a Euphoria file in another file, only the identifiers declared using a scope modifier are accessible to the file doing the include. The other declarations in the included file are invisible to the file doing the include, and you will get an error message, "Errors resolving the following references", if you try to use them.

There is a variant of the **include** statement, called **public include**, which will be discussed later and behaves differently on **public** symbols.

Note that **constant** and **enum** declarations must be outside of any subroutine.

Euphoria encourages you to restrict the scope of identifiers. If all identifiers were automatically global to the whole program, you might have a lot of naming conflicts, especially in a large program consisting of files written by many different programmers. A naming conflict might cause a compiler error message, or it could lead to a very subtle bug, where different parts of a program accidentally modify the same variable without being aware of it. Try to use the most restrictive scope that you can. Make variables **private** to one routine where possible, and where that is not possible, make them **local** to a file, rather than **global** to the whole program. And whenever an identifier needs to be known from a few files only, make it **public** or **export** so as to hide it from whoever does not need to see it -- and might some day define the same identifier.



For example:

```
-- sublib.e
export procedure bar()
?0
end procedure

-- some_lib.e
include sublib.e
export procedure foo()
?1
end procedure
bar() -- ok, declared in sublib.e

-- my_app.exw
include some_lib.e
foo() -- ok, declared in some_lib.e
bar() -- error! bar() is not declared here
```

Why not declare `foo` as global, as it is meant to be used anywhere? Well, one could, but this will increase the risks of name conflicts. This is why, for instance, all public identifiers from the standard library have **public** scope. **global** should be used rarely, if ever. Because earlier versions of Euphoria didn't have **public** or **export**, it has to remain there for a while. One should be very sure of not polluting any foreign file's symbol table before using **global** scope. Built-in identifiers act as if declared as **global** -- but they are not declared in any Euphoria coded file.

## Using namespaces

Euphoria namespaces are used to disambiguate between symbols (routines, variables, constants, etc) with the same names in different files. They may be declared as a default namespace in a file for the convenience of the users of that file, or they may be declared at the point where a file is included. Note that unlike namespaces in some other languages, this does not provide a sandbox around the symbols in the file. It is just an easy way to tell euphoria to look for a symbol in a particular file.

Identifiers marked as `global`, `public` or `export` are known as *exposed* variables because they can be used in files other than the one they were declared in.

All other identifiers can only be used within their own file. This information is helpful when maintaining or enhancing the file, or when learning how to use the file. You can make changes to the internal routines and variables, without having to examine other files, or notify other users of the include file.

Sometimes, when using include files developed by others, you will encounter a naming conflict. One of the include file authors has used the same name for a exposed identifier as one of the other authors. One of way of fixing this, if you have the source, is to simply edit one of the include files to correct the problem, however then you'd have repeat this process whenever a new version of the include file was released.

Euphoria has a simpler way to solve this. Using an extension to the include statement, you can say for example:

```
include johns_file.e as john
include bills_file.e as bill

john:x += 1
bill:x += 2
```

In this case, the variable `x` was declared in two different files, and you want to refer to both variables in your file. Using the *namespace identifier* of either `john` or `bill`, you can attach a prefix to `x` to indicate which `x` you are referring to. We sometimes say that `john` refers to one *namespace*, while `bill` refers to another distinct *namespace*. You can attach a namespace identifier to any user-defined variable, constant, procedure or function. You can do it to solve a conflict, or simply to make things clearer. A namespace identifier has local scope. It is known only within the file that declares it, i.e. the file that

contains the include statement. Different files might define different namespace identifiers to refer to the same included file.

There is a special, reserved namespace, **eu** for referring to built-in Euphoria routines. This can be useful when a built-in routine has been overridden:

```
procedure puts( integer fn, object text )
    eu:puts(fn, "Overloaded puts says: "& text )
end procedure

puts(1, "Hello, world!\n")
eu:puts(1, "Hello, world!\n")
```

Files can also declare a default namespace to be used with the file. When a file with a default namespace is included, if the include statement did not specify a namespace, then the default namespace will be automatically declared in that file. If the include statement declares a namespace for the newly included file, then the specified namespace will be available instead of the default. No two files can use the same namespace identifier. If two files with the same default namespaces are included, at least one will be required to have a different namespace to be specified.

To declare a default namespace in a file, the first token (whitespace and comments are ignored) should be 'namespace' followed by the desired name:

```
-- foo.e : this file does some stuff
namespace foo
```

A namespace that is declared as part of an include statement is local to the file where the include statement is. A default namespace declared in a file is considered a public symbol in that file. Namespaces and other symbols (e.g., variables, functions, procedures and types) can have the same name without conflict. A namespace declared through an include statement will mask a default namespace declared in another file, just like a normal local variable will mask a public variable in another file. In this case, rather than using the default namespace, declare a new namespace through the include statement.

Note that declaring a namespace, either through the include statement or as a default namespace does not **require** that every symbol reference must be qualified with that namespace. The namespace simply **allows** the user to deconflict symbols in different files with the same name, or to allow the programmer to be explicit about where symbols are coming from for the purposes of clarity, or to avoid possible future conflicts.

A qualified reference does not absolutely restrict the reference to symbols that actually reside within the specified file. It can also apply to symbols included by that file. This is especially useful for multi-file libraries. Programmers can use a single namespace for the library, even though some of the visible symbols in that library are not declared in the main file:

```
-- lib.e
namespace lib

public include sublib.e

public procedure main()
...

-- sublib.e
public procedure sub()
...

-- app.ex
include lib.e

lib:main()
lib:sub()
```

Now, what happens if you do not use 'public include'?

```
-- lib2.e
include sublib.e
...

-- app2.ex
include lib.e
lib:main()
lib:sub() -- error.  sub() is visible in lib2.e but not in app2.ex
```

## The visibility of public and export identifiers

When a file needs to see the public or exported identifiers in another file that includes the first file, the first file must include that other (including) file.

For example,

```
-- Parent file: foo.e --
public integer Foo = 1
include bar.e -- bar.e needs to see Foo
showit() -- execute a routine in bar.e
```

```
-- Included file: bar.e --
include foo.e -- included so I can see Foo
constant xyz = Foo + 1

public procedure showit()
? xyz
end procedure
```

*Public* symbols can only be seen by the file that explicitly includes the file where those public symbols are declared.

For example,

```
-- Parent file: foo.e --
include bar.e
showit() -- execute a public routine in bar.e
```

If however, a file wants a third file to also see the symbols that it can, it needs to do a public include.

For example,

```
-- Parent file: foo.e --
public include bar.e
showit() -- execute a public routine in bar.e

public procedure fooer()
. . .
end procedure
```

```
-- Appl file: runner.ex --
include foo.e
showit() -- execute a public routine that foo.e can see in bar.e
fooer() -- execute a public routine in foo.e
```

The public include facility is designed to make having a library composed of multiple files easy for an application to use. It allows the main library file to expose symbols in files that *it* includes as if the application had actually included them. That way, symbols meant for the end user can be declared in files other than the main file, and the library can still be organized however the author prefers without affecting the end user.

### Another example

Given that we have two files LIBA.e and LIBB.e ...

```
-- LIBA.e --
public constant
    foo1 = 1,
    foo2 = 2

export function foobarr1()
    return 0
end function

export function foobarr2()
    return 0
end function
```

and

```
-- LIBB.e --
-- I want to pass on just the constants not
-- the functions from LIBA.e.
public include LIBA.e
```

The export scope modifier is used to limit the extent that symbols can be accessed. It works just like public except that export symbols are only ever passed up one level only. In other words, if a file wants to use an export symbol, that file must include it explicitly.

In this example above, code in LIBB.e can see both the public and export symbols declared in LIBA.e (foo1, foo2 foobarr1 and foobarr2) because it explicitly includes LIBA.e. And by using the public prefix on the include of LIBA.e, it also allows any file that includes LIBB.e to the public symbols from LIBA.e but they will not see any export symbols declared in LIBA.e.

In short, a public include is used expose public symbols that are included, up one level but not any export symbols that were include.

## The complete set of resolution rules

**Resolution** is the process by which the interpreter determines which specific symbol will actually be used at any given point in the code. This is usually quite easy as most symbol names in a given scope are unique and so Euphoria does not have to choose between them. However, when the same symbol name is used in different but enclosing scopes, Euphoria has to make a decision about which symbol the coder is referring to.

When Euphoria sees an identifier name being used, it looks for the name's declaration starting from the current scope and moving outwards through the enclosing scopes until the name's declaration is found.

The hierarchy of scopes can be viewed like this:

```
global/public/export
  file
    routine
      block 1
        block 2
          ...
        block n
```

So, if a name is used at a block level, Euphoria will first check for its declaration in the same block, and if not found will check the enclosing blocks until it reaches the routine level, in which case it checks the routine (including parameter names), and then check the file that the block is declared in and finally check the global/public/export symbols.

By the way, Euphoria will not allow a name to be declared if it is already declared in the same scope, or enclosing block or enclosing routine. Thus the following examples are illegal...

```
integer a
```

```
if x then
  integer a -- redefinition not allowed.
end if
```

```
if x then
  integer a
  if y then
    integer a -- redefinition not allowed.
  end if
end if
```

```
procedure foo(integer a)
if x then
  integer a -- redefinition not allowed.
end if
end procedure
```

But note that this below is valid ...

```
integer a = 1
procedure foo()
  integer a = 2
  ? a
end procedure
? a
```

In this situation, the second declaration of 'a' is said to *shadow* the first one. The output from this example will be ...

```
2
1
```

Symbols all declared in the same file (be they in blocks, routines or at the file level) are easy to check by Euphoria for scope clashes. However, a problem can arise when symbol names declared as global/public/export in different files are placed in the same scope during include processing. As it is quite possible for these files to come from independent developers that are not aware of each other's symbol names, the potential for name clashes is high. A name clash is just when the same name is declared in the same scope but in different files. Euphoria cannot generally decide which name you were referring to when this happens, so it needs you help to resolve it. This is where the namespace concept is used.

A namespace is just a name that you assign to an include file so that your code can exactly specify where an identifier that your code is using actually comes from. Using a namespace with an identifier, for example:

```
include somefile.e as my_lib
include another.e
my_lib:foo()
```

enables Euphoria to resolve the identifier (foo) as explicitly coming from the file associated with the namespace "my lib". This means that if foo was also declared as global/public/export in *another.e* then that foo would be ignored and the foo in *somefile.e* would be used instead. Without that namespace, Euphoria would have complained (Errors resolving the following references:)

If you need to use both foo symbols you can still do that by using two different namespaces. For example:

```
include somefile.e as my_lib
include another.e as her_ns
my_lib:foo() -- Calls the one in somefile.e
her_ns:foo() -- Calls the one in another.e
```

Note that there is a reserved namespace name that is always in use. The special

namespace **eu** is used to let Euphoria know that you are accessing a built-in symbol rather than one of the same name declared in someone's file.

For example,

```
include somefile.e as my_lib
result = my_lib:find(something) -- Calls the 'find' in somefile.e
xy = eu:find(X, Y) -- Calls Euphoria's built-in 'find'
```

The controlling variable used in a **for statement** is special. It is automatically declared at the beginning of the loop block, and its scope ends at the end of the for-loop. If the loop is inside a function or procedure, the loop variable cannot have the same name as any other variable declared in the routine or enclosing block. When the loop is at the top level, outside of any routine, the loop variable cannot have the same name as any other file-scoped variable. You can use the same name in many different for-loops as long as the loops are not nested. You do not declare loop variables as you would other variables because they are automatically declared as atoms. The range of values specified in the for statement defines the legal values of the loop variable.

Variables declared inside other types of blocks, such as a **loop**, **while**, **if** or **switch** statement use the same scoping rules as a for-loop index.

## The override qualifier

There are times when it is necessary to replace a global, public or export identifier. Typically, one would do this to extend the capabilities of a routine. Or perhaps to supersede the user defined type of some public, export or global variable, since the type itself may not be global.

This can be achieved by declaring the identifier as **override**:

```
override procedure puts(integer channel,sequence text)
    eu:puts(log_file, text)
    eu:puts(channel, text)
end procedure
```

A warning will be issued when you do this, because it can be very confusing, and would probably break code, for the new routine to change the behavior of the former routine. Code that was calling the former routine expects no difference in service, so there should not be any.

If an identifier is declared global, public or export, but not override, and there is a built-in of the same name, Euphoria will not assume an override, and will choose the built-in. A warning will be generated whenever this happens.

## Deprecation

Beginning in Euphoria 4.1, procedures and functions can be marked as deprecated. Deprecation is a computer software term that assigns a status to a particular item to indicate that it should be avoided, typically because it has been superseded. Deprecated routines remain in the language or library but should be avoided.

The deprecate modifier will cause a warning to appear if that routine is used. It serves no more purpose but is a powerful way to keep an evolving library clean, slim and fit for the task. Instead of simply removing an old routine authors are encouraged to use the deprecate modifier on a routine and leave it a part of the library for at least one major version increment. It can then be removed. This allows your users time to upgrade their code to the new recommended routine. Deprecated routines should be included in your manual, state when and why they were deprecated and what is the path future for accomplishing the same task.

```
--**
-- Say hello to someone
--
-- Parameters:
```

```
-- * name - name of person to say hello to
--
-- Deprecated:
-- ##say_hello## has been deprecated in favor of the new greet routine.
--

deprecate public procedure say_hello(sequence name)
    printf(1, "Hello, %s\n", { name })
end procedure

public procedure greet(sequence name="World", sequence greeting="Hello")
    printf(1, "%s, %s\n", { greeting, name })
end procedure
```

When deprecating a routine, the keyword `deprecate` should occur before any scope modifier.

# Transact

To **transact** is "to conduct the business of data input and data output." A *transaction* generalizes the idea of data transfer to a wider reach than just thinking about simple *input/output* (I/O).

## Assignment

You can input data into a program by assigning values to variables and by typing literal values in expressions.

Editing source-code for data input is an unusual suggestion. But, Euphoria is a fast interpreter so for small programs (for your own use) editing source-code is surprisingly effective.

## Simple Output

**? print**

**puts**

**display**

See Also:

? | [print](#) | [puts](#) | [display](#)

## More Output

**printf**

**sprintf**

**pretty\_print**

**ppp**

**serialize**

See Also:

[printf](#) | [sprintf](#) | [pretty\\_print](#) | [ppp](#) | [std/serialize.e](#)

## Command Line

Everything you type on the command line to start a program is available to your application; this is true for both text and gui based applications.

See Also:

[Command Line Handling](#) | [std/cmdline.e](#)

## Unix Pipe

A traditional Unix technique is for an application to "pipe in" a file and "pipe out" the results to a new file. Applications can be chained together so that a few small programs



can provide tremendous flexibility.

## Keyboard

### Prompted Input

```
include std/console.e

object s = prompt_string( "Enter some text: " )

object a = prompt_number( "Enter a number: ", {1,10} )

object a = any_key()

object a = maybe_any_key()
```

#### See Also:

[prompt\\_string](#) | [prompt\\_number](#) | [any\\_key](#) | [maybe\\_any\\_key](#)

### Single Keypress

*Immediately* return a the keycode (key press) from the keyboard or -1 (minus one) if no key was pressed.

```
atom a = get_key()
```

Wait for a keypress and then return the keycode (key press) from the keyboard.

```
include std/console.e
atom a = wait_key()
```

#### See Also:

[get\\_key](#) | [wait\\_key](#)

## with / without

These special statements affect the way that Euphoria translates your program into internal form. Options to the with and without statement come in two flavors. One simply turns an option on or off, while the others have multiple states.

### On / Off options

Default	Option
without	<code>profile</code>
without	<code>profile_time</code>
without	<code>trace</code>
without	<code>batch</code>
with	<code>type_check</code>
with	<code>indirect_includes</code>
with	<code>inline</code>

with turns **on** one of the options and without turns **off** one of the options.

For more information on the `profile`, `profile_time` and `trace` options, see [Debugging and Profiling](#). For more information on the `type_check` option, see [Performance Tips](#).

There is also a rarely-used special with option where a code number appears after with. In previous releases this code was used by RDS to make a file exempt from adding to the statement count in the old "Public Domain" Edition. This is not used any longer, but does not cause an error.

You can select any combination of settings, and you can change the settings, but the changes must occur *between* subroutines, not within a subroutine. The only exception is that you can only turn on one type of profiling for a given run of your program.

An **included file** inherits the **with/without** settings in effect at the point where it is included. An included file can change these settings, but they will revert back to their original state at the end of the included file. For instance, an included file might turn off warnings for itself and (initially) for any files that it includes, but this will not turn off warnings for the main file.

**indirect\_includes**, This with/without option changes the way in which global symbols are resolved. Normally, the parser uses the way that files were included to resolve a usage of a global symbol. If without `indirect_includes` is in effect, then only direct includes are considered when resolving global symbols.

This option is especially useful when a program uses some code that was developed for a prior version of Euphoria that uses the pre-4.0 standard library, when all exposed symbols were global. These can often clash with symbols in the new standard library. Using without `indirect_includes` would not force a coder to use namespaces to resolve symbols that clashed with the new standard library.

Note that this setting does not propagate down to included files, unlike most with/without options. Each file begins with `indirect_includes` turned on.

**with batch**, Causes the program to not present the "Press Enter" prompt if an error occurs. The exit code will still be set to 1 on error. This is helpful for programs that run in a mode where no human may be directly interacting with it. For example, a CGI application or a CRON job.

You can also set this option via a [command line parameter](#).

### Complex with / without options

#### with / without warning

Any warnings that are issued will appear on your screen after your program has finished execution. Warnings indicate minor problems. A warning will never terminate the execution of your program. You will simply have to hit the Enter key to keep going -- which may stop the program on an unattended computer.

The forms available are ...

with warning

enables all warnings

without warning

disables all warnings

with warning {*warning name list*}

with warning = {*warning name list*}

enables only these warnings, and disables all other

without warning {*warning name list*}

without warning = {*warning name list*}

enables all warnings except the warnings listed

with warning &= {*warning name list*}

with warning += {*warning name list*}

enables listed warnings in addition to whichever are enabled already

without warning &= {*warning name list*}

without warning += {*warning name list*}

disables listed warnings and leaves any not listed in its current state.

with warning save

saves the current warning state, i.e. the list of all enabled warnings.  
This destroys any previously saved state.

with warning restore

causes the previously saved state to be restored.

without warning strict

overrides some of the warnings that the -STRICT command line option tests for, but only until the end of the next function or procedure. The warnings overridden are \* default\_arg\_type \* not\_used \* short\_circuit \* not\_reached \* empty\_case \* no\_case\_else

The **with/without warnings** directives will have no effect if the -STRICT command line switch is used. The latter turns on all warnings and ignores any **with/without warnings** statement. However, it can be temporarily affected by the "without warning strict" directive.

---

## Warning Names

---

Name	Meaning
none	When used with the with option, this turns off all warnings. When used with the without option, this turns on all warnings.
resolution	an identifier was used in a file, but was defined in a file this file doesn't (recursively) include.
short_circuit	a routine call may not take place because of short circuit evaluation in a conditional clause.
override	a built-in is being overridden
builtin_chosen	an unqualified call caused Euphoria to choose between a built-in and another global which does not override it. Euphoria chooses the built-in.
not_used	A variable has not been used and is going out of scope.
no_value	A variable never got assigned a value and is going out of scope.
custom	Any warning that was defined using the warning procedure.

not_reached	After a keyword that branches unconditionally, the only thing that should appear is an end of block keyword, or possibly a label that a goto statement can target. Otherwise, there is no way that the statement can be reached at all. This warning notifies this condition.
translator	An option was given to the translator, but this option is not recognized as valid for the C compiler being used.
cmdline	A command line option was not recognized.
mixed_profile	For technical reasons, it is not possible to use both with profile and with profile_time in the same section of code. The profile statement read last is ignored, and this warning is issued.
empty_case	In switch that have without fallthru, an empty case block will result in no code being executed within the switch statement.
default_case	A switch that does not have a case else clause.
default_arg_type	Reserved (not in use yet)
deprecated	Reserved (not in use yet)
all	Turns all warnings on. They can still be disabled by with/without warning directives.

## Example

```
with warning save
without warning &= (builtin_chosen, not_used)
. . . -- some code that might otherwise issue warnings
with warning restore
```

Initially, only the following warnings are enabled:

- resolution
- override
- builtin\_chosen
- translator
- cmdline
- mixed\_profile
- not\_reached
- custom

This set can be changed using -W or -X command line switches.

## with / without define

As mentioned about [ifdef statement](#), this top level statement is used to define/undefine tags which the ifdef statement may use.

The following tags have a predefined meaning in Euphoria:

- WINDOWS: platform is any version of Windows (tm) from '95 on to Vista and beyond
- WINDOWS: platform is any kind of Windows system
- UNIX: platform is any kind of Unix style system
- LINUX: platform is Linux
- FREEBSD: platform is FreeBSD
- OSX: platform is OS X for Macintosh
- SAFE: turns on a slower debugging version of memory.e called safe.e when defined. Switching mode by renaming files **no longer works**.
- EU4: defined on all versions of the version 4 interpreter
- EU4\_0: defined on all versions of the interpreter from 4.0.0 to 4.0.X
- EU4\_0\_0: defined only for version 4.0.0 of the interpreter

The name of a tag may contain any character that is a valid identifier character, that is A-Za-z0-9\_. It is not required, but by convention defined words are upper case.

## with / without inline

This directive allows coders some flexibility with inlined routines. The default is for

inlining to be on. Any routine that is defined when without inline is in effect will never be inlined.

with inline takes an optional integer parameter that defines the largest routine (by size of IL code) that will be considered for inlining. The default is 30.

# Scope

## Program Files

### Base and Include Files

All files in Euphoria are created as equals.

A **main file** is "the one with the filename used as an argument to invoke the Euphoria interpreter." A main file could have the `.ex` (any program or maybe a text program) or `.exw` (maybe a gui program) or have no extension at all (maybe a *unix* program). A main file has no special properties compared to any other file. A main file does not have to contain or call a main subroutine.

A **main subroutine** "contains the first executable line for your program; a call to the main subroutine would officially start your program execution." Euphoria does not require you to create and call a main subroutine. Some programmers like to create a main subroutine because they prefer to organize their code in that style. Either a procedure or a function could be used if you like this style of code writing.

A **base file** "contains an include statement that refers to an external *include* file." An **include file** is "the file mentioned by the include statement of a base file." The standard library consists of include files written in Euphoria. You can create your own include files. A base file and an include file have otherwise identical properties.

For example, you could have File A and File B as part of your program. File A is a base file when it includes File B; File B is a base file when it includes File A. If you want to share identifiers between any two files then A *must* include B and B *must* include A.

## Space and Level, Scope and Layer

Space and level is about *where* you can type your code. Scope and layer is about *visibility* of your code.

A *space* is where lines of code are written.

The *interpreter space* contains built-in identifiers; these subroutines are available to all other spaces.

*File-space* is where you save your program code. The *top-level* is where you start typing in a file; definitions, interpreter options, and statements can be written at the top-level. The *block-level* is found nested inside a structured statement; structured statements (and blocks as a result) can be nested; definitions and interpreter options can not be written in a block.

Most programs are composed of several files. Each file has *identical properties*; scope rules for each file are identical; any file can include another file; scope rules for identifiers are identical for each file. It is convenient to view an included file as being an *include space* inside a including file as a reminder that scope rules control what identifiers are visible to the base file.

Any space can be viewed as built from stacked *levels* of code. Top-level code is executed when the interpreter encounters each statement. The default is that no identifiers are visible to any external file. The scope modifier keywords `export`, `public`, and `global` create layers of code that allow limited visibility of identifiers to external files.

The purpose of scope is to prevent name-clashes. As programs get larger it becomes

increasingly difficult to invent unique names. By understanding how scope works you can reuse names and still keep them distinct from each other.

## Scope

**Scope** is "where an identifier is visible to the interpreter."

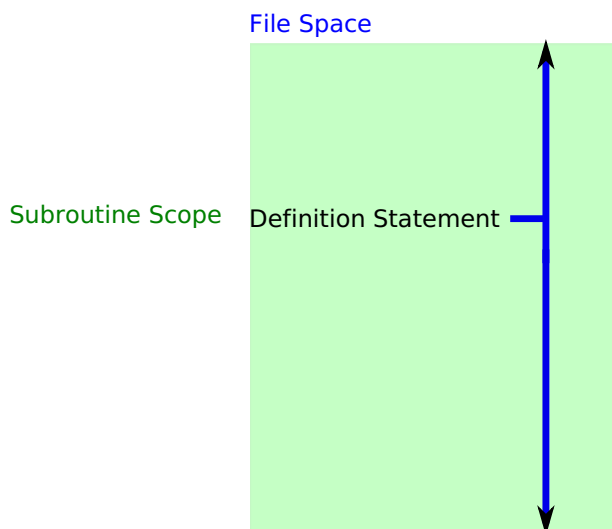
Subroutine **definition scope** (procedure, function, type) is "for the entire file space." Yes, you can define a subroutine at any top-level line of your source-code; a subroutine name is visible everywhere in your file space.

Data-object **declaration scope** (variable, constant, enum) is "from the point of declaration to the end of space." For example a variable declared at the top-level will be visible to the end of the file space. You can declare a variable inside a block; it will be visible only from the declaration to the end of the block.

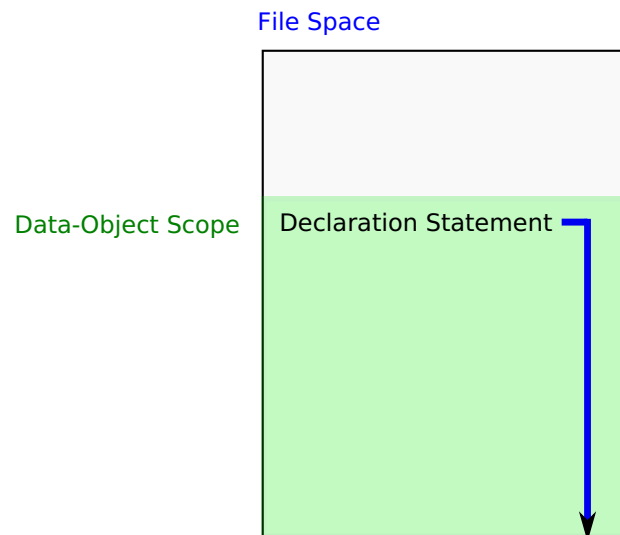
Important:

Every file is treated identically by the interpreter. Between *any two files* there is a balanced include and included relationship. An including file sees no identifiers from the included file. Equally, the included file sees no identifiers from the including file.

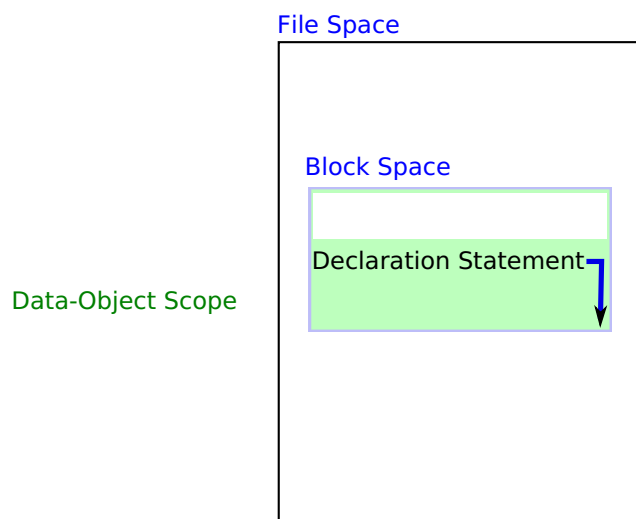
### Subroutine Visibility



### Data-object Visibility



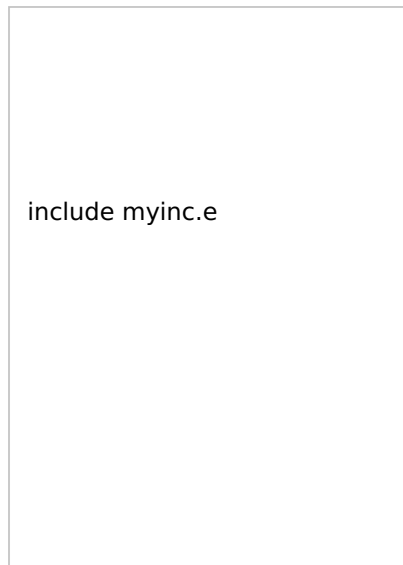
### Data-object Visibility



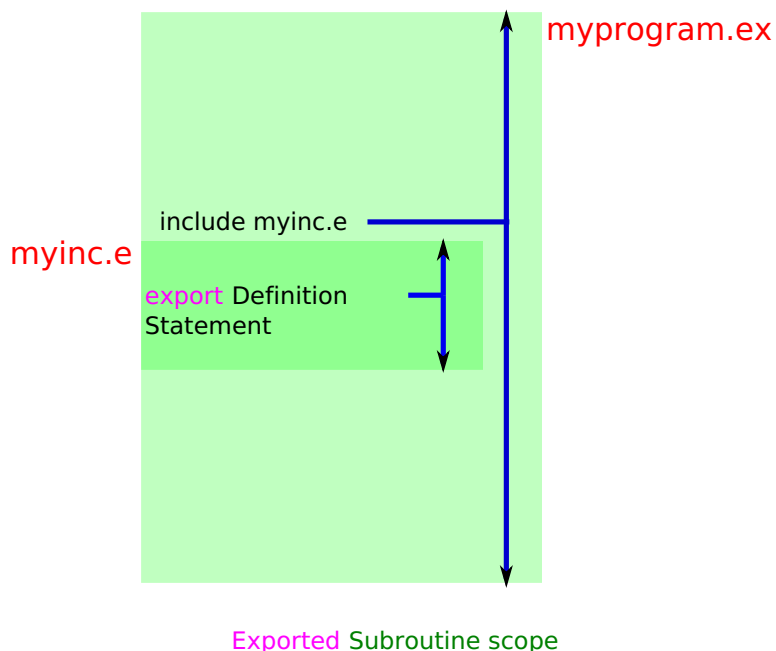
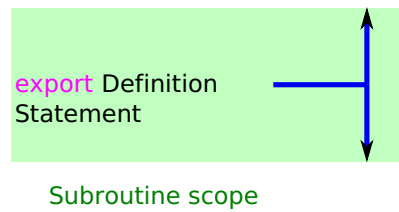
When a file is included the basic rules for scope are extended but only if there is a scope modifier (export, public, global) preceding the definition or declaration.

The *definition scope* is for the entire including file.

myprogram.ex



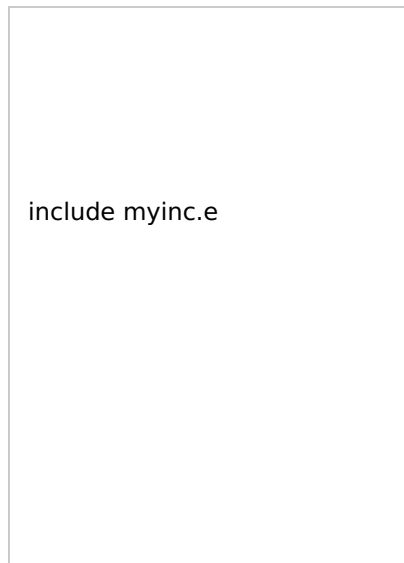
myinc.e



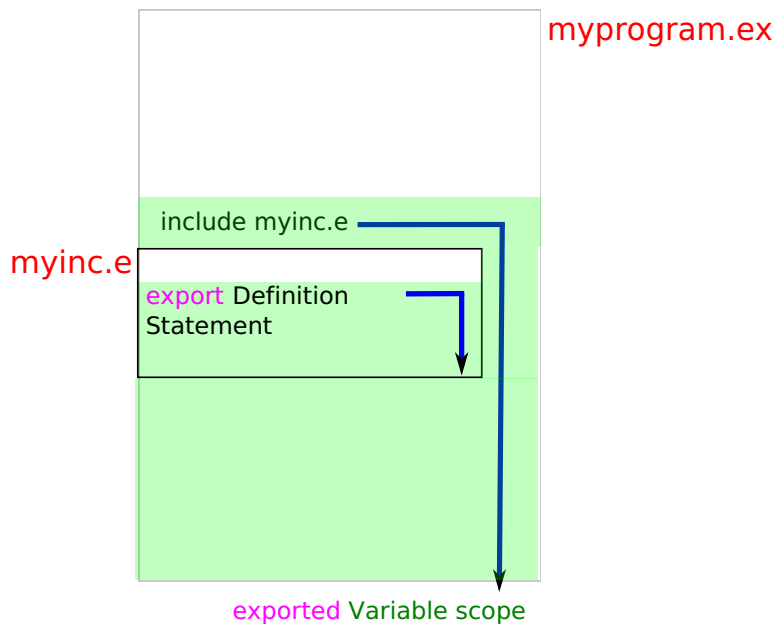
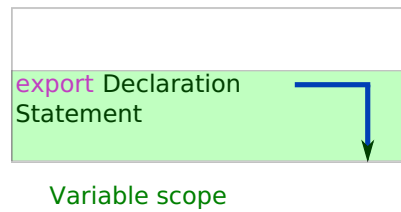
The *declaration scope* is *within* the included file is from the declaration to the end of local space. For the including file the *declaration scope* is from the include statement to the end of local space.



myprogram.ex



myinc.e



See Also:

[recursion](#) | [mutual recursion](#)

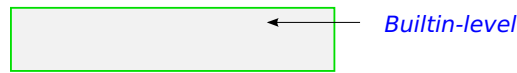
## Space

A **space** is "where write your lines source-code for a Euphoria program." A **subspace** is "a space nested within another space."

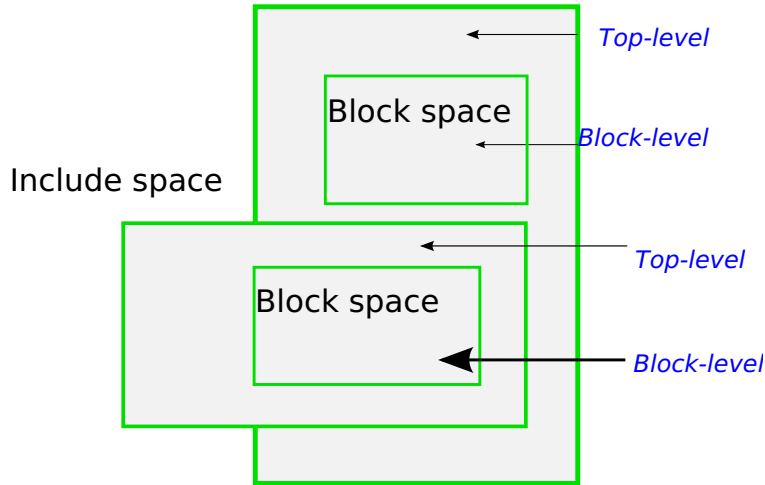
There are several *spaces* within a program:

- file space; provides top level space.
- block space; a sub space within a statement.
- include space; a sub space within a file.

## Interpreter space

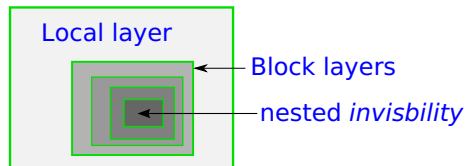


## File space



A **namespace** is "a name assigned to an include space." The namespace is used as a prefix joined with a : colon to a name to form a complete identifier. Including two files that use the same name results in a name-clash; just add the namespace prefix and the name becomes an identifier unique to the included file.

## File space



## Level

A **level** is "the permitted visibility of a statement."

The **global-level** is "at the level of the interpreter itself; the built-in names and subroutines from the interpreter are *globally* available to all files of a program." Using the global scope modifier you can make an identifier (defined subroutine or declared data object) visible to all files.

The **top-level** of a file "is where you can define subroutines, send options to the interpreter, and write various statements." The top-level is available the moment you start typing in a file; nothing extra is needed to write statements at the top-level. Top-level statements are executed by the interpreter as they are encountered.

The **export-level** "permits *one* including file to see identifiers marked with the export scope modifier."

The **public-level** "permits *several chained* including files to see public identifiers when the public include statement is used." When a file is include its top-level statements will be executed but only the first time the file is included; repeated include statements are quietly ignored.

A **local-level** is "restricted to the file space, include space, or block space where the identifier originates."

The **block-level** "is found in certain structured statements (subroutines, condition, repetition)." Statements in a block-level become visible only after a structured statement is executed. Block-level space does not permit subroutine definitions but allows data-object declarations and other statements. Each block is a *local-level*.

## File Space

A **file space** is "the contents of a file." A file space lets you write top-level statements and structured statements that contain a block or blocks of sub-space.

**Top-level** code "is a statement written in file space but outside the block of any sub-space." Some statements (subroutine definition statements, constant and enum declarations, include statement, option statements) can only be written at the top-level.

There are two kinds of top level statements:

- interpreter directives
- executable statements

Top-level code is executed when it is encountered. An interpreter directive alters how the interpreter works but does not result in any action in your program. The top-level code of an include space is executed just once for the *first* include statement; duplicate include statements are quietly ignored.

Subspace code of a block is only executed under specific conditions. A subroutine **call** is "required before any subroutine block code may execute." A **condition** must be evaluate to non-zero (true) before a block of code in an branching or repetition statement may be executed.

An **directive** statement is "how you send information to the interpreter itself." An interpreter directive will control how the interpreter (shrouder, binder, or compiler) will behave. For example the include statement determines how external files are read into your program. An interpreter directive is only written at the top-level of file space. A directive statement is executed by the interpreter but does not cause an action in your program.

A **definition** statement "creates a subroutine (procedure, function, or type.)" A definition can only be written at the top-level of file space. When a definition is executed by the interpreter syntax is checked but there is no action. A subroutine **call** is "required before the block code of a subroutine is executed."

An **action** "is an change in state of your program." Examples of actions are: assignment, input, output, altered flow of execution. This contrasts with a directive that does not cause a visible action.

A **declaration** "creates a named object (variable, constant, or enum)." A constant or enum can only be declared at the top-level of file space. A variable declaration can be written at the top-level of file space or in the subspace of a block. Declarations do not execute but block code is checked for syntax.

A **executable statement** "results in action by the interpreter." A top-level statement will be executed by the interpreter. Statements in the subspace of a block will not be immediately executed.

A **call** "permits the block code of a subroutine to execute."

A **test** "permits the block of code in a conditional or repetition statement to be executed when the test evaluates to non-zero or true."

## Block Space

A **block** is "subspace within some enclosing structured statement." A structured statement can be written inside another structured statement with the result that you can have nested blocks of space. Structured statements (subroutine, repetition, conditional) may contain one or more blocks of subspace.

A block may have zero or more lines of code. Code written inside a block is never considered to be top-level code.

For example, a subroutine contains a block that starts after the subroutine signature and stops at the end keyword.

```
function twotimes( atom x ) ★ START BLOCK
```

```
    x = x * 2
    return x BLOCK
```

```
END BLOCK ★ end function
```

A simple if statement has one block:

```
if x = 5 then ★ START BLOCK
```

```
    puts(1, "five" ) BLOCK
```

```
END BLOCK ★ end if
```

A compound if statement can have several blocks:

```
if x = 5 then ★ START BLOCK
```

```
    puts(1, "five" ) BLOCK one
```

```
END BLOCK ★ elseif x = 6 then ★ START BLOCK
```

```
    puts(1, "six" ) BLOCK two
```

```
END BLOCK ★ else ★ START BLOCK
```

```
    puts(1,"none") BLOCK three
```

```
END BLOCK ★ end if
```

A **file scoped name** is "is an name declared or defined at the top-level of a file space." A **block scoped name** is "a name declared inside a block space." Where a declaration is made has a major impact on scope.

Recognizing the extent of a block is important because a block can act like an independent space within your program. Inside a block you can declare a variable and even use an existing name without causing a name-clash. But, you can not define a subroutine inside any block.

```
atom x = 5          -- file variable
                   -- declared in file space

procedure show( )
    -- block
    atom x = 3      -- declared in block space
    ? x             -- block variable
end procedure

? x
--> 5  -- file space 'x'

show()
--> 3  -- block space 'x'
```

Any source-code file may include another file. Most practical programs contain include files which in turn may include files themselves. An **include space** is "the file space of an include file that is injected into your program at the location of the include statement."

## Include Space

The **include statement** has "the keyword include followed by a filename."

```
include <filename>
```

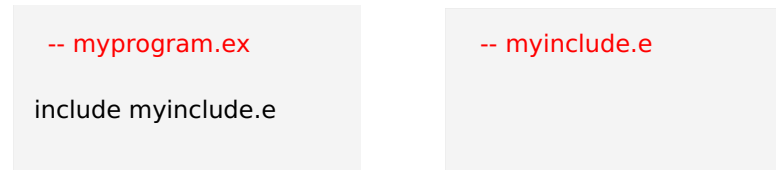
When the interpreter encounters an include statement it stops reading the current file, reads the entire contents of the included file, and then continues with the following statement after include.

You can use existing library files or create your own include files. The file extension is optional and is not important. It is just a convention that ".e" is popular for include files: ".e" (a general include), ".ew" (include for a gui), or ".eu" (include for unix). You are free to use any file extension you wish.

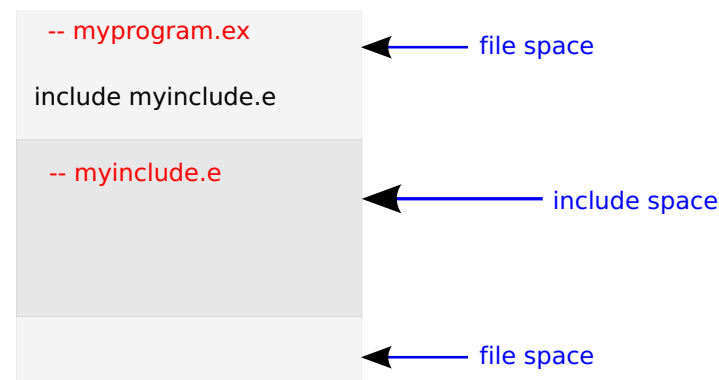
```
include std/console.e    -- include a library file
include myinc.e          -- include your own file
```

*Only at first glance* is including a file like pasting the contents of another file into the current including file:

#### BEFORE



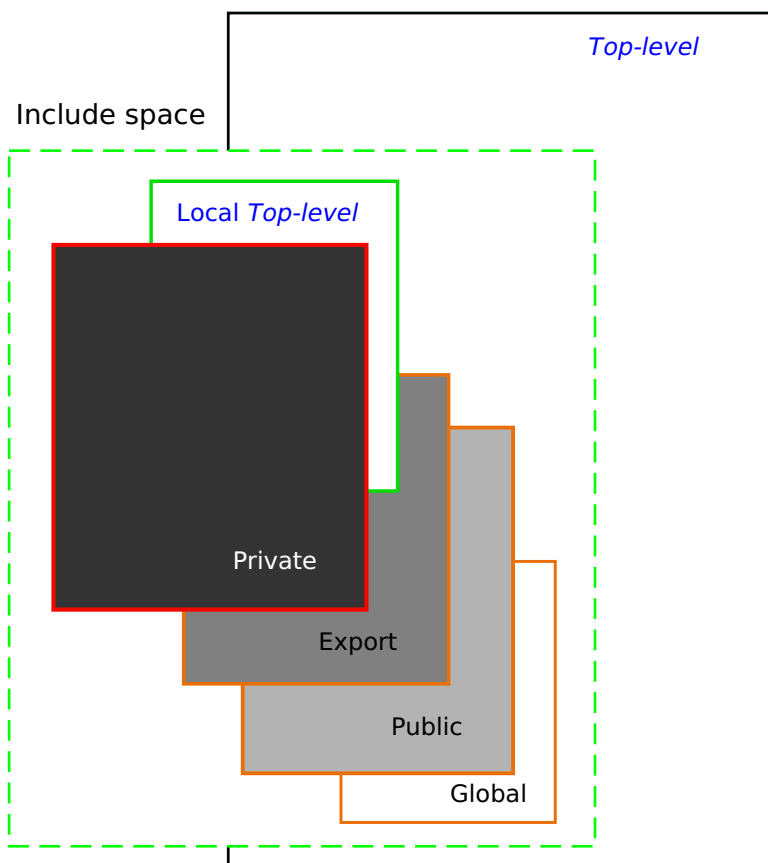
#### AFTER



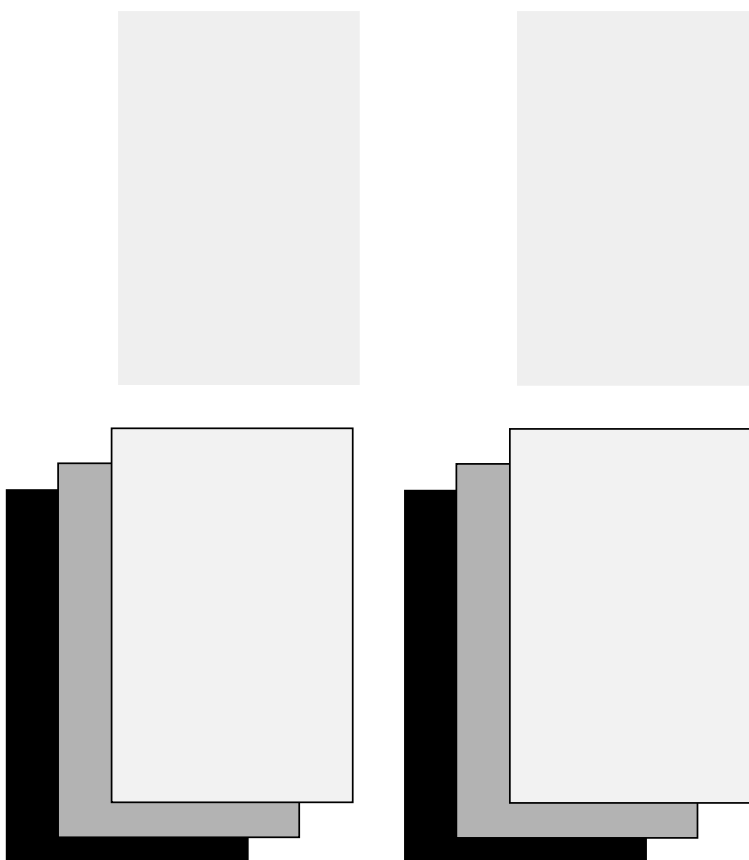
If the including process was a simple pasting of one file into another the result would be a big mess. The basic rules for including are:

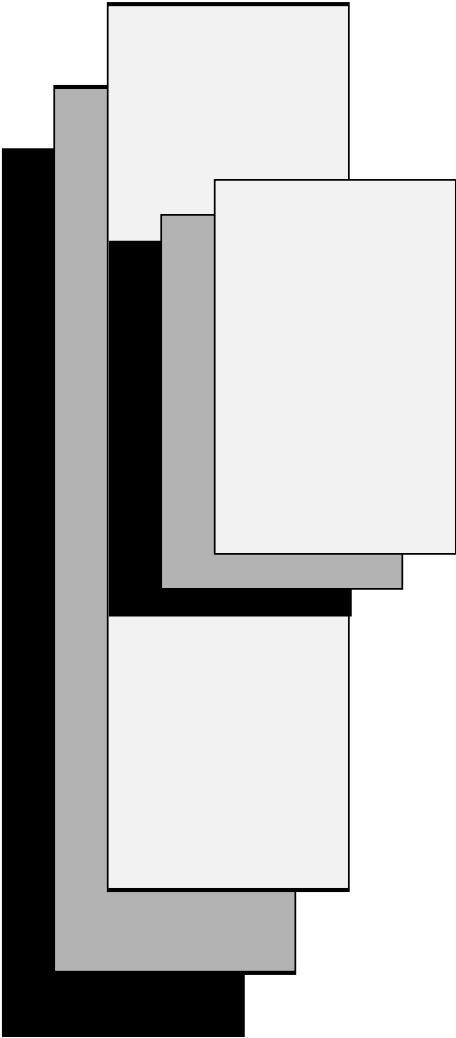
- You can include the same file multiple times; the surplus includes do no harm.
- Each include file is read just once; there are no unneeded file reads.
- Any file can include any other file; this is even sometimes required.
- Top-level statements are executed the *first time* a file is included.
- The default is that no name or identifier from the included file is visible to the including file.
- Special keywords ( export, public, global) used in the included file make a name or an identifier visible to the including file.
- If the include file has names or identifiers that conflict with those in the including file then you can use the namespace prefix to prevent name clashes.

## File space



## Diagrams





# String

Euphoria does not have a dedicated string data-type. However, you can use a sequence in a variety of flexible ways represent any kind of string based text data.

A **string** is a **character sequence** which is a "flat sequence of character values."

A **character** is "a written symbol that is used to represent speech." An **alphabet** is "a set of characters that includes letters used to write a language."

The challenge is how to encode characters so they can be represented with a computer.

The description of the Ubuntu font family states "Coverage 1,200 glyphs, 200-250 languages (native languages of 3 billion people!)."

From wikipedia: "Unicode defines a codespace of 1,114,112 code points in the range 0hex to 10FFFFhex."

A **code point** is a "number assigned to unique character or symbol." There are enough code points for every human language (current, extinct, and artifical) and utility characters like mathematical symbols.

## Seven Bits

The Euphoria language is written with **ASCII** (American Standard Code for Information Interchange) characters using seven bit numbers to define a 128 character alphabet.

**Plain text** is text written in ASCII code.

To understand ASCII you have to imagine a time before computers were invented. Because of its history it is a standard that everyone can agree to. This alphabet corresponds to the keys on a typical keyboard.

Characters from 0 to 31 are used for the control of hardware and many of these codes are now obsolete; they are often called **control characters** which have the property of not printing anything visible.

Characters from 32 to 126 let you write in English with letters, digits, and some punctuation symbols; these are the **printable characters**.

All operating systems can work with plain text. The Unicode standard incorporates all of the ASCII characters with identical code point values.

## Eight Bits

With eight bit numbers you can have an alphabet of 254 characters. There is a variety of conflicting eight bit standards. You can now write in a few more human languages or use graphical characters but not all at once. These alphabets are called **extended ASCII**.

A Windows console will display extended ASCII characters. A Linux terminal will optionally display extended ASCII characters.

The Unicode standard does not recognize these disparate alphabets.

## UTF-8

A UTF is a **unicode transformation format**. UTF-8 is "the Unicode standard that represents a single character as a group (from one to four) of eight-bit numbers."

UTF-8 has the advantage of being memory efficient and the disadvantage that indexing characters is not simple because of variable length encodings. To find an individual character you must search from the head of a string and identify the multi-length encoded characters everytime you need to index characters in a string. Only if you use



ASCII valued characters will a UTF-8 string provide simple indexing. A UTF-8 string can be longer than the number of characters because of variable length encodings.

The UTF-8 standard incorporates ASCII characters seamlessly and is the standard used for the internet.

Linux supports UTF-8 by default in terminal, office, and text editor software. Windows does not support UTF-8 as smoothly.

Euphoria lets you have sequences containing UTF-8 encoded strings that work seamlessly on Linux.

### Example 1:

The value of the A character is 65 which is the same in ASCII, UTF-8, or UTF-32.

```
-- the character A

? 'A'
--> 65

-- a string with one character

? "A"
--> { 65 }
```

## Sixteen Bits

Using sixteen-bit numbers you can represent a much larger alphabet which is sufficient for many users. The 655345 available characters are not enough for all languages; notably Asiatic languages need variable length encodings.

This is the Microsoft standard.

### UTF-16

The Unicode standard is UTF-16.

## Thirty-two Bits

A single thirty-two bit number will represent every code point in the Unicode standard. Using a single number for any character makes indexing a sequence especially simple.

### UTF-32

The Unicode standard is UTF-32

A Euphoria sequence works well with UTF-32 encoded strings. All indexing, slicing, and sequence operations work with these strings.

Since UTF-8 is the common standard you will have to convert to and from UTF-32 when inputting and outputting strings.

### Example 2:

The ☺ smiley face needs three values when encoded in UTF-8. Note that you need a sequence to represent this one character.

```
? "☺"
--> {226,152,186}
```

The code point U+263A is the ☺ smiley character. For example in LibreOffice Insert/Special Character from the main menu displays a table of characters and code points written in U+ style that you can inset into your documents. The integer 9786 is the

decimal equivalent.

Euphoria lets you write a UTF-32 character in a style similar to U+ notation:

Unicode Code Point	Euphoria Notation UTF-32	Decimal Value
U+263A	U"263A"	9786

## UTF String Literals

- using word strings hexadecimal (for utf-16) and double word hexadecimal (for utf-32) e.g.

```
u"65 66 67 AE" -- ==> {#65,#66,#67,#AE}
U"65 66 67 AE" -- ==> {#65,#66,#67,#AE}
```

The value of the strings above are equivalent. Spaces separate values to other elements. When you put too many hex characters together for the kind of string they are split up appropriately for you:

```
x"6566 67AE" -- 8-bit ==> {#65,#66,#67,#AE}
u"6566 67AE" -- 16-bit ==> {#6566,#67AE}
U"6566 67AE" -- 32-bit ==> {#6566,#67AE}
U"6566_67AE" -- 32-bit ==> {#656667AE}
                -- Uses '_' to aid readability for long values.
U"656667AE" -- 32-bit ==> {#656667AE}
```

String literals encoded as ASCII, UTF-8, UTF-16, UTF-32 or really any encoding that uses elements that are 32-bits long or shorter can be built with U"" syntax. Literals of encodings that have 16-bit long or shorter or 8-bit long or shorter elements can be built using u"" syntax or x"" syntax respectively. Use delimiters, such as spaces and underscores, to break the ambiguity and improve readability.

The following is code with a valid UTF8 encoded string:

```
sequence utf8_val = x"3e 65" -- This is ">e"
```

**However**, it is up to the coder to know the correct code-point values for these to make any sense in the encoding the coder is using. That is to say, it is possible for the coder to use the x"", u"", and U"" syntax to create literals that are **not valid** UTF strings.

Hexadecimal strings can be used to encode UTF-8 strings, even though the resulting string does not have to be a valid UTF-8 string.

## Rules for Unicode Strings

1. they begin with the pair u" for UTF-16 and U" for UTF-32 strings, and end with a double-quote (") character
2. they can only contain hexadecimal digits (0-9 A-F a-f), and space, underscore, tab, newline, carriage-return. Anything else is invalid.
3. an underscore is simply ignored, as if it was never there. It is used to aid readability.
4. For UTF-16 strings, each set of four contiguous hex digits represent a single sequence element with a value from 0x0000 to 0xFFFF
5. For UTF-32 strings, each set of eight contiguous hex digits represent a single sequence element with a value from 0x0000 to 0xFFFFFFFF
6. they can span multiple lines
7. The non-hex digits are treated as punctuation and used to delimit individual values.
8. The resulting string does not have to be a valid UTF-16/UTF-32 string.

```
u"1 2 34 5678AbC" == {0x0001, 0x0002, 0x0034, 0x5678, 0x0ABC}
U"1 2 34 5678AbC" == {0x0000_0001, 0x0000_0002, 0x0000_0034, 0x05678ABC}
U"1 2 34 5_678_AbC" == {0x0000_0001, 0x0000_0002, 0x0000_0034, 0x0567_8ABC}
```

## UTF String Literals

- using word strings hexadecimal (for utf-16) and double word hexadecimal (for utf-32) e.g.

```
u"65 66 67 AE" -- ==> {#65,#66,#67,#AE}
U"65 66 67 AE" -- ==> {#65,#66,#67,#AE}
```

The value of the strings above are equivalent. Spaces separate values to other elements. When you put too many hex characters together for the kind of string they are split up appropriately for you:

```
x"6566 67AE" -- 8-bit ==> {#65,#66,#67,#AE}
u"6566 67AE" -- 16-bit ==> {#6566,#67AE}
U"6566 67AE" -- 32-bit ==> {#6566,#67AE}
U"6566_67AE" -- 32-bit ==> {#656667AE}
                -- Uses '_' to aid readability for long values.
U"656667AE"  -- 32-bit ==> {#656667AE}
```

String literals encoded as ASCII, UTF-8, UTF-16, UTF-32 or really any encoding that uses elements that are 32-bits long or shorter can be built with U"" syntax. Literals of encodings that have 16-bit long or shorter or 8-bit long or shorter elements can be built using u"" syntax or x"" syntax respectively. Use delimiters, such as spaces and underscores, to break the ambiguity and improve readability.

The following is code with a valid UTF8 encoded string:

```
sequence utf8_val = x"3e 65" -- This is ">e"
```

**However**, it is up to the coder to know the correct code-point values for these to make any sense in the encoding the coder is using. That is to say, it is possible for the coder to use the x"", u"", and U"" syntax to create literals that are **not valid** UTF strings.

Hexadecimal strings can be used to encode UTF-8 strings, even though the resulting string does not have to be a valid UTF-8 string.

## Rules for Unicode Strings

1. they begin with the pair u" for UTF-16 and U" for UTF-32 strings, and end with a double-quote (") character
2. they can only contain hexadecimal digits (0-9 A-F a-f), and space, underscore, tab, newline, carriage-return. Anything else is invalid.
3. an underscore is simply ignored, as if it was never there. It is used to aid readability.
4. For UTF-16 strings, each set of four contiguous hex digits represent a single sequence element with a value from 0x0000 to 0xFFFF
5. For UTF-32 strings, each set of eight contiguous hex digits represent a single sequence element with a value from 0x0000 to 0xFFFFFFFF
6. they can span multiple lines
7. The non-hex digits are treated as punctuation and used to delimit individual values.
8. The resulting string does not have to be a valid UTF-16/UTF-32 string.

```
u"1 2 34 5678AbC" == {0x0001, 0x0002, 0x0034, 0x5678, 0x0ABC}
U"1 2 34 5678AbC" == {0x0000_0001, 0x0000_0002, 0x0000_0034, 0x05678ABC}
U"1 2 34 5_678_AbC" == {0x0000_0001, 0x0000_0002, 0x0000_0034, 0x0567_8ABC}
```

# PRODUCTS



Software for running programs and making applications is a Euphoria **product**.

Product	Name	Use
eui	<b>Interpreter</b>	Run programs by executing source-code.
eushroud	<b>Shrouder</b>	Compile source-code into intermediate language.
eub	<b>Backend</b>	Run programs by executing intermiate language file.
eubind	<b>Binder</b>	Create application by combining intermiate langue with the backend.
euc	<b>Compiler</b>	Create application by <i>translating</i> source-code into C and then compiling with an external C compiler.
eucoverage		
eudist		
eutest		
eudis		

## UTILITIES

- `bench`
- `bugreport`
- `buildcpdb`
- `edx`
- `euloc`

## Backend

# Euphoria To C Translator

## Introduction

The **Euphoria to C Translator** (translator) will translate any Euphoria program into equivalent C source code.

There are versions of the translator for *Windows* and *Unix* operating Systems. After translating a Euphoria program to C, you can compile and link using one of the supported C compilers. This will give you an executable file that will typically run much faster than if you used the Euphoria interpreter.

The translator can translate and then compile *itself* into an executable file for each platform. The translator is also used in translating/compiling the front-end portion of the interpreter. The source code for the translator is in euphoria\source. It is written 100% in Euphoria.

## Supported Compilers

The **Translator** currently works with GNU C on *Unix* OSes, GNU C on *Windows* from **MinGW** or **Cygwin** using the `-gcc` option and with **Watcom C** (the default) on *Windows*. These are all **free** compilers.

GNU C will exist already on your *Unix* system. The others can be downloaded from their respective Web sites.

### Notes:

- Warnings are turned off when compiling directly or with makefiles. If you turn them on, you may see some harmless messages about variables declared but not used, labels defined but not used, function prototypes not declared etc.
- For the `-gcc` option on *Windows* you will need `aeu.a` compiled with **MinGW** or **Cygwin**. The official distribution may only contain `eu.lib` compiled with **Watcom**. Also, the `-stack` and `-con` options may not produce the expected result with **GCC C**.
- Currently, only 32-bit compilers are supported on 64-bit platforms.

## Run the Translator

Running the **Translator** is similar to running the **Interpreter**:

```
euc -con allsorts.ex
```

Note: that on *Unix* the demos might be installed to `/usr/share/euphoria/demo`

Instead of running the `allsorts.ex` program, the **Translator** will create several C source files in a temporary build directory, compile them and result in a native executable file. For this to work, you have to have a supporting compiler installed (mentioned above). The optional parameter used in this example, `-con`, will be explained in full detail below.

When the C compiling and linking is finished, you will have a file called `allsorts.exe` or simply `allsorts` on *\*nix* systems. The C source files will have been removed to avoid clutter.

When you run the `allsorts` executable, it should run the same as if you had typed:

```
eui allsorts
```

to run it with the **Interpreter**, except that it should run faster, showing reduced times for the various sorting algorithms in `euphoria\demo\allsorts.ex`.

After creating your executable file, the translator removes all the C files that were

created. If you want to look at these files, you'll need to run the translator again, using either the `-keep` or `-makefile` options.

## Command-Line Options

### **-arch - Set architecture**

The translator generally produces cross platform code. However, the euphoria source code may have different code for different architectures. The default is to use the architecture of the translator binary that is being used. To target a different architecture, you can use one of three supported architectures:

- X86
- X86\_64
- ARM

### **-build-dir dir**

Use the specified directory to write translated C files and compiled objects. The final executable is still output by default to the current directory (or however the `-o` flag specifies). When not specified, euphoria will create a temporary, randomly named build directory.

The specified directory cannot contain any wildcards ('\*', '?') or be an existing file.

```
$ euc -build-dir temp_dir myapp.ex
```

### **-cc-prefix - Compiler prefix**

Some compilers, especially MinGW (the Windows version of gcc) may prefix their normal names with platform prefixes. The `-cc-prefix` switch allows the developer to specify this special prefix. This can also be useful for having a system with both the 32bit and 64bit versions installed. Cross compilers generally require this.

For example, on Windows, to build with MinGW installed as `i686-w64-mingw32`:

```
euc -gcc -cc-prefix i686-w64-mingw32- pretend.exw
```

### **-cflags FLAGS - Compiler Flags**

Specifies the flags to pass to the compiler.

### **-com DIR - Compiler directory**

Tells the translator where to find `include/euphoria.h`, which is the header file required when translating code.

### **-con - Console based program**

To make a *Windows* console program instead of a *Windows* GUI program, add `-con` to the command line. e.g.



```
euc -con myprog.exw
```

When creating a Windows GUI program, if the `-con` option is used, when running your Windows program, you will have a blank console window appear and remain the duration of your application. By default, a GUI program is assumed.

### **-debug - Debug mode**

To compile your program with debugging information, usable with a debugger compatible with your compiler, use the `-debug` option:

```
euc -debug myapp.ex
```

### **-dll / -so - Shared Library**

To make a shared dynamically loading library, just add `-dll` to the command line. e.g.

```
euc -dll mylib.ew
```

Note: On \*nix systems, you can also use `-so`. Both will produce a \*nix shared library.

Please see [Dynamic Link Libraries](#)

### **-extra-cflags - Extra Compiler Flags**

Supply extra compiler flags to supplement the flags used automatically by the translator or supplied via the `-cflags` option.

### **-extra-lflags - Extra Linker Flags**

Supply extra linker flags to supplement the flags used automatically by the translator or supplied via the `-lflags` option.

### **-gcc, -wat**

If you happen to have more than one C compiler for a given platform, you can select the one you want to use with a command-line option:

```
-wat  -- Watcom compiler  
-gcc  -- GCC compiler (MinGW on Windows)
```

For example, to compile with GCC (or MinGW on Windows):

```
euc -gcc pretend.exw
```

Note: *Watcom* is the default on *Windows* and `-wat` is assumed.

### **-keep**

Normally, after building your `.exe` file, the translator will delete all C files and object files

produced by the Translator. If you want it to keep these files, add the `-keep` option to the Translator command-line. e.g.

```
euc -keep sanity.ex
```

## **-lflags FLAGS - Linker Flags**

Specifies the flags to pass to the linker.

## **-lib - User defined library**

It is sometimes useful to link your translated code to a Euphoria runtime library other than the default supplied library. This ability is probably mostly useful for testing and debugging the runtime library itself, or to give additional debugging information when debugging translated Euphoria code. Note that only the default library is supplied. Use the `-lib {library}` option:

```
euc -lib decu.a myapp.ex
```

## **-lib-pic - User defined library for PIC mode**

Some platforms and architectures (e.g., x86-64) require that shared libraries be built in Position Independent Code mode, which requires that the euphoria run time library also be built with PIC. This option is similar to the `-lib - User defined library` option, except that it specifies the library to use for PIC code:

```
euc -lib-pic euso.a myapp.ex
```

## **-makefile / -makefile-partial - Using makefiles**

You can optionally have the translator create a makefile that you can use to build your program instead of building directly. Using a makefile like this can be convenient if you want or need to alter the translated C code, or change compiling or linking options before building your program. To do so:

```
$ euc -makefile myapp.ex
Translating code, pass: 1 2 3 4 generating
3.c files were created.
To build your project, type make -f myapp.mak
```

Then, as the message indicates, simply type:

```
$ make -f myapp.mak
```

On Windows, when using Watcom, the message will refer to `wmake`, the Watcom version of `make`. On BSD platforms, you may need to use `gmake`, as the generated makefiles are in GNU format, not BSD.

You can also get a partial makefile using the `-makefile-partial` switch. This generates a makefile that you can use to include into another makefile for a larger project. This is useful for including the file dependencies for your code into the larger project.

## **-maxsize NUMBER**

Specifies the maximum number of C statements to go into a single file before the translated file is split into multiple C files.

## **-plat - Set platform**

The translator has the capability of translating Euphoria code to C code for a platform other than the host platform. This can be done with the `-plat` option. It takes one parameter, the platform code:

- FREEBSD
- LINUX
- OSX
- WINDOWS
- NETBSD
- OPENBSD

Use one of these options to translate code into C for the specified platform. The default will always be the host platform of the translator that is executed, so `euc.exe` will default to *Windows*, and `euc` will default to the platform upon which it was built.

The resulting output can be compiled by the appropriate compiler on the specified platform, or, possibly a cross platform compiler, if you have one configured.

## **-rc-file - Resource File**

On Windows, `euc` can automatically compile and link in an application specific resource file. This resource file can contain product and version information, an application icon or any other valid resource data.

```
euc -rc-file myapp.rc myapp.ex
```

The resulting executable will contain all the resources from `myapp.rc` compiled into the executable. Please see [Using Resource Files](#).

## **-silent**

Do not display status messages.

## **-stack - Stack size**

To increase or decrease the total amount of stack space reserved for your program, add `-stack nnnn` to the command line. e.g.

```
euc -stack 100000 myprog.ex
```

The total stack space (in bytes) that you specify will be divided up among all the tasks that you have running (assuming you have more than one). Each task has its own private stack space. If it exceeds its allotment, you'll get a run-time error message identifying the task and giving the size of its stack space. Most non-recursive tasks can run with call stacks as small as 2000 bytes, but to be safe, you should allow more than this. A deeply-recursive task could use a great deal of space. It all depends on the maximum levels of calls that a task might need. At run-time, as your program creates more simultaneously-active tasks, the stack space allotted to each task will tend to decrease.

## Dynamic Link Libraries

Simply by adding `-dll` (or `-so`) to the command line, the **Translator** will build a shared dynamically loading library instead of an executable program.

You can translate and compile a set of useful Euphoria routines, and share them with other people, without giving them your source. Furthermore, your routines will likely run much faster when translated and compiled. Both translated/compiled and interpreted programs will be able to use your library.

Only the global Euphoria procedures and functions, i.e. those declared with the "global", "public" or "export" keyword, will be exported from the shared dynamically loaded library.

Any Euphoria program, whether translated or compiled or interpreted, can link with a Euphoria shared dynamically loading library using the same mechanism that lets you link with a shared dynamically loading library written in C. The program first calls `open_dll` to open the file, then it calls `define_c_func` or `define_c_proc` for any routines that it wants to call. It calls these routines using `c_func` and `c_proc`.

The routine names exported from a Euphoria shared dynamically loading library will vary depending on which C compiler you use.

GNU C on *Unix* exports the names exactly as they appear in the C code produced by the **Translator**, e.g. a Euphoria routine

```
global procedure foo(integer x, integer y)
```

would be exported as `"_0foo"` or maybe `"_1foo"` etc. The underscore and digit are added to prevent naming conflicts. The digit refers to the Euphoria file where the identifier is defined. The main file is numbered as 0. The include files are numbered in the order they are encountered by the compiler. You should check the C source to be sure.

For Watcom, the **Translator** also creates an `EXPORT` command, added to `objfiles.lnk` for each exported identifier, so `foo` would be exported as `"foo"`.

With Watcom, if you specify the `-makefile` option, you can edit the `objfiles.lnk` file to rename the exported identifiers, or remove ones that you do not want to export. Then build with the generated makefile.

Having nice exported names is not critical, since the name need only appear once in each Euphoria program that uses the shared dynamically loading library, i.e. in a single `define_c_func` or `define_c_proc` statement. The author of a shared dynamically loading library should probably provide his users with a Euphoria include file containing the necessary `define_c_func` and `define_c_proc` statements, and he might even provide a set of Euphoria "wrapper" routines to call the routines in the shared dynamically loading library.

When you call `open_dll`, any top-level Euphoria statements in the shared dynamically loading library will be executed automatically, just like a normal program. This gives the library a chance to initialize its data structures prior to the first call to a library routine. For many libraries no initialization is required.

To pass Euphoria data (atoms and sequences) as arguments, or to receive a Euphoria object as a result, you will need to use the following constants in `euphoria\include\dll.e`:

```
-- Euphoria types for shared dynamically loading library arguments
-- and return values:

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

Use these in `define_c_proc` and `define_c_func` just as you currently use `C_INT`, `C_UINT` etc. to call C shared dynamically loading libraries.

Currently, file numbers returned by `open`, and routine id's returned by `routine_id`, can be passed and returned, but the library and the main program each have their own separate ideas of what these numbers mean. Instead of passing the file number of an open file, you could instead pass the file name and let the shared dynamically loading library open it. Unfortunately there is no simple solution for passing routine id's. This might be fixed in the future.

A Euphoria shared dynamically loading library currently may not execute any multitasking operations. The Translator will give you an error message about this.

Euphoria shared dynamically loading library can also be used by C programs as long as only 31-bit integer values are exchanged. If a 32-bit pointer or integer must be passed, and you have the source to the C program, you could pass the value in two separate 16-bit integer arguments (upper 16 bits and lower 16 bits), and then combine the values in the Euphoria routine into the desired 32-bit atom.

## Using Resource Files

When creating an executable file to deliver to your users on Windows, its best to link in a resource file that at minimum sets your application icon but better if it sets product and version information.

When the resource compiler is launched by `euc`, a single macro is defined named `SRCDIR`. This can be used in your resource files to reference your application source path for including other resource files, icon files, etc...

A simple resource file to attach an icon to your executable file is as simple as:

```
myapp ICON SRCDIR\myapp.ico
```

Remember that `SRCDIR` will be expanded to your application source path.

A more complex resource file containing an icon and product/version information may look like:

```
1 VERSIONINFO

FILEVERSION 4,0,0,9
PRODUCTVERSION 4,0,0,9

FILEFLAGSMASK 0x3fL
FILEFLAGS 0x0L
FILEOS 0x4L
FILETYPE 0x1L
FILESUBTYPE 0x0L

BEGIN
    BLOCK "StringFileInfo"
        BEGIN
            BLOCK "040904B0"
                BEGIN
                    VALUE "Comments", "http://myapplication.com\0"
                    VALUE "CompanyName", "John Doe Computing\0"
                    VALUE "FileDescription", "Cool App\0"
                    VALUE "FileVersion", "4.0.0\0"
                    VALUE "InternalName", "coolapp.exe\0"
                    VALUE "LegalCopyright", "Copyright (c) 2022 by John Doe Computing\0"
                    VALUE "LegalTrademarks1", "Trademark Pending\0"
                    VALUE "LegalTrademarks2", "\0"
                    VALUE "OriginalFilename", "coolapp.exe\0"
                    VALUE "ProductName", "Cool Application\0"
                    VALUE "ProductVersion", "4.0.0\0"
                END
            END
        END
    BLOCK "VarFileInfo"
        BEGIN
            VALUE "Translation", 0x409, 1200
        END
    END
```

```

    END
END

coolapp ICON SRCDIR\coolapp.ico

```

One other item you may wish to include is a manifest file which lets Windows know that controls should use the new theming engines available in  $\geq$  Windows XP. Simply append:

```
1 24 "coolapp.manifest"
```

to the end of your resource file. The coolapp.manifest file is:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity
    version="0.64.1.0"
    processorArchitecture="x86"
    name="euphoria"
    type="win32"
  />
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
        processorArchitecture="X86"
        publicKeyToken="6595b64144ccf1df"
        language="*"
      />
    </dependentAssembly>
  </dependency>
</assembly>

```

Version, Product and Manifest information may change with new releases of Microsoft Windows. You should consult MSDN for up to date information about using resource files with your application. [MSDN About Resource Files](#).

## Executable Size and Compression

The translator does not compress your executable file. If you want to do this we suggest you try the free [UPX](#) compressor.

Large Win32Lib-based .exe's produced by the Translator can be compressed by UPX to about 15% of their original size, and you won't notice any difference in start-up time.

The **Translator** deletes routines that are not used, including those from the standard Euphoria include files. After deleting unused routines, it checks again for more routines that have now become unused, and so on. This can make a big difference, especially with Win32Lib-based programs where a large file is included, but many of the included routines are not used in a given program.

Nevertheless, your compiled executable file will likely be larger than the same Euphoria program bound with the interpreter **back-end**. This is partly due to the **back-end** being a compressed executable. Also, Euphoria statements are extremely compact when stored in a bound file. They need more space after being translated to C, and compiled into machine code. Future versions of the **Translator** will produce faster and smaller executables.

## Interpreter versus Translator

All Euphoria programs can be translated to C, and with just a few exceptions noted below, will run the same as with the **Interpreter** (but hopefully faster).

The **Interpreter** and **Translator** share the same parser, so you will get the same syntax errors, variable not declared errors etc. with either one.

The **Interpreter** automatically expands the call stack (until memory is exhausted), so you can have a huge number of levels of nested calls. Most C compilers, on most systems, have a pre-set limit on the size of the stack. Consult your compiler or linker manual if you want to increase the limit, for example if you have a recursive routine that might need thousands of levels of recursion. Modify the link command in your makefile, or use the `-lflags` option when calling the translator. For Watcom C, use `OPTION STACK=nnnn`, where `nnnn` is the number of bytes of stack space.

### Note:

The **Translator** assumes that your program has no run-time errors in it that would be caught by the **Interpreter**. The **Translator** does not check for: subscript out of bounds, variable not initialized, assigning the wrong type of data to a variable, etc.

You should **debug** your program with the **Interpreter**. The Translator checks for certain run-time errors, but in the interest of speed, most are not checked. When translated C code crashes you'll typically get a very cryptic machine exception. In most cases, the first thing you should do is run your program with the **Interpreter**, using the same inputs, and preferably with `type_check` turned on. If the error only shows up in translated code, you can use `with trace` and `trace(3)` to get a `ctrace.out` file showing a circular buffer of the last 500 Euphoria statements executed. If a translator-detected error message is displayed (and stored in `ex.err`), you will also see the offending line of Euphoria source whenever `with trace` is in effect. `with trace` will slow your program down, and the slowdown can be extreme when `trace(3)` is also in effect.

## Legal Restrictions

As far as RDS is concerned, any executable programs or shared dynamically loading libraries that you create with this **Translator** without modifying an RDS translator library file, may be distributed royalty-free. You are free to incorporate any Euphoria files provided by RDS into your application.

In general, if you wish to use Euphoria code written by 3rd parties, please honor any restrictions that apply. If in doubt, you should ask for permission.

On *Linux*, *FreeBSD*, the GNU Library licence will normally not affect programs created with this **Translator**. Simply compiling with GNU C does not give the Free Software Foundation any jurisdiction over your program. If you statically link their libraries you will be subject to their Library licence, but the standard compile/link procedure does not statically link any FSF libraries, so there should be no problem.

## Disclaimer:

This is what we believe to be the case. We are not lawyers. If it's important to you, you should read **all** licences and the legal comments in them, to form your own judgment. You may need to get professional legal opinion as well.

## Frequently Asked Questions

### How much of a speed-up should I expect?

It all depends on what your program spends its time doing. Programs that use mainly integer calculations, don't call run-time routines very often, and don't do much I/O will see the greatest improvement, currently up to about 5x faster. Other programs may see only a few percent improvement.

The various C compilers are not equal in optimization ability.

### What if I want to change the compile or link options in my generated makefile?

Feel free to do so, that's one reason for producing a makefile.

### How can I make my program run even faster?

It's important to declare variables as integer where possible. In general, it helps if you choose the most restrictive type possible when declaring a variable.

Typical user-defined types will not slow you down. Since your program is supposed to be free of type check errors, types are ignored by the Translator, unless you call them directly with normal function calls. The one exception is when a user-defined type routine has side-effects (i.e. it sets a global variable, performs pokes into memory, I/O etc.). In that case, if with type\_check is in effect, the Translator will issue code to call the type routine and report any type\_check failure that results.

On *Windows* we have left out the /ol loop optimization for Watcom's wcc386. We found in a couple of rare cases that this option led to incorrect machine code being emitted by the Watcom C compiler. If you add it back in to your own makefile you might get a slight improvement in speed, with a slight risk of buggy code.

On *Linux* or *FreeBSD* you could try the -O3 option of gcc instead of -O2. It will "in-line" small routines, improving speed slightly, but creating a larger executable. You could also try the [Intel C++ Compiler for Linux](#). It's compatible with GNU C, but some adjustments to your makefile might be required.

## Common Problems

Many large programs have been successfully translated and compiled using each of the supported C compilers, and the Translator is now quite stable.

### Note:

On *Windows*, if you call a C routine that uses the cdecl calling convention (instead of stdcall), you must specify a '+' character at the start of the routine's name in `define_c_proc` and `define_c_func`. If you don't, the call may not work when running the eui Interpreter.

In some cases a huge Euphoria routine is translated to C, and it proves to be too large for the C compiler to process. If you run into this problem, make your Euphoria routine smaller and simpler. You can also try turning off C optimization in your makefile for just the .c file that fails. Breaking up a single constant declaration of many variables into separate constant declarations of a single variable each, may also help. Euphoria has no limits on the size of a routine, or the size of a file, but most C compilers do. The Translator will automatically produce multiple small .c files from a large Euphoria file to avoid stressing the C compiler. It won't however, break a large routine into smaller routines.





# eudist Distributing Source-Code

## Introduction

The **eudist** "tool makes distributing your program easier." It is designed to gather all of the Euphoria files that your program uses and put them into a directory. This can also be useful for sending example code for bug reports.

## Command Line Options

You can use the standard -i and -c switches with eudist. There are additional options:

- --clear Clear the output directory before copying files
- -d <dir> Specify the output directory for the files
- -e <file> --exclude-file <file> Exclude a file from being copied
- -ed <dir> --exclude-directory <file> Exclude a directory from being copied
- -edr <dir> --exclude-directory-recursively <file> Exclude a directory and all subdirectories from being copied

## euteest Unit Testing

### Introduction

The testing system gives you the ability to check if the library, interpreter and translator works properly by use of *unit tests*. The unit tests are Euphoria *include* files that include `unittest.e` at the top, several test-routines for comparison between expected value and true value and at the end of the program a call to `test_report`. There are error control files for when we expect the interpreter to fail but we want it to fail with a particular error message. You may use this section as an outline for testing your own code.

### euteest Tool

#### Synopsis, Running Tests

```
euteest [-D NO_INET ] [-D NO_INET_TESTS ]  
        [-verbose] [-log] [-i include path] [-cc wat|gcc] [-exe interpreter]  
        [-ec translator] [-lib binary library path]  
        [optional list of unit test files]
```

#### Synopsis, Report From Log

```
euteest -process-log [-html]
```

#### General Behavior

If you want to test translation of your tests as well as interpreted tests, you can specify it with `-ec`.

If you don't specify unit tests on the command line euteest will scan the directory for unit test files using the pattern `t_*.e`. If you specify a pattern it will interpret the pattern, as some shells do not do this for programs.

#### Options Detail

- `-D REC`: Is for creating control files, use only when on tests that work already on an interpreter that correctly works or correctly *\*fails\** with them. This option must come before the `euteest.ex` program itself in the command line and is the option with that requirement.
- `-log`: Is for creating a log file for later processing
- `-verbose`: Is for euteest.ex to give you detail of what it is doing
- `-i`: is for specify the include path which will be passed to both the interpreter and the translator when interpreting and translating the test.
- `-cc`: is for specifying the compiler. This can be any one of `-wat`, `djg`, or `gcc`. Each of these represent the kind of compiler we will request the translator to use.
- `-process-log`: Is for processing a log created by a previous invocation of `euteest.ex` output is sent to standard output as a report of how the tests went. By default this is in ASCII format. Use `-html` to make it HTML format.
- `-html`: Is for making the report creation to be in HTML format
- `-D NO_INET`: This is for keeping tests from trying to use the Internet. The tests have to be written to support them by using `ifdef/end ifdef` statements. Since in some Euphoria unit tests `"-D NO_INET_TESTS"` is used in its place, you must use both options to prevent them from trying to connect through the Internet.
- `-D NO_INET_TESTS`: See `NO_INET`

## The Unit Test Files

Unit test files must match a pattern `t_*.e`. If the unit test file matches `t_c_*.e` the test program will expect the program to fail, if there is an error control file in a directory with its same name and `'d'` extension it will also expect it to fail according to the control file's contents found in the said directory. Such a failure is marked as a successful test run. However, if there is no such directory or file the counter test will be marked as a failed test run.

### Trivial Example

The following is a minimal unit test file:

```
include std/unittest.e

test_report()
```

Please see the [Unit Testing Framework](#), for information on how to construct these files.

## The Error Control Files

There are times when we expect something to fail. We want good EUPHORIA code to do the correct thing and there is a correct thing to do also for *\*bad\** code. The interpreter must return with an error message of why it failed and the error must be correct and it must get written to `ex.err`. We must thus check the `ex.err` file to see if it has the correct error message.

If the unit test is `t_foo.e` then the location for its control file can be in the following locations:

- `t_foo.d/interpreter/OSNAME/control.err`
- `t_foo.d/OSNAME/control.err`
- `t_foo.d/control.err`

The *OSNAME* is the name of the operating system. Which is either UNIX or Win32.

Now, if `t_foo.d/Win32/control.err` exists, then the testing program `euteest.exe` expects `t_foo.e` to fail when run with the *Windows* interpreter. However, this is not necessarily true for other platforms. In *Windows* `euteest` runs it, watches it fail, then compares the `ex.err` file to `t_foo.d/Win32/control.err`. If they `ex.err` is different from `control.err` an error message is written to the log. Now on, say NetBSD, `t_file.e` is tested with the expectation it will return 0 and the tests will all pass unless `t_foo.d/UNIX/control.err` or `t_foo.d/control.err` also exist. Thus you can have different expectations for differing platforms. Some feature that is not possible to implement under *Windows* can be put into a unit test and the resulting `ex.err` file can be put into a control file for *Windows*. This means we do not need to have all of these errors that we expect to get drawing our attention away from errors that need our attention. On the other hand, if an unexpected error message not like `t_foo.d/Win32/control.err` gets generated in the *Windows* case then `euteest` will tell us that.

How do we construct these control files? You don't really need to, you can take an `ex.err` file that results from running a stable interpreter on a test and rename it and move it to the appropriate place.

## Test Coverage

When writing and evaluating the results of unit tests, it is important to understand which parts of your code are and are not being tested. The Euphoria interpreter has a built in capability to instrument your code to analyze how many times each line of your code is executed during your suite of tests. The data is output into an EDS database. Euphoria also comes with a coverage data post-processor that generates html reports to make analysis of your coverage easy.

The coverage capabilities can be used manually, with arguments supplied on the command line, or passed to eutest. Indeed, eutest simply passes these along to the interpreter. The Euphoria suite of unit tests can be run via the makefiles, and there is a special target to run a coverage analysis of the standard library:

```
Windows:  
> wmake coverage  
  
Unix:  
$ make coverage
```

Then, in your build directory, eutest will run the tests to create the coverage database unit-test.edb, and will post-process the results, placing the HTML reports into a unit-test subdirectory from your build directory.

## Coverage Command Line Switches

- `-coverage [file|dir]` This specifies what files to gather stats for. If you supply a directory, it recurses on child directories. Only files that are obviously Euphoria are included ( `.e`, `.ew`, `.eu`, `.ex`, `.exw`, `.exu` ).
- `-coverage-db <file>` This one allows you to specify a specific location and name for the database where coverage information is stored. It is an EDS database. By default, the DB is `eui-cvg.edb`.
- `-coverage-erase` Tells the interpreter to start over. By default, multiple runs accumulate coverage in the DB to allow coverage analysis based on a suite of unit test files.
- `-coverage-exclude <pattern>` Specifies a regular expression that is used to exclude files from coverage analysis.
- `-coverage-pp <post-processor>` Supported by eutest only (i.e., not the interpreter itself). Tells eutest how to post process the coverage data. `<post-processor>` must be the path to a the post processing application. After running the suite of tests, eutest will execute this program with the path to the coverage db as an argument.

## Coverage Post-Processing

Once you have run tests to generate a coverage database, the data is not easily viewed. Euphoria comes with a post-processor called `eucoverage.ex`, which is installed in the bin directory. On a *Unix* packaged install, you should be able to simply use `eucoverage`, which is configured to run `eucoverage.ex`.

The post-processor generates an index page, with coverage stats for each file, and individual html files, linked from the index page, for each file analyzed for test coverage. At the file level, statistics are presented for total and executed routines and lines of code. The files are sorted in descending order of lines that were never executed, in order to highlight the parts of your code that are less tested. The page for each file shows this information, as well as a similar breakdown by routine, displaying the number of lines in each routine that was executed. The routines are also sorted in descending order by the most unexecuted lines.

Additionally, the source of the file is displayed below the statistics. The routines are linked to their place in the code. Each line is colored either green red or white. White lines are those that are not executed. These are typically blank, comments, declarations or "end" clauses of code blocks that do not create any executable code. Red lines are those that were never executed, and lines that were executed are colored green. The line number is displayed in the left margin, and the number of times each line was executed is displayed just to the left of where the source code begins.

## Command Line Switches

- `-o <dir>` Specify the output directory. The default is to create a subdirectory, from the same directory as the coverage database, with the name of the base filename (without extension) of the coverage database.
- `-v` Verbose output

## eudis Source-code Disassembly

# Euphoria Database System (EDS)

## Introduction

While you can connect Euphoria to most databases (such as: MySQL, SQLite, PostgreSQL), sometimes you do not need that kind of power.

The **Euphoria Database System (EDS)** is "a simple, easy-to-use, flexible, Euphoria-oriented database for storing data." EDS works better for many cases where you need more than a text file and do not quite need or want the power and complexity of larger database packages.

## EDS API

More details on using EDS, including complete coverage of the EDS API, can be found at [Euphoria Database \(EDS\)](#).

## EDS Database Structure

An **EDS database** is "a single file with a .edb file extension." Each **database** "contains zero or more tables." Each **table** has "a name, and contains zero or more records." Each **record** consists "of a key part, and a data part." A **key** can be "any Euphoria object--an atom, a sequence, a deeply-nested sequence, whatever." Similarly **data** can be "any Euphoria object." There are *no* constraints on the size or structure of the key or data. Within a given table, the keys are all unique. That is, no two records in the same table can have the same key part.

The records of a table are stored in ascending order of key value. An efficient binary search is used when you refer to a record by key. You can also access a record directly, with no search, if you know its current record number within the table. **Record numbers** are "integers from one to the length (current number of records) of the table." By incrementing the record number, you can efficiently step through all the records, in order of key. Note however that a record's number can change whenever a new record is inserted, or an existing record is deleted.

The keys and data parts are stored in a compact form, but *no* accuracy is lost when saving or restoring floating-point numbers or *any* other Euphoria data.

The EDS database std/eds.e will work as is, on all platforms. EDS database files can be copied and shared between programs running on all platforms as well. When sharing EDS database files, be sure to make an exact byte-for-byte copy using "binary" mode copying, rather than "text" or "ASCII" mode, which could change the line terminators.

Example:

```
database: "mydata.edb"
  first table: "passwords"
    record #1: key: "jones"    data: "euphor123"
    record #2: key: "smith"   data: "billgates"

    second table: "parts"
      record #1: key: 134525   data: {"hammer", 15.95, 500}
      record #2: key: 134526   data: {"saw", 25.95, 100}
      record #3: key: 134530   data: {"screw driver", 5.50, 1500}
```

It is up to you to interpret the meaning of the key and data. *In keeping with the spirit of Euphoria, you have total flexibility.* Unlike most other database systems, an EDS record is *not* required to have either a fixed number of fields, or fields with a preset maximum length.

In many cases there will not be any natural key value for your records. In those cases you should simply create a meaningless, but unique, integer to be the key. Remember that you can always access the data by record number. It is easy to loop through the records looking for a particular field value.



## Accessing Data

To reduce the number of arguments that you have to pass, there is a notion of the current database, and current table.

### Current Database

The **current database** is "the database that is active in the present time." Any data operation or table operation assumes there is a current database being defined. You set the current database by opening, creating or selecting a database. Deleting the current database leaves the current database undefined.

### Current Table

The **current table** is "the table that is active in the present time." All data operations assume there is a current table being defined. You must create, select or rename a table in order to make it current. Deleting the current table leaves the current table undefined.

## Accessing Data

Most routines use these *current* values automatically. You normally start by opening (or creating) a database file, then selecting the table that you want to work with.

You can map a key to a record number using `db_find_key`. It uses an efficient binary search. Most of the other record-level routines expect the record number as a argument. You can very quickly access any record, given it's number. You can access all the records by starting at record number one and looping through to the record number returned by `db_table_size`.

## Storage Recycling

When you delete something, such as a record, the space for that item gets put on a free list, for future use. Adjacent free areas are combined into larger free areas. When more space is needed, and no suitable space is found on the free list, the file will grow in size. Currently there is no automatic way that a file will shrink in size, but you can use a `db_compress` to completely rewrite a database, removing the unused spaces.

## Security and Multi-user Access

This release provides a simple way to lock an entire database to prevent unsafe access by other processes.

## Scalability

Internal pointers are 4 bytes. In theory that limits the size of a database file to 4 Gb. In practice, the limit is 2 Gb because of limitations in various C file functions used by Euphoria. Given enough user demand, EDS databases could be expanded well beyond 2 Gb in the future.

The current algorithm allocates four bytes of memory per record in the current table. So you will need at least 4 Mb RAM per million records on disk.

The binary search for keys should work reasonably well for large tables.

Inserts and deletes take slightly longer as a table gets larger.

At the low end of the scale, it is possible to create extremely small databases without incurring much disk space overhead.

## Disclaimer

Do not store valuable data without a backup. RDS will not be responsible for any damage or data loss.

## Warning: Use Binary File Mode

.edb files are binary files, not text files. You **must** use BINARY mode when transferring a .edb file via FTP from one machine to another. You must also avoid loading a .edb file into an editor and saving it. If you open a .edb file directly using Euphoria's `open`, which is not recommended, you must use binary mode (not text mode). Failure to follow these rules could result in 10 (line-feed) and 13 (carriage-return) bytes being changed, leading to subtle and not-so-subtle forms of corruption in your database.

## bench.ex

Utility for running benchmark tests.

## bugreport.ex

Utility to help file a bugreport.

**buildcpdb.ex**

## edx.ex Demo Editor

Utility for editing source-code; a demonstration of Euphoria programming.



# USING \_\_\_\_\_



- `cfg`
- `advance`
- `indirect`
- `preprocess`
- `tasking`
- `debug`
- `GUI`
- `gui_windows`
- `gui_unix`
- `gui_multi`
- `ARCHIVE`
- `rds`
- `rdsindex`



# cfg Product Configuration File

# Shrouding and Binding

## Interpreter Design

The Euphoria interpreter has two "ends." The interpreter **frontend** "reads your source-code, parses main and included files, and produces an Intermediate Language (IL) version of your program." The Euphoria **Intermediate Language (IL)** is "source-code parsed and processed into a format that is machine readable by the interpreter." The interpreter **backend** "reads Intermediate Language (IL) and performs the actual interpretation."

When you run eui both "ends" combine seamlessly and your program simply executes. Euphoria also lets you run each "end" independently.

- run frontend only: eushroud
- run backend only: eub

The eushroud utility runs just the frontend, parses files, excludes unused code, produces an IL, and saves an `.il` file. The result is a single file, unreadable by humans, containing only code actually used by your program.

Run the filename.il file:

- `./filename` (Unix platform)
- `eub filename` ( any platform )

An **application** is "a single file executable produced from Euphoria source-code."

If you do not have a C compiler installed, then you can use eubind make an application from your source-code by binding the IL of your program with the backend. The result is an application which can be executed from your operating system.

```
eubind filename
```

If you have a C compiler installed, then you can use euc which translates Euphoria source-code into C language code and then immediately compiles your program. The result is an application which can be executed from your operating system.

```
euc filename
```

## Sample File Sizes

- mt.ex, blank file with zero statements
- hello.ex, classic one line hello world
- edx.ex, the demonstration code editor

File sizes reported as 1000 \* bytes ( KB ).

Category	mt.ex	hello.ex	edx.ex
source-code	0.0	0.02	60
shrouded intermediate	0.44	0.46	250
bound application	1900	1900	2100
compiled application	186	186	490

A bound application has a minimum size (that of the backend) that increases very slowly as a program increases in size.

## eushroud Utility

## eushroud

```
eushroud [-full_debug] [-list] [-quiet] [-out shrouded_file] filename.ex[w|u]
```

The eushroud.ex utility converts a Euphoria program, typically consisting of a main file plus many include files, into a single compact file. A single file is easier to distribute, and it allows you to distribute your program to others without releasing your source code.

A shrouded file does not contain any Euphoria source code statements. Rather, it contains a low-level Intermediate Language (IL) that is executed by the back-end of the interpreter. A shrouded file does not require any parsing. It starts running immediately, and with large programs you will see a quicker start-up time. Shrouded files must be run using the eub interpreter backend:

eubw.exe (*Windows*) or eub.exe (*Unix*).

Only one copy of eub is needed to run any number of bound programs. It is stored in ../euphoria/bin in the Euphoria interpreter package. You can run your **myprog.il** file with:

On *Windows* use:

```
eub myprog.il
eubw myprog.il
```

On *Unix* use:

```
eub myprog.il
```

Although it does not contain any source statements, an .il file will generate a useful ex.err dump in case of a run-time error.

The .il file produced by eushroud excludes all unused routines and variables. Only the required portions of included files and standard library files remain. The result is a compact single file version of your original source-code.

The .il file is unreadable by humans. The .il file can not be used to recover the original source-code used to produce it.

## eushroud Options

Command	Argument	Description
-full_debug		Make a somewhat larger .il file that contains enough debug information to provide a full ex.err dump when a crash occurs. Normally, variable names and line-number information is stripped out of the .il file, so the ex.err will simply have "no-name" where each variable name should be, and line numbers will only be accurate to the start of a routine or the start of a file. Only the private variable values are shown, not the global or local values. In addition to saving space, some people might prefer that the shrouded file, and any ex.err file, not expose as much information.
-list		Produce a listing in deleted.txt of the routines and constants that were deleted.
-quiet		Suppress normal messages and statistics. Only report errors.
-out	shrouded_file	Write the output to shrouded_file.

The Euphoria interpreter will not perform tracing on a shrouded file. You must trace your original source.

On *Unix* `eushroud` will make the file executable, and adds a **shebang** ( a `#!` statement as the top line) instructing `eub` to run when the file is opened. You can override this default `#!` line by specifying your own `#!` line at the top of your main Euphoria source-code file. ( A shebang line on *Windows* is simply ignored.)

Always keep a copy of your original source. There is no way to recover it from a shrouded file.

## eubind Utility

### eubind

```
eubind [-c config-file] [-con] [-copyright] [-eub path-to-backend]
        [-full_debug] [-i dir] [-icon file] [-list] [-quiet]
        [-out executable_file] [-shroud_only filename.ex]
```

Both `eubind -shroud_only filename` and `eushroud filename` produce identical IL.

It then combines your shrouded `.il` file with the interpreter backend (`eub.exe`, `eubw.exe` or `eub`) to make a *single file application* that you can conveniently use and distribute. Your users need not have Euphoria installed. Each time your executable file is run, a quick integrity check is performed to detect any tampering or corruption. Your program will start up very quickly since no parsing is needed.

The Euphoria interpreter will not perform tracing on a bound file since the source statements are not there.

### eubind Options

Command	Argument	Description
-c	config-file	A Euphoria config file to use when binding.
-con		<b>Windows only:</b> This option will create a <i>Windows</i> console program instead of a <i>Windows</i> GUI program. Console programs can access standard input and output, and they work within the current console window, rather than popping up a new one.
-eub	path-to-backend	Allows specification of the backend runner to use instead of the default, installed version.
-full_debug		Same as <code>eushroud</code> above. If Euphoria detects an error, your executable will generate either a partial, or a full, <code>ex.err</code> dump, according to this option.
-i	dir	A directory to add to the paths to use for searching for included files.
-icon	filename[.ico]	<b>(Windows only):</b> When you bind a program, you can patch in your own customized icon, overwriting the one in <code>euiw.exe</code> . In <code>../euphoria/source</code> there are two icons: <code>euphoria.ico</code> is the <i>Mongoose</i> Logo, and <code>eufile.ico</code> is the <i>Claws</i> Logo. <code>eui.exe</code> contains a 32x32 icon using 256 colors. <i>Windows</i> will display this shape beside <code>euiw.exe</code> , and beside your bound program, in file listings. You can also load this icon as a resource, using the name "euiw" (see <code>...\euphoria/demo/win32/window.exw</code> for an example). When you bind your program, you can substitute your own 32x32 256-color icon file of size 2238 bytes or less. Other dimensions may also work as long as the file is 2238 bytes or less. The file must contain a single icon image ( <i>Windows</i> will create a smaller or larger image as necessary).

-list		Produce a listing in deleted.txt of the routines and constants that were deleted.
-quiet		Suppress normal messages and statistics. Only report errors.
-out	executable_file	This option lets you choose the name of the executable file created by the binder. Without this option, eubind will choose a name based on the name of the main Euphoria source file.

A one-line Euphoria program will result in an executable file as large as the back-end you are binding with, but the size increases very slowly as you add to your program.

The first two items returned by `command_line` will be slightly different when your program is bound. See the function description for the details.

A **bound executable** file *can* handle standard input and output redirection as with this syntax:

```
myprog.exe < file.in > file.out
```

If you were to write a small .bat file, say myprog.bat, that contained the line "eui myprog.ex" you would *not* be able to redirect input and output. The following will not work:

```
myprog.bat < file.in > file.out
```

You *could* however use redirection on individual lines *within* the .bat file.

# Advanced Language Features

## Scope

## Indirect Routine Calling

Euphoria does not have function pointers. However, it enables you to call any routine, including some internal to the interpreter, in an indirect way, using two different sets of identifiers.

### Indirect Calling a Routine Coded in Euphoria

The following applies to any routine coded in Euphoria that your program uses, whether it is defined in the standard library, any third party library or your own code. It does not apply to routines implemented in the backend.

#### Getting a Routine Identifier

Whenever a routine is in scope, you can supply its name to the builtin `routine_id` function, which returns a small integer:

```
include get.e
constant value_id = routine_id("value")
```

Because `value` is defined as public, that routine is in scope. This ensures the call succeeds. A failed call returns -1, else a small nonnegative integer.

You can then feed this integer to `call_func` or `call_proc` as appropriate. It does not matter whether the routine is still in scope at the time you make that call. Once the id is gotten, it's valid.

#### Calling Euphoria Routines by Id

This is very similar to using `c_func` or `c_proc` to interface with external code.

#### Function Calling

This is done as follows:

```
result = call_func(id_of_the_routine, argument_sequence)
```

where

- `id_of_the_routine` is an id you obtained from `routine_id`.
- `argument_sequence` is the list of the parameters to pass, enclosed into curly braces

```
include get.e

constant value_id = routine_id("value")
result = call_func(value_id, {"Model 36A", 6, GET_LONG_ANSWER})
-- result is {GET_SUCCESS, 36, 4, 1}
```

This is equivalent to

```
result = value("Model 36A", 6, GET_LONG_ANSWER)
```

#### Procedure Calling

The same formalism applies, but using `call_proc` instead. The differences are almost the same as between `c_func` and `c_proc`.

```
include std/pretty.e
```

```
constant pretty_id = routine_id("pretty_print")

call_proc(pretty_id,{1, some_object, some_options})
```

This does the same as a straightforward

```
include std/pretty.e

pretty_print(1, some_object, some_options)
```

The difference with `c_proc` is that you can call an external function using `c_proc` and thus ignore its return value, like in C. Note that you cannot use `call_proc` to invoke a Euphoria function, only C functions.

## Why Call Indirectly?

Calling functions and procedures indirectly can seem more complicated and slower than just calling the routine directly, but indirect calls can be used when the name of the routine you want to call might not be known until run-time.

```
integer foo_id

function bar(integer x)
    return call_func(foo_id,{x})
end function

function foo_dev1(integer y)
    return y + 1
end function

function foo_dev2(integer y)
    return y - 1
end function

function foo_dev3(integer y)
    return y * y - 3
end function

function user_opt(object x)
    ...
end function

-- Initialize foo ID
switch user_opt("dev") do
case 1 then
    foo_id = routine_id("foo_dev1")
case 2 then
    foo_id = routine_id("foo_dev2")
case else
    foo_id = routine_id("foo_dev3")
end switch
```

One last word: when calling a routine indirectly, its **full** parameter list must be passed, even if some of its parameters are defaulted. This limitation may be overcome in future versions.

## Calling Euphoria Internals

A number of Euphoria routines are defined in different ways depending on the platform they will run on. It would be cumbersome, and at times downright impossible, to put such code in include files or to make the routine fully builtin.

A solution to this is provided by `machine_func` and `machine_proc`. User code normally never needs to use these. Various examples are to be found in the standard library.



These primitives are called like this:

```
machine_proc(id, argument)
result = machine_func(id, argument)
```

argument is either an atom, or a sequence standing for one or more parameters. Since the first parameter does not need to be a constant, you may use some sort of dynamic calling. The circumstances where it is useful are rare.

The complete list of known values for id is to be found in the file `source/execute.h`.

Defining new identifiers and overriding `machine_func` or `machine_proc` to handle them is an easy way to extend the capabilities of the interpreter.

# Preprocessor

A **pre-processor** is "a program that translates code written in an alternative syntax into standard Euphoria syntax."

The OpenEuphoria pre-processor was developed by Jeremy Cowgar.

A pre-processor lets you turn Euphoria into a custom programming language that is exclusive to your exact needs.

## Pre-Processor

A pre-processor is a program that produces a program written in Euphoria syntax. The pre-processor itself can be written in any language and use any kind of input. How the pre-processor works is limited only by your imagination. Naturally, our examples show pre-processors written in Euphoria. What the pre-processor must do is produce a program listing, using valid Euphoria syntax, that the interpreter can execute.

### Pre-processor requirements:

The pre-processor is a command line program with two arguments:

- `-i <filename>`
- `-o <filename>`

## Datestamp

The user defined **pre-processor**, developed by Jeremy Cowgar, opens a world of possibilities to the Euphoria programmer. In a sentence, it allows one to create (or use) a translation process that occurs transparently when a program is run. This mini-guide is going to explore the pre-processor interface by first giving a quick example, then explaining it in detail and finally by writing a few useful pre-processors that can be put immediately to work.

Any program can be used as a pre-processor. It must, however, adhere to a simple specification:

1. Accept a parameter "-i filename" which specifies which file to read and process.
2. Accept a parameter "-o filename" which specifies which file to write the result to.
3. Exit with a zero error code on success or a non-zero error code on failure.

It does not matter what type of program it is. It can be a Euphoria script, an executable written in the C programming language, a script/batch file or anything else that can read one file and write to another file. As Euphoria programmers, however, we are going to focus on writing pre-processors in the Euphoria programming language. As a benefit, we will describe later on how you can easily convert your pre-processor to a shared library that Euphoria can make use of directly thus improving performance.

## Quick Example

The problem in this case is that you want the copyright statement and the about screen to show what date the program was compiled on but you do not want to manually maintain this date. So, we are going to create a simple pre-processor that will read a source file, replace all instances of @DATE@ with the current date and then write the output back out.

Before we get started, let me say that we will expand on this example later on. Up front, we are going to do almost no error checking for the purpose of showing off the pre-processor not for the sake of making a production quality application.

We are going to name this file datesub.ex.

```
-- datesub.ex
include std/datetime.e -- now() and format()
include std/io.e       -- read_file() and write_file()
include std/search.e   -- match_replace()

sequence cmds = command_line()
sequence inFileName, outFileName

for i = 3 to length(cmds) do
    switch cmds[i] do
        case "-i" then
            inFileName = cmds[i+1]
        case "-o" then
            outFileName = cmds[i+1]
        end switch
    end for

sequence content = read_file(inFileName)

content = match_replace("@DATE@", content, format(now()))

write_file(outFileName, content)

-- programs automatically exit with ZERO error code, if you want
-- non-zero, you exit with abort(1), for example.
```

So, that is our pre-processor. Now, how do we make use of it? First let's create a simple test program that we can watch it work with. Name this file thedate.ex.

```
-- thedate.ex

puts(1, "The date this was run is @DATE@\n")
```

Rather simple, but it shows off the pre-processor we have created. Now, let's run it, but first without a pre-processor hook defined.

NOTE: Through this document I am going to assume that you are working in *Windows*. If not, you can make the appropriate changes to the shell type examples.

```
C:\MyProjects\datesub> eui thedate.ex
The date this was run is @DATE@
```

Not very helpful? Ok, let's tell Euphoria how to use the pre-processor that we just created and then see what happens.

```
C:\MyProjects\datesub> eui -p eui:datesub.ex thedate.ex
The date this was run is 2009-08-05 19:36:22
```

If you got something similar to the above output, good job, it worked! If not, go back up and check your code for syntax errors or differences from the examples above.

What is this -p paramater? In short, -p tells eui or euc that there is a pre-processor. The definition of the pre-processor comes next and can be broken into 2 required sections and 1 optional section. Each section is divided by a colon (:). For example, -p e,ex:datesub.ex

1. e,ex tells Euphoria that when it comes across a file with the extension e or ex that it should run a pre-processor
2. datesub.ex tells Euphoria which pre-processor should be run. This can be a .ex file or any other executable command.
3. An optional section exists to pass options to the pre-processor but we will go into this later.

That's it for the quick introduction. I hope that the wheels are turning in your head already as to what can be accomplished with such a system. If you are interested, please continue reading and see where things will get very interesting!

## Pre-process Details

Euphoria manages when the pre-processor should be called and with what arguments. The pre-processor does not need to concern itself as to if it should run, what filename it is reading or what filename it will be writing to. It should simply do as Euphoria tells it to do. This is because Euphoria monitors what the modification time is on the source file and what time the last pre-process call was made on the file. If nothing has changed in the source file then the pre-processor is not called again. Pre-processing does have a slight penalty in speed as the file is processed twice. For example, the `datesub.ex` pre-processor read the entire file, searched for `@DATE@`, wrote the file and then Euphoria picked up from there reading the output file, parsing it and finally executing it. To minimize the time taken, Euphoria caches the output of the pre-processor so that the interim process is not normally needed after it has been run once.

## Command Line Options

### -p - Define a pre-processor

The primary command line option that you will use is the `-p` option which defines the pre-processor. It is a two or three section option. The first section is a comma delimited list of file extensions to associate with the pre-processor, the second is the actual pre-processor script/command and the optional third is parameters to send to the pre-processor in addition to the `-i` and `-o` parameters.

Let's go over some examples:

- `-p e:datesub.ex` - This will be executed for every `.e` file and the command to call is `datesub.ex`.
- `-p "de,dex,dew:dot4.dll:-verbose -no-dbc"` - Files with `de`, `dex`, `dew` extensions will be passed to the `dot4.dll` process. `dot4.dll` will get the optional parameters `-verbose -no-dbc` passed to it.

Multiple pre-processors can be defined at the same time. For instance,

```
C:\MyProjects\datesub> eui -p e,ex:datesub.ex -p de,dex:dot4.dll \
    -p le,lex:iterate.ex hello.ex
```

is a valid command line. It's possible that `hello.ex` may include a file named `greeter.le` and that file may include a file named `person.de`. Thus, all three pre-processors will be called upon even though the main file is only processed by `datesub.ex`

### -pf - Force pre-processing

When writing a pre-processor you may run into the problem that your source file did not change, therefore, Euphoria is not calling your pre-processor. However, your pre-processor has changed and you want Euphoria to re-process your unchanged source file. This is where `-pf` comes into play. `-pf` causes Euphoria to force the pre-processing, regardless of the cached state of any file. When used, Euphoria will always call the pre-processor for all files with a matching pre-processor definition.

## Use of a configuration file

Ok, so who wants to type these pre-processor definitions in all the time? I don't either. That's where the standard Euphoria configuration file comes into play. You can simply create a file named `eu.cfg` and place something like this into it.

```
-p le,lex:iterate.ex
-p e,ex:datesub.ex
... etc ...
```

Then you can execute any of those files directly without the `-p` parameters on the command line. This `eu.cfg` file can be local to a project, local to a user or global on a system. Please read about the [eu.cfg](#) file for more information.

## DLL/Shared Library Interface

A pre-processor may be a Euphoria file, ending with an extension of .ex, a compiled Euphoria program, .exe or even a compiled Euphoria DLL file, .dll. The only requirements are that it must accept the two command line options, -i and -o described above and exit with a ZERO status code on success or non-ZERO on failure.

The DLL file (or shared library on *Unix*) has a real benefit in that with each file that needs to be pre-processed does not require a new process to be spawned as with an executable or a Euphoria script. Once you have the pre-processor written and functioning, it's easy to convert your script to use the more advanced, better performing shared library. Let's do that now with our datesub.ex pre-processor. Take a moment to review the code above for the datesub.ex program before continuing. This will allow you to more easily see the changes that we make here.

```
-- datesub.ex
include std/datetime.e -- now() and format()
include std/io.e       -- read_file() and write_file()
include std/search.e   -- match_replace()

public function preprocess(sequence inFileName, sequence outFileName,
                           sequence options={})

    sequence content = read_file(inFileName)

    content = match_replace("@DATE@", content, format(now()))

    write_file(outFileName, content)

    return 0
end function

ifndef not EUC_DLL then
    sequence cmds = command_line()
    sequence inFileName, outFileName

    for i = 3 to length(cmds) do
        switch cmds[i] do
            case "-i" then
                inFileName = cmds[i+1]
            case "-o" then
                outFileName = cmds[i+1]
            end switch
        end for

        preprocess(inFileName, outFileName)
    end ifdef
```

It's beginning to look a little more like a well structured program. You'll notice that we took the actual pre-processing functionality out the the top level program making it into an exported function named preprocess. That function takes three parameters:

1. inFileName - filename to read from
2. outFileName - filename to write to
3. options - options that the user may wish to pass on verbatim to the pre-processor

It should return 0 on no error and non-zero on an error. This is to keep a standard with the way error levels from executables function. In that convention, it's suggested that 0 be OK and 1, 2, 3, etc... indicate different types of error conditions. Although the function could return a negative number, the main routine cannot exit with a negative number.

To use this new process, we simply translate it through euc,

```
C:\MyProjects\datesub> euc -dll datesub.ex
```

If all went correctly, you now have a datesub.dll file. I'm sure you can guess on how it

should be used, but for the sake of being complete,

```
C:\MyProjects\datesub> eui -p e,ex:datesub.dll thedate.ex
```

On such a simple file and such a simple pre-processor, you probably are not going to notice a speed difference but as things grow and as the pre-processor gets more complicated, compiling to a shared library is your best option.

## Advanced Examples

### Finish datesub.ex

Before we move totally away from our datesub.ex example, let's finish it off by adding some finishing touches and making use of optional parameters. Again, please go back and look at the Shared Library version of datesub.ex before continuing so that you can see how we have changed things.

```
-- datesub.ex
include std/cmdline.e -- command line parsing
include std/datetime.e -- now() and format()
include std/io.e      -- read_file() and write_file()
include std/map.e     -- map accessor functions (get())
include std/search.e  -- match_replace()

sequence cmdopts = {
  { "f", 0, "Date format", { NO_CASE, HAS_PARAMETER, "format" } }
}

public function preprocess(sequence inFileName, sequence outFileName,
  sequence options={})
  map opts = cmd_parse(cmdopts, options)
  sequence content = read_file(inFileName)

  content = match_replace("@DATE@", content, format(now(), map:get(opts,
    "f")))

  write_file(outFileName, content)

  return 0
end function

ifdef not EUC_DLL then
  cmdopts = {
    { "i", 0, "Input filename", { NO_CASE, MANDATORY, HAS_PARAMETER,
      "filename" } },
    { "o", 0, "Output filename", { NO_CASE, MANDATORY, HAS_PARAMETER,
      "filename" } }
  } & cmdopts

  map opts = cmd_parse(cmdopts)
  preprocess(map:get(opts, "i"), map:get(opts, "o"),
    "-f " & map:get(opts, "f", "%Y-%m-%d"))
end ifdef
```

Here we simply used cmdline.e to handle the command line parsing for us giving out command line program a nice interface, such as parameter validation and an automatic help screen. At the same time we also added a parameter for the date format to use. This is optional and if not supplied, %Y-%m-%d is used.

The final version of datesub.ex and thedate.ex are located in the demo/preproc directory of your Euphoria installation.

## Others

TODO: this needs done still.

Euphoria includes two more demos of pre-processors. They are ETML and literate. Please explore demo/preproc for these examples and explanations.

## Other examples of pre-processors include

- eSQL - Allows you to include a .sql file directly. It parses CREATE TABLE and CREATE INDEX statements building common routines to create, destroy, get by id, find by any index, add, remove and save entities.
- make40 - Will process any 3.x script on the fly making sure that it will run in 4.x. It does this by converting variables, constants and routine names that are the same as new 4.x keywords into something acceptable to 4.x. Thus, 3.x programs can run in the 4.x interpreter and translator with out any user intervention.
- dot4 - Adds all sorts of syntax goodies to Euphoria such as structured sequence access, one line if statements, DOT notation for any function/routine call, design by contract and more.

## Other Ideas

- Include a *Windows* .RC file that defines a dialog layout and generate code that will create the dialog and interact with it.
- Object Oriented system for Euphoria that translates into pure Euphoria code, thus has the raw speed of Euphoria.
- Include a Yacc, Lex, ANTLR parser definition directly that then generates a Euphoria parser for the given syntax.
- Instead of writing interpreters such as a QBasic clone, simply write a pre-processor that converts QBasic code into Euphoria code, thus you can run `eui -p bas:qbasic.ex hello.bas` directly.
- Include a XML specification, which in turn, gives you nice accessory functions for working with XML files matching that schema.

If you have ideas of helpful pre-processors, please put the idea out on the forum for discussion.

# Multitasking

## Introduction

Euphoria allows you to set up multiple, independent tasks. Each task has its own current statement that it is executing, its own call stack, and its own set of private variables. Tasks run in parallel with each other. That is, before any given task completes its work, other tasks can be given a chance to execute. Euphoria's task scheduler decides which task should be active at any given time.

## Why Multitask?

Most programs do not need to use multitasking and would not benefit from it. However it is very useful in some cases:

- Action games where numerous characters, projectiles etc. need to be displayed in a realistic way, as if they are all independent of one another. Language War is a good example.
- Situations where your program must sometimes wait for input from a human or other computer. While one task in your program is waiting, another separate task could be doing some computation, disk search, etc.
- All operating systems today have special API routines that let you initiate some I/O, and then proceed without waiting for it to finish. A task could check periodically to see if the I/O is finished, while another task is performing some useful computation, or is perhaps starting another I/O operation.
- Situations where your program might be called upon to serve many users simultaneously. With multiple tasks, it's easy to keep track of the state of your interaction with all these separate users.
- Perhaps you can divide your program into two logical processes, and have a task for each. One produces data and stores it, while the other reads the data and processes it. Maybe the first process is time-critical, since it interacts with the user, while the second process can be executed during lulls in the action, where the user is thinking or doing something that doesn't require quick response.

## Types of Tasks

Euphoria supports two types of tasks: real-time tasks, and time-share tasks.

**Real-time tasks** are scheduled at intervals, specified by a number of seconds or fractions of a second. You might schedule one real-time task to be activated every 3 seconds, while another is activated every 0.1 seconds. In Language War, when the Euphoria ship moves at warp 4, or a torpedo flies across the screen, it's important that they move at a steady, timed pace.

**Time-share tasks** need a share of the CPU but they needn't be rigidly scheduled according to any clock.

It's possible to reschedule a task at any time, changing its timing or its slice of the CPU. You can even convert a task from one type to the other dynamically.

## A Small Example

This example shows the main task (which all Euphoria programs start off with) creating two additional real-time tasks. We call them real-time because they are scheduled to get control every few seconds.

You should try copy/pasting and running this example. You'll see that task 1 gets control every 2.5 to 3 seconds, while task 2 gets control every 5 to 5.1 seconds. In between, the main task (task 0), has control as it checks for a 'q' character to abort execution.

```
constant TRUE = 1, FALSE = 0

type boolean(integer x)
```



```

    return x = 0 or x = 1
end type

boolean t1_running, t2_running

procedure task1(sequence message)
  for i = 1 to 10 do
    printf(1, "task1 (%d) %s\n", {i, message})
    task_yield()
  end for
  t1_running = FALSE
end procedure

procedure task2(sequence message)
  for i = 1 to 10 do
    printf(1, "task2 (%d) %s\n", {i, message})
    task_yield()
  end for
  t2_running = FALSE
end procedure

puts(1, "main task: start\n")

atom t1, t2

t1 = task_create(routine_id("task1"), {"Hello"})
t2 = task_create(routine_id("task2"), {"Goodbye"})

task_schedule(t1, {2.5, 3})
task_schedule(t2, {5, 5.1})

t1_running = TRUE
t2_running = TRUE

while t1_running or t2_running do
  if get_key() = 'q' then
    exit
  end if
  task_yield()
end while

puts(1, "main task: stop\n")
-- program ends when main task is finished

```

## Comparison with earlier multitasking schemes

In earlier releases of Euphoria, Language War already had a mechanism for multitasking, and some people submitted to User Contributions their own multitasking schemes. These were all implemented using plain Euphoria code, whereas this new multitasking feature is built into the interpreter. Under the old Language War tasking scheme a scheduler would *call* a task, which would eventually have to *return* to the scheduler, so it could then dispatch the next task.

In the new system, a task can call the built-in procedure `task_yield` at any point, perhaps many levels deep in subroutine calls, and the scheduler, which is now part of the interpreter, will be able to transfer control to any other task. When control comes back to the original task, it will resume execution at the statement after `task_yield`, with its call stack and all private variables intact. Each task has its own call stack, program counter (i.e. current statement being executed), and private variables. You might have several tasks all executing a routine at the same time, and each task will have its own set of private variable values for that routine. Global and local variables are shared between tasks.

It's fairly easy to take any piece of code and run it as a task. Just insert a few `task_yield` statements so it will not hog the CPU.

## Comparison with multithreading

When people talk about threads, they are usually referring to a mechanism provided by the operating system. That's why we prefer to use the term "multitasking". Threads are generally "preemptive", whereas Euphoria multitasking is "cooperative". With preemptive threads, the operating system can force a switch from one thread to another at virtually any time. With cooperative multitasking, each task decides when to give up the CPU and let another task get control. If a task were "greedy" it could keep the CPU for itself for long intervals. However since a program is written by one person or group that wants the program to behave well, it would be silly for them to favor one task like that. They will try to balance things in a way that works well for the user. An operating system might be running many threads, and many programs, that were written by different people, and it would be useful to enforce a reasonable degree of sharing on these programs. Preemption makes sense across the whole operating system. It makes far less sense within one program.

Furthermore, threading is notorious for causing subtle bugs. Nasty things can happen when a task loses control at just the wrong moment. It may have been updating a global variable when it loses control and leaves that variable in an inconsistent state. Something as trivial as incrementing a variable can go awry if a thread-switch happens at the wrong moment. e.g. consider two threads. One has:

```
x = x + 1
```

and the other also has:

```
x = x + 1
```

At the machine level, the first task loads the value of `x` into a register, then loses control to the second task which increments `x` and stores the result back into `x` in memory. Eventually control goes back to the first task which also increments `x` \*using the value of `x` in the register\*, and then stores it into `x` in memory. So `x` has only been incremented once instead of twice as was intended. To avoid this problem, each thread would need something like:

```
lock x
x = x + 1
unlock x
```

where `lock` and `unlock` would be special primitives that are safe for threading. It's often the case that programmers forget to lock data, but their program seems to run ok. Then one day, many months after they've written the code, the program crashes mysteriously.

Cooperative multitasking is much safer, and requires far fewer expensive locking operations. Tasks relinquish control at safe points once they have completed a logical operation.

## Summary

For a complete function reference, refer to the Library Documentation [Multitasking](#).

# Debugging and Profiling

## Debugging

Extensive run-time checking provided by the Euphoria interpreter catches many bugs that in other languages might take hours of your time to track down. When the interpreter catches an error you will always get a brief report on your screen and a detailed report in a file called `ex.err`. These reports include a full English description of what happened along with a call-stack traceback. The file `ex.err` will also have a dump of all variable values, and optionally a list of the most recently executed statements. For extremely large sequences only a partial dump is shown. If the name `ex.err` is not convenient or if a nondefault path is required you can choose another file name, anywhere on your system, by calling `crash_file`.

In addition, you are able to create `user-defined types` that precisely determine the set of legal values for each of your variables. An error report will occur the moment that one of your variables is assigned an illegal value.

Sometimes a program will misbehave without failing any run-time checks. In any programming language it may be a good idea to simply study the source code and rethink the algorithm that you have coded. It may also be useful to insert print statements at strategic locations in order to monitor the internal logic of the program. This approach is particularly convenient in an interpreted language like Euphoria since you can simply edit the source and rerun the program without waiting for a re-compile/re-link.

## Trace Directives: with, without

The interpreter provides you with additional powerful tools for debugging. Using `trace(1)` you can **trace** the execution of your program on one screen while you witness the output of your program on another. `trace(2)` is the same as `trace(1)` but the trace screen will be in monochrome. Finally, using `trace(3)`, you can log all executed statements to a file called `ctrace.out`.

The **with/without trace** special statements select the parts of your program that are available for tracing. Often you will simply insert a `with trace` statement at the very beginning of your source code to make it all traceable. Sometimes it is better to place the first `with trace` after all of your `user-defined types`, so you don't trace into these routines after each assignment to a variable. At other times, you may know exactly which routine or routines you are interested in tracing, and you will want to select only these ones. Of course, once you are in the trace window, you can skip viewing the execution of any routine by pressing down-arrow on the keyboard rather than Enter. However, once inside a routine, you must step through till it returns, even if stepping in was an mistake.

Only traceable lines can appear in `ctrace.out` or in `ex.err` as "Traced lines leading up to the failure", should a run-time error occur. If you want this information and didn't get it, you should insert a `with trace` and then rerun your program. Execution will be slower when lines compiled with trace are executed, especially when `trace(3)` is used.

After you have predetermined the lines that are traceable, your program must then dynamically cause the trace facility to be activated by executing a `trace` statement. You could simply say:

```
with trace
trace(1)
```

However, you cannot dynamically set or free breakpoints while tracing. You must abort program, edit, change setting, save, and run again.

At the top of your program, so you can start tracing from the beginning of execution.

More commonly, you will want to trigger tracing when a certain routine is entered, or when some condition arises.

```
if x < 0 then
  trace(1)
end if
```

You can turn off tracing by executing a `trace(0)` statement. You can also turn it off interactively by typing 'q' to quit tracing. Remember that with `trace` must appear *outside* of any routine, whereas `trace` can appear *inside* a routine *or outside*.

You might want to turn on tracing from within a `type`. Suppose you run your program and it fails, with the `ex.err` file showing that one of your variables has been set to a strange, although not illegal value, and you wonder how it could have happened. Simply create a type for that variable that executes `trace(1)` if the value being assigned to the variable is the strange one that you are interested in.

```
type positive_int(integer x)
  if x = 99 then
    trace(1) -- how can this be???
    return 1 -- keep going
  else
    return x > 0
  end if
end type
```

When `positive_int` returns, you will see the exact statement that caused your variable to be set to the strange value, and you will be able to check the values of other variables. You will also be able to check the output screen to see what has happened up to this precise moment. If you define `positive_int` so it returns zero for the strange value (99) instead of one, you can force a diagnostic dump into `ex.err`.

Remember that the argument to `trace` does not need to be a constant. It only needs to be 0, 1, 2 or 3, but these values may be the result from any expression passed to `trace`. Other values will cause `trace` to fail.

## Trace Screen

When a `trace(1)` or `trace(2)` statement is executed by the interpreter, your main output screen is saved and a **trace screen** appears. It shows a view of your program with the statement that will be executed next highlighted, and several statements before and after showing as well. You cannot scroll the window further up or down though. Several lines at the bottom of the screen are reserved for displaying variable names and values. The top line shows the commands that you can enter at this point:

Command	Action	Comments
F1	display output screen	display main output screen take a look at your program's output so far
F2	redisplay trace screen	redisplay trace screen Press this key while viewing the main output screen to return to the trace display.
Enter	execute current	execute current the currently-highlighted statement only
down-arrow	continue execution	continue execution and break when any statement coming after this one in the source listing is about to be executed. This lets you skip over subroutine calls. It also lets you stop on the first statement following the end of a loop without having to witness all iterations of the loop.

?	display variable	display the value of a variable. After hitting ? you will be prompted for the name of the variable. Many variables are displayed for you automatically as they are assigned a value. If a variable is not currently being displayed, or is only partially displayed, you can ask for it. Large sequences are limited to one line on the trace screen, but when you ask for the value of a variable that contains a large sequence, the screen will clear, and you can scroll through a pretty-printed display of the sequence. You will then be returned to the trace screen, where only one line of the variable is displayed. Variables that are not defined at this point in the program cannot be shown. Variables that have not yet been initialized will have "< NO VALUE >" beside their name. Only variables, not general expressions, can be displayed. As you step through execution of the program, the system will update any values showing on the screen. Occasionally it will remove variables that are no longer in scope, or that haven't been updated in a long time compared with newer, recently-updated variables.
q	quit	quit tracing and resume normal execution. Tracing will start again when the next trace(1) is executed.
Q	quit	quit tracing and let the program run freely to its normal completion. trace statements will be ignored.
!	abort	this will abort execution of your program. A traceback and dump of variable values will go to ex.err.

As you trace your program, variable names and values appear automatically in the bottom portion of the screen. Whenever a variable is assigned to, you will see its name and new value appear at the bottom. This value is always kept up-to-date. Private variables are automatically cleared from the screen when their routine returns. When the variable display area is full, least-recently referenced variables will be discarded to make room for new variables. The value of a long sequence will be cut off after 80 characters.

For your convenience, numbers that are in the range of printable ASCII characters (32-127) are displayed along with the ASCII character itself. The ASCII character will be in a different color (or in quotes in a mono display). This is done for all variables, since Euphoria does not know in general whether you are thinking of a number as an ASCII character or not. You will also see ASCII characters (in quotes) in ex.err. This can make for a rather "busy" display, but the ASCII information is often very useful.

The trace screen adopts the same graphics mode as the main output screen. This makes flipping between them quicker and easier.

When a traced program requests keyboard input, the main output screen will appear, to let you type your input as you normally would. This works fine for a `gets` (read one line) input. When a `get_key` (quickly sample the keyboard) is called you will be given 8 seconds to type a character, otherwise it is assumed that there is no input for this call to `get_key`. This allows you to test the case of input and also the case of no input for `get_key`.

## Trace File

When your program calls `trace(3)`, tracing to a file is activated. The file, `ctrace.out` will be created in the current directory. It contains the last 500 Euphoria statements that your program executed. It is set up as a circular buffer that holds a maximum of 500 statements. Whenever the end of `ctrace.out` is reached, the next statement is written back at the beginning. The very last statement executed is always followed by "=== THE END ===". Because it's circular, the last statement executed could appear anywhere in `ctrace.out`. The statement coming after "=== THE END ===" is the 500th-last.

This form of tracing is supported by both the interpreter and the the Euphoria to C translator. It is particularly useful when a machine-level error occurs that prevents Euphoria from writing out an `ex.err` diagnostic file. By looking at the last statement executed, you may be able to guess why the program crashed. Perhaps the last statement was a poke into an illegal area of memory. Perhaps it was a call to a C routine.

In some cases it might be a bug in the interpreter or the translator.

The source code for a statement is written to `ctrace.out`, and flushed, just *before* the statement is performed, so the crash will likely have happened *during* execution of the final statement that you see in `ctrace.out`.

## Count Profile

If you specify a `with profile` or `with profile_time` (*Windows only*) directive, then a special listing of your program, called a **profile**, will be produced by the interpreter when your program finishes execution. This listing is written to the file **ex.pro** in the current directory.

There are two types of profiling available: execution-count profiling, and time profiling. You get **execution-count** profiling when you specify `with profile`. You get **time profiling** when you specify `with profile_time`. You can not mix the two types of profiling in a single run of your program. You need to make two separate runs.

We ran the `sieve8k.ex` benchmark program in `demo\bench` under both types of profiling. The results are in `sieve8k.pro` (execution-count profiling) and `sieve8k.pro2` (time profiling).

Execution-count profiling shows precisely how many times each statement in your program was executed. If the statement was never executed the count field will be blank.

Time profiling shows an estimate of the total time spent executing each statement. This estimate is expressed as a percentage of the time spent profiling your program. If a statement was never sampled, the percentage field will be blank. If you see 0.00 it means the statement was sampled, but not enough to get a score of 0.01.

Only statements compiled with `profile` or `with profile_time` are shown in the listing. Normally you will specify either `with profile` or `with profile_time` at the top of your main `.ex*` file, so you can get a complete listing.

Profiling can help you in many ways:

- It lets you see which statements are heavily executed, as a clue to speeding up your program
- It lets you verify that your program is actually working the way you intended
- It can provide you with statistics about the input data
- It lets you see which sections of code were never tested -- do not let your users be the first!

Sometimes you will want to focus on a particular action performed by your program. For example, in the **Language War** game, we found that the game in general was fast enough, but when a planet exploded, shooting 2500 pixels off in all directions, the game slowed down. We wanted to speed up the explosion routine. We did not care about the rest of the code. The solution was to call `profile(0)` at the beginning of *Language War*, just after `with profile_time`, to turn off profiling, and then to call `profile(1)` at the beginning of the explosion routine and `profile(0)` at the end of the routine. In this way we could run the game, creating numerous explosions, and logging a lot of samples, just for the explosion effect. If samples were charged against other lower-level routines, we knew that those samples occurred during an explosion. If we had simply profiled the whole program, the picture would not have been clear, as the lower-level routines would also have been used for moving ships, drawing phasors etc. `profile` can help in the same way when you do execution-count profiling.

## Time Profile

With each click of the system clock, an interrupt is generated. When you specify `with profile_time` *Euphoria* will sample your program to see which statement is being executed at the exact moment that each interrupt occurs.

Each sample requires four bytes of memory and buffer space is normally reserved for 25000 samples. If you need more than 25000 samples you can request it:

```
with profile_time 100000
```

will reserve space for 100000 samples (for example). If the buffer overflows you'll see a warning at the top of **ex.pro**. At 100 samples per second your program can run for 250 seconds before using up the default 25000 samples. It's not feasible for Euphoria to dynamically enlarge the sample buffer during the handling of an interrupt. That is why you might have to specify it in your program. After completing each top-level executable statement, Euphoria will process the samples accumulated so far, and free up the buffer for more samples. In this way the profile can be based on more samples than you have actually reserved space for.

The percentages shown in the left margin of **ex.pro**, are calculated by dividing the number of times that a particular statement was sampled, by the total number of samples taken. e.g. if a statement were sampled 50 times out of a total of 500 samples, then a value of 10.0 (10 per cent) would appear in the margin beside that statement. When profiling is disabled with `profile(0)`, interrupts are ignored, no samples are taken and the total number of samples does not increase.

By taking more samples you can get more accurate results. However, one situation to watch out for is the case where a program synchronizes itself to the clock interrupt, by waiting for `time` to advance. The statements executed just after the point where the clock advances might *never* be sampled, which could give you a very distorted picture.

```
while time() < LIMIT do
end while
x += 1 -- This statement will never be sampled
```

Sometimes you will see a significant percentage beside a return statement. This is usually due to time spent deallocating storage for temporary and private variables used within the routine. Significant storage deallocation time can also occur when you assign a new value to a large sequence.

If disk swapping starts to happen, you may see large times attributed to statements that need to access the swap file, such as statements that access elements of a large swapped-out sequence.



# MINIGUIDES



- [Regular Expressions](#)
- [OOP Style Programming](#)
- [Database Access](#)
- [Windows GUI](#)
- [Unix GUI](#)
- [Multi-platform GUI](#)
- [RDS Website](#)
- [Archive](#)



# Regular Expressions Tutorial

A regular expression is a way of describing, in a cleverly coded way, a fragment of text that you may search for in a larger string of text. It is all about pattern matching. Regular expressions is a coding system, and it's akin to learning a new language.

## Simple Match: Euphoria

To locate the position of a particular character in a string you would normally write the following:

```
? match("pho", "Euphoria")
--          |
--          pho    ---the match you want

-- 3    ---output from match()

---the matched pattern starts at index 3
```

Where "pho" is the pattern, "Euphoria" is the string you are searching, and 3 is the result of your search. Searches of this kind are best done with match().

## Simple Match: Regular Expressions

The previous example with match() looks like this expressed using regular expressions:

```
include std/regex.e as re
regex r = re:new( "pho" )
? re:find( r, "Euphoria" )
--          |
--          pho    ---the match you want

-- {
-- {3, 5}
-- }    ---output from re:new()

---matched pattern starts at index 3 and ends at index 5

re:free( r )
```

The fragment "pho" is a text pattern that is a literal match for the three characters found in the search string "Euphoria". A blank space (invisible as it is) is also a valid character.

## Demo program

Experimenting with regular expression is easier if you have a demonstration program. Try the following code as a starting point:

```
include std/console.e
include std/graphics.e

clear_screen()

puts(1, "Test out regular expressions.\n\n" &
      "Enter string values without \" delimiters\n" &
      "(do not use #/ / notation)\n\n" )

sequence haystack = prompt_string( "enter the haystack... " )
sequence needle   = prompt_string( "enter the needle..... " )

include regex.e as re
```

```

regex r = re:new( needle )
object result = re:find_all( r, haystack )

position(15,1) puts(1, haystack )
text_color( YELLOW ) bk_color( BLUE )

sequence slice
for i=1 to length( result ) do
    slice = result[i][1]
    for k=slice[1] to slice[2] do
        position(15, k )
        puts(1, haystack[ k ] )
    end for
end for

text_color( BLACK ) bk_color( BRIGHT_WHITE )
end for

position( 20, 1 ) puts(1, "results: " ) print(1, result )
puts(1, "\n\npatterns matched: " ) ? length( result )

puts(1, "\n...done...\n" )

```

## Simple Find: Euphoria

The Euphoria `find()` will search for a "needle" object as an element of a Euphoria "haystack" object.

```

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

```

Think of `find()` as an extension that goes beyond finding a slice out of a string.

## Using regex.e

- To prevent identifier name clashes we write:
  - include `std/regex.e` as `re`
  - The choice **re** is common because it suggests "regular expression." The library routines then become `re:new()` instead of just `new()`.
- Creating a regex involves a pair of routines: `new()` and `free()`.
  - By analogy think of the `open()` and `close()` pair you use with files.
  - We write a regex as a text string, but must "compile" it into an encoded value--hence the variable "r" and the "regex-type." The `new()` function creates the compiled form we need.
  - The compiled regex is then used in a variety of searching routines, such as the `find()` function.
  - When done, you `free()` the memory used by the compiled regex. You may now create another compiled regex.

## Metacharacters

Regular expressions are about exotic text patterns that you may use for searching. You must learn the rules of regular expressions before I can show you what can be done. I will often use **regex** as shorthand for "regular expression."

Some characters have special meaning when used in regular expressions. They are analogous to keywords in Euphoria, and are called **metacharacters**.

```
\ . ^ $ * + ? @ # ( ) [ - ] { , }
```

Metacharacters are used to define and control how a regular expression is created and used. They are words of the regex language.

When you see one of them, you must reason out what special purpose they are serving.

If you need to use a metacharacter in its literal sense, that character must be "escaped" first. The backslash( \ )is the escape character. When placed before a metacharacter, the metacharacter loses its special meaning.

For example \* has special meaning, but \\* is just a star.

If you escape the escape symbol, \ , its special meaning is lost and it is just a backslash.

The escape \ is also used to create special meaning. For example the letter c on its own is just a letter. But combined with an escape the c loses its normal meaning and gains special meaning. The \c is now a command for "lower case" searching.

Another caution, the meaning of a metacharacter may depend on the location where it is used. For example the ^ changes meaning if it's inside or outside square brackets: **^[ ]** is very different from **[^ ]**.

## Writing style

A regex is coded using metacharacters and regular characters.

```
[a-z]+i[a-z]*
```

The coded regex is then written as a Euphoria string. A simple regex just needs( " )delimiters and is used as an argument to the re:new() function:

```
regex sample = re:new( "[a-z]+i[a-z]*" )
```

Here is an alternative way of writing the same thing using #/ and / as delimiters:

```
regex sample = re:new( #[a-z]+[a-z]*/ )
```

A regex can become messy very quickly if there are backslashes. Consider this regex:

```
\*
```

To search for a literal \* you need to escape it. A \ is *also* an escape inside a Euphoria string--the Euphoria escape and regex escape are in competition. The consequence is you must add an *extra* \ to write the regex as a string:

```
regex sample = re:new( "\\*" )
```

The alternative writing style lets you write a regex without doubling up on the escape characters:

```
regex sample = re:new( #/\*/ )
```

You will see the style using( / )forward slash delimiters used to describe regex patterns in outside references such as books and the web:

```
/[a-z]+i[a-z]*/
```

## Euphoria wildcard matching

The Euphoria wildcard match lets you create fancier searches.

The plain Euphoria( ? )represents any one character, while the plain Euphoria( \* )represents several characters.

For example:

```
include std/wildcard
integer i
i = wildcard_match("A?B*", "AQBXXYY")
? i
    -- i is 1 (TRUE)

i = wildcard_match("*xyz*", "AAAbbbxyz")
? i
    -- i is 1 (TRUE)

i = wildcard_match("A*B*C", "a111b222c")
? i
    -- i is 0 (FALSE) because upper/lower case doesn't match
```

You can not search for literal ? or \* using match() or wildcard\_match().

If your searches are simple, then use match() and wildcard\_match().

## Regex "wildcard" matching

The advantage of a regex is that you can do a lot more than is possible with a simple search.

Warning: when using a regex, the meaning of ? and \* is not the same as in plain Euphoria.

The vocabulary for a "wildcard" regex is:

metacharacter	meaning
.	any character
.*	any character 0 or more times
.+	any character 1 or more times
.?	any character 0 or 1 times

Using a regex for the previous examples would look like:

```
include std/regex.e as re
regex r = re:new( "A.B" )
? re:find( r, "AQBXXYY" )
    --      |      {
    --      AQB      {1, 3}
    --      }
```

```
include std/regex.e as re
regex r = re:new( ".*xyz.*" )
? result = re:find( r, "AAAbbbxyz" )
    --      |      {
    --      AAAabbxyz      {1,9}
    --      }
```

```
include std/regex.e as re
regex r = re:new( "A.*B.*C" )
? re:find( r, "a111b222c" )
    --      |
    --      -1
```

The ( . ) is working like a wildcard. The ( \* + ? ) are **quantifiers** acting on the ( . ) letting you be very specific about what will be matched.

Using quantifiers does have its gotchas, since what is matched may not be what you expect at first glance.

## The Regex Routines

A regex search is like finding a "needle" in a "haystack." Or, a slice out of a string.

Sometimes it is just enough to know that a **"needle" exists** within the "haystack."

- The function `is_match()` returns false ( 0 ) or true ( 1 ) if your regex "needle" matches the **entire** "haystack" string.
- The function `has_match()` returns false ( 0 ) or true ( 1 ) if your regex "needle" matches a **portion** of the "haystack" string.

At other times you want to find the **"needle" and all details** about the search. The "needle" is now described by a slice that is part of the search string. A "needle" slice is a sequence pair with start and end indices: { start, end } .

- The function `find()` returns false ( 0 ) if a search fails, or the **first slice** that matches the regular expression.
- The function `find_all()` returns false ( 0 ) if a search fails, or **all slices** that match the regular expression.

The output of the "find" functions may contain even more information. Complex regular expressions may be composed of simpler regular expressions. The output of `find()` or `find_all()` will also include the simple slices that were matched as part of matching the large complex regex.

### First match

The function `has_match()` will tell you if a match exists.

The function `find()` is designed to display the first, from the left, valid match it can locate.

```
include std/regex.e as re
regex r = re:new( "dog" )

? re:has_match( r, "cats and dogs and cats and dogs and dogs" )
-- 1

? re:find( r, "cats and dogs and cats and dogs and dogs" )
--           |                               {
--           dog                               {10,12}
--                                           }
```

### All matches

The `find_all()` function will display all possible matches.

```
include std/regex.e as re
regex r = re:new( "dog" )
? re:find_all( r, "cats and dogs and cats and dogs and dogs" )
--           |           |           |           {
--           dog           dog         dog         {10,12},
--                                           {28,30},
--                                           {37,39}
--                                           }
```

## ^ \$ search anchors

You may use **anchor** metacharacters to specify that your match must be in a specific location within the search string.

The metacharacter( **^** )is an anchor that requires the match to *start* at the beginning of the string.

The metacharacter( **\$** )is an anchor that requires the match to *finish* at the end of the string.

```
include std/regex.e as re
regex r

r = re:new( "cat" ) ---no anchors
? re:find( r, "dog chases cat chasing another cat" )
--          |          {
--          cat          {12,14}
--          }

r = re:new( "^cat" ) ---anchor start of string
? re:find( r, "dog chases cat chasing another cat" )
--          |
--          -1

r = re:new( "^dog" ) ---anchor start of string
? re:find( r, "dog chases cat chasing another cat" )
--          |          {
--          dog          { 1, 3 }
--          }

r = re:new( "cat$" ) ---anchor end of string
? re:find( r, "dog chases cat chasing another cat" )
--          |          {
--          cat          {32,34}
--          }

r = re:new( "cat$" ) --- anchor end of string
? find( r, "dog chases cat chasing another cat or maybe a mouse" )
--          |
--          -1
```

You may use **^** and **\$** together to force a regex to match an entire string.

```
include std/regex.e as re
regex r = re:new( "^.+$" )
? re:find( r, "Match this entire string, please." )
--          |          {
--          |          {1,33}
--          }
```

## \b word anchor

The metacharacter( **\b** )is used to anchor a search using the boundary of a word:

```
include std/regex.e as re

regex r = re:new( "cat" ) --- can be anywhere

? find_all( r, "dogchasescatcatchinganothercatormouse" )
--          |          |          {
--          |          |          {
--          cat          |          {10,12}
--          |          |          },
--          |          |
```

```
--
--                               |      {
--                               cat    {27,29}
--                               }
--                               }
```

```
r = re:new( "\\bcat" ) --- must start a word

? re:find_all( r, "dogchases catchasinganothercatormouse" )
--          |
--          -1

? re:find( r, "dogchases catchasinganothercatormouse" )
--          |      {
--          cat    {11,13}
--          }
```

```
r = re:new( "cat\\b" ) --- must end a word

? re:find_all( r, "dogchases catchasinganothercatormouse" )
--          |
--          0

? re:find_all( r, "dogchases catchasinganothercat ormouse" )
--          |      {
--          cat    {28,30}
--          }
```

```
r = re:new( "\\bcat\\b" ) --- must be a complete word

? re:find_all( r, "dogchases catchasinganothercatormouse" )
--          |
--          {}

? re:find_all( r, "dogchases cat chasinganothercatormouse" )
--          | |      {
--          | |      {
--          cat    {11,13}
--          }
--          }

r = re:new( #/\bcat\b/ )
? re:find_all( r, "dogchases cat chasinganothercatormouse" )
-- {          | |
-- {
-- {11,13}
-- }
-- }
--- same as above but with alternative notation
```

## \B word anchor

The metacharacter( **\B** )requires the match to be *inside* a word boundary:

```
include std/regex.e
r = re:new( #/\Bcat/ )
? re:find( r, "dogchases catchasinganothercatormouse cat" )
--          |      {
--          cat    {28,30}
--          }
```

## [] range

- user defined

- predefined range

Square brackets( `[ ]` )are used to describe a **class** of characters.

Between the brackets you list the characters that belong to the class. They may be listed individually: `[abc]` . They may also be listed as part of a **range** of characters: `[a-c]` . In this case the range starts at 'a', continues on as indicated by the metacharacter( `-` )dash separator, and ends at 'c'.

The meaning of `[a-c]` is that **any one character** in the range may result in a proper match.

```
include regex.e as re
regex r = re:new( "[aeiou]" )           -- find a vowel
? re:find_all( r, "The simple, powerful language" )

--      | | | | | | | | | | | | | | | | {
--      e | | | | | | | | | | | | | | { {3,3} },
--      i | | | | | | | | | | | | | | { {6,6} },
--      e | | | | | | | | | | | | | | { {10,10} },
--      o | | | | | | | | | | | | | | { {14,14} },
--      e | | | | | | | | | | | | | | { {16,16} },
--      u | | | | | | | | | | | | | | { {19,19} },
--      a | | | | | | | | | | | | | | { {23,23} },
--      u | | | | | | | | | | | | | | { {26,26} },
--      a | | | | | | | | | | | | | | { {27,27} },
--      e | | | | | | | | | | | | | | { {29,29} },
--                                     }
```

## [^] range complement

When a range starts with the metacharacter pair `[^` then everything **not** in the class is included. The `^` is a range complement operator.

If `[aeiou]` are all of the vowels, then `[^aeiou]` means all characters that are *not* vowels.

```
include regex.e as re
regex r = re:new( "[^aeiou]" )
? re:find_all( r, "The simple, powerful language" )
```

## Units and Groups

The regex coding system operates on one **unit** of the regex expression at a time. Any single ordinary character is easy to recognize as being a unit.

An *escaped metacharater*, such as `\$` is also a unit. It is a unit composed of two symbols.

It is easy for a unit to "get lost" in a regular expression:

```
abc\$
```

In a regex you may use round brackets: `( )` to group symbols. You may also nest brackets.

This example is identical in meaning to the one above, but it emphasizes that `\$` is a single unit:

```
-- haystack string: abc$defg
-- regular expression: abc(\$)

--matches abc$
```

The brackets make it clear that you are searching for a literal `$` and not using the `$` as an anchor.



Brackets are used to create elaborate groups of symbols in exotic regex codes.

When brackets are used, the group automatically also becomes a **named group**. A named group gets an identifier. That identifier can be used as a constant in the rest of the regex expression--saving a lot of typing. The section on named groups will explain how this works.

Brackets are metacharacters themselves, so you must escape them if you need to use brackets in their literal meaning.

## Matching metacharacters

When you see a metacharacter you must consider what special meaning it may have.

Inside a class `*[ ]*`, metacharacters are, mostly, just characters (no special meaning). Of course, there are gotchas!

```
regex r = re:new( "[rds^@]" )
```

The class includes the characters 'r', 'd', 's', '^', and '@'.

```
regex r = re:new( "[^rds@]" )
```

When you see `^[^ ]*` it means that none of the characters that follow the are to be matched. That is to say, any other character will match. The `^` just after the `[` has a special metacharacter meaning. The actual characters in the class are the **complement** of the characters listed.

If you want `[` or `]` to be part of you character class, you need a special notation to include these metacharacters in your class. The `( \ )`backslash is known as the "escape" character. It is used to turn a metacharacter into an ordinary character.

```
regex r = re:new( "[rds\[]" )
```

The characters are now 'r', 'd', 's', and '['. The bracket has been escaped from its metacharacter meaning.

To include a `\` in the regex you are creating, you must also "escape" the backslash. That means you must write `*\***` to represent one backslash.

BUT! The `\` is also an escape character in a Euphoria sequence that you will be searching. The two escape characters are working against each other!

```
include std/regex.e as re
regex r = re:new( "\\\\" )
object result = re:find( r, "Eup\\horia" )
? result

-- {
--   {4,4}
-- }
```

To include one backslash in a string you need `\\` (escaping in the string). For each `\` in a regular expression you also need `\\` (escaping the regular expression). The painful result is that you need `\\` to match `\\` which is in reality just the `\` character.

## Metacharacter shortcuts and escaped characters

The `\` backslash is also used to to code commonly used classes of characters.

shortcut	class meaning	long form
<code>\d</code>	any decimal digit character	<code>[0-9]</code>
<code>\D</code>	an non-digit character	<code>[^0-9]</code>
<code>\s</code>	any whitespace character	<code>[ \t\n\r]</code>

\S	any non-whitespace character	[^\t\n\r]
\w	any alphanumeric character	[a-zA-Z0-9]
\W	any non-alphanumeric character	[^a-zA-Z0-9]

It may be hard to notice (depending on the font you are using) that a blank character ( ' ' ) is included in the class of whitespace characters. A blank space is significant when writing a regular expression.

Notice that '\t', '\n', ... are recognized as "escaped" characters that represent the non-printing characters like TAB and NEWLINE.

```
include std/regex.e as re
regex r = re:new( "\\n" )
result = re:find( r, "Eup\\horia\\n" )
? result
-- {
--   {10,10}
-- }
```

In a Euphoria string \n is a single character. Therefore in a regular expression you only need \\n to search for it.

If you encounter a regex shortcut, then it is considered to be a single character. Shortcuts can be entered into a class and may be part of a larger class definition.

```
regex r = re:new( "[\s.!,;,:?]" )
```

The regex for whitespace is now extended with punctuation markers.

## | Alternation

? as an optional

The metacharacter( | )is the **alternation** symbol. It allows one of two choices to be included in the regex match.

```
include std/regex.e as re
regex r = re:new( "a|b" ) -- a or b
? re:find_all( r, "xxxxaxxxxbbxxxxxx" )
--      |      |      {
--      a      |      {5,5},
--      b      |      {11,11}
--      }      }
```

Thus 'a' **or** 'b' will result in a valid match.

Groups are also recognized:

```
include std/regex.e as re

regex r = re:new( "sink|swim" )

? re:find_all( r, "I am afraid I will sink if I try to swim" )
--      |      |      {
--      |      |      {
--      sink  |      {20,23} -- sink
--      |      |      },
--      |      |      {
--      swim  |      {37,40} -- swim
--      |      |      }
--      |      |      }
```

## Repeated patterns

Regular expressions are about patterns formed from characters. A simple pattern occurs when a unit or group is repeated several times within the string being searched.

The ( **+** ) metacharacter is used to indicate that a pattern can be matched **one or more** times. The **+** "quantifier" applies to the unit or group to its left.

```
include std/regex.e as re
regex r = re:new( "a+" )

? re:find_all( r, "cat" ) -- one
--      |      {
--      a      {2,2}
--      }

? re:find_all( r, "caat" ) -- two
--      |      {
--      aa     {2,3}
--      }

? re:find_all( r, "caaatatonic" ) -- many
--      |      |      {
--      aaaa |      {2,5},
--      a    {7,7}
--      }
```

The ( **\*** ) metacharacter is used to indicate that a pattern can be matched **zero or more** times. The **\*** "quantifier" applies to the unit or group to its left.

```
include std/regex.e as re
regex r = re:new( "a*" )

? re:find_all( r, "cat" ) -- one
--      |      {
--      |      {1,0},
--      a      {2,2},
--      {3,2}
--      }
```

Surprise! You may have expected only one match, but three matches are reported. The first and last matches are "zero length" matches indicated by {1,0} and {3,2} which are "illegal" Euphoria indexes. The **\*** allows for *zero or more* matches, and it is true that 'c' is a zero match for 'a'.

```
include std/regex.e as re
regex r = re:new( "a*" )
```

The ( **?** ) metacharacter is used to indicate that a pattern can be matched **zero or one** times. The **?** "quantifier" applies to the unit or group to its left.

```
include std/regex.e as re
regex r = re:new( "a?" )

? re:find_all( r, "cat" ) -- one
--      |      {
--      |      {1,0},
--      a      {2,2},    -- a
--      {3,2}
--      }

? re:find_all( r, "caat" ) -- two
--      ||     {
--      ||     {1,0},
--      a|     {2,2},    -- a
--      a     {3,3},    -- a
--      {4,3}
--      }
```

```
? re:find_all( r, "caaaatatonic" ) -- many
--                                     {
--                                     {1,0},
--                                     {2,2},      -- a
--                                     {3,3},      -- a
--                                     {4,4},      -- a
--                                     {5,5},      -- a
--                                     {6,5},
--                                     {7,7},      -- a
--                                     {8,7},
--                                     {9,8},
--                                     {10,9},
--                                     {11,10},
--                                     {12,11}
--                                     }
```

The quantifiers may be used to create complex regex searches.

```
include std/regex.e as re
regex r = re:new( "A+B*C?D" )

? re:find( r, "AAAD" )
--      |      {
--      AAAD    {1,4}
--      }

? re:find( r, "ABBBBCD" )
--      |      {
--      ABBBBCD {1,7}
--      }

? re:find( r, "BBBCD" )
--      |
--      0

? re:find( r, "ABCCD" )
--      |
--      0

? re:find( r, "AAABBC" )
--      |
--      0
```

## Extent of matching: Greedy, Non-Greedy

left to right searching, and backing up to find match

problems in getting too much, too little

## Numeric quantifiers

The meta characters( { , } )are used to specify **numeric quantifiers**.

If there is only one number, such as {2}, then an **exact** match is required:

```
include std/regex.e as re
regex r = re:new( "a{2} b{3} c{4}" )

? re:find_all( r, "aa bbb cccc" )
--      |      |      |      {
--      |      |      |      {
--      aa bbb cccc {1,11}
--      }
--      }

? re:find_all( r, "aaa bb cccccc" )
```

```
--{ }
```

A numeric quantifier can represent a **specific range** of numbers { minimum, maximum }:

A numeric quantifier can also represent an **implied range** of numbers { minimum, } where the upper limit is implied to be infinity, and { , maximum } where the lower limit is implied to be zero.

## Named units and backtracking

When you create groups using ( ) brackets, you also automatically create named constants of the form \1 through \9. The groups are created and named in order from left to right.

Once a group gets named, it may be used in the regular expression as a shortcut for that group--this saves retyping the group. This is also known as **backtracking**, since only groups to the left (i.e. backwards) can be used in writing the rest of the regular expression.

```
include regex.e as re
regex r

r = re:new( "(abc|xyz) \1" )
? re:find_all( r, "abc abc" )
-- same as next example

r = re:new( #/(abc|xyz) \1/ )

? re:find_all( r, "abc abc" )
--      |      |      {
--      |      |      {
--      abc abc      {1,7},
--      abc          {1,3}
--      }
--      }
```

The find\_all() functions returns the complete match and all of the sub-matches.

## Search and Replace

## OOP Style Programming

From Wikipedia: "Imperative programming is focused on describing how a program operates. In computer science terminology, imperative programming is a programming paradigm that describes computation in terms of statements that change a program state."

Typically imperative computer programs are easy to understand. Imperative programs translate well into the native machine language of a computer resulting in efficient and fast executing applications.

Complex code is organized using subroutines in the form of procedures and functions. For this reason Euphoria can also be called a procedural language.

**Object Oriented Programming (OOP)** is a style of imperative programming. We substitute the word *entity* for the term "object" as it applied to OOP languages--to avoid confusion with the Euphoria object. OOP programming is still imperative programming but with an extra layer of organization. An **entity** is "a named structure that combines subroutines and data-objects." Using dot-notation it is possible to select a subroutine or a data-object of a particular entity. An OOP language has built in syntax that makes using entities easy.

Euphoria was designed as a strictly imperative language. The simplicity and speed of Euphoria is a result of this design choice.

However, you can still program in an OOP style using an external library. The Archive has several OOP libraries you can try:

Mike Nelson has written *Method Euphoria*

<http://www.rapideuphoria.com/method%20euphoria.zip> described as: "An OOP library for Euphoria. Single inheritance with interfaces, exceptions, event handling. Full cleanup after errors including Eu runtime errors. Now supports task switching."

Mike Nelson also wrote *Diamond-Lite Object Oriented Library*

<http://www.rapideuphoria.com/diamondlite.zip> described as: "A simpler version of his Diamond Object Oriented Programming Library. This lite version is intended for beginner and intermediate OOP programmers. Mar 6: it now prints an error file for debugging."

Jeremy Cowgar updated a preprocessor written by David Cuny  
*Dot Notation for Euphoria (v2)*

<http://www.rapideuphoria.com/dot2.zip> described as: "An update to David Cuny's DOT program that supports new Euphoria 4.0 features such as namespaces, enums, new built-in functions and more. It also adds DOT notation for sequence access. REQUIRES 4.x."

Mathew Lewis wrote *OOEU*

<http://ooeu.sourceforge.net/> which is a modified Euphoria Euphoria 2.5 interpreter; you can download source-code and binary versions for Windows and Linux. Its description: "As-is, this code allows you to use some object oriented techniques with Euphoria, without the added overhead or clumsy syntax that an added Object

Oriented library requires."

# Database Access

If you are interested in interfacing Euphoria with an external database use The Archive as a resource. The following links are provided as a starting point.

## SQLite

SQLite describes itself as "Small. Fast. Reliable. Choose any three."

- Website: <http://www.sqlite.org/>
- Install database: using package manager
- EuSQLite: A wrapper for SQLite. First created by Ray Smith. Now updated by Chris Burch.
- Download: <http://eusqlite.wikispaces.com/>

## MySQL

MySQL describes itself as "The world's most popular open source database." Now owned by Oracle.

- Website: <http://www.mysql.com/>
- Install database:
- MySQL for Euphoria: By Fabio Ramirez  
<http://www.rapideuphoria.com/emyslib.zip>
- MySQL 5.0 for Euphoria from Fabio Ramirez: By MC Wong  
<http://www.rapideuphoria.com/emysql.zip>

## MariaDB

MariaDB describes itself as "An enhanced, drop-in replacement for MySQL."

Note: MySQL wrappers not tested yet.

## PostgreSQL

PostgreSQL describes itself as "The world's most advanced open source database."

- Website: <http://www.postgresql.org/>
- PostgreSQL8.1 wrapper: By Jeremy Peterson.  
<http://www.rapideuphoria.com/postgressqlwrap1.zip>

## Firebird

Firebird describes itself as the "True universal open source database."

- Website: <http://www.firebirdsql.org/>
- WIN Firebird Server ODBC 179K Steve Baxter Jan 14/10 Demonstration program for Firebird Server ODBC access using odbcc.e and wxEuphoria



with two-tone flexible datagrid. Uses the EMPLOYEE.FDB example database that comes standard with Firebird install.

<http://www.rapideuphoria.com/fbodbc.zip>

## ODBC

**ODBC** is "a standard interface for accessing database management systems."

WIN ODBC Library 269K Tone Skoda Mar 4/08 3.00

With ODBC you can manipulate many database formats, including Access .mdb files. This library lets you loop through selected records of a table and quickly get and/or change fields. A lot was learned from `odbc.ew` by Matthew Lewis. Mar 4: Uploaded to RDS site

[http://www.rapideuphoria.com/tslibrary\\_odbc.zip](http://www.rapideuphoria.com/tslibrary_odbc.zip)

WIN Firebird Server ODBC 179K Steve Baxter Jan 14/10

Demonstration program for Firebird Server ODBC access using `odbc.e` and `wxEuphoria` with two-tone flexible datagrid. Uses the EMPLOYEE.FDB example database that comes standard with Firebird install.

- GEN ODBC Database Connectivity 36K Matthew Lewis Mar 10/05 14.00

A library that wraps the ODBC API for Euphoria, allowing Euphoria programs access to almost any type of database. Mar 10: v1.34: Connections can now be properly closed. <http://www.rapideuphoria.com/odbc.zip>

- WIN SQL Utility 5K Jonas Temple Mar 12/01

Inspired by Matt Lewis' ODBC wrapper, this utility allows entry of SQL statements and displays the results in a dynamically built list view. requires: Win32Lib 0.55.1 and Matt's ODBC wrapper. Mar 12: Only return first 100 records of a set. Uses latest version of Matt Lewis' ODBC wrapper.

<http://www.rapideuphoria.com/sq4less.zip>

- WIN Sample `odbc.e` cursors connection to AS/400 (iSeries) 1K MC Wong Sep 26/09
- - Sample code using Matthew Lewis' `odbc.e` to access AS/400 tables (2 tables). Matthew's demo program uses `wxWidget` so this may be helpful to someone. This demo program uses cursors to read the entire file. Same DSN setup from sample1  
<http://www.rapideuphoria.com/sample2.zip>
- WIN Sample `odbc.e` connection to AS/400 (iSeries) 71K MC Wong Sep 25/09

Sample code using Matthew Lewis' `odbc.e` to access AS/400 tables (2 tables). Matthew's demo program uses `wxWidget` so this may be helpful to someone.

<http://www.rapideuphoria.com/sample1.zip>

## Windows GUI Toolkits

tinEWG & Designer [euphoria.indonesianet.de](http://euphoria.indonesianet.de)

# Unix Gui Toolkit

# Multiplatform GUI Toolkits

**wxeu and wxWidgets**



**Selected Archive Index**

# DESIGN \_\_\_\_\_



- syntax
- internals

# BNF

A **BNF** is a "language used to formally describe a computer language." The official description of the OpenEuphoria language is the interpreter. A BNF description of Euphoria is provided as a way to better understand how the interpreter works.

The following description of BNF is taken directly from the documentation of BNF\_CHK written by Umberto Salsi.

## Syntax Checker

A file that BNF\_CHK can parse must be a sequence of zero or more rules. The special character "#" starts a comment and all the characters up to the end of the line are ignored.

## Rules

Every rule begins with its rule identifier, followed by an equal sign =, followed by an expression and terminated by semicolon:

identifier = expression ;

## Rule identifiers

Any sequence of letters, digits and underscore starting with a letter or an underscore is a rule identifier (non-terminal symbol).

## Expressions

An expression is one or more terms separated by a vertical bar:

term1 | term2 | term3

## Terms

A term is a product of factors. Factors can be simply written in the order, no "product" symbol is required. Spaces are allowed to separate contiguous identifiers.

## Factors

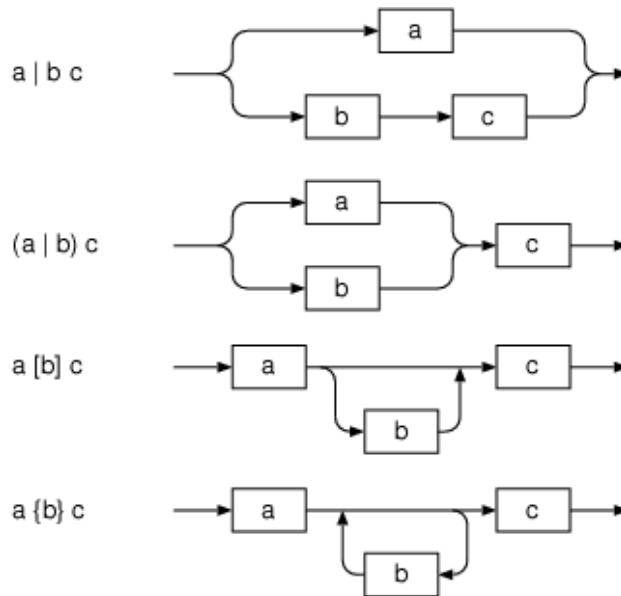
A factor can be:

- a rule name `some_rule`
- a literal string enclosed between double-quotes `"some text"`
- a range of characters `"A".."Z"`
- an optional expression enclosed between square parentheses `[expression]`
- an expression that can match zero or more times enclosed between curly braces `{expression}`
- an expression enclosed between round parentheses just to alter the order of precedence between factors and terms `(a|b|c)`.



Literal words of the language must be enclosed within double quotes. Only ASCII printable characters are allowed. The character double quote itself can be expressed through the special sequence \" (backslash, double quote) and the character backslash can be expressed as (backslash, backslash). Commonly used escaped sequences \"a\\b\\n\\r\\t\" are also allowed. Any other byte can be expressed in hexadecimal form as \\xHH where HH are two hexadecimal digits.

The figure below illustrates some features of the EBNF syntax through syntax diagrams: the square boxes are the rules, while the arrows indicate the allowed paths between rules.



Algebraic syntax and syntax diagrams compared.

## BNF Described

The BNF syntax allowed by the program can be expressed in terms of the BNF syntax itself as follows:

1.  $\text{bnf\_file} = \{ \text{rule}_2 \} ;$
2.  $\text{rule} = \text{identifier}_3 \text{ "=" } \text{expression}_6 \text{ " ;" } ;$
3.  $\text{identifier} = ( \text{letter}_4 \mid \text{"\_"} ) \{ \text{letter}_4 \mid \text{digit}_5 \mid \text{"\_"} \} ;$
4.  $\text{letter} = \text{"a" .. "z"} \mid \text{"A" .. "Z"} ;$
5.  $\text{digit} = \text{"0" .. "9"} ;$
6.  $\text{expression} = \text{term}_7 \{ \text{"|"} \text{term}_7 \} ;$
7.  $\text{term} = \text{factor}_8 \{ \text{factor}_8 \} ;$
8.  $\text{factor} = \text{identifier}_3 \mid \text{literal}_9 \mid \text{range}_{10} \mid \text{"(" } \text{expression}_6 \text{ ")" } \mid \text{"{" } \text{expression}_6 \text{ "}" } \mid \text{"[" } \text{expression}_6 \text{ "]" } ;$
9.  $\text{literal} = \text{" " } \{ \text{char}_{11} \} \text{" " } ;$
10.  $\text{range} = \text{" " } \text{char}_{11} \text{" " } \text{" .. " } \text{" " } \text{char}_{11} \text{" " } ;$

11. `char = plain_char12 | escaped_char13 | hex_char14 ;`
12. `plain_char = "\x20".." \x7F" | "\x80".." \xFF" ;`
13. `escaped_char = "\" ( "\"" | "\" | "a" | "b" | "n" | "r" | "t" ) ;`
14. `hex_char = "\x" hex15 hex15 ;`
15. `hex = "0".."9" | "a".."f" | "A".."F" ;`

### See Also:

- BNF\_CHK checker

online version: [http://www.icosaedro.it/bnf\\_chk/bnf\\_chk-on-line.html](http://www.icosaedro.it/bnf_chk/bnf_chk-on-line.html)

offline version: [http://www.icosaedro.it/bnf\\_chk/index.html](http://www.icosaedro.it/bnf_chk/index.html)

# Syntax

## Basics

The syntax of Euphoria is described using a form of BNF notation.

```

ALPHA ==: ('a' - 'z') | ('A' - 'Z')
DIGIT ==: ('0' - '9')
USCORE ==: '_'
EOL ==: new line character

IDENTIFIER ==: ( ALPHA | USCORE ) [(ALPHA | DIGIT | USCORE) ... ]

EXPRESSION ==: NUMEXPR | STREXPR | SEQEXPR | BOOLEXPR

NUMEXPR ==: (an expression that evaluates to an atom)

STREXPR ==: (an expression that evaluates to a string sequence)

SEQEXPR ==: (an expression that evaluates to an sequence)

BOOLEXPR ==: (an expression that evaluates to an atom in which zero represents
               falsehood and non-zero represents truth)

BINARYEXPR ==: [ EXPRESSION BINOP EXPRESSION ]

BINOP ==: 'and' | 'or' | 'xor' | '+' | '-' | '*' | '/'

UNARYEXPR ==: [ UNARYOP EXPRESSION ]

UNARYOP ==: 'not' | '-'

STATEMENT ==:

STMTBLK ==: STATEMENT [STATEMENT ...]

LABEL      ==: 'label' STRINGLIT

LISTDELIM  ==: ','

STRINGLIT  ==: SIMPLESTRINGLIT | RAWSTRINGLIT

SIMPLESTRINGLIT ==: SSLITSTART [ (CHAR | ESCCHAR) ... ] SSLITEND
SSLITSTART  ==: '"'
SSLITEND    ==: '"'
CHAR        ==: (any byte value)
ESCCHAR     ==: ESCLEAD ( 't' | 'n' | 'r' | '\' | '"' \ ' ')
ESCLEAD     ==: '\'

RAWSTRINGLIT ==: DQRAWSTRING | BQRAWSTRING
DQRAWSTRING ==: '""' [ MARGINSTR ] [CHAR ...] '""'
BQRAWSTRING ==: '`' [ MARGINSTR ] [CHAR ...] '`'
MARGINSTR   ==: '_' ...

SCOPETYPE ==: 'global' | 'public' | 'export' | 'override'

DATATYPE ==: 'atom' | 'integer' | 'sequence' | 'object' | IDENTIFER

```

## Statements

### Directives

```

INCLUDESTMT
WITHSTMT

```

NAMESPACE

## Variables, Constants, Enums

VARDECLARE  
CONSTDECLARE  
ENUMDECLARE  
SLICING

## Flow Control

IFSTMT  
SWITCHSTMT  
BREAKSTMT  
CONTINUESTMT  
RETRYSTMT  
EXITSTMT  
FALLTHRUSTMT  
FORSTMT  
WHILESTMT  
LOOPSTMT  
GOTOSTMT  
CALL  
IFDEFSTMT

## Subroutines

PROCDECLARE  
FUNCDECLARE  
TYPEDECLARE  
RETURN

## include

INCLUDESTMT

```
INCLUDESTMT ==: 'include' FILEREF [ 'as' NAMESPACEID ] EOL
FILEREF    ==: A file path that may be enclosed in double-quotes.
NAMESPACEID ==: IDENTIFIER
```

**NOTE** that after the file reference, the only text allowed is the keyword 'as' or the start of a comment. Nothing else is permitted on the same text line.

### See Also:

[include statement](#)

## Sequence Slice

SLICING

```
SLICE      ==: SLICESTART INTEXPRESSION SLICEDELIM INTEXPRESSION SLICEEND
SLICESTART ==: '['
SLICEDELIM ==: '..'
SLICEEND   ==: ']'
```

### See Also:

[Slicing of Sequences](#)

## if

## IFSTMT

```

IFSTMT ==: IFTEST [ ELSIF ... ] [ELSE] ENDIF
IFTEST ==: 'if' ATOMEXPR [ LABEL ] 'then' [ STMTBLOCK ]
ELSIF  ==: 'elsif' ATOMEXPR 'then' [ STMTBLOCK ]
ELSE   ==: 'else' [ STMTBLOCK ]
ENDIF  ==: 'end' 'if'

```

### See Also:

[if statement](#)

## ifdef

### IFDEFSTMT

```

IFDEFSTMT ==: IFDEFTEST [ ELSDEFIF ... ] [ELSEDEF] ENNDEFIF
IFDEFTEST ==: 'ifdef' DEFEXPR 'then' [ STMTBLOCK ]
ELSDEFIF  ==: 'elsifdef' DEFEXPR 'then' [ STMTBLOCK ]
ELSEDEF   ==: 'elsedef' [ STMTBLOCK ]
ENNDEFIF  ==: 'end' 'ifdef'
DEFEXPR   ==: DEFTERM [ DEFOP DEFTERM ]
DEFTERM   ==: [ 'not' IDENTIFIER ]
DEFOP     ==: 'and' | 'or'

```

### See Also:

[ifdef statement](#)

## switch

### SWITCHSTMT

```

SWITCHSTMT ==: SWITCHTEST CASE [ CASE ... ] [ CASEELSE ] [ ENDSWITCH ]
SWITCHTEST ==: 'switch' EXPRESSION [ WITHFALL ] [ LABEL ] 'do'
WITHFALL   ==: ('with' | 'without') 'fallthru'
CASE       ==: 'case' CASELIST 'then' [ STMTBLOCK ]
CASELIST   ==: EXPRESSION [(LISTDELIM EXPRESSION) ...]
CASEELSE   ==: 'case' 'else'
ENDSWITCH  ==: 'end' 'switch'

```

### See Also:

[switch statement](#)

## break

### BREAKSTMT

```

BREAKSTMT ==: 'break' [ STRINGLIT ]

```

### See Also:

[break statement](#)

## continue

### CONTINUESTMT

```

CONTINUESTMT ==: 'continue' [ STRINGLIT ]

```

### See Also:

[continue statement](#)

## retry

RETRYSTMT

```
RETRYSTMT      ==:  'retry' [ STRINGLIT ]
```

[See Also:](#)

[retry statement](#)

## exit

EXITSTMT

```
EXITSTMT       ==:  'exit' [ STRINGLIT ]
```

[See Also:](#)

[exit statement](#)

## fallthru

FALLTHRUSTMT

```
FALLTHRUSTMT   ==:  'fallthru'
```

[See Also:](#)

[switch statement](#)

## for

FORSTMT

```
FORSTMT ==: 'for' FORIDX [ LABEL ] 'do' [STMTBLK] 'end' 'for'  
FORIDX  ==: IDENTIFIER '=' NUMEXPR 'to' NUMEXPR ['by' NUMEXPR]
```

[See Also:](#)

[for statement](#)

## while

WHILESTMT

```
WHILESTMT ==:  
    'while' BOOLEXP [WITHENTRY] [LABEL] 'do' STMTBLK [ENTRY] 'end' 'while'  
WITHENTRY ==: 'with' 'entry'  
ENTRY ==: 'entry' [STMTBLK]
```

[See Also:](#)

[while statement](#)

## loop

LOOPSTMT

```
LOOPSTMT ==:
  'loop' [WITHENTRY] [LABEL] 'do' STMTBLK [ENTRY] 'until' BOOLEXP 'end' 'loop'
```

### See Also:

[loop until statement](#)

## goto

GOTOSTMT

```
GOTOSMT ==: 'goto' LABEL
```

### See Also:

[goto statement](#)

## Declare a Variable

VARDECLARE

```
VARDECLARE ==: [SCOPE] DATATYPE IDENTLIST
IDENTLIST ==: IDENT [',' IDENTLIST]
IDENT ==: IDENTIFIER [ '=' EXPRESSION ]
```

Notes: The type of the EXPRESSION must be compatible with the DATATYPE.

## Declare a Constant

CONSTDECLARE

```
CONSTDECLARE ==: [SCOPE] 'constant' IDENTLIST
```

## Declare an Enumerated Value

ENUMDECLARE

```
ENUMDECLARE ==: [SCOPE] [ ENUMVAL | ENUMTYPE ]
ENUMVAL ==: 'enum' [ 'by' ENUMDELTA ] IDENTLIST
ENUMDELTA ==: [ '+' | '-' | '*' | '/' ] NUMEXPR
ENUMTYPE ==: 'enum' 'type' [ 'by' ENUMDELTA ] IDENTLIST 'end' 'type'
```

## Call a Subroutine

CALL Used to call (invoke) either a procedure or a function.

```
CALL ==: IDENTIFIER '(' [ARGLIST] ')'
ARGLIST ==: ARGUMENT [',' ARGLIST]
```

### See Also:

[procedures](#) | [functions](#) | [types](#)

## Define a Procedure

PROCDECLARE

```
PROCDECLARE ==: [SCOPE] 'procedure' IDENTIFIER '(' [PARMLIST] ')' [STMTBLK] 'end' 'procedure'
PARMLIST ==: PARAMETER [',' PARMLIST]
PARAMETER ==: DATATYPE IDENTIFIER
```

Notes: The procedure statement block **must not** contain a return statement.

[See Also:](#)

[procedures](#)

## Define a Function

FUNCDECLARE

```
FUNCDECLARE ==: [SCOPETYPE] 'function' IDENTIFIER '(' [PARMLIST] ')' [STMTBLK] 'end' 'function'
PARMLIST ==: PARAMETER [',' PARMLIST]
PARAMETER ==: DATATYPE IDENTIFIER
```

Notes:

- The function statement block **must** contain a return statement.

[See Also:](#)

[functions](#)

## Define a User Defined Type

TYPEDECLARE

```
TYPEDECLARE ==: [SCOPETYPE] 'type' IDENTIFIER '(' PARAMETER ')' [STMTBLK] 'end' 'type'
PARAMETER ==: DATATYPE IDENTIFIER
```

Notes:

- The type statement block **must** contain a return statement.
- It must return an integer; 0 means that the supplied argument is not of the correct type.

[See Also:](#)

[types](#)

## Return the Result of a Function

RETURN

```
RETURN ==: 'return' EXPRESSION
```

[See Also:](#)

[types](#)

## Default Namespace

```
NAMESPACE ==: 'namespace' IDENTIFIER EOL
```

[See Also:](#)

[Using namespaces](#)

## with Options

WITHSTMT



```
WITHSTMT ==: [ "with" | "without" ] WITHOPTION
WITHOPTION ==: [ "profile" | "profile_time" | "trace" | "batch" |
                 "type_check" | "indirect_includes" | "inline" | WITHWARNING ]
WITHWARNING ==: "warning" [ WARNOPT]
WARNOPT ==: SETWARN | ADDWARN | SAVEWARN | RESTOREWARN | STRICTWARN
SETWARN ==: ['='] '{' WARNLIST '}'
ADDWARN ==: ['+='] ['&='] '{' WARNLIST '}'
SAVEWARN ==: 'save'
RESTOREWARN ==: 'restore'
STRICTWARN ==: 'strict'
```

## See Also:

[with / without](#)

# Internals

The interpreter has four binary components:

- Interpreter
- Translator
- Backend
- Library

The Euphoria interpreter has two parts: the frontend and the backend. The **frontend** is a parser that converts source-code into a set of **Intermediate Language** (IL) instructions. The **backend** then takes the IL instructions and executes the program.

When the *interpreter* executes source-code, the frontend parses and prepares the code, and then the backend executes the code.

When the *shrouder* executes source-code, only the frontend is run producing an .il file. This .il file may be run by the backend as an independent step to execute the program.

When the *binder* executes source-code, the .il instructions produced by the frontend are combined with the backend to produce a stand-alone executable program. The executable program may then be run independently at any time.

When the *translator* executes source-code, the .il instructions are translated into C-code. This C-code is compiled with an installed C compiler producing an executable program.

The *library* is called by the backend for the many builtins included in Euphoria.

## Euphoria Data Structures

### Euphoria Representation of Object

Every Euphoria object is stored as-is. A special unlikely floating point value is used for NOVALUE. NOVALUE signifies that a variable has not been assigned a value or the end of a sequence.

### C Representation of Object

Every Euphoria object is either stored as is, or as an encoded pointer. A Euphoria integer is stored in a 32-bit signed integer. If the number is too big for a Euphoria integer, it is assigned to a 64-bit double float in a structure and an encoded pointer to that structure is stored in the said 32-bit memory space. Sequences are stored in a similar way.

```

32 bit number range:
 0X8      0XA      0XC      0XE      0X0      0X2      0X4      0X6      0X8
-4*2^29  -3*2^29  -2*2^29-1  -2^29    0*2^29    1*2^29    2*2^29    3*2^29  4*2^29
 *-----*-----*-----*-----*-----*-----*-----*-----o
                                o NOVALUE = -2*2^29-1
      o<-----ATOM_INT-----[-2*2^29..4*2^29)----->o
      |<-----ATOM_DBL-----[-3*2^29..4*2^29)----->o
-->|      |<--- IS_SEQUENCE [-4*2^29..-3*2^29)
-->|      o<--- IS_DBL_OR_SEQUENCE [-4*2^29..-2*2^29-1)
-->|sequence|<-----
      |<-----atom----->|
----->|double|<-----
      |<-----integer----->|
|<-----object----->|

```

Euphoria integers are stored in object variables as-is. An object variable is a four byte signed integer. Legal integer values for Euphoria integers are between -1,073,741,824 (  $-2^{30}$  ) and +1,073,741,823 (  $2^{30}-1$  ). Unsigned hexadecimal numbers from C000\_0000 to

FFFF\_FFFF are the negative integers and numbers from 0000\_0000 to 3FFF\_FFFF are the positive integers. The hexadecimal values not used as integers are thus 4000\_0000 to BFFF\_FFFF. Other values are for encoded pointers. Pointers are always 8 byte aligned. So a pointer is stored in 29-bits instead of 32 and can fit in a hexadecimal range 0x2000\_0000 long. The pointers are encoded in such a way that their encoded values will never be in the range of the integers. Pointers to sequence structures (struct s1) are encoded into a range between 8000\_0000 to 9FFF\_FFFF. Pointers to structures for doubles (struct d) are encoded into a range between A000\_0000 to BFFF\_FFFF. A special value NOVALUE is at the end of the range of encoded pointers is BFFF\_FFFF and it signifies that there is no value yet assigned to a variable and it also signifies the end of a sequence. In C, values of this type are stored in the 'object' type. The range 4000\_0000 to 7FFF\_FFFF is unused.

A double structure 'struct d' could indeed contain a value that is legally in the range of a Euphoria integer. So the encoded pointer to this structure is recognized by the interpreter as an 'integer' but in this internals document when we say Euphoria integer we mean it actually is a C integer in the legal Euphoria integer range.

## The C Representations of Sequence and Atom

```
// Sequence Header
struct s1
{
    object_ptr base;    // base is such that base[1] is the first element
    long length;        // this is the sequence length
    long ref;           // ref is the number of as virtual copies of this sequence
    long postfill;      // is how many extra objects could fit at the end of base
    cleanup_ptr cleanup; // this is a pointer to a Euphoria routine that is run
                        // just before the sequence is freed.
}
```

However, we allocate more than this structure. Inside the allocated data but past the structure, there also is an area of 'pre free space'; sequence data pointed to by base[1] to base[\$], \$ being the length; a NOVALUE terminator for the sequence, and an area of post fill space. In memory, immediately following the structure there is the following data stored:

```
object pre_fill_space[]; // could have 0 (not exist) or more elements before used data
object base[1..$];       // sequence members pointed to by base
object base[$+1];        // a magic number terminating the sequence members (NOVALUE)
object post_fill_space[]; // could have 0 (not exist) or more elements after used data
```

Taken together these are what get represented in memory.

```
base length ref postfill cleanup pre fill space base[1..$] NOVALUE post fill space
```

By their nature, sequences are variable length, dynamic entities and so the C structure needs to cater for this. When a sequence is created, we allocate enough RAM for the combined header and the initial storage for the elements.

Field	Description
base	This contains the address of the first element less the length of one element. Thus base[1] points to the first element and base[0] points to a fictitious element just before the first one, which is never used. Initially, base contains the address of the last member of the sequence header but as the sequence is resized, it can point to the last member or anywhere after.
length	Contains the current number of elements in the sequence.
ref	Contains the count of references to this sequence. Only when this is zero, can the RAM used by the sequence be returned to the system for reuse.

postfill	The size of 'post fill space' in element spaces. Rather than using bytes, postfill is measured in objects which are each address wide elements. If this is non-zero, we can append to the sequence with at most postfill new elements before needing to reallocate RAM.
cleanup	If not null, it points to a routine that is called immediately before the sequence is deleted.
pre fill space	There are 0 or more spaces before base[1]. We can calculate the free space in *objects* at the front of a sequence, s1, in C by $(\&s1.base[1] - (object\_ptr)(1+\&s1))$ . In EUPHORIA, you will have to divide by the size of a C POINTER on the difference. When elements are removed from the front of a sequence, we simply adjust the address in base to point to the new <i>first</i> element and reduce the length count. If we want to prepend and this pre fill space has some positive size, then we make room by decrementing base and increment the length. The new data is then assigned to base[1].
base[1]..base[length] sequence data	This is actual data.
base[\$+1]	This is always set to NOVALUE.
post fill space	There are 0 or more spaces after base[length+1]. The number of spaces is stored in postfill. If postfill is non-zero we can append by incrementing the length, decrementing postfill and assigning the new data to base[\$]. When we remove from the end of the sequence, we increment postfill and decrement the length.

```
// Atom Header
struct d
{
    double dbl;           // the actual value of a double number.
    long ref;             // ref is the number of virtual copies of this double
    cleanup_ptr cleanup;  // this is a pointer to a Euphoria routine that is run
                        // just before the sequence is freed.
}
```

Now offset of the 'ref' in struct d must be the same as the offset of the 'ref' in struct s1. To this end, the 64bit implementation of 4.1 has these members in a different order.

## Euphoria Object Macros and Functions

### Description

The macros are imperfect. For example, IS\_SEQUENCE(NOVALUE) returns TRUE and IS\_ATOM\_DBL will return TRUE for integer values as well as encoded pointers to 'struct d's. This is why there is an order that these tests are made: We test IS\_ATOM\_INT and if that fails we can use IS\_ATOM\_DBL and then that will only be true if we pass an encoded pointer to a double. We must be sure that something is not NOVALUE before we use IS\_SEQUENCE on it.

*Often we know foo is not NOVALUE before getting into this:*

```
// object foo
if (IS_ATOM_INT(foo)) {
    // some code for a Euphoria integer
} else if (IS_ATOM_DBL(foo)) {
    // some code for a double
} else {
    // code for a sequence foo
}
```

A sequence is held in a 'struct s1' type and a double is contained in a 'struct d'.

## Type Value Functions and Macros

## IS\_ATOM\_INT

```
<internal> int IS_ATOM_INT( object o )
```

### Returns

true if object is a Euphoria integer and not an encoded pointer.

### Note

IS\_ATOM\_INT will return true even though the argument is out of the Euphoria integer range when the argument is positive. These values are not possible encoded pointers.

## IS\_ATOM\_DBL

```
<internal> int IS_ATOM_DBL( object o )
```

### Returns

true if the object is an encoded pointer to a double struct.

### Assumption

*o* must not be a Euphoria integer.

## IS\_ATOM

```
<internal> int IS_ATOM( object o )
```

### Returns

true if the object is a Euphoria integer or an encoded pointer to a 'struct d'.

## IS\_SEQUENCE

```
<internal> int IS_SEQUENCE( object o )
```

### Returns

true if the object is an encoded pointer to a 'struct s1'.

### Assumption

*o* is not NOVALUE.

## IS\_DBL\_OR\_SEQUENCE

```
<internal> int IS_DBL_OR_SEQUENCE( object o )
```

### Returns

true if the object is an encoded pointer of either kind of structure.

## Type Conversion Functions and Macros

## MAKE\_INT

```
<internal> object MAKE_INT( signed int x )
```

### Returns

an object with the same value as x. x must be with in the integer range of a legal Euphoria integer type.

## MAKE\_UINT

```
<internal> object MAKE_UINT( unsigned int x )
```

### Returns

an object with the same value as x.

### Assumption

x must be an **unsigned** integer with in the integer range of a C unsigned int type.

### Example

MAKE\_UINT(4\*1000\*1000\*1000) will make a Euphoria value of four billion by creating a double.

## MAKE\_SEQ

```
<internal> object MAKE_SEQ( struct s1 * sptr )
```

### Returns

an object with an argument of a pointer to a 'struct s1' The pointer is encoded into a range for sequences and returned.

## NewString

```
<internal> object NewString(char *s)
```

### Returns

an object representation of a Euphoria byte string s. The returned encoded pointer is a sequence with all of the bytes from s copied over.

## MAKE\_DBL

```
<internal> object MAKE_DBL( struct d * dptr )
```

### Returns

an object with an argument of a pointer to a 'struct d' The pointer is encoded into a range for doubles and returned.

## NewDouble

```
<internal> object NewDouble( double dbl )
```

### Returns

an object with an argument a double dbl. A struct d is allocated and dbl is assigned to the value part of that structure. The pointer is encoded into the range for doubles and returned.

## DBL\_PTR

```
<internal> struct d * DBL_PTR( object o )
```

### Returns

The pointer to a 'struct d' from the object o.

### Assumption

IS\_ATOM\_INT(o) is FALSE and IS\_ATOM\_DBL(o) is TRUE.

## SEQ\_PTR

```
<internal> struct s1 * SEQ_PTR( object o )
```

### Returns

The pointer to a 'struct s1' from the object o.

### Assumption

IS\_SEQUENCE(o) is TRUE and o is not NOVALUE.

## get\_pos\_int

```
#include be_machine.h
<internal> uintptr_t get_pos_int(char *where, object x)
```

### Returns

a unsigned long value by truncating what x's value is to an integer

### Comment

Any object may be passed. A sequence results in a runtime failure. There may be a cast of a double to a smaller ranged long type.

## Creating Objects

## NewS1

```
<internal> object NewS1 ( long size )
```

## Returns

A sequence object with size members which are not yet set to a value.

## Object Constants

Use MAXINT and MININT to check for overflow and underflow, NOVALUE to check if a variable has not been assigned, and use NOVALUE to terminate a sequence.

### NOVALUE

```
<internal> object NOVALUE
```

Indicates that a variable has not been assigned and also terminates a sequence.

### MININT

```
<internal> signed int MININT
```

The minimal Euphoria integer. This is  $-(2^{30})$ .

### MAXINT

```
<internal> signed int MAXINT
```

The maximal Euphoria integer. This is  $2^{30}-1$ .

### HIGH\_BITS

```
<internal> signed int HIGH_BITS
```

HIGH\_BITS is an integer value such that if another integer value  $c$  lies outside of the range between MININT and MAXINT,  $c + \text{HIGH\_BITS}$  will be non-negative.

## Proof that HIGH\_BITS is #C000\_0000 on 32-bit version of EUPHORIA.

- In the following expressions powers have higher precedence than unary minus.\* if  $c$  is a non-ATOM-INT value, then

$c$  belongs to the set  $[-2^{31}, -2^{30}-1 (= \text{NOVALUE})] \cup [2^{30}, 2^{31}]$ .

$c + -2^{30}$  belongs to the set  $[-2^{31}-2^{30}, -2^{30}-1-2^{30}] \cup [2^{30}-2^{30}, 2^{30}]$  which is  $[-3*2^{30}, -2^{31}-1] \cup [0, 2^{30}]$ . However the lower values wrap around to non-negative numbers:

$-2^{31}-1$  wraps to  $2^{31}-1$ .  $-3*2^{30}$  wraps around to  $2^{30}$ .

$c + -2^{30}$  belongs to the set  $[2^{30}, 2^{31}-1] \cup [0, 2^{30}] = [0, 2^{31}-1]$

This is the set of all non-negative numbers that can fit into 32-bit signed longs.  $-2^{30}$  is the unsigned version of #C000\_0000. QED.

A visual way of looking at it is, adding #C000\_0000 to the set of non-ATOM\_INTS rotates the set to the negative side by  $-\text{MININT}$  ( $2^{30}$ ). The already negative ones wrap around to the positive; the positive numbers stay positive and hug the zero. Since adding



#C000\_0000 on registers is 1-1 and onto, we also know that ATOM\_INTs will all be mapped to negative signed longs.

## Testing for Overflow

There are two ways to test for overflow:

1.  $(c > \text{MAXINT}) \parallel (c < \text{MININT})$
2.  $(c + \text{HIGH\_BITS}) \geq 0$

## Parser

Inserting tokens into the token buffer is the easiest way to add features to the Euphoria parser. The tokens are two-element sequences one of the class of token and the other the token's value:

```
{<class>,<value>}
```

Each of the class values are capitalized words for some keyword or VARIABLE. The list of constants is in `reswords.e`. Often it is enough to only examine the class. In the case of variables, it is important to know which variable. In this case the second element, comes into play.

You can use `putback` to put tokens into the token buffer. The tokens will be pulled out by the parser in a *filio* manner, like a stack.

## Backend Instructions

After the Parser processes the instructions. It creates Backend instructions that are easily translated or interpreted. The system uses opcodes and some parameters which are put on a stack. This backend language is similar to assembler. You have opcodes (instructions) and parameters. These parameters must be integers themselves but some may serve as pointers to arbitrary Euphoria objects. As a developer of Euphoria itself, rather than a developer that uses Euphoria, it is important to know exactly what these opcodes do and what they are for. In this section we will document what they are for, and how they manipulate the instruction pointer, and stack.

IF instruction:

```
[ IF instruction ] [ test value ] [ failure address ]
```

INTEGER\_CHECK instruction:

The `INTEGER_CHECK` is used to ensure that something has a value considered to be 'integer' to the EUPHORIA language definition. The instruction takes the next argument as a pointer to a value and determines whether this value is in the legal integer range, regardless of how that number is represented. If not in legal range, then the program ends execution in a type-check failure error message.

```
[ INTEGER_CHECK instruction ] [ test pointer ]
```

ATOM\_CHECK instruction:

The `ATOM_CHECK` is used to determine whether something has a numeric value rather than a sequence. The instruction takes an argument as a pointer to a value and determines whether the value is an atom. If it is not an atom, then the program ends execution in a type-check failure error message.

```
[ ATOM_CHECK instruction ] [ test pointer ]
```

IS\_AN\_INTEGER instruction:

The `IS_AN_INTEGER` instruction is used to determine whether something has a value considered to be 'integer' to the EUPHORIA language definition. The instruction takes the argument as a pointer to a value and determines whether this value is in the legal integer range, regardless of how that number is represented. If it is in the 'integer'

range then the value pointed by the second argument will be 1 otherwise it will be 0.

[ IS\_AN\_INTEGER instruction ] [ test pointer ][ return value pointer ]

# DOCUMENTING \_\_\_\_\_



eudoc

creole

creole cheatsheet

wkhtmltopdf

LaTeX

- [eudoc](#)
- [creole](#)
- [topdf-doc](#)
- [docbuild](#)



## Creole Formatting

Home page: <http://www.wikicreole.org/>

The wikicreole.org cheatsheet:

//italics//	→ <i>italics</i>
**bold**	→ <b>bold</b>
* Bullet list * Second item ** Sub item	→ • Bullet list • Second item ..• Sub item
# Numbered list # Second item ## Sub item	→ 1. Numbered list 2. Second item 2.1 Sub item
Link to [[wikipage]]	→ Link to <a href="#">wikipage</a>
[[URL linkname]]	→ <a href="#">linkname</a>
== Large heading === Medium heading ==== Small heading	→ <b>Large heading</b> <b>Medium heading</b> <b>Small heading</b>
No linebreak!	→ No linebreak!
Use empty row	Use empty row
Force\\linebreak	→ Force linebreak
Horizontal line: ----	→ Horizontal line: _____
{{Image.jpg title}}	→ Image with title
=table =header   a table row   b table row	→ Table
{{ == [[Nowiki]]; /**don't** format// }}}	→ == [[Nowiki]]; /**don't** format//

[www.wikicreole.org](http://www.wikicreole.org)

- EuWeb extensively uses the creole syntax in news, forum messages, tickets, comments and wiki.
- EuWikiEngine uses creole syntax.
- Practice editing and learn by example in the SandBox.
- Learn more about Wikis in general on the [Original Wiki](#).

## Headings

Markup	Effect
= Heading 1	<b>Heading 1</b>
= = Heading 2	<b>Heading 2</b>
= = = Heading 3	<b>Heading 3</b>
= = = = Heading 4	<b>Heading 4</b>
= = = = = Heading 5	<b>Heading 5</b>
= = = = = Heading 6	<b>Heading 6</b>

## Heading Examples

# Heading 1

## Heading 2

### Heading 3

#### Heading 4

#### Heading 5

## Heading 6

## Fonts

Markup	Effect
<b>**</b>	<b>Strong</b>
<i>//</i>	<i>Emphasis</i>
<u>__</u>	<u>Underline</u>
<sup>^^</sup>	<sup>Superscript</sup>
<sub>,,</sub>	<sub>Subscript</sub>
<u>++</u>	<u>Addition</u>
<del>--</del>	<del>Deletion</del>
<b>##</b>	Monospaced
{{{ }}}}	Escape creole
!!	Comment

## Objects

### Links

CamelCase

CamelCase words create internal links automatically.

[[Euphoria]]

**Euphoria** Forced internal link

[[http://en.wikipedia.org/wiki/CamelCase]]

**http://en.wikipedia.org/wiki/CamelCase** External Link

[[Euphoria|Cool Programming Language]]

**Cool Programming Language** Link with label override.

[[OpenEuphoria.org -> http://openeuphoria.org]]

**OpenEuphoria.org** Alternate Link with label override.

Links to the online manual

```
[[man:std_wildcard.html#is_match|Wildcard is_match]]
```

Interwiki link failed for Wildcard is\_match

Links to our ticket system

```
View [[ticket:123]] for more information
```

View Interwiki link failed for ticket:123 for more information

## Images

```
{{http://fluidae.com/images/euphoria.png}}
```

## Unordered Lists

Unordered lists use the asterisk `*` as a line prefix. Each consecutive asterisk increases the indentation of the list items.

```
* Point 1
** Point 1.1
* Point 2
** Two levels
*** Three levels
```

- Point 1
  - Point 1.1
- Point 2
  - Two levels
    - Three levels

## Ordered Lists

Ordered lists use the hash `#` as a line prefix. Each consecutive hash increases the indentation of the list items.

```
# Point 1
## Point 1.1
# Point 2
## Two levels
### Three levels
```

1. Point 1
  1. Point 1.1
2. Point 2
  1. Two levels
    1. Three levels

## Line breaks

```
This is how you\\make a line break.
```

This is how you  
make a line break.

## Horizontal Rule

```
This text is
----
separated
```

This text is

Tables

Tables use the following syntax:

=Heading1	=Heading2	
One	Two	
Three	Four	

**Heading1** **Heading2**

One	Two
Three	Four

To insert a *bar* character "|" inside a table definition, prefix the bar with a tilde "~".

column1	column2
	puts(1, "~" )

Definitions

; Word	
:Definition	
Word	Definition

Special Sections

Plain Text

{{{This text is //not// parsed.}}}
This text is //not// parsed.
{{{ This text is //not// parsed.  }}}
This text is //not// parsed.

Forced New Lines

[[[ This forces a line break after each new line in the source text ]]]
This forces a line break after each new line in the source text



## Source Code

```
<eucode>
s = sprintf("%08d", 12345)
-- s is "00012345"
</eucode>
```

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

## Float Right

```
%%style=floatright
%(
This text will be in a float-right box.
)%
```

Euphoria is a programming language. It is used to write general purpose programs that can run on a number of platforms, such as Windows and Linux. Actually, you can write any type of applications in Euphoria, but it is suited to some types more than others.

Euphoria is a programming language. It is used to write general purpose programs that can run on a number of platforms, such as Windows and Linux. Actually, you can write any type of applications in Euphoria, but it is suited to some types more than others.

This text will be in a float-right box.

## Float Left

```
%%style=floatleft
%(
This text will be in a float-left box.
)%
```

Euphoria is a programming language. It is used to write general purpose programs that can run on a number of platforms, such as Windows and Linux. Actually, you can write any type of applications in Euphoria, but it is suited to some types more than others.

This text will be in a float-left box.

Euphoria is a programming language. It is used to write general purpose programs that can run on a number of platforms, such as Windows and Linux. Actually, you can write any type of applications in

Euphoria, but it is suited to some types more than others.

## Embedded

Euphoria is a programming language. It is used to write general purpose programs that can run on a number of platforms, such as Windows and Linux. Actually, you can write any type of applications in Euphoria, but it is suited to some types more than others.

```
%%style=embedded
%(
This text will be in a box.
)%
```

Euphoria is a programming language. It is used to write general purpose programs that can run on a number of platforms, such as Windows and Linux. Actually, you can write any type of applications in Euphoria, but it is suited to some types more than others.

Euphoria is a programming language. It is used to write general purpose programs that

can run on a number of platforms, such as Windows and Linux. Actually, you can write any type of applications in Euphoria, but it is suited to some types more than others.

This text will be in a box.

Euphoria is a programming language. It is used to write general purpose programs that can run on a number of platforms, such as Windows and Linux. Actually, you can write any type of applications in Euphoria, but it is suited to some types more than others.

## Plugins

### Table of Contents

```
<<TOC>>  
<<TOC heading=no>>
```

### Inserting Other Pages

This will insert the entire contents of a wiki page.

```
<<wiki page=Euphoria>>  
<<wiki heading=no page=Euphoria>>  
<<wiki heading=yes page=Euphoria>>
```

**Name:**

wkhtmltopdf 0.12.1 (with patched qt)

**Synopsis:**

wkhtmltopdf [GLOBAL OPTION]... [OBJECT]... <output file>

**Document objects:**

wkhtmltopdf is able to put several objects into the output file, an object is either a single webpage, a cover webpage or a table of content. The objects are put into the output document in the order they are specified on the command line, options can be specified on a per object basis or in the global options area. Options from the Global Options section can only be placed in the global options area

A page objects puts the content of a single webpage into the output document.

(page)? <input url/file name> [PAGE OPTION]...

Options for the page object can be placed in the global options and the page options areas. The applicable options can be found in the Page Options and Headers And Footer Options sections.

A cover objects puts the content of a single webpage into the output document, the page does not appear in the table of content, and does not have headers and footers.

cover <input url/file name> [PAGE OPTION]...

All options that can be specified for a page object can also be specified for a cover.

A table of content object inserts a table of content into the output document.

toc [TOC OPTION]...

All options that can be specified for a page object can also be specified for a toc, further more the options from the TOC Options section can also be applied. The table of content is generated via XSLT which means that it can be styled to look however you want it to look. To get an aide of how to do this you can dump the default xslt document by supplying the

--dump-default-toc-xsl, and the outline it works on by supplying

--dump-outline, see the Outline Options section.

**Description:**

Converts one or more HTML pages into a PDF document, using wkhtmltopdf patched qt.

**Global Options:**

--collate	Collate when printing multiple copies (default)
--no-collate	Do not collate when printing multiple copies
--cookie-jar <path>	Read and write cookies from and to the supplied cookie jar file
--copies <number>	Number of copies to print into the pdf file (default 1)
-d, --dpi <dpi>	Change the dpi explicitly (this has no effect on X11 based systems)
-H, --extended-help	Display more extensive help, detailing less common command switches
-g, --grayscale	PDF will be generated in grayscale
-h, --help	Display help
--htmldoc	Output program html help
--image-dpi <integer>	When embedding images scale them down to this dpi (default 600)
--image-quality <integer>	When jpeg compressing images use this quality (default 94)
--license	Output license information and exit
-l, --lowquality	Generates lower quality pdf/ps. Useful to shrink the result document space
--manpage	Output program man page
-B, --margin-bottom <unitreal>	Set the page bottom margin

-L, --margin-left <unitreal>	Set the page left margin (default 10mm)
-R, --margin-right <unitreal>	Set the page right margin (default 10mm)
-T, --margin-top <unitreal>	Set the page top margin
-O, --orientation <orientation>	Set orientation to Landscape or Portrait (default Portrait)
--page-height <unitreal>	Page height
-s, --page-size <Size>	Set paper size to: A4, Letter, etc. (default A4)
--page-width <unitreal>	Page width
--no-pdf-compression	Do not use lossless compression on pdf objects
-q, --quiet	Be less verbose
--read-args-from-stdin	Read command line arguments from stdin
--readme	Output program readme
--title <text>	The title of the generated pdf file (The title of the first document is used if not specified)
--use-xserver	Use the X server (some plugins and other stuff might not work without X11)
-V, --version	Output version information and exit

#### Outline Options:

--dump-default-toc-xsl	Dump the default TOC xsl style sheet to stdout
--dump-outline <file>	Dump the outline to a file
--outline	Put an outline into the pdf (default)
--no-outline	Do not put an outline into the pdf
--outline-depth <level>	Set the depth of the outline (default 4)

#### Page Options:

--allow <path>	Allow the file or files from the specified folder to be loaded (repeatable)
--background	Do print background (default)
--no-background	Do not print background
--cache-dir <path>	Web cache directory
--checkbox-checked-svg <path>	Use this SVG file when rendering checked checkboxes
--checkbox-svg <path>	Use this SVG file when rendering unchecked checkboxes
--cookie <name> <value>	Set an additional cookie (repeatable), value should be url encoded.
--custom-header <name> <value>	Set an additional HTTP header (repeatable)
--custom-header-propagation	Add HTTP headers specified by --custom-header for each resource request.
--no-custom-header-propagation	Do not add HTTP headers specified by --custom-header for each resource request.
--debug-javascript	Show javascript debugging output
--no-debug-javascript	Do not show javascript debugging output (default)
--default-header	Add a default header, with the name of the page to the left, and the page number to the right, this is short for: --header-left='[webpage]' --header-right='[page]/[toPage]' --top 2cm --header-line
--encoding <encoding>	Set the default text encoding, for input
--disable-external-links	Do not make links to remote web pages
--enable-external-links	Make links to remote web pages (default)
--disable-forms	Do not turn HTML form fields into pdf form fields (default)
--enable-forms	Turn HTML form fields into pdf form fields
--images	Do load or print images (default)
--no-images	Do not load or print images
--disable-internal-links	Do not make local links
--enable-internal-links	Make local links (default)
-n, --disable-javascript	Do not allow web pages to run javascript
--enable-javascript	Do allow web pages to run javascript (default)
--javascript-delay <msec>	Wait some milliseconds for javascript finish (default 200)
--load-error-handling <handler>	Specify how to handle pages that fail to load: abort, ignore or skip (default

```

                                abort)
--load-media-error-handling <handler> Specify how to handle media files
                                that fail to load: abort, ignore or skip
                                (default ignore)
--disable-local-file-access      Do not allowed conversion of a local file
                                to read in other local files, unless
                                explicitly allowed with --allow
--enable-local-file-access      Allowed conversion of a local file to read
                                in other local files. (default)
--minimum-font-size <int>       Minimum font size
--exclude-from-outline           Do not include the page in the table of
                                contents and outlines
--include-in-outline            Include the page in the table of contents
                                and outlines (default)
--page-offset <offset>          Set the starting page number (default 0)
--password <password>           HTTP Authentication password
--disable-plugins               Disable installed plugins (default)
--enable-plugins                Enable installed plugins (plugins will
                                likely not work)
--post <name> <value>           Add an additional post field (repeatable)
--post-file <name> <path>       Post an additional file (repeatable)
--print-media-type              Use print media-type instead of screen
--no-print-media-type           Do not use print media-type instead of
                                screen (default)
-p, --proxy <proxy>            Use a proxy
--radiobutton-checked-svg <path> Use this SVG file when rendering checked
                                radiobuttons
--radiobutton-svg <path>        Use this SVG file when rendering unchecked
                                radiobuttons
--run-script <js>              Run this additional javascript after the
                                page is done loading (repeatable)
--disable-smart-shrinking        Disable the intelligent shrinking strategy
                                used by WebKit that makes the pixel/dpi
                                ratio none constant
--enable-smart-shrinking         Enable the intelligent shrinking strategy
                                used by WebKit that makes the pixel/dpi
                                ratio none constant (default)
--stop-slow-scripts             Stop slow running javascripts (default)
--no-stop-slow-scripts          Do not Stop slow running javascripts
--disable-toc-back-links        Do not link from section header to toc
                                (default)
--enable-toc-back-links         Link from section header to toc
--user-style-sheet <url>        Specify a user style sheet, to load with
                                every page
--username <username>           HTTP Authentication username
--viewport-size <>              Set viewport size if you have custom
                                scrollbars or css attribute overflow to
                                emulate window size
--window-status <>windowStatus> Wait until window.status is equal to this
                                string before rendering page
--zoom <float>                 Use this zoom factor (default 1)

```

#### Headers And Footer Options:

```

--footer-center <text>          Centered footer text
--footer-font-name <name>       Set footer font name (default Arial)
--footer-font-size <size>       Set footer font size (default 12)
--footer-html <url>            Adds a html footer
--footer-left <text>           Left aligned footer text
--footer-line                   Display line above the footer
--no-footer-line                Do not display line above the footer
                                (default)
--footer-right <text>           Right aligned footer text
--footer-spacing <real>         Spacing between footer and content in mm
                                (default 0)
--header-center <text>          Centered header text
--header-font-name <name>       Set header font name (default Arial)
--header-font-size <size>       Set header font size (default 12)
--header-html <url>            Adds a html header
--header-left <text>           Left aligned header text
--header-line                   Display line below the header
--no-header-line                Do not display line below the header
                                (default)

```

```
--header-right <text>           Right aligned header text
--header-spacing <real>         Spacing between header and content in mm
                                (default 0)
--replace <name> <value>       Replace [name] with value in header and
                                footer (repeatable)
```

#### TOC Options:

```
--disable-dotted-lines         Do not use dotted lines in the toc
--toc-header-text <text>       The header text of the toc (default Table
                                of Contents)
--toc-level-indentation <width> For each level of headings in the toc
                                indent by this length (default 1em)
--disable-toc-links            Do not link from toc to sections
--toc-text-size-shrink <real>  For each level of headings in the toc the
                                font is scaled by this factor (default
                                0.8)
--xsl-style-sheet <file>       Use the supplied xsl style sheet for
                                printing the table of content
```

#### Page sizes:

The default page size of the rendered document is A4, but using this `--page-size` option this can be changed to almost anything else, such as: A3, Letter and Legal. For a full list of supported pages sizes please see <http://qt-project.org/doc/qt-4.8/qprinter.html#PaperSize-enum>.

For a more fine grained control over the page size the `--page-height` and `--page-width` options may be used

#### Reading arguments from stdin:

If you need to convert a lot of pages in a batch, and you feel that wkhtmltopdf is a bit too slow to start up, then you should try `--read-args-from-stdin`,

When `--read-args-from-stdin` each line of input sent to wkhtmltopdf on stdin will act as a separate invocation of wkhtmltopdf, with the arguments specified on the given line combined with the arguments given to wkhtmltopdf

For example one could do the following:

```
echo "http://qt-project.org/doc/qt-4.8/qapplication.html qapplication.pdf" >> cmds
echo "cover google.com http://en.wikipedia.org/wiki/Qt_(software) qt.pdf" >> cmds
wkhtmltopdf --read-args-from-stdin --book < cmds
```

#### Specifying A Proxy:

By default proxy information will be read from the environment variables: `proxy`, `all_proxy` and `http_proxy`, proxy options can also be specified with the `-p` switch

```
<type> := "http://" | "socks5://"
<serif> := <username> (":" <password>)? "@"
<proxy> := "None" | <type>? <string>? <host> (":" <port>)?
```

Here are some examples (In case you are unfamiliar with the BNF):

```
http://user:password@myproxyserver:8080
socks5://myproxyserver
None
```

#### Footers And Headers:

Headers and footers can be added to the document by the `--header-*` and `--footer*` arguments respectively. In header and footer text string supplied to e.g. `--header-left`, the following variables will be substituted.

```
* [page]           Replaced by the number of the pages currently being printed
* [frompage]       Replaced by the number of the first page to be printed
* [topage]         Replaced by the number of the last page to be printed
* [webpage]        Replaced by the URL of the page being printed
* [section]        Replaced by the name of the current section
* [subsection]     Replaced by the name of the current subsection
* [date]           Replaced by the current date in system local format
* [time]           Replaced by the current time in system local format
* [title]          Replaced by the title of the of the current page object
```

- \* [doctitle] Replaced by the title of the output document
- \* [sitepage] Replaced by the number of the page in the current site being converted
- \* [sitepages] Replaced by the number of pages in the current site being converted

As an example specifying `--header-right "Page [page] of [toPage]"`, will result in the text "Page x of y" where x is the number of the current page and y is the number of the last page, to appear in the upper left corner in the document.

Headers and footers can also be supplied with HTML documents. As an example one could specify `--header-html header.html`, and use the following content in `header.html`:

```
<html><head><script>
function subst() {
  var vars={};
  var x=window.location.search.substring(1).split('&');
  for (var i in x) {var z=x[i].split('=');vars[z[0]] = unescape(z[1]);}
  var x=['frompage','topage','page','webpage','section','subsection','subsubsection'];
  for (var i in x) {
    var y = document.getElementsByClassName(x[i]);
    for (var j=0; j<y.length; ++j) y[j].textContent = vars[x[i]];
  }
}
</script></head><body style="border:0; margin: 0;" onload="subst()">
<table style="border-bottom: 1px solid black; width: 100%">
  <tr>
    <td class="section"></td>
    <td style="text-align:right">
      Page <span class="page"></span> of <span class="topage"></span>
    </td>
  </tr>
</table>
</body></html>
```

As can be seen from the example, the arguments are sent to the header/footer html documents in get fashion.

#### Outlines:

Wkhtmltopdf with patched qt has support for PDF outlines also known as book marks, this can be enabled by specifying the `--outline` switch. The outlines are generated based on the `<h?>` tags, for a in-depth description of how this is done see the Table Of Content section.

The outline tree can sometimes be very deep, if the `<h?>` tags were spread to generous in the HTML document. The `--outline-depth` switch can be used to bound this.

#### Table Of Content:

A table of content can be added to the document by adding a `toc` object to the command line. For example:

```
wkhtmltopdf toc http://qt-project.org/doc/qt-4.8/qstring.html qstring.pdf
```

The table of content is generated based on the H tags in the input documents. First a XML document is generated, then it is converted to HTML using XSLT.

The generated XML document can be viewed by dumping it to a file using the `--dump-outline` switch. For example:

```
wkhtmltopdf --dump-outline toc.xml http://qt-project.org/doc/qt-4.8/qstring.html qstring.pdf
```

The XSLT document can be specified using the `--xsl-style-sheet` switch. For example:

```
wkhtmltopdf toc --xsl-style-sheet my.xsl http://qt-project.org/doc/qt-4.8/qstring.html qstring.pdf
```

The `--dump-default-toc-xsl` switch can be used to dump the default XSLT style sheet to stdout. This is a good start for writing your own style sheet

```
wkhtmltopdf --dump-default-toc-xsl
```

The XML document is in the namespace "http://wkhtmltopdf.org/outline" it has a root node called "outline" which contains a number of "item" nodes. An item can contain any number of item. These are the outline subsections to the section the item represents. A item node has the following attributes:

- \* "title" the name of the section.
- \* "page" the page number the section occurs on.
- \* "link" a URL that links to the section.
- \* "backLink" the name of the anchor the the section will link back to.

The remaining TOC options only affect the default style sheet so they will not work when specifying a custom style sheet.

Contact:

If you experience bugs or want to request new features please visit  
<<https://github.com/wkhtmltopdf/wkhtmltopdf/issues>>



# Documentation Building

You can create your own version of the Euphoria documentation and you can use the same process to document your own projects.

Documentation is written in two locations: embedded within source-code and as separate text files. Creole wiki style markup is used to add formatting information.

Location	Contents
../euphoria/docs	Documentation written as text files.
../euphoria/docs/images	Artwork in svg and png formats.
../euphoria/include/std	Source-code for the standard library. Each file contains embedded documentation.

The Euphoria eudoc utility extracts documentation written inside source-code, combines source-code and text file sources together, and produces a single output file.

The Euphoria creole utility reads a text file formatted using Creole wiki markup and produces html or LaTeX formatted documentation. The html documentation can be viewed in any browser; html documentation can be converted into a PDF. The LaTeX files can be converted to a PDF.

The wkhtmltopdf utility reads html files and produces a PDF. A 15 MB download. Used to create this documentation.

The htmldoc is an alternative utility that reads html files and produces a PDF. A 5 MB download. Works with jpg and png images but not with svg graphics. Does not work with css style sheets. Convenient gui interface. A simple way to make a complete PDF book.

Libre Office will read html files and produce a PDF.

A TeX installation and texworks editor can be used to convert LaTeX files into a PDF. Requires a 50+ MB download to install.

## Software Used

### Sansation Font

The Sansation font is the headline font used for titles and sections. It was created by Bernd Montag; this is a "freeware" font.

Download this font from [www.dafont.com](http://www.dafont.com); search for "sansation."

As an administrator, the downloaded fonts are copied to /user/share/fonts/ttf/sansation; then update the font cache from the terminal `sudo fc-cache -fv`.

For general text: a serif font is used for text and a monospace font is used for source-code.

### eudoc.ex

From source-code and independent files to *plain text documentation*.

Source-code is in the OpenEuphoria SCM <http://scm.openeuphoria.org/>

### creole.ex

From plain text documentation--written with creole formatting--to *html documentation*.

Produces html files from text files containing creole wiki formatting.

Source-code is in the OpenEuphoria SCM <http://scm.openeuphoria.org/>

## wkhtmltopdf

From html documentation to *PDF documentation*.

Website <http://wkhtmltopdf.org/>

For a *unix* system goto the wkhtmltopdf website:

- An "ubuntu" download will provide a .deb package for Ubuntu, Mint, Debian, and derivatives.
- A "centos" download will provide a .rpm package for Redhat and derivatives.

The Synaptic Package Manager on Mint Linux installs and downloads an older version than that provided by the website links.

For a *windows* system the wkhtmltopdf website provides links for an XP version and a current Windows version.

## PDF Creation Process

- Latest eudoc and creole

You may want to download the latest files from the Euphoria SCM. Compiled versions are used in this tutorial.

- Edit manual.af

The ../euphoria/docs/manual.af file is a list of files (source-code and plain text) that will be combined into one documentation file.

- Run eudoc

From the ../euphoria/docs directory run eudoc from a terminal:

```
$ eudoc -a manual.af -o doctmp.txt --verbose
```

- Create Temp Directory

Create a temporary directory, tempdoc for example, to hold intermediate files. Copy the ../euphoria/docs/images directory here. Copy ../euphoria/docs/style.css here. Copy doctmp.txt here. Copy ../euphoria/docs/manual.af here.

- Run creole

From your ../tempdoc directory run creole from a terminal:

```
$ creole doctmp.txt -v
```

- Create the Coverpage

Convert the ../euphoria/docs/cover.txt into an html [file](#):

```
$ creole cover.txt
```

Copy the html file to tempdoc.

- Run topdf.ex

The wkhtmltopdf utility requires a list of html files that will be combined to create the PDF.

The Euphoria `topdf.ex` demonstrates how convert `manual.af` into a form used by `wkhtmltopdf`:

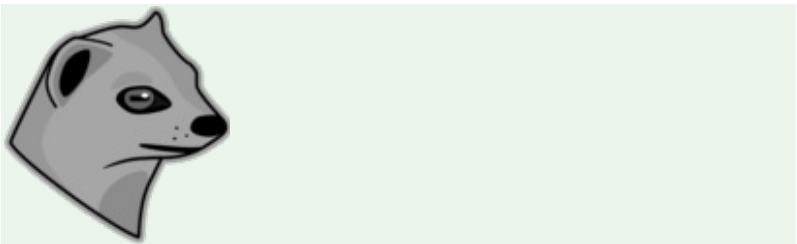

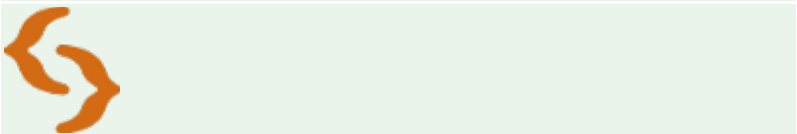
Run the `$ eui topdf`

```
$ eui topdf
```

### See Also:

[eudoc](#) | [creole](#) | [Creole markup](#) | [wkhtmltopdf](#) | [htmldoc](#) | [tex](#)

# Artwork

Graphic	Name
	Mongoose trademark
	openEuphoria wordmark
	Claws trademark

Original artwork by Smudger.

# MONGOOSE

Our adopted animal friend is the mongoose. Here is a story of a mongoose, really just bits of the story, hopefully enough to make you interested in reading the whole book.



## The Jungle Book, by Rudyard Kipling

"Rikki-Tikki-Tavi"

This is the story of the great war that Rikki-tikki-tavi fought single-handed, through the bath-rooms of the big bungalow in Segowlee cantonment. Darzee, the Tailorbird, helped him, and Chuchundra, the musk-rat, who never comes out into the middle of the floor, but always creeps round by the wall, gave him advice, but Rikki-tikki did the real fighting.

He was a mongoose, rather like a little cat in his fur and his tail, but quite like a weasel in his head and his habits. His eyes and the end of his restless nose were pink. He could scratch himself anywhere he pleased with any leg, front or back, that he chose to use. He could fluff up his tail till it looked like a bottle brush, and his war cry as he scuttled through the long grass was: "Rikk-tikk-tikki-tikki-tchk!"

It is the hardest thing in the world to frighten a mongoose, because he is eaten up from nose to tail with curiosity. The motto of all the mongoose family is "Run and find out," and Rikki-tikki was a true mongoose. He looked at the cotton wool, decided that it was not good to eat, ran all round the table, sat up and put his fur in order, scratched himself, and jumped on the small boy's shoulder.

"Don't be frightened, Teddy," said his father. "That's his way of making friends."

"Ouch! He's tickling under my chin," said Teddy.

"Who is Nag?" said he. "I am Nag. The great God Brahm put his mark upon all our people, when the first cobra spread his hood to keep the sun off Brahm as he slept. Look, and be afraid!"

He spread out his hood more than ever, and Rikki-tikki saw the spectacle-mark on the back of it that looks exactly like the eye part of a hook-and-eye fastening. He was afraid for the minute, but it is impossible for a mongoose to stay frightened for any length of time, and though Rikki-tikki had never met a live cobra before, his mother had fed him on dead ones, and he knew that all a grown mongoose's business in life was to fight and eat snakes. Nag knew that too and, at the bottom of his cold heart, he was afraid.

If you read the old books of natural history, you will find they say that when

the mongoose fights the snake and happens to get bitten, he runs off and eats some herb that cures him. That is not true. The victory is only a matter of quickness of eye and quickness of foot--snake's blow against mongoose's jump--and as no eye can follow the motion of a snake's head when it strikes, this makes things much more wonderful than any magic herb. Rikki-tikki knew he was a young mongoose, and it made him all the more pleased to think that he had managed to escape a blow from behind. It gave him confidence in himself, and when Teddy came running down the path, Rikki-tikki was ready to be petted.

"He saved our lives and Teddy's life," she said to her husband. "Just think, he saved all our lives."

Rikki-tikki had a right to be proud of himself. But he did not grow too proud, and he kept that garden as a mongoose should keep it, with tooth and jump and spring and bite, till never a cobra dared show its head inside the walls.

[See Also:](#)

[Project Gutenberg](#)



- small
- coherent
- powerful
- fast
- benchmark

The look and feel of Euphoria syntax has been carefully crafted. Easy to learn, easy to read, easy to write are Euphoria design principles that are often missing in other languages.

- No surprises.
- Coherent syntax.
- Explicit.
- Simple.
- Dual alternatives.
- Predictable.

## Pseudocode

Authors of books on computer algorithms often resort to **pseudocode** which is "a simplified syntax that makes it easier to reveal how an algorithm works." Once you understand an algorithm you are expected to translate it into your working language-- which is not used at first because it is difficult to read.

Pseudocode translates into Euphoria with very little effort. Pseudocode was invented to make programs easier to read; Euphoria is naturally easy to read.

## No Surprises

Euphoria does not use invisible syntax. Euphoria does not use extra punctuation to make invisible syntax work.

```
def fcn:
  if x:
    continue
  elif:
    return x
```

## Programmer Personality

Programmer personality and language personality go together. Some people like dense compact code. Some people like the expressiveness of obscure syntax that makes a particular line of code easy to write. Some people like raw mathematical rigor. There is a language designed for every personality type, but to any outsider each of these languages will be difficult to learn and use.

Euphoria is easy to learn and easy to write. Everyone can understand Euphoria regardless of their background.

Critique of *clean syntax* supporters. Syntax is based on indentation but an invisible space and an invisible tab are not the same. For some reason you must add extra colons because indentation is insufficient to write this code. The invisible end to the function and the invisible end to the conditional become harder to see as the nesting level of code increases. Because syntax is invisible, you must use the keyword `continue` as a placeholder. You are not allowed any freedom in spacing your code. You need a specialized editor to avoid making indentation errors.

```
function fcn()
  if x then
  elsif
    return x
  end if
end function
```

Regardless of how you type this code both you and the interpreter will be able to understand it.

"Clean" code has its appeal but it is harder to learn and harder to write.



## Small



Powerful

**Fast**

## Benchmark

testing

sample results

benchmark programs

# API



- builtin
- standard library
- optional library

# BUILTINS

## Built-in Subroutines

A **built-in** is "a subroutine that is part of the Euphoria interpreter." A built-in does not require any include file.

A built-in routine has global scope and belongs to the eu namespace. The identifier for the name print is eu:print.

?	abort	and_bits	append
arctan	atom	c_func	c_proc
call	call_func	call_proc	clear_screen
close	command_line	compare	cos
date	delete	delete_routine	equal
find	floor	get_key	getc
getenv	gets	hash	head
include_paths	insert	integer	length
log	machine_func	machine_proc	match
mem_copy	mem_set	not_bits	object
open	option_switches	or_bits	peek
peek2s	peek2u	peek4s	peek4u
peek8s	peek8u	peek longs	peek_longu
peek_pointer	peeks	peek_string	pixel
platform	poke	poke2	poke4
poke8	poke_long	poke_pointer	position
power	prepend	print	printf
puts	rand	remainder	remove
repeat	replace	routine_id	sequence
sin	splice	sprintf	sqrt
system	system_exec	tail	tan
task_clock_start	task_clock_stop	task_create	task_list
task_schedule	task_self	task_status	task_suspend
task_yield	time	trace	xor_bits

New built-in for *alternative literals branch*:

name\_of

Built-in names are not reserved; you can use names from this table in a definition or declaration and override the built-in name.

# GENERAL

- [cmdline](#)
- [console](#)
- [datetime](#)
- [filesys](#)
- [io](#)
- [os](#)
- [pipeio](#)
- [pretty](#)
- [task](#)
- [types](#)
- [utils](#)



# Command Line Handling

## Constants

### NO\_PARAMETER

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

This option switch does not have a parameter. See [cmd\\_parse](#)

### HAS\_PARAMETER

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

This option switch does have a parameter. See [cmd\\_parse](#)

### NO\_CASE

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

This option switch is not case sensitive. See [cmd\\_parse](#)

### HAS\_CASE

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

This option switch is case sensitive. See [cmd\\_parse](#)

### MANDATORY

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

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

### OPTIONAL

```
include std/cmdline.e
```

```
namespace cmdline
public constant OPTIONAL
```

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

## ONCE

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

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

## MULTIPLE

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

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

## HELP

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

This option switch triggers the 'help' display. See [cmd\\_parse](#)

## HEADER

```
include std/cmdline.e
namespace cmdline
public constant HEADER
```

This option switch is simply a help display header to group like options together. See [cmd\\_parse](#)

## VERSIONING

```
include std/cmdline.e
namespace cmdline
public constant VERSIONING
```

This option switch sets the program version information. If this option is chosen by the user `cmd_parse` will display the program version information and then end the program with a zero error code.

## enum

```
include std/cmdline.e
```

```
namespace cmdline
public enum
```

```
HELP_RID | VALIDATE_ALL | NO_VALIDATION |
NO_VALIDATION_AFTER_FIRST_EXTRA | SHOW_ONLY_OPTIONS | AT_EXPANSION |
NO_AT_EXPANSION | PAUSE_MSG | NO_HELP | NO_HELP_ON_ERROR
```

## HELP\_RID

```
include std/cmdline.e
namespace cmdline
HELP_RID
```

Additional help routine id. See [cmd\\_parse](#)

## VALIDATE\_ALL

```
include std/cmdline.e
namespace cmdline
VALIDATE_ALL
```

Validate all parameters (default). See [cmd\\_parse](#)

## NO\_VALIDATION

```
include std/cmdline.e
namespace cmdline
NO_VALIDATION
```

Do not cause an error for an invalid parameter. See [cmd\\_parse](#)

## NO\_VALIDATION\_AFTER\_FIRST\_EXTRA

```
include std/cmdline.e
namespace cmdline
NO_VALIDATION_AFTER_FIRST_EXTRA
```

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

For instance:

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

-D TEST is meant for eui, but -name and -greeting options are meant for greet.ex. See [cmd\\_parse](#)

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

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

## SHOW\_ONLY\_OPTIONS

```
include std/cmdline.e
namespace cmdline
SHOW_ONLY_OPTIONS
```

Only display the option list in `show_help`. Do not display other information (such as program name, options, and so on) See [cmd\\_parse](#)

## AT\_EXPANSION

```
include std/cmdline.e
namespace cmdline
AT_EXPANSION
```

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

## NO\_AT\_EXPANSION

```
include std/cmdline.e
namespace cmdline
NO_AT_EXPANSION
```

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

## PAUSE\_MSG

```
include std/cmdline.e
namespace cmdline
PAUSE_MSG
```

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

## NO\_HELP

```
include std/cmdline.e
namespace cmdline
NO_HELP
```

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

## NO\_HELP\_ON\_ERROR

```
include std/cmdline.e
namespace cmdline
NO_HELP_ON_ERROR
```

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

## enum

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

[OPT\\_IDX](#) | [OPT\\_CNT](#) | [OPT\\_VAL](#) | [OPT\\_REV](#)

## OPT\_IDX

```
include std/cmdline.e
namespace cmdline
OPT_IDX
```

An index into the opts list. See [cmd\\_parse](#)

## OPT\_CNT

```
include std/cmdline.e
namespace cmdline
OPT_CNT
```

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

## OPT\_VAL

```
include std/cmdline.e
namespace cmdline
OPT_VAL
```

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

## OPT\_REV

```
include std/cmdline.e
namespace cmdline
OPT_REV
```

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

## EXTRAS

```
include std/cmdline.e
namespace cmdline
public constant EXTRAS
```

The extra parameters on the cmd line, not associated with any specific option. See [cmd\\_parse](#)

## Subroutines

## command\_line

```
<built-in> function command_line()
```

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

### Returns:

A **sequence**, of strings

1. Executing name or pathname: interpreter or application.
2. Program name.
3. List of user arguments.

### Comments:

The command line syntax is:

```
command_line( [argument] [, argument] )
```

The returned sequence syntax:

```
{ ( interpreter name | interpreter full pathname | application full pathname ), ( program name | application name ), [argument], [, argument] }
```

An argument is a number, a word with no delimiters, or a string using double quotation marks; each argument becomes an string item in the sequence returned by `command_line`.

First item is one of:

- Interpreter name `eui` (unix)
- Interpreter pathname ending in `eui.exe`, `euid.exe`, or `euiw.exe` (windows)
- Application full pathname (windows, unix)

Second item:

- Name of program or application, exactly as typed (unix)
- Full pathname of program or application (windows)

Remaining zero or more items:

- The arguments typed by the user.

The returned sequence always has two items plus any arguments typed by the user.

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

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

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

The output depends on the operating system. The `myprog.ex` test program.

```
--myprog.ex--
include std/console.e

sequence cmd = command_line()
console:display( cmd )
```

### Example 1:

With no extra command line arguments; execute as:

```
eui myprog.ex
```

Output under windows:

```
{"C:\EUPHORIA\BIN\EUI.EXE", "myprog.ex" }
```

Output under unix:

```
{ "eui", "myprog.ex" }
```

### Example 2:

With command line arguments; execute as:

```
eui myprog myfile.dat 12345 "the end"
```

Output under windows:

```
{"C:\EUPHORIA\BIN\EUI.EXE",  
    "myprog",  
    "myfile.dat",  
    "12345",  
    "the end"}
```

### Example 3:

Make an application using eubind or euc .

Execute under windows:

```
myprog.exe
```

Output under windows:

```
{"C:\MYFILES\MYPROG.EXE", "C:\MYFILES\MYPROG.EXE"
```

Execute under unix:

```
./myprog
```

Output under unix:

```
{ "/home/guest/myprograms/myprog", "./myprog"}
```

### Example 4:

```
-- Your program is bound with the name "myprog.exe"  
-- and is stored in the directory c:\myfiles  
-- The user types: myprog myfile.dat 12345 "the end"  
  
cmd = command_line()  
  
-- cmd will be:  
{"C:\MYFILES\MYPROG.EXE",  
    "C:\MYFILES\MYPROG.EXE", -- place holder  
    "myfile.dat",  
    "12345",  
    "the end"  
}  
  
-- Note that all arguments remain the same as in Example 1  
-- except for the first two. The second argument is always
```

```
-- the same as the first and is inserted to keep the numbering
-- of the subsequent arguments the same, whether your program
-- is bound or translated as a .exe, or not.
```

### Example 5:

```
-- a simple parser
sequence cmds = command_line()
sequence inFileName, outFileName

for i = 3 to length(cmds) do
    switch cmds[i] do
        case "-i" then
            inFileName = cmds[i+1]
        case "-o" then
            outFileName = cmds[i+1]
    end switch
end for
```

### See Also:

[build\\_commandline](#) | [option\\_switches](#) | [getenv](#) | [cmd\\_parse](#) | [show\\_help](#)

## option\_switches

```
<built-in> function option_switches()
```

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

### Returns:

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

### Comments:

All switches are recorded in upper case.

### Example 1:

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

### See Also:

[Command line switches](#)

## show\_help

```
include std/cmdline.e
namespace cmdline
public procedure show_help(sequence opts, object add_help rid = - 1,
    sequence cmds = command_line(), object parse_options = {})
```

shows the help message for the given opts.

### Arguments:

1. `opts` : a sequence of options. See the [cmd\\_parse](#) for details.
2. `add_help rid` : an object. Either a routine `_id` or a set of text strings. The default is `-1` meaning that no additional help text will be used.



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

## Comments:

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

## Example 1:

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

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

## Outputs:

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

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

## Example 2:

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

procedure sh()
    for i = 1 to length(description) do
        printf(1, " >> %s <<\n", {description[i]})
    end for
end procedure

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

## Outputs:

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

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

## See Also:

## cmd\_parse

```
include std/cmdline.e
namespace cmdline
public function cmd_parse(sequence opts, object parse_options = {},
    sequence cmds = command_line())
```

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

### Arguments:

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

### Returns:

A **map**, containing the set of actual options used in `cmds`. The returned map has one special key, `EXTRAS` that are values passed on the command line that are not part of any of the defined options. This is commonly used to get the list of files entered on the command line. For instance, if the command line used was *myprog -verbose file1.txt file2.txt* then the `EXTRAS` data value would be {"file1.txt", "file2.txt"}.

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

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

These modifiers can be any combination of:

1. `VALIDATE_ALL` -- The default. All options will be validated for all possible errors.
2. `NO_VALIDATION` -- Do not validate any parameter.
3. `NO_VALIDATION_AFTER_FIRST_EXTRA` -- Do not validate any parameter after the first extra was encountered. This is helpful for programs such as the Interpreter itself: `eui -D TEST greet.ex -name John. -D TEST` should be validated but anything after "greet.ex" should not as it is meant for greet.ex to handle, not eui.
4. `HELP_RID` -- The next Parse Option must either a routine id or a set of text strings. The routine is called or the text is displayed when a parse error (invalid option given, mandatory option not given, no parameter given for an option that requires a parameter, etc...) occurs. This can be used to provide additional help text. By default, just the option switches and their descriptions will be displayed. However you can provide additional text by either supplying a `routine_id` of a procedure that accepts no parameters, or a sequence containing lines of text (one line per element). The procedure is expected to write text to the stdout device.
5. `NO_HELP_ON_ERROR` -- Do not show a list of options on a command line error.
6. `NO_HELP` -- Do not automatically add the switches `'-h'`, `'-?'`, and `'--help'` to display the help text (if any).
7. `NO_AT_EXPANSION` -- Do not expand arguments that begin with `'@'`.
8. `AT_EXPANSION` -- Expand arguments that begin with `'@'`. The name that follows `@` will be opened as a file, read, and each trimmed non-empty line that does not begin with a `'#'` character will be inserted as arguments in the command line. These lines replace the original `'@'` argument as if they had been entered on the original command line.
  - If the name following the `'@'` begins with another `'@'`, the extra `'@'` is removed and the remainder is the name of the file. However, if that file cannot be read, it is simply ignored. This allows *optional* files to be included on the

command line. Normally, with just a single '@', if the file cannot be found the program aborts.

- Lines whose first non-whitespace character is '#' are treated as a comment and thus ignored.
- Lines enclosed with double quotes will have the quotes stripped off and the result is used as an argument. This can be used for arguments that begin with a '#' character, for example.
- Lines enclosed with single quotes will have the quotes stripped off and the line is then further split up use the space character as a delimiter. The resulting 'words' are then all treated as individual arguments on the command line.

An example of parse options:

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

## Comments:

Token types recognized on the command line:

1. a single '-'. Simply added to the 'extras' list
2. a single "--". This signals the end of command line options. What remains of the command line is added to the 'extras' list, and the parsing terminates.
3. -shortName. The option will be looked up in the short name field of opts.
4. /shortName. Same as -shortName.
5. -!shortName. If the 'shortName' has already been found the option is removed.
6. /!shortName. Same as -!shortName
7. --longName. The option will be looked up in the long name field of opts.
8. --!longName. If the 'longName' has already been found the option is removed.
9. anything else. The word is simply added to the 'extras' list.

For those options that require a parameter to also be supplied, the parameter can be given as either the next command line argument, or by appending '=' or ':' to the command option then appending the parameter data.

For example, **-path=/usr/local** or as **-path /usr/local**.

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

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

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

You can also remove all of these switches as *automatic* help switches by using the NO\_HELP parsing option. This just means that these switches are not automatically used as 'help' switches, regardless of whether they are used explicitly or not. So if NO\_HELP is used, and you want to give the user the ability to display the 'help' then you must explicitly set up your own switch to do so. **N.B.**, the 'help' is still displayed if an invalid command line switch is used at runtime, regardless of whether NO\_HELP is used or not.

Option records have the following structure:

1. a sequence representing the (short name) text that will follow the "-" option format. Use an atom if not relevant
2. a sequence representing the (long name) text that will follow the "--" option format. Use an atom if not relevant
3. a sequence, text that describes the option's purpose. Usually short as it is displayed when "-h"/"--help" is on the command line. Use an atom if not required.
4. An object ...
  - If an **atom** then it can be either HAS\_PARAMETER or anything else if there is no parameter for this option. This format also implies that the option is optional, case-sensitive and can only occur once.
  - If a **sequence**, it can contain zero or more processing

flags in any order ...

- MANDATORY to indicate that the option must always be supplied.
  - HAS\_PARAMETER to indicate that the option must have a parameter following it. You can optionally have a name for the parameter immediately follow the HAS\_PARAMETER flag. If one isn't there, the help text will show "x" otherwise it shows the supplied name.
  - NO\_CASE to indicate that the case of the supplied option is not significant.
  - ONCE to indicate that the option must only occur once on the command line.
  - MULTIPLE to indicate that the option can occur any number of times on the command line.
  - If both ONCE and MULTIPLE are omitted then switches that also have HAS\_PARAMETER are only allowed once but switches without HAS\_PARAMETER can have multiple occurrences but only one is recorded in the output map.
5. an integer; a [routine\\_id](#). This function will be called when the option is located on the command line and before it updates the map.  
Use -1 if `cmd_parse` is not to invoke a function for this option.  
The user defined function must accept a single sequence parameter containing four values. If the function returns 1 then the command option does not update the map. You can use the predefined index values `OPT_IDX`, `OPT_CNT`, `OPT_VAL`, `OPT_REV` when referencing the function's parameter elements.
1. An index into the opts list.
  2. The number of times that the routine has been called by `cmd_parse` for this option
  3. The option's value as found on the command line
  4. 1 if the command line indicates that this option is to remove any earlier occurrences of it.

One special circumstance exists and that is an option group header. It should contain only two elements:

1. The header constant: `HEADER`
2. A sequence to display as the option group header

When assigning a value to the resulting map, the key is the long name if present, otherwise it uses the short name. For options, you must supply a short name, a long name or both.

If you want `cmd_parse` to call a user routine for the extra command line values, you need to specify an Option Record that has neither a short name or a long name, in which case only the `routine_id` field is used.

For more details on how the command line is being pre-parsed, see [command\\_line](#).

### Example 1:

```
-- simple usage

map args = cmd_parse({
  { "o", 0, "Output directory", { HAS_PARAMETER } },
  { "v", 0, "Verbose mode" }
})

if map:get(args, "v") then
  printf(1, "Output directory is %s\n", { map:get(args, "o") })
end if
```

## Example 2:

```
-- complex usage

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

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

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

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

map:map opts = cmd_parse(option_definition, NO_HELP)

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

## See Also:

[show\\_help](#) | [command\\_line](#)

## build\_commandline

```
include std/cmdline.e
namespace cmdline
public function build_commandline(sequence cmds)
```

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

### Arguments:

1. `cmds` : A sequence. Contains zero or more strings.

### Returns:

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

### Comments:

Typically, this is used to ensure that arguments on a command line are properly formed before submitting it to the shell.

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

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

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

### Example 1:

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

### Example 2:

You can use this to run things that might be difficult to quote out. Suppose you want to run a program that requires quotes on its command line? Use this function to pass quotation marks:

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

### See Also:

[parse\\_commandline](#) | [system](#) | [system\\_exec](#) | [command\\_line](#) | [canonical\\_path](#) | [TO\\_SHORT](#)

## parse\_commandline

```
include std/cmdline.e
namespace cmdline
public function parse_commandline(sequence cmdline)
```

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

### Arguments:

1. `cmdline` : Command line sequence (string)

## Returns:

A **sequence**, of command line options

## Example 1:

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

## See Also:

[build\\_commandline](#)

# Console

## Information

### has\_console

```
include std/console.e
namespace console
public function has_console()
```

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

#### Returns:

An **atom**,

- 1 one (*true*) if there is more than one process attached to the current console,
- 0 zero (*false*) if a console does not exist or only one process (Euphoria) is attached to the current console.

#### Comments:

- On *unix* systems always returns 1 one (*true*).
- On *windows* client systems earlier than Windows XP the function always returns 0 zero (*false*).
- On *windows* server systems earlier than Windows Server 2003 the function always returns 0 zero (*false*).

#### Example 1:

```
include std/console.e

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

--> Hello Console!
```

#### See Also:

[free\\_console](#)

### key\_codes

```
include std/console.e
namespace console
public function key_codes(object codes = 0)
```

gets and sets the keyboard codes used internally by Euphoria.

#### Arguments:

1. **codes** : Either a sequence of exactly 256 integers or an atom (the default).

#### Returns:

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



## Comments:

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

## Example 1:

```

                                include std/console.e
                                sequence kc

kc = key_codes() -- Get existing set.
? kc[KC_LEFT]
--> 1016384

kc[KC_LEFT] = 263 -- Change the code for the left-arrow press.
key_codes(kc)     -- Set the new codes.
? kc[KC_LEFT]
--> 263

```

## Key Code Names

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

KC_LBUTTON	#01 + 1,	Left mouse button
KC_RBUTTON	#02 + 1,	Right mouse button
KC_CANCEL	#03 + 1,	Control-break processing
KC_MBUTTON	#04 + 1,	Middle mouse button (three-button mouse)
KC_XBUTTON1	#05 + 1,	Windows 2000/XP: X1 mouse button
KC_XBUTTON2	#06 + 1,	Windows 2000/XP: X2 mouse button
KC_BACK	#08 + 1,	BACKSPACE key
KC_TAB	#09 + 1,	TAB key
KC_CLEAR	#0C + 1,	CLEAR key NUMPAD-5
KC_RETURN	#0D + 1,	ENTER key NUMPAD-ENTER
KC_SHIFT	#10 + 1,	SHIFT key
KC_CONTROL	#11 + 1,	Control key
KC_MENU	#12 + 1,	ALT key
KC_PAUSE	#13 + 1,	PAUSE key
KC_CAPITAL	#14 + 1,	CAPS LOCK key
KC_KANA	#15 + 1,	Input Method Editor (IME) Kana mode
KC_JUNJA	#17 + 1,	IME Junja mode
KC_FINAL	#18 + 1,	IME final mode
KC_HANJA	#19 + 1,	IME Hanja mode
KC_ESCAPE	#1B + 1,	ESC key

KC_CONVERT	#1C + 1,	IME convert
KC_NONCONVERT	#1D + 1,	IME nonconvert
KC_ACCEPT	#1E + 1,	IME accept
KC_MODECHANGE	#1F + 1,	IME mode change request
KC_SPACE	#20 + 1,	SPACEBAR
KC_PRIOR	#21 + 1,	PAGE UP key
KC_NEXT	#22 + 1,	PAGE DOWN key
KC_END	#23 + 1,	END key
KC_HOME	#24 + 1,	HOME key
KC_LEFT	#25 + 1,	LEFT ARROW key
KC_UP	#26 + 1,	UP ARROW key
KC_RIGHT	#27 + 1,	RIGHT ARROW key
KC_DOWN	#28 + 1,	DOWN ARROW key
KC_SELECT	#29 + 1,	SELECT key
KC_PRINT	#2A + 1,	PRINT key
KC_EXECUTE	#2B + 1,	EXECUTE key
KC_SNAPSHOT	#2C + 1,	PRINT SCREEN key
KC_INSERT	#2D + 1,	INS key
KC_DELETE	#2E + 1,	DEL key
KC_HELP	#2F + 1,	HELP key
KC_LWIN	#5B + 1,	Left Windows key (Microsoft Natural keyboard)
KC_RWIN	#5C + 1,	Right Windows key (Natural keyboard)
KC_APPS	#5D + 1,	Applications key (Natural keyboard)
KC_SLEEP	#5F + 1,	Computer Sleep key
KC_NUMPAD0	#60 + 1,	Numeric keypad 0 key
KC_NUMPAD1	#61 + 1,	Numeric keypad 1 key
KC_NUMPAD2	#62 + 1,	Numeric keypad 2 key
KC_NUMPAD3	#63 + 1,	Numeric keypad 3 key
KC_NUMPAD4	#64 + 1,	Numeric keypad 4 key
KC_NUMPAD5	#65 + 1,	Numeric keypad 5 key
KC_NUMPAD6	#66 + 1,	Numeric keypad 6 key

KC_NUMPAD7	#67 + 1,	Numeric keypad 7 key
KC_NUMPAD8	#68 + 1,	Numeric keypad 8 key
KC_NUMPAD9	#69 + 1,	Numeric keypad 9 key
KC_MULTIPLY	#6A + 1,	Multiply key NUMPAD
KC_ADD	#6B + 1,	Add key NUMPAD
KC_SEPARATOR	#6C + 1,	Separator key
KC_SUBTRACT	#6D + 1,	Subtract key NUMPAD
KC_DECIMAL	#6E + 1,	Decimal key NUMPAD
KC_DIVIDE	#6F + 1,	Divide key NUMPAD
KC_F1	#70 + 1,	F1 key
KC_F2	#71 + 1,	F2 key
KC_F3	#72 + 1,	F3 key
KC_F4	#73 + 1,	F4 key
KC_F5	#74 + 1,	F5 key
KC_F6	#75 + 1,	F6 key
KC_F7	#76 + 1,	F7 key
KC_F8	#77 + 1,	F8 key
KC_F9	#78 + 1,	F9 key
KC_F10	#79 + 1,	F10 key
KC_F11	#7A + 1,	F11 key
KC_F12	#7B + 1,	F12 key
KC_F13	#7C + 1,	F13 key
KC_F14	#7D + 1,	F14 key
KC_F15	#7E + 1,	F15 key
KC_F16	#7F + 1,	F16 key
KC_F17	#80 + 1,	F17 key
KC_F18	#81 + 1,	F18 key
KC_F19	#82 + 1,	F19 key
KC_F20	#83 + 1,	F20 key
KC_F21	#84 + 1,	F21 key
KC_F22	#85 + 1,	F22 key

KC_F23	#86 + 1,	F23 key
KC_F24	#87 + 1,	F24 key
KC_NUMLOCK	#90 + 1,	NUM LOCK key
KC_SCROLL	#91 + 1,	SCROLL LOCK key
KC_LSHIFT	#A0 + 1,	Left SHIFT key
KC_RSHIFT	#A1 + 1,	Right SHIFT key
KC_LCONTROL	#A2 + 1,	Left CONTROL key
KC_RCONTROL	#A3 + 1,	Right CONTROL key
KC_LMENU	#A4 + 1,	Left MENU key
KC_RMENU	#A5 + 1,	Right MENU key
KC_BROWSER_BACK	#A6 + 1,	Windows 2000/XP: Browser Back key
KC_BROWSER_FORWARD	#A7 + 1,	Windows 2000/XP: Browser Forward key
KC_BROWSER_REFRESH	#A8 + 1,	Windows 2000/XP: Browser Refresh key
KC_BROWSER_STOP	#A9 + 1,	Windows 2000/XP: Browser Stop key
KC_BROWSER_SEARCH	#AA + 1,	Windows 2000/XP: Browser Search key
KC_BROWSER_FAVORITES	#AB + 1,	Windows 2000/XP: Browser Favorites key
KC_BROWSER_HOME	#AC + 1,	Windows 2000/XP: Browser Start and Home key
KC_VOLUME_MUTE	#AD + 1,	Windows 2000/XP: Volume Mute key
KC_VOLUME_DOWN	#AE + 1,	Windows 2000/XP: Volume Down key
KC_VOLUME_UP	#AF + 1,	Windows 2000/XP: Volume Up key
KC_MEDIA_NEXT_TRACK	#B0 + 1,	Windows 2000/XP: Next Track key
KC_MEDIA_PREV_TRACK	#B1 + 1,	Windows 2000/XP: Previous Track key
KC_MEDIA_STOP	#B2 + 1,	Windows 2000/XP: Stop Media key
KC_MEDIA_PLAY_PAUSE	#B3 + 1,	Windows 2000/XP: Play/Pause Media key
KC_LAUNCH_MAIL	#B4 + 1,	Windows 2000/XP: Start Mail key
KC_LAUNCH_MEDIA_SELECT	#B5 + 1,	Windows 2000/XP: Select Media key
KC_LAUNCH_APP1	#B6 + 1,	Windows 2000/XP: Start Application 1 key
KC_LAUNCH_APP2	#B7 + 1,	Windows 2000/XP: Start Application 2 key
KC_OEM_1	#BA + 1,	Used for miscellaneous characters; it can vary by keyboard. Windows 2000/XP: For the US standard keyboard, the ';' key
KC_OEM_PLUS	#BB + 1,	Windows 2000/XP: For any country/region, the '+' key

KC_OEM_COMMA	#BC + 1,	Windows 2000/XP: For any country/region, the ',' key	
KC_OEM_MINUS	#BD + 1,	Windows 2000/XP: For any country/region, the '-' key	
KC_OEM_PERIOD	#BE + 1,	Windows 2000/XP: For any country/region, the '.' key	
KC_OEM_2	#BF + 1,	Used for miscellaneous characters; it can vary by keyboard. Windows 2000/XP: For the US standard keyboard, the '/'-' key	
KC_OEM_3	#C0 + 1,	Used for miscellaneous characters; it can vary by keyboard. Windows 2000/XP: For the US standard keyboard, the '`~' key	
KC_OEM_4	#DB + 1,	Used for miscellaneous characters; it can vary by keyboard. Windows 2000/XP: For the US standard keyboard, the '[' key	
KC_OEM_5	#DC + 1,	Used for miscellaneous characters; it can vary by keyboard. Windows 2000/XP: For the US standard keyboard, the '\' key	,
KC_OEM_6	#DD + 1,	Used for miscellaneous characters; it can vary by keyboard. Windows 2000/XP: For the US standard keyboard, the ']' key	
KC_OEM_7	#DE + 1,	Used for miscellaneous characters; it can vary by keyboard. Windows 2000/XP: For the US standard keyboard, the 'single-quote/double-quote' key	
KC_OEM_8	#DF + 1,	Used for miscellaneous characters; it can vary by keyboard.	
KC_OEM_102	#E2 + 1,	Windows 2000/XP: Either the angle bracket key or the backslash key on the RT 102-key keyboard	
KC_PROCESSKEY	#E5 + 1,	Windows 95/98/Me, Windows NT 4.0, Windows 2000/XP: IME PROCESS key	
KC_PACKET	#E7 + 1,	Windows 2000/XP: Used to pass Unicode characters as if they were keystrokes. The KC_PACKET key is the low word of a 32-bit Virtual Key value used for non-keyboard input methods. For more information, see Remark in KEYBDINPUT, SendInput, WM_KEYDOWN, and WM_KEYUP	
KC_ATTN	#F6 + 1,	Attn key	
KC_CRSEL	#F7 + 1,	CrSel key	
KC_EXSEL	#F8 + 1,	ExSel key	
KC_EREOF	#F9 + 1,	Erase EOF key	
KC_PLAY	#FA + 1,	Play key	
KC_ZOOM	#FB + 1,	Zoom key	
KC_NONAME	#FC + 1,	Reserved	
KC_PA1	#FD + 1,	PA1 key	
KC_OEM_CLEAR	#FE + 1,	Clear key	
KM_CONTROL	#1000,	Ctrl modifier	
KM_SHIFT	#2000,	Shift modifier	
KM_ALT	#4000,	Alt modifier	

See Also:

[key\\_codes](#)

## KC\_LBUTTON

```
include std/console.e
namespace console
public constant KC_LBUTTON
```

### See Also:

[Key Code Names](#)

## set\_keycodes

```
include std/console.e
namespace console
public function set_keycodes(object kcfile)
```

changes the default codes returned by the keyboard.

### Arguments:

1. kcfile : Either the name of a text file or the handle of an opened (for reading) text file.

### Returns:

An **integer**,

- 0 means no error.
- -1 means that the supplied file could not be loaded in to `amap`.
- -2 means that a new key value was not an integer.
- -3 means that an unknown key name was found in the file.

### Comments:

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

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

### Example 1:

A text file containing keycode information:

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

Test program to set keycodes:

```
include std/console.e
sequence kc

kc = key_codes() -- get existing codes
? kc[KC_F1]
--> 1017600

? set_keycodes( "mykeycodes.txt" )
--> 0
? kc[KC_F1]
--> 1017600
```

### See Also:

## Windows Cursor Style

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

### Note:

*Windows* only.

### See Also:

[cursor](#)

#### NO\_CURSOR

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

#### UNDERLINE\_CURSOR

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

#### THICK\_UNDERLINE\_CURSOR

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

#### HALF\_BLOCK\_CURSOR

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

#### BLOCK\_CURSOR

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

## Keyboard Related Routines

#### get\_key

```
<built-in> function get_key()
```

returns the key that was pressed by the user, without waiting. Special codes are returned for the function keys, arrow keys, and so on.

### Returns:

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

### Comments:

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

Run the `.../euphoria/demo/key.ex` program to see what key code is generated for each key on your keyboard.

### Example 1:

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

--> No key waiting.
```

### See Also:

`wait_key`

## allow\_break

```
include std/console.e
namespace console
public procedure allow_break(types :boolean b)
```

sets the behavior of Control+C and Control+Break keys.

### Arguments:

1. `b` : a boolean,

!=0 not zero (*true*) to enable the trapping of Control+C and Control+Break,

0 zero (*false*) to disable it.

### Comments:

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

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

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

### Example 1:

```
include std/console.e

-- do not allow user to kill program!
```



```
allow_break(0)

-- an endless loop
-- you cannot exit!
while 1 do
    ? wait_key()
end while
```

### See Also:

[check\\_break](#)

## check\_break

```
include std/console.e
namespace console
public function check_break()
```

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

### Returns:

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

### Comments:

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

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

### Example 1:

```
include std/console.e
integer k

allow_break(0)

-- an endless loop
while 1 do
    k = wait_key()
    ? k
    if check_break() then
        exit
    end if
end while

puts(1, "exited endless loop" )

? gets(0)
```

### Example 2:

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

```
end if
```

## See Also:

[allow\\_break](#)

## wait\_key

```
include std/console.e
namespace console
public function wait_key()
```

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

## Returns:

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

## Example 1:

```
include std/console.e
integer k

while 1 do
    k = wait_key()
    if k='q' then exit end if
end while

puts(1, "exited loop" )
```

## See Also:

[get\\_key](#) | [getc](#)

## any\_key

```
include std/console.e
namespace console
public procedure any_key(sequence prompt = "Press Any Key to continue...", integer con = 1)
```

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

## Arguments:

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

## Comments:

This wraps [wait\\_key](#) by giving a clue that the user should press a key, and perhaps do some other things as well.

## Example 1:

```
include std/console.e

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

## Example 2:

```
include std/console.e
```

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

### See Also:

[wait\\_key](#)

## maybe\_any\_key

```
include std/console.e
namespace console
public procedure maybe_any_key(sequence prompt = "Press Any Key to continue...",
    integer con = 1)
```

displays a prompt to the user and waits for any key. *Only* if the user is running under a GUI environment.

### Arguments:

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

### Comments:

This wraps [wait\\_key](#) by giving a clue that the user should press a key, and perhaps do some other things as well.

Requires Windows XP or later or Windows 2003 or later to work. Earlier versions of *Windows* or O/S will always pause even when not needed.

On *Unix* systems this will not pause even when needed.

### Example 1:

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

### Example 2:

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

### See Also:

[wait\\_key](#)

## prompt\_number

```
include std/console.e
namespace console
public function prompt_number(sequence prompt, sequence range)
```

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

### Arguments:

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

### Returns:

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

## Errors:

If `puts` cannot display `st` on standard output, or if the first or second item of `s` is a sequence, a runtime error will be raised.

If user tries cancelling the prompt by pressing Control+Z the program will abort as well, issuing a type check error.

## Comments:

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

If this routine is too simple for your needs then examine the source-code to make your own more specialized version.

## Example 1:

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

## Example 2:

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

## See Also:

[puts](#) | [prompt\\_string](#)

## prompt\_string

```
include std/console.e
namespace console
public function prompt_string(sequence prompt)
```

prompts the user to enter a string of text.

## Arguments:

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

## Returns:

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

## Comments:

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

## Example 1:

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

## See Also:

[prompt\\_number](#)

## Cross Platform Text Graphics

## See Also:

[Graphics - Cross Platform](#)

## positive\_int

```
include std/console.e
namespace console
public type positive_int(object x)
```

## clear\_screen

```
<built-in> procedure clear_screen()
```

clears the screen using the current background color.

### Comments:

The background color can be set by `bk_color` ).

### See Also:

`bk_color`

## get\_screen\_char

```
include std/console.e
namespace console
public function get_screen_char(positive_atom line, positive_atom column, integer fgbg = 0)
```

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

### Arguments:

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

### Returns:

- If fgbg is zero then a **sequence** of *two* elements, {character, attribute\_code} for the specified location.
- If fgbg is not zero then a **sequence** of *three* elements, {characterfg\_color, bg\_color}.

### Comments:

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

### Color Table

color number	name
0	black
1	dark blue
2	green
3	cyan
4	crimson

5	purple
6	brown
7	light gray
8	dark gray
9	blue
10	bright green
11	light blue
12	red
13	magenta
14	yellow
15	white

### Example 1:

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

### Example 2:

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

### See Also:

[put\\_screen\\_char](#) | [save\\_text\\_image](#)

## put\_screen\_char

```
include std/console.e
namespace console
public procedure put_screen_char(positive_atom line, positive_atom column, sequence char_attr)
```

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

### Arguments:

1. line : the 1-based line at which to start writing.
2. column : the 1-based column at which to start writing.
3. char\_attr : a sequence of alternated characters and attribute codes.

### Comments:

char\_attr must be in the form {character, attribute code, character, attribute code, ...}.

### Errors:

The length of char\_attr must be a multiple of two.

### Comments:

The attributes atom contains the foreground color, background color, and possibly other platform-dependent information controlling how the character is displayed on the screen. If char\_attr has 0 length, nothing will be written to the screen. The characters are written to the *active page*. It is faster to write several characters to the screen with a single call to put\_screen\_char than it is to write one character at a time.

### Example 1:

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

### See Also:

[get\\_screen\\_char](#) | [display\\_text\\_image](#)

## attr\_to\_colors

```
include std/console.e
namespace console
public function attr_to_colors(integer attr_code)
```

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

### Arguments:

1. `attr_code` : integer, an attribute code.

### Returns:

A **sequence**, of two elements -- {fgcolor, bgcolor}

### Example 1:

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

### See Also:

[get\\_screen\\_char](#) | [colors\\_to\\_attr](#)

## colors\_to\_attr

```
include std/console.e
namespace console
public function colors_to_attr(object fgbg, integer bg = 0)
```

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

### Arguments:

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

### Returns:

An **integer**, an attribute code.

### Example 1:

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

### See Also:

[get\\_screen\\_char](#) | [put\\_screen\\_char](#) | [attr\\_to\\_colors](#)

## display\_text\_image

```
include std/console.e
namespace console
public procedure display_text_image(text_point xy, sequence text)
```

displays a text image in any text mode.

### Arguments:

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

### Comments:

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

You might use [save\\_text\\_image](#) and [display\\_text\\_image](#) in a text-mode graphical user interface, to allow "pop-up" dialog boxes, and drop-down menus to appear and disappear without losing what was previously on the screen.

### Example 1:

```
clear_screen()
display_text_image({1,1}, {{ 'A', WHITE, 'B', GREEN},
                           { 'C', RED+16*WHITE},
                           { 'D', BLUE}})

-- displays:
--   AB
--   C
--   D
-- at the top left corner of the screen.
-- 'A' will be white with black (0) background color,
-- 'B' will be green on black,
-- 'C' will be red on white, and
-- 'D' will be blue on black.
```

### See Also:

[save\\_text\\_image](#) | [put\\_screen\\_char](#)

## save\_text\_image

```
include std/console.e
namespace console
public function save_text_image(text_point top_left, text_point bottom_right)
```

copies a rectangular block of text out of screen memory.

### Arguments:

1. top\_left : the coordinates, given as a pair, of the upper left corner of the area to save.
2. bottom\_right : the coordinates, given as a pair, of the lower right corner of the area to save.

### Returns:

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

### Comments:

The returned value is appropriately handled by [display\\_text\\_image](#).

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



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

### Example 1:

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

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

### See Also:

[display\\_text\\_image](#) | [get\\_screen\\_char](#)

## text\_rows

```
include std/console.e
namespace console
public function text_rows(positive_int rows)
```

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

### Arguments:

1. rows : an integer, the desired number of rows.

### Platform:

*Windows*

### Returns:

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

### Comments:

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

### See Also:

[graphics\\_mode](#) | [video\\_config](#)

## CURSOR

```
include std/console.e
namespace console
public procedure cursor(integer style)
```

selects a style of a *windows* cursor.

### Arguments:

1. style : an integer defining the cursor shape.

### Platform:

*Windows*

### Comments:

In pixel-graphics modes no cursor is displayed.

## Example 1:

```
cursor(BLOCK_CURSOR)
```

Cursor Type Constants:

- `NO_CURSOR`
- `UNDERLINE_CURSOR`
- `THICK_UNDERLINE_CURSOR`
- `HALF_BLOCK_CURSOR`
- `BLOCK_CURSOR`

## See Also:

`graphics_mode` | `text_rows`

## free\_console

```
include std/console.e
namespace console
public procedure free_console()
```

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

## Comments:

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

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

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

After freeing the console window, you can create a new console window by printing something to the screen, 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, and so on.

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

## See Also:

`clear_screen` `has_console`

## display

```
include std/console.e
namespace console
public procedure display(object data_in, object args = 1, integer finalnl = - 918_273_645)
```

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

### Arguments:

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

### Comments:

- If `data_in` is an atom or integer, it is simply displayed.
- If `data_in` is a simple text string, then `args` can be used to produce a formatted output with `data_in` providing the `text:format` string and `args` being a sequence containing the data to be formatted.
  - If the last character of `data_in` is an underscore character then it is stripped off and `finalnl` is set to zero. Thus ensuring that a new line is **not** output.
  - The formatting codes expected in `data_in` are the ones used by `text:format`. It is not mandatory to use formatting codes, and if `data_in` does not contain any then it is simply displayed and anything in `args` is ignored.
- If `data_in` is a sequence containing floating-point numbers, sub-sequences or integers that are not characters, then `data_in` is forwarded on to the `pretty_print` to display.
  - If `args` is a non-empty sequence, it is assumed to contain the `pretty_print` formatting options.
  - if `args` is an atom or an empty sequence, the assumed `pretty_print` formatting options are assumed to be {2}.

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

### Example 1:

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

Output would be:

```
Some plain text
Your answer:cat
Your answer:
Your answer:dog
{
"abc",
3.44554
}
The answer to 'why' was 42
""
```

```
{51'3',362,71'G'}
```

### See Also:

[pretty\\_print](#), [print](#), [puts](#)

# Date and Time

## Localized Variables

### month\_names

```
include std/datetime.e
namespace datetime
public sequence month_names
```

Month Names

### month\_abbrs

```
include std/datetime.e
namespace datetime
public sequence month_abbrs
```

Abbreviations of Month Names

### day\_names

```
include std/datetime.e
namespace datetime
public sequence day_names
```

Day Names

### day\_abbrs

```
include std/datetime.e
namespace datetime
public sequence day_abbrs
```

Abbreviations of Day Names

### ampm

```
include std/datetime.e
namespace datetime
public sequence ampm
```

AM and PM

## Date and Time Type Accessors

These accessors can be used with the `datetime` data-type.

## enum

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

YEAR | MONTH | DAY | HOUR | MINUTE | SECOND

## YEAR

```
include std/datetime.e
namespace datetime
YEAR
```

Year (full year, i.e. 2010, 1922,...)

## MONTH

```
include std/datetime.e
namespace datetime
MONTH
```

Month (1-12)

## DAY

```
include std/datetime.e
namespace datetime
DAY
```

Day (1-31)

## HOUR

```
include std/datetime.e
namespace datetime
HOUR
```

Hour (0-23)

## MINUTE

```
include std/datetime.e
namespace datetime
MINUTE
```

Minute (0-59)

## SECOND

```
include std/datetime.e
namespace datetime
SECOND
```

Second (0-59)

## Intervals

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

### enum

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

`YEARS` | `MONTHS` | `WEEKS` | `DAYS` | `HOURS` | `MINUTES` | `SECONDS` | `DATE`

### YEARS

```
include std/datetime.e
namespace datetime
YEARS
```

Years

### MONTHS

```
include std/datetime.e
namespace datetime
MONTHS
```

Months

### WEEKS

```
include std/datetime.e
namespace datetime
WEEKS
```

Weeks

### DAYS

```
include std/datetime.e
namespace datetime
DAYS
```

Days

## HOURS

```
include std/datetime.e
namespace datetime
HOURS
```

Hours

## MINUTES

```
include std/datetime.e
namespace datetime
MINUTES
```

Minutes

## SECONDS

```
include std/datetime.e
namespace datetime
SECONDS
```

Seconds

## DATE

```
include std/datetime.e
namespace datetime
DATE
```

Date

## Types

### datetime

```
include std/datetime.e
namespace datetime
public type datetime(object o)
```

datetime type

#### Arguments:

1. obj : any object, so no crash takes place.

#### Comments:

A datetime type consists of a sequence of length six in the form {year, month, day\_of\_month, hour, minute, second}. Checks are made to guarantee those values are in range.

#### Note:

All items must be integers except for seconds which could either integer or atom values.



## Subroutines

### time

```
<built-in> function time()
```

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

#### Returns:

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

#### Comments:

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

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

#### Example 1:

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

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

t0 = time()
for i = 1 to ITERATIONS do
  p = power(2, 20)
end for

-- calculates time (in seconds) for one call to power
? (time() - t0 - loop_overhead)/ITERATIONS
--> 3.725290298e-15
```

#### See Also:

`date` | `now`

### date

```
<built-in> function date()
```

returns a sequence with information on the current date.

#### Returns:

A **sequence** of eight items, laid out as follows:

1. year -- since 1900
2. month -- January = 1
3. day -- day of month, starting at 1
4. hour -- 0 to 23
5. minute -- 0 to 59
6. second -- 0 to 59

7. day of the week -- Sunday = 1
8. day of the year -- January 1st = 1

### Comments:

The value returned for the year is actually the number of years since 1900 (not the last two digits of the year). In the year 2000 this value was 100. In 2001 it was 101, and so on.

### Example 1:

```
sequence s

s = date()
? s
--> s is
-- {95, 3, 24, 23, 47,38, 6, 83 }
-- 1995, March, 24, 11pm, 47:38 minute:second, Friday, 83 day of the year
-- 1900+95 = 1995
```

### See Also:

[time](#) | [now](#)

## from\_date

```
include std/datetime.e
namespace datetime
public function from_date(sequence src)
```

converts a sequence formatted according to the built-in date function to a valid **datetime** sequence.

### Arguments:

1. **src** : a sequence which date might have returned

### Returns:

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

### Example 1:

```
include std/datetime.e
sequence s
datetime d

-- the current date and time

s = date()
? s
--> s is {114,11,25,11,7,40,3,329}

d = from_date(s)
? d
--> d is {2014,11,25,11,7,40}
```

### See Also:

[date](#) | [from\\_unix](#) | [now](#) | [new](#)

## now

```
include std/datetime.e
namespace datetime
public function now()
```

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

### Returns:

A **sequence**, more precisely a datetime data-type corresponding to the current moment in time.

### Example 1:

```
include std/datetime.e
datetime dt

dt = now()
? dt
--> dt is {2014,11,25,11,18,48}
-- the current date and time
```

### See Also:

[from\\_date](#) | [from\\_unix](#) | [new](#) | [new\\_time](#) | [now\\_gmt](#)

## now\_gmt

```
include std/datetime.e
namespace datetime
public function now_gmt()
```

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

### Comments:

This function will return a datetime that is GMT no matter what timezone the system is running under.

### Example 1:

```
include std/datetime.e
datetime dt

dt = now_gmt()
? dt
--> dt is {2014,11,25,16,44,46}

-- this is a "local" time for the Prime Meridian, at Greenwich England
```

### Example 2:

```
include std/datetime.e
datetime dnow
datetime dgmt

dnow = now()
? dnow
--> dnow is {2014,11,25,11,47,38}
-- local time for the timezone where Euphoria was first created

dgmt = now_gmt()
```

```
? dgmt
--> dgmt is {2014,11,25,16,47,38}
    -- local time for Prime Meridian, Greenwich England

? dnow - dgmt
--> {0,0,0,-5,0,0}
    -- Euphoria was first developed in the timezone 5 hours to the
    -- west of the Prime Meridian
```

## See Also:

[now](#)

## new

```
include std/datetime.e
namespace datetime
public function new(integer year = 0, integer month = 0, integer day = 0, integer hour = 0,
    integer minute = 0, atom second = 0)
```

creates a new datetime value.

## Arguments:

1. year -- the full year.
2. month -- the month (1-12).
3. day -- the day of the month (1-31).
4. hour -- the hour (0-23) (defaults to 0)
5. minute -- the minute (0-59) (defaults to 0)
6. second -- the second (0-59) (defaults to 0)

## Returns:

A **sequence**, datetime data-type.

- If not one argument is provided the return value is the same as returned by [thenow](#) function.
- If at least one argument is provided, then a datetime value with the input arguments (and 0 zero for unspecified arguments.)

## Example 1:

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

## See Also:

[from\\_date](#), [from\\_unix](#), [now](#), [new\\_time](#)

## new\_time

```
include std/datetime.e
namespace datetime
public function new_time(integer hour, integer minute, atom second)
```

creates a new datetime value with a date of zeros.

## Arguments:

1. hour : is the hour (0-23)

2. minute : is the minute (0-59)
3. second : is the second (0-59)

### Example 1:

```

                                include std/datetime.e
                                datetime dt

    dt = new_time(10, 30, 55)
    ? dt
--> dt is {0,0,0,10,30,35}
-- 10:30:55 AM

```

### See Also:

[from\\_date](#) | [from\\_unix](#) | [now](#) | [new](#)

## weeks\_day

```

include std/datetime.e
namespace datetime
public function weeks_day(datetime dt)

```

gets the day of week of the datetime dt.

### Arguments:

1. dt : a datetime to be queried.

### Returns:

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

### Example 1:

```

                                include std/datetime.e
                                datetime d
                                object day

    d = new(2008, 5, 2, 0, 0, 0)
    ? d
--> d is {2008,5,2,0,0,0}
-- 2008 May 2

    day = weeks_day(d)
    ? day
--> day is 6
-- a Friday

```

## years\_day

```

include std/datetime.e
namespace datetime
public function years_day(datetime dt)

```

gets the Julian day of year of the supplied date.

### Arguments:

1. dt : a datetime to be queried.

### Returns:

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

### Comments:

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

### Example 1:

```

                                include std/datetime.e
                                datetime d
                                integer day

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

```

## is\_leap\_year

```

include std/datetime.e
namespace datetime
public function is_leap_year(datetime dt)

```

determines if dt falls within leap year.

### Arguments:

1. dt : a datetime to be queried.

### Returns:

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

### Example 1:

```

                                include std/datetime.e
                                datetime d

    d = new(2008, 1, 1, 0, 0, 0)
    ? is_leap_year(d)
--> prints 1 (true)

    d = new(2005, 1, 1, 0, 0, 0)
    ? is_leap_year(d)
--> prints 0 (false)

```

### See Also:

[days\\_in\\_month](#)

## days\_in\_month

```

include std/datetime.e
namespace datetime
public function days_in_month(datetime dt)

```

returns the number of days in the month of dt.

### Comments:

This takes into account leap year.

### Arguments:

1. `dt` : a datetime to be queried.

### Example 1:

```

                                include std/datetime.e
                                datetime dt
                                dt = new(2008, 1, 1, 0, 0, 0) -- a regular month
? days_in_month(dt)
--> prints 31

                                dt = new(2008, 2, 1, 0, 0, 0) -- Feb in a leap year
? days_in_month(dt)
--> prints 29

```

### See Also:

[is\\_leap\\_year](#)

## days\_in\_year

```

include std/datetime.e
namespace datetime
public function days_in_year(datetime dt)

```

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

### Comments:

This takes into account leap year.

### Arguments:

1. `dt` : a datetime to be queried.

### Example 1:

```

                                include std/datetime.e
                                datetime dt
                                dt = new(2007, 1, 1, 0, 0, 0)
? days_in_year(dt)
--> 365
-- a regular year
                                dt = new(2008, 1, 1, 0, 0, 0)
? days_in_year(dt)
--> 366 a leap year

```

### See Also:

[is\\_leap\\_year](#) | [days\\_in\\_month](#)

## to\_unix

```

include std/datetime.e
namespace datetime
public function to_unix(datetime dt)

```

converts a datetime value to the *unix* numeric format (seconds since EPOCH\_1970).

## Arguments:

1. `dt` : a datetime to be queried.

## Returns:

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

## Example 1:

```

include std/datetime.e
atom secs_since_epoch

secs_since_epoch = to_unix(now())
? secs_since_epoch
--> secs_since_epoch is 1416998627
-- the current seconds since epoch

```

## See Also:

[from\\_unix](#) | [format](#)

## from\_unix

```

include std/datetime.e
namespace datetime
public function from_unix(atom unix)

```

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

## Arguments:

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

## Returns:

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

## Example 1:

```

include std/datetime.e
datetime dt

dt = from_unix(0)
? dt
--> dt is {1970,1,1,0,0,0}
-- 1970-01-01 00:00:00 date of the EPOCH
-- (zero seconds since EPOCH)

```

## See Also:

[to\\_unix](#) | [from\\_date](#) | [now](#) | [new](#)

## format

```

include std/datetime.e
namespace datetime
public function format(datetime d, sequence pattern = "%Y-%m-%d %H:%M:%S")

```

formats the date according to the format pattern string.

## Arguments:



1. `d` : a datetime which is to be printed out
2. `pattern` : a format string, similar to the onessprintf uses, but with some Unicode encoding.  
The default is "%Y-%m-%d %H:%M:%S".

Returns:

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

Comments:

Pattern string can include the following specifiers:

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

Example 1:

```
include std/datetime.e
datetime dt
sequence s

dt = new(2008, 5, 2, 12, 58, 32)
s = format(dt, "%Y-%m-%d %H:%M:%S" )
puts(1,s)
--> s is "2008-05-02 12:58:32"
```

Example 2:

```
include std/datetime.e
datetime dt
sequence s

dt = new(2008, 5, 2, 12, 58, 32 )
s = format(dt, "%A, %B %d '%y %H:%M%p")
puts(1,s)
--> s is "Friday, May 2 '08 12:58AM"
```

See Also:

[to\\_unix](#) | [parse](#)

## parse

```
include std/datetime.e
namespace datetime
public function parse(sequence val, sequence fmt = "%Y-%m-%d %H:%M:%S",
    integer yylower = - 80)
```

parses a datetime string according to the given format.

### Arguments:

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

### Returns:

A **datetime**, value.

### Comments:

Only a subset of the format specification is currently supported:

Format	Use	Comment
%d	day of month	(such as: 01)
%H	hour	(00..23)
%m	month	(01..12)
%M	minute	(00..59)
%S	second	(00..60)
%y	2-digit year	(YY)
%Y	4-digit year	(CCYY)

More format codes will be added in future versions.

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

All non-digits in the input string are ignored.

#### Parsing Two Digit Years:

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

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

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

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

YY	Diff	CCYY
18	-92/+8	2018
95	-15/+85	1995

```
33 -77/+23 2033
```

```
29 -81/+19 2029
```

### Note:

- Since 4.0.1 -- 2-digit year parsing and `yylower` parameter.

### Example 1:

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

### Example 2:

```
include std/datetime.e
datetime dt
dt = parse("05/01/44", "%m/%d/%y", -50) -- -50/+50 rule
? dt
--> dt is { 2044, 5, 14, 0, 0, 0 }
```

### See Also:

[format](#)

## add

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

adds a number of *intervals* to a datetime.

### Arguments:

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

### Returns:

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

### Comments:

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

Do not confuse the item access constants (such as `YEAR`, `MONTH`, `DAY` ) with the interval constants (`YEARS`, `MONTHS`, `DAYS` ).

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

When adding `YEARS` leap years are taken into account. Adding four `YEARS` to a date may result in a different day of month number due to leap year.

### Example 1:

```
include std/datetime.e
datetime d
datetime ds, dw, dy
```

```

d = {2014,11,26,11,16,29}

ds = add(d, 35, SECONDS) -- add 35 seconds to d
? ds
--> ds is      {2014,11,26,11,17,4}

dw = add(d, 7, WEEKS)    -- add 7 weeks to d
? dw
--> dw is      {2015,1,14,11,16,29}

dy = add(d, 19, YEARS)   -- add 19 years to d
? dy
--> dy is      {2033,11,26,11,16,29}

```

### See Also:

[subtract](#) | [diff](#)

## subtract

```

include std/datetime.e
namespace datetime
public function subtract(datetime dt, atom qty, integer interval)

```

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

### Arguments:

1. dt : the base datetime
2. qty : the number of *intervals* to subtract. It should be positive.
3. interval : which kind of interval to subtract.

### Returns:

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

### Comments:

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

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

### Example 1:

```

include std/datetime.e
datetime d
datetime dm, dM, dh

d = {2014,11,26,11,24,29}

dm = subtract(d, 18, MINUTES) -- subtract 18 minutes from d
? dm
--> dm is      {2014,11,26,11,6,29}

dM = subtract(d, 7, MONTHS)   -- subtract 7 months from d
? dM
--> dM is      {2014,4,26,11,24,29}

dh = subtract(d, 12, HOURS)   -- subtract 12 hours from d
? dh
--> dh is      {2014,11,25,23,24,29}

```

### See Also:

[add](#) | [diff](#)

## diff

```
include std/datetime.e
namespace datetime
public function diff(datetime dt1, datetime dt2)
```

computes the difference, in seconds, between two dates.

### Arguments:

1. dt1 : the end datetime
2. dt2 : the start datetime

### Returns:

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

### Comments:

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

### Example 1:

```
include std/datetime.e
include std/os.e
datetime d1, d2
integer i

d1 = now()
? d1
--> d1 is {2014,11,26,11,33,6}

sleep(15) -- sleep for 15 seconds

d2 = now()
? d2
--> d2 is {2014,11,26,11,33,21}

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

### See Also:

[add](#) | [subtract](#)

# File System

Cross platform file operations for Euphoria

## Constants

### Note:

To write a `'\'` backslash character in a string you must write `'\\'` an *escaped* backslash character.

### SLASH

```
public constant SLASH
```

platform's path separator character.

### Comments:

- *windows* is `\` backslash
- *unix* is `/` slash

### SLASHES

```
public constant SLASHES
```

all possible platform path separators.

### Comments:

- *windows* is `\` backslash, `:` colon, and `/` slash (where slash is recognized on new versions.)
- *unix* is `/` slash.

### EOLSEP

```
public constant EOLSEP
```

platform's line separator.

### Comments:

- *windows* is `"\r\n"` return newline {13,10}
- *unix* is `'\n'` newline 10

### EOL

```
public constant EOL
```

Euphoria newline character.

### Comments:

The '\n' newline character is the Euphoria line separator. The EOLSEP is replaced by EOL for *all* platforms when a file is read by Euphoria.

## PATHSEP

```
public constant PATHSEP
```

platform's path separator character.

### Comments:

- *windows* ; semi-colon
- *unix* : colon

## NULLDEVICE

```
public constant NULLDEVICE
```

platform's null device path.

### Comments:

- *windows* NUL:
- *unix* /dev/null

## SHARED\_LIB\_EXT

```
public constant SHARED_LIB_EXT
```

platform's shared library extension.

### Comments:

- *windows* dll
- *unix* so
- *osx* dylib

## Directory Handling

### enum

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

D\_NAME, D\_ATTRIBUTES, D\_SIZE, D\_YEAR, D\_MONTH, D\_DAY, D\_HOUR, D\_MINUTE,  
D\_SECOND, D\_MILLISECOND, D\_ALTNAME

## W\_BAD\_PATH

```
public constant W_BAD_PATH
```

Bad path error code. See [walk\\_dir](#)

## W\_SKIP\_DIRECTORY

```
public constant W_SKIP_DIRECTORY
```

### dir

```
include std/filesys.e
namespace filesys
public function dir(sequence name)
```

returns directory information for the specified file or directory.

#### Arguments:

1. `name` : a sequence, the name to be looked up in the file system.

#### Returns:

An **object**, -1 if no match found, else a sequence of sequence entries

#### Errors:

The length of `name` should not exceed 1\_024 characters.

#### Comments:

An **entry** is "a sequence of file attributes for one file or directory returned by the `dir` function." The returned information is similar to what you would get from the `DIR` command. A sequence is returned where each item is an entry (a sequence) that describes one file or subdirectory.

The argument `name` can also contain `*` and `?` wildcards to select multiple files.

- If `name` refers to a directory then you may have entries for `"."` and `".."`, just as with the `DIR` command.
- If `name` refers to a file and has no wildcards then the returned sequence will have just one entry (that is its length will be 1).
- If `name` contains wildcards then you may have multiple entries.

Each entry contains the name, attributes and file size as well as the time of the last modification.

You can index the items of an entry with the following constants:

#### File Attributes Index Comments

		<i>unix</i>	<i>windows</i>
D_NAME	1	filename	long filename
D_ATTRIBUTES	2		
D_SIZE	3		
D_YEAR	4		
D_MONTH	5		
D_DAY	6		
D_HOUR	7		
D_MINUTE	8		
D_SECOND	9		
D_MILLISECOND	10	0	
D_ALTNAME	11	0	short filename

The `D_ATTRIBUTES` item is a string sequence containing characters chosen from:



Attribute		Description
<i>unix</i>	<i>windows</i>	
""	""	a normal file
'd'	'd'	directory
	'r'	read only file
	'h'	hidden file
	's'	system file
	'v'	volume-id entry
	'a'	archive file
	'c'	compressed file
	'e'	encrypted file
	'N'	not indexed
	'D'	a device name
	'O'	offline
	'R'	reparse point or symbolic link
	'S'	sparse file
	'T'	temporary file
	'V'	virtual file

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

The top level directory ( therefore c:\ does not have "." or ".." entries).

This function is often used just to test if a file or directory exists.

Under *windows* the argument can have a long file or directory name anywhere in the path.

Under *unix* the only attribute currently available is 'd' and the milliseconds are always zero.

Under *windows*: The file name returned in [D\_NAME] will be a long file name. If [D\_ALTNAME] is not zero, it contains the 'short' name of the file.

### Example 1:

```
d = dir(current_dir())

-- console:display(d) could look like:
-- {
--   {".",      "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}
-- }

--console:display( d[3][D_NAME] )
-- would be "fred"
```

### See Also:

[walk\\_dir](#)

## current\_dir

```
include std/filesys.e
namespace filesys
public function current_dir()
```

returns the name of the current working directory.

### Returns:

A **sequence**, the name of the current working directory

### Comments:

Normally there will be no trailing / slash or \ backslash on the end of the current directory; except under *windows* at the top-level of a drive (such as C:\).

### Example 1:

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

### See Also:

[dir](#) | [chdir](#)

## chdir

```
include std/filesys.e
namespace filesys
public function chdir(sequence newdir)
```

sets a new value for the current directory.

### Arguments:

*newdir* : a sequence, the name for the new working directory.

### Returns:

An **integer**, 0 on failure, 1 on success.

### Comments:

After setting the current directory you can refer to files in that directory using just the file name.

The `current_dir` function will now return the name of the *new* current directory.

On *windows* the current directory is a public property shared by all the processes running under one shell.

On *unix* a subprocess can change the current directory for itself but this will not affect the current directory of its parent process.

### Example 1:

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

### See Also:

[current\\_dir](#) | [dir](#)

```
include std/filesys.e
namespace filesys
public integer my_dir
```

**Deprecated**, so therefore not documented.

## walk\_dir

```
include std/filesys.e
namespace filesys
public function walk_dir(sequence path_name, object your_function,
    integer scan_subdirs = types :FALSE,
    object dir_source = types :NO_ROUTINE_ID)
```

Generalized Directory Walker

### Arguments:

1. `path_name` : a sequence, the name of the directory to walk through
2. `your_function` : the routine id of a function that will receive each path returned from the result of `dir_source`, one at a time. Optionally, to include extra data for your function, `your_function` can be a 2 element sequence, with the `routine_id` as the first element and other data as the second element.
3. `scan_subdirs` : an optional integer, 1 to also walk though subfolders, 0 (the default) to skip them all.
4. `dir_source` : an optional integer. A routine id of a user-defined routine that returns the list of paths to pass to `your_function`. If omitted, the `dir()` function is used. If your routine requires an extra parameter, `dir_source` may be a 2 element sequence where the first element is the routine id and the second is the extra data to be passed as the second parameter to your function.

### Returns:

An **object**,

- 0 on success
- `W_BAD_PATH` if an error occurred
- the return value from the custom function when it stops `walk_dir`

### Comments:

This routine will *walk* through a directory named `path_name`. For each entry in the directory it will call a function whose `routine_id` is `your_function`. If `scan_subdirs` is !0 non-zero (true) then the subdirectories in `path_name` will be walked through recursively in the very same way.

The routine that you supply should accept two sequences, the *path name* and *dir* entry for each file and subdirectory. It should return 0 to keep going, `W_SKIP_DIRECTORY` to avoid scan the contents of the supplied path name (if a directory), or non-zero to stop `walk_dir`. Returning `W_BAD_PATH` is taken as denoting some error.

This mechanism allows you to write a simple function that handles one file at a time while `walk_dir` handles the process of walking through all the files and subdirectories.

By default, the files and subdirectories will be visited in alphabetical order. To use a different order, use the `dir_source` to pass the `routine_id` of your own modified `dir` function that sorts the directory entries differently.

The path that you supply to `walk_dir` must not contain wildcards (\* or ?). Only a single directory (and its subdirectories) can be searched at one time.

On *windows* systems all / slash characters in `path_name` are replaced with \ backslash characters.

All trailing slash and whitespace characters are removed from `path_name`.

### Example 1:

```
function look_at(sequence path_name, sequence item)
-- this function accepts two sequences as arguments
-- it displays all C/C++ source files and their sizes
  if find('d', item[D_ATTRIBUTES]) then
    -- Ignore directories
    if find('s', item[D_ATTRIBUTES]) then
      return W_SKIP_DIRECTORY -- Don't recurse a system directory
    else
      return 0 -- Keep processing as normal
    end if
  end if
  if not find(fileext(item[D_NAME]), {"c","h","cpp","hpp","cp"}) then
    return 0 -- ignore non-C/C++ files
  end if
  printf(STDOUT, "%s%s%s: %d\n",
    {path_name, {SLASH}, item[D_NAME], item[D_SIZE]})
  return 0 -- keep going
end function

function mysort(sequence path)
  object d

  d = dir(path)
  if atom(d) then
    return d
  end if
  -- Sort in descending file size.
  return sort_columns(d, {-D_SIZE})
end function

exit_code = walk_dir("C:\\MYFILES\\", routine_id("look_at"), TRUE,
  routine_id("mysort"))
```

### See Also:

[dir](#) | [sort](#) | [sort\\_columns](#)

## create\_directory

```
include std/filesys.e
namespace filesys
public function create_directory(sequence name, integer mode = 448, integer mkparent = 1)
```

creates a new directory.

### Arguments:

1. `name` : a sequence, the name of the new directory to create
2. `mode` :
  - \* on *unix* systems permissions for the new directory. Default is 448 (all rights for owner, none for others).
  - \* on *windows* ignored
3. `mkparent` : If 1 (*true*) the parent directories are also created if needed.

### Returns:

An **integer**, 0 (*false*) on failure, 1 (*true*) on success.

### Comments:

The argument `mode` is ignored on *windows* platforms.

### Example 1:

```
include std/error.e
include std/filesys.e

-- creates a directory if it does not exist

if not create_directory("the_new_folder") then
    error:crash("Filesystem problem - could not create the new folder")
end if
```

### Example 2:

```
-- Creates parent "myapp/" and child "myapp/interface/"
-- if they don't exist.

if not create_directory("myapp/interface/letters") then
    error:crash("Filesystem problem - could not create the new folder")
end if
```

### Example 3:

```
-- This example will NOT create "myapp/" and "myapp/interface/"
-- if they don't exist.

if not create_directory("myapp/interface/letters",,0) then
    error:crash("Filesystem problem - could not create the new folder")
end if
```

### See Also:

[remove\\_directory](#) | [chdir](#)

## create\_file

```
include std/filesys.e
namespace filesys
public function create_file(sequence name)
```

creates a new file.

### Arguments:

1. name : a sequence, the name of the new file to create

### Returns:

An **integer**, 0 (*false*) on failure, 1 (*true*) on success.

### Comments:

If name exists it will be over-written with a new empty file.

The created file will be empty; it has a length of zero.

The created file will not be open when this function returns.

### Example 1:

```
if not create_file("the_new_file") then
    crash("Filesystem problem - could not create the new file")
end if
```

## See Also:

[create\\_directory](#) | [delete\\_file](#)

## delete\_file

```
include std/filesys.e
namespace filesys
public function delete_file(sequence name)
```

deletes a file.

## Arguments:

1. name : a sequence, the name of the file to delete.

## Returns:

An **integer**, 0 (*false*) on failure, 1 (*true*) on success.

## See Also:

[create\\_file](#) | [create\\_directory](#) |

## curdir

```
include std/filesys.e
namespace filesys
public function curdir(integer drive_id = 0)
```

returns the current directory or current directory for a selected drive on *windows*.

## Arguments:

1. drive\_id : The drive letter for the current directory. The current drive is the default.

## Returns:

A **sequence**, the current directory.

## Comments:

On a *windows* platform the argument drive selects a drive. A *windows* platform maintains a current directory for each disk drive. You would use this routine if you wanted the current directory for a drive that may not be the current drive.

On a *unix* platform the argument is ignored; a *unix* platform has only one current directory at any time.

## Note:

This function always returns a string with a trailing SLASH character.

## Example 1:

```
res = curdir('D') -- Find the current directory on the D: drive.
-- res might be "D:\backup\music\"

res = curdir()    -- Find the current directory on the current drive.
-- res might be "C:\myapp\work\"
```

## See Also:

[current\\_dir](#)

### init\_curdir

```
include std/filesys.e
namespace filesys
public function init_curdir()
```

returns the original current directory.

## Arguments:

1. None.

## Returns:

A **sequence**, the current directory at the time the program started running.

## Comments:

You would use this if the program might change the current directory during its processing and you wanted to return to the original directory.

## Note:

This always ensures that the returned value has a trailing SLASH character.

## Example 1:

```
res = init_curdir() -- Find the original current directory.
```

## See Also:

[current\\_dir](#) | [curdir](#) | [chdir](#)

### clear\_directory

```
include std/filesys.e
namespace filesys
public function clear_directory(sequence path, integer recurse = 1)
```

clears directory (and possibly subdirectories) by deleting files while keeping the directory structure.

## Arguments:

1. name : a sequence, the name of the directory whose files you want to deleted.
2. recurse : an integer, whether or not to delete files in the directory's sub-directories; Defaults to 1 .
  - \* If 0 then this function is identical to `remove_directory`.
  - \* If 1 then recursively clear subdirectories.

## Returns:

An **integer**, 0 (*false*) on failure, otherwise the number of files removed plus 1 one (*true*).

## Comments:

This never removes a directory. It only deletes files. It is used to clear a directory

structure of all existing files leaving the structure intact.

### Example 1:

```
integer cnt = clear_directory("the_old_folder")
if cnt = 0 then
    crash("Filesystem problem - could not remove one or more of the files.")
end if
printf(1, "Number of files removed: %d\n", cnt - 1)
```

### See Also:

[remove\\_directory](#) | [delete\\_file](#)

## remove\_directory

```
include std/filesys.e
namespace filesys
public function remove_directory(sequence dir_name, integer force = 0)
```

removes a directory.

### Arguments:

1. name : a sequence, the name of the directory to remove.
2. force : an integer, default 0
  - \* if 0 then only removes the directory if it is already empty.
  - \* if 1 this will also remove files and sub-directories in the directory.

### Returns:

An **integer**, 0 (*false*) on failure, 1 (*true*) on success.

### Example 1:

```
if not remove_directory("the_old_folder") then
    crash("Filesystem problem - could not remove the old folder")
end if
```

### See Also:

[create\\_directory](#) | [chdir](#) | [clear\\_directory](#)

## File Name Parsing

### enum

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

PATH\_DIR, PATH\_FILENAME, PATH\_BASENAME, PATH\_FILEEXT, PATH\_DRIVEID

### pathinfo

```
include std/filesys.e
namespace filesys
public function pathinfo(sequence path, integer std_slash = 0)
```



parses a fully qualified pathname.

### Arguments:

1. `path` : a sequence, the path to parse

### Returns:

A **sequence**, of length five. Each of these items is a string:

- The path name. For *windows* this excludes the drive id.
- The full unqualified file name
- the file name, without extension
- the file extension
- the drive id

### Comments:

The host operating system path separator is used in the parsing.

### Example 1:

```
include std/filesys.e
include std/console.e

sequence info = pathinfo("C:\\euphoria\\docs\\readme.txt")
display( info )

-- on windows info is
-- {
--  "C:\\euphoria\\docs",
--  "readme.txt",
--  "readme",
--  "txt",
--  "C"
-- }

-- on unix info is
-- {
--  "",
--  "C:\\euphoria\\docs\\readme.txt",
--  "C:\\euphoria\\docs\\readme",
--  "txt",
--  ""
-- }
```

### Example 2:

```
include std/filesys.e
include std/console.e

sequence info = pathinfo("/opt/euphoria/docs/readme.txt")
display( info )

-- on windows and unix info is
-- {
--  "/opt/euphoria/docs",
--  "readme.txt",
--  "readme",
--  "txt",
--  ""
-- }
```

### Example 3:

```
-- no extension
info = pathinfo("/opt/euphoria/docs/readme")
```

```
-- on windows and unix info is
-- info is {"/opt/euphoria/docs", "readme", "readme", "", ""}
```

### See Also:

[driveid](#) | [dirname](#) | [filename](#) | [fileext](#) | [PATH\\_BASENAME](#) | [PATH\\_DIR](#) | [PATH\\_DRIVEID](#) | [PATH\\_FILEEXT](#) | [PATH\\_FILENAME](#)

## dirname

```
include std/filesys.e
namespace filesys
public function dirname(sequence path, integer pcd = 0)
```

returns the directory name of a fully qualified filename.

### Arguments:

1. path : the path from which to extract information
2. pcd : If not zero and there is no directory name in path then "." is returned. The default (0) will just return any directory name in path.

### Returns:

A **sequence**, the full file name part of path.

### Comments:

The host operating system path separator is used.

### Example 1:

```
fname = dirname("/opt/euphoria/docs/readme.txt")
-- fname is "/opt/euphoria/docs"
```

### See Also:

[driveid](#) | [filename](#) | [pathinfo](#)

## pathname

```
include std/filesys.e
namespace filesys
public function pathname(sequence path)
```

returns the directory name of a fully qualified filename.

### Arguments:

1. path : the path from which to extract information
2. pcd : If not zero and there is no directory name in path then "." is returned. The default (0) will just return any directory name in path.

### Returns:

A **sequence**, the full file name part of path.

### Comments:

The host operating system path separator is used.

### Example 1:

```
fname = dirname("/opt/euphoria/docs/readme.txt")
-- fname is "/opt/euphoria/docs"
```

### See Also:

[driveid](#) | [filename](#) | [pathinfo](#)

## filename

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

returns the file name portion of a fully qualified filename.

### Arguments:

1. path : the path from which to extract information

### Returns:

A **sequence**, the file name part of path.

### Comments:

The host operating system path separator is used.

### Example 1:

```
sequence fname = filename("/opt/euphoria/docs/readme.txt")

-- on windows and unix
-- fname is "readme.txt"
```

### See Also:

[pathinfo](#) | [filebase](#) | [fileext](#)

## filebase

```
include std/filesys.e
namespace filesys
public function filebase(sequence path)
```

returns the base filename of path.

### Arguments:

1. path : the path from which to extract information

### Returns:

A **sequence**, the base file name part of path.

TODO: Test

### Example 1:

```
base = filebase("/opt/euphoria/readme.txt")
```

```
-- base is "readme"
```

## See Also:

[pathinfo](#) | [filename](#) | [fileext](#)

## fileext

```
include std/filesys.e
namespace filesys
public function fileext(sequence path)
```

returns the file extension of a fully qualified filename.

## Arguments:

1. path : the path from which to extract information

## Returns:

A **sequence**, the file extension part of path.

## Comments:

The host operating system path separator is used.

## Example 1:

```
fname = fileext("/opt/euphoria/docs/readme.txt")
-- fname is "txt"
```

## See Also:

[pathinfo](#) | [filename](#) | [filebase](#)

## driveid

```
include std/filesys.e
namespace filesys
public function driveid(sequence path)
```

returns the drive letter of the path on *windows* platforms.

## Arguments:

1. path : the path from which to extract information

## Returns:

A **sequence**, the file extension part of path.

TODO: Test

## Example 1:

```
sequence letter = driveid("C:\\EUPHORIA\\Readme.txt")
display( letter )

-- on windows letter is {C}
-- on unix letter is {}
```

## See Also:

[pathinfo](#) | [dirname](#) | [filename](#)

### defaulttext

```
include std/filesys.e
namespace filesys
public function defaulttext(sequence path, sequence defext)
```

returns the supplied filepath with the supplied extension, if the filepath does not have an extension already.

## Arguments:

1. path : the path to check for an extension.
2. defext : the extension to add if path does not have one.

## Returns:

A **sequence**, the path with an extension.

## Example 1:

```
-- ensure that the supplied path has an extension,
-- but if it doesn't use "tmp".
include std/filesys.e
include std/console.e

sequence UserFileName

UserFileName = "hello.txt"
display( defaulttext( UserFileName, "tmp" ) )
-- result is "hello.txt"

UserFileName = "hello"
display( defaulttext( UserFileName, "tmp" ) )
-- result is "hello.tmp"
```

## See Also:

[pathinfo](#)

### absolute\_path

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

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

## Arguments:

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

## Returns:

An **integer**, 0 (*false*) if filename is a relative path or 1 (*true*) otherwise.

## Comments:

A **relative** path is "one which is relative to the current directory." An **absolute** path is "one that does not need to know the current directory to find the file."

### Example 1:

```
include std/filesys.e
include std/console.e

-- on all platforms
? absolute_path("") -- returns 0
? absolute_path("/usr/bin/abc") -- returns 1
? absolute_path("../abc") -- returns 0
? absolute_path("local/abc.txt") -- returns 0
? absolute_path("abc.txt") -- returns 0
? absolute_path("c:..\abc") -- returns 0

? absolute_path("\\temp\\somefile.doc")
-- on unix returns 0
-- on windows returns 1
? absolute_path("c:\\windows\\system32\\abc")
-- on unix returns 0
-- on windows returns 1
? absolute_path("c:/windows/system32/abc")
-- on unix returns 0
-- on windows returns 1
```

## enum

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

AS\_IS | TO\_LOWER | CORRECT | TO\_SHORT

## case\_flagset\_type

```
include std/filesys.e
namespace filesys
public type case_flagset_type(integer x)
```

## canonical\_path

```
include std/filesys.e
namespace filesys
public function canonical_path(sequence path_in, integer directory_given = 0,
    case_flagset_type case_flags = AS_IS)
```

returns the full path and file name of the supplied file name.

### Arguments:

1. **path\_in** : A sequence. This is the file name whose full path you want.
2. **directory\_given** : An integer. This is zero if **path\_in** is to be interpreted as a file specification otherwise it is assumed to be a directory specification. The default is zero.
3. **case\_flags** : An integer. This is a combination of flags.  
 AS\_IS = Includes no flags  
 TO\_LOWER = If passed will convert the part of the path not affected by other case flags to lowercase.  
 CORRECT = If passed will correct the parts of the filepath that exist in the current

filesystem in parts of the filesystem that is case insensitive. This should work on *Windows* or SMB mounted volumes on *Unix* and all OS X filesystems.

TO\_LOWER = If passed alone the entire path is converted to lowercase.

or\_bits(TO\_LOWER,CORRECT) = If these flags are passed together the the part that exists has the case of that of the filesystem. The part that does not is converted to lower case.

TO\_SHORT = If passed the elements of the path that exist are also converted to their *Windows* short names if available.

### Returns:

A **sequence**, the full path and file name.

### Comments:

- The supplied file/directory does not have to actually exist.
- path\_in can be enclosed in quotes, which will be stripped off.
- If path\_in begins with a tilde '~' then that is replaced by the contents of \$HOME in *Unix* platforms and %HOMEDRIVE%%HOMEPATH% in *Windows*.
- In *Windows* all '/' characters are replaced by '\' characters.
- Does not (yet) handle UNC paths or *Unix* links.

### Example 1:

```
include std/console.e
include std/filesys.e

display( curdir() )
-- Assuming the current directory is
-- "/usr/foo/bar"

sequence res = canonical_path("../abc.def")
display( res )

-- res is now
-- "/usr/foo/abc.def"
```

### Example 2:

```
-- res is "C:\Program Files" on systems that have that directory.
res = canonical_path("c:\pRoGrAm FiLeS", CORRECT)
-- on Windows Vista this would be "c:\Program Files" for Vista uses lowercase for its drives.
```

## abbreviate\_path

```
include std/filesys.e
namespace filesys
public function abbreviate_path(sequence orig_path, sequence base_paths = {})
```

returns a path string to the supplied file which is shorter than the given path string.

### Arguments:

1. orig\_path : A sequence. This is the path to a file.
2. base\_paths : A sequence. This is an optional list of paths that may prefix the original path. The default is an empty list.

### Returns:

A **sequence**, an equivalent path to orig\_path which is shorter than the supplied path. If a shorter one cannot be formed, then the original path is returned.

### Comments:

- This function is primarily used to get the shortest form of a file path for output to a file or screen.
- It works by first trying to find if the `orig_path` begins with any of the `base_paths`. If so it returns the parameter minus the base path prefix.
- Next it checks if the `orig_path` begins with the current directory path. If so it returns the parameter minus the current directory path.
- Next it checks if it can form a relative path from the current directory to the supplied file which is shorter than the parameter string.
- Failing all of that, it returns the original parameter.
- In *Windows* the shorter result has all `'` characters are replaced by `\` characters.
- The supplied path does not have to actually exist.
- `orig_path` can be enclosed in quotes, which will be stripped off.
- If `orig_path` begins with a tilde `~` then that is replaced by the contents of `$HOME` in *Unix* platforms and `%HOMEDRIVE%%HOMEPATH%` in *Windows*.

### Example 1:

```
include std/console.e
include std/filesys.e

sequence res

display( curdir() )
    -- Assuming the current directory is "/home/mint17/foo"

res = abbreviate_path("/home/mint17/foo/abc.def")
display(res)
    -- res is now "abc.def"

res = abbreviate_path("/home/mint17/foo/inc/abc.def")
display(res)
    -- res is now "inc/abc.def"
```

## split\_path

```
include std/filesys.e
namespace filesys
public function split_path(sequence fname)
```

splits a filename into individual string items.

### Arguments:

- `fname` -- filename to split

### Returns:

A **sequence**, of strings representing each path item found in `fname`.

### Example 1:

```
sequence path_items = split_path("/usr/home/john/hello.txt")
display( path_items )
    -- path_items would be { "usr", "home", "john", "hello.txt" }
```

### Versioning:

- Added in 4.0.1

### See Also:

[join\\_path](#)



## join\_path

```
include std/filesys.e
namespace filesys
public function join_path(sequence path_elements)
```

joins individual path items into a complete string filename.

### Arguments:

- path\_elements -- sequence of path item strings.

### Returns:

A **sequence**, a string representing the path itmes formatted to the given platform.

### Example 1:

```
include std/console.e
include std/filesys.e

sequence fname = join_path({ "usr", "home", "john", "hello.txt" })
display( fname )

-- on unix fname is "/usr/home/john/hello.txt"
-- on windows fname is "\\usr\\home\\john\\hello.txt"
```

### Versioning:

- Added in 4.0.1

### See Also:

[split\\_path](#)

## File Types

### enum

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

FILETYPE\_UNDEFINED | FILETYPE\_NOT\_FOUND | FILETYPE\_FILE |  
FILETYPE\_DIRECTORY

### file\_type

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

gets the type of a file.

### Arguments:

1. filename : the name of the file to query. It must not have wildcards.

### Returns:

An **integer**,

- FILETYPE\_UNDEFINED (-1)  
if file could be multiply defined (for example contains any wildcards - '\*' or '?')
- FILETYPE\_NOT\_FOUND (0)  
if filename does not exist
- FILETYPE\_FILE (1)  
if filename is a file
- FILETYPE\_DIRECTORY  
(2) if filename is a directory

### Example 1:

```
include std/filesys.e

system( `ls` )
  -- sample directory contains three items
  -- ex1.ex  ex.err  foo

? file_type( "ex.*" )
  --> is -1
? file_type( "Ex1.ex" )
  --> is 0
? file_type( "ex1.ex" )
  --> is 1
? file_type( "foo" )
  --> is 2
```

### See Also:

[dir](#) | [FILETYPE\\_DIRECTORY](#) | [FILETYPE\\_FILE](#) | [FILETYPE\\_NOT\\_FOUND](#) | [FILETYPE\\_UNDEFINED](#)

## File Handling

### enum

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

[SECTORS\\_PER\\_CLUSTER](#) | [BYTES\\_PER\\_SECTOR](#) | [NUMBER\\_OF\\_FREE\\_CLUSTERS](#) | [TOTAL\\_NUMBER\\_OF\\_CLUSTERS](#)

### enum

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

[TOTAL\\_BYTES](#) | [FREE\\_BYTES](#) | [USED\\_BYTES](#)

### enum

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

[COUNT\\_DIRS](#) | [COUNT\\_FILES](#) | [COUNT\\_SIZE](#) | [COUNT\\_TYPES](#)

## enum

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

EXT\_NAME | EXT\_COUNT | EXT\_SIZE

## file\_exists

```
include std/filesys.e
namespace filesys
public function file_exists(object name)
```

checks to see if a file exists.

### Arguments:

1. name : filename to check existence of

### Returns:

An **integer**, 1 (*true*) on yes, 0 (*false*) on no.

### Example 1:

```
if file_exists("abc.e") then
  puts(1, "abc.e exists already\n")
end if
```

### See Also:

[file\\_type](#)

## file\_timestamp

```
include std/filesys.e
namespace filesys
public function file_timestamp(sequence fname)
```

gets the timestamp of the file.

### Arguments:

1. name : the filename to get the date of

### Returns:

A **sequence**, a valid `datetime` type representing the files date and time or -1 if the file's date and time could not be read.

### Example 1:

```
include std/filesys.e
? file_timestamp( "foo.txt" )
--> {2014,10,28,13,59,11}
```

### See Also:

[std/datetime.e](#)

## copy\_file

```
include std/filesys.e
namespace filesys
public function copy_file(sequence src, sequence dest, integer overwrite = 0)
```

copies a file.

### Arguments:

1. `src` : a sequence, the name of the file or directory to copy
2. `dest` : a sequence, the new name or location of the file
3. `overwrite` : an integer; 0 (the default) will prevent an existing destination file from being overwritten. Non-zero will overwrite the destination file.

### Returns:

An **integer**, 0 (*false*) on failure, 1 (*true*) on success.

### Comments:

If `overwrite` is true, and if `dest` file already exists, the function overwrites the existing file and succeeds.

### Example 1:

```
include std/filesys.e

-- assuming "ex1.ex" exists

? copy_file( "ex1.ex", "example.ex" )

-- returns 1 the first time run
-- returns 0 for subsequent runs
```

### See Also:

[move\\_file](#) | [rename\\_file](#)

## rename\_file

```
include std/filesys.e
namespace filesys
public function rename_file(sequence old_name, sequence new_name, integer overwrite = 0)
```

rename a file.

### Arguments:

1. `old_name` : a sequence, the name of the file or directory to rename.
2. `new_name` : a sequence, the new name for the renamed file
3. `overwrite` : an integer, 0 (the default) to prevent renaming if destination file exists, 1 to delete existing destination file first

### Returns:

An **integer**, 0 on failure, 1 on success.

### Comments:

- If `new_name` contains a path specification, this is equivalent to moving the file, as well as possibly changing its name. However, the path must be on the same drive for this to work.

- If overwrite was requested but the rename fails, any existing destination file is preserved.

### Example 1:

```
include std/filesys.e

-- assume "foo.ex" exists

? rename_file( "foo.ex", "boo.txt" )
--> 1 first time run
--> 0 second time run
```

### See Also:

[move\\_file](#) | [copy\\_file](#)

## move\_file

```
include std/filesys.e
namespace filesys
public function move_file(sequence src, sequence dest, integer overwrite = 0)
```

moves a file to another location.

### Arguments:

1. `src` : a sequence, the name of the file or directory to move
2. `dest` : a sequence, the new location for the file
3. `overwrite` : an integer, 0 (the default) to prevent overwriting an existing destination file, 1 to overwrite existing destination file

### Returns:

An **integer**, 0 (*false*) on failure, 1 (*true*) on success.

### Comments:

If overwrite was requested but the move fails, any existing destination file is preserved.

### Example 1:

```
include std/filesys.e
-- assume directory "foo" exists and file "boo.txt" exists

? move_file( "boo.txt", "foo/boo.txt" )
--> returns 1 first run
--> returns 0 second run
```

### See Also:

[rename\\_file](#) | [copy\\_file](#)

## file\_length

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

returns the size of a file.

### Arguments:

1. filename : the name of the queried file

### Returns:

An **atom**, \* the file size in bytes  
 \* 0 for a directory  
 \* -1 if file is not found.

### Comments:

This function does not compute the total size for a directory; it returns 0 zero bytes for directory size.

### Example 1:

```
include std/filesys.e
? file_length( "example1.ex" )
--> the integer 48
--
```

See Also: [dir](#)

## locate\_file

```
include std/filesys.e
namespace filesys
public function locate_file(sequence filename, sequence search_list = {},
    sequence subdir = {})
```

locates a file by searching a set of directories.

### Arguments:

1. filename : a sequence, the name of the file to search for.
2. search\_list : a sequence, the list of directories to look in. By default this is "", meaning that a predefined set of directories is scanned. See comments below.
3. subdir : a sequence, the sub directory within the search directories to check. This is optional.

### Returns:

A **sequence**, the located file path if found, else the original file name.

### Comments:

If filename is an absolute path, it is just returned and no searching takes place.

If filename is located, the full path of the file is returned.

If search\_list is supplied, it can be either a sequence of directory names, of a string of directory names delimited by ':' in *unix* and ';' in *windows*.

If the search\_list is omitted or "", this will look in the following places:

- The current directory
- The directory that the program is run from.
- The directory in \$HOME (\$HOMEDRIVE & \$HOMEPATH in *Windows*)
- The parent directory of the current directory
- The directories returned by include\_paths
- \$EUDIR/bin
- \$EUDIR/docs
- \$EUDIST/
- \$EUDIST/etc
- \$EUDIST/data

- The directories listed in \$USERPATH
- The directories listed in \$PATH

If the subdir is supplied, the function looks in this sub directory for each of the directories in the search list.

### Example 1:

```
res = locate_file("abc.def", {"/usr/bin", "/u2/someapp", "/etc"})
res = locate_file("abc.def", "/usr/bin:/u2/someapp:/etc")
res = locate_file("abc.def")
    -- Scan default locations.
res = locate_file("abc.def", , "app")
    -- Scan the 'app' sub directory in the default locations.
```

### Example 2:

```
include std/filesys.e
include std/console.e

display( locate_file( "PROJECTS.tar.gz", "/media/500B" ) )
    -- file NOT located since path is incorrect
    -- returns "PROJECTS.tar.gz"

display( locate_file( "PROJECTS.tar.gz", "/media/mint17/500B" ) )
    -- file IS located
    -- returns "/media/mint17/500B/PROJECTS.tar.gz"
```

## disk\_metrics

```
include std/filesys.e
namespace filesys
public function disk_metrics(object disk_path)
```

returns some information about a disk drive.

### Arguments:

1. disk\_path : A sequence. This is the path that identifies the disk to inquire upon.

### Returns:

A **sequence**, containing

- \* SECTORS\_PER\_CLUSTER,
- \* BYTES\_PER\_SECTOR,
- \* NUMBER\_OF\_FREE\_CLUSTERS,
- \* TOTAL\_NUMBER\_OF\_CLUSTERS

### Example 1:

```
res = disk_metrics("C:\\")
min_file_size = res[SECTORS_PER_CLUSTER] * res[BYTES_PER_SECTOR]
```

### Example 2:

```
include std/console.e
include std/filesys.e

display( disk_metrics( "/media/mint17/500B" ) )
    --> for example {8,512,1711777,120147634}

display( disk_metrics( "/" ) )
    --> for example {8,512,2036789,7665091}
```

## disk\_size

```
include std/filesys.e
namespace filesys
public function disk_size(object disk_path)
```

returns the amount of space for a disk drive.

### Arguments:

1. `disk_path` : A sequence. This is the path that identifies the disk to inquire upon.

### Returns:

A **sequence**, containing

- \* `TOTAL_BYTES`,
- \* `USED_BYTES`,
- \* `FREE_BYTES`,
- \* a string which represents the filesystem name

### Example 1:

```
res = disk_size("C:\\")
printf(1, "Drive %s has %3.2f%% free space\n", {
    "C:", res[FREE_BYTES] / res[TOTAL_BYTES]
})
```

### Example 2:

```
include std/filesys.e
include std/console.e

display( disk_size( "/" ) )

-- {
-- 3.139621274e+10,
-- 8342589440,
-- 2.143516262e+10,
-- "/dev/sda4"
-- }
```

### Example 3:

```
display( disk_size( `c:\` ) )

-- on unix
-- df: 'c: ': No such file or directory
-- {0,0,0,""}

-- on windows
-- {
-- 1.362561556e+11,
-- 1.265119273e+11,
-- 9744228352,
-- "c:\""
-- }
```

## dir\_size

```
include std/filesys.e
namespace filesys
public function dir_size(sequence dir_path, integer count_all = 0)
```



returns the amount of space used by a directory.

### Arguments:

1. `dir_path` : A sequence. This is the path that identifies the directory to inquire upon.
2. `count_all` : An integer. Used by *Windows* systems. If zero (the default) it will not include *system* or *hidden* files in the count, otherwise they are included.

### Returns:

A **sequence**, containing four elements; the number of sub-directories [`COUNT_DIRS`], the number of files [`COUNT_FILES`], the total space used by the directory [`COUNT_SIZE`], and breakdown of the file contents by file extension [`COUNT_TYPES`].

### Comments:

- The total space used by the directory does not include space used by any sub-directories.
- The file breakdown is a sequence of three-element sub-sequences. Each sub-sequence contains the extension [`EXT_NAME`], the number of files of this extension [`EXT_COUNT`], and the space used by these files [`EXT_SIZE`]. The sub-sequences are presented in extension name order. On *Windows* the extensions are all in lowercase.

### Example 1:

```
include std/filesys.e
include std/console.e

sequence res = dir_size("/usr/local/bin")
printf(1, "Directory %s contains %d files\n", {
    "/usr/localbin", res[COUNT_FILES]
})
for i = 1 to length(res[COUNT_TYPES]) do
    printf(1, "Type: %s (%d files %d bytes)\n", {
        res[COUNT_TYPES][i][EXT_NAME],
        res[COUNT_TYPES][i][EXT_COUNT],
        res[COUNT_TYPES][i][EXT_SIZE]
    })
end for

-- sample display
-- Directory /usr/localbin contains 35 files
-- Type: (26 files 102211699 bytes)
-- Type: 1 (2 files 358883 bytes)
-- Type: dat (1 files 25328 bytes)
-- Type: ex (6 files 99991 bytes)
```

## temp\_file

```
include std/filesys.e
namespace filesys
public function temp_file(sequence temp_location = "", sequence temp_prefix = "",
    sequence temp_extn = "_T_", integer reserve_temp = 0)
```

returns a file name that can be used as a temporary file.

### Arguments:

1. `temp_location` : A sequence. A directory where the temporary file is expected to be created.
  - If omitted (the default) the 'temporary' directory will be used. The temporary directory is defined in the "TEMP" environment symbol, or failing that the "TMP" symbol and failing that "C:\TEMP\" is used on *Windows* systems and "/tmp/" is used on *Unix* systems.
  - If `temp_location` was supplied,

- If it is an existing file, that file's directory is used.
  - If it is an existing directory, it is used.
  - If it doesn't exist, the directory name portion is used.
- 2. **temp\_prefix** : A sequence: The is prepended to the start of the generated file name. The default is "" .
- 3. **temp\_extn** : A sequence: The is a file extension used in the generated file. The default is "\_T" .
- 4. **reserve\_temp** : An integer: If not zero an empty file is created using the generated name. The default is not to reserve (create) the file.

### Returns:

A **sequence**, A generated file name.

### Example 1:

```
-- no directory specified

display( temp_file() )
--> /tmp/714670._T

-- directory "/home/mint17/foo" exists

display( temp_file( "/home/mint17/foo" ) )
--> /home/mint17/foo/147310._T

display( temp_file( "/home/mint17/foo", "myapp", "tmp" ) )
--> /home/mint17/foo/myapp411172.tmp

-- directory "/home/mint17/boo" does NOT exit

display( temp_file( "/home/mint17/boo", "myapp", "tmp" ) )
--> /home/mint17/myapp053651.tmp
```

## checksum

```
include std/filesys.e
namespace filesys
public function checksum(sequence filename, integer size = 4, integer username = 0,
    integer return_text = 0)
```

returns a checksum value for the specified file.

### Arguments:

1. **filename** : A sequence. The name of the file whose checksum you want.
2. **size** : An integer. The number of atoms to return. Default is 4
3. **username**: An integer. If not zero then the actual text of filename will affect the resulting checksum. The default (0) will not use the name of the file.
4. **return\_text**: An integer. If not zero, the check sum is returned as a text string of hexadecimal digits otherwise (the default) the check sum is returned as a sequence of size atoms.

### Returns:

A **sequence** containing size atoms.

### Comments:

- The larger the size value, the more unique will the checksum be. For most files and uses, a single atom will be sufficient as this gives a 32-bit file signature. However, if you require better proof that the content of two files are different then use higher values for

- size. For example, size = 8 gives you 256 bits of file signature.
- If size is zero or negative, an empty sequence is returned.
  - All files of zero length will return the same checksum value when username is zero.

### Example 1:

```
include std/console.e
include std/filesys.e

-- Example values. The exact values depend on the contents of the file.

display( checksum("myfile") )    --> {23448, 239807, 79283749, 427370} -- default

display( checksum("myfile", 1) )  --> {92837498}
display( checksum("myfile", 2) )  --> {1238176, 87192873}
display( checksum("myfile", 2,,1)) --> "0012E480 05327529"
display( checksum("myfile", 4) )  --> {23448, 239807, 79283749, 427370}
```

# I/O

## Constants

### STDIN

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

Standard Input

### STDOUT

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

Standard Output

### STDERR

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

Standard Error

### SCREEN

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

Screen (Standard Out)

### EOF

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

End of file

## Read and Write Subroutines

### ?

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

displays an object using numbers and braces.

### Note:

This procedure does not use parenthesis to delimit the single argument. This is a unique shortcut in the Euphoria syntax.

### Comments:

The ? question mark shortcut is the same as `pretty_print( STDOUT, x, {} )` . An object or an expression is printed to the standard output with braces and indentation to show the structure.

### Example 1:

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

### See Also:

[print](#)

## print

```
<built-in> procedure print(integer fn, object x)
```

displays an object using numbers and braces.

### Comments:

All data objects are in *binary* format within computer hardware; something that is easy to forget. An output routine must convert these binary values into "text" to be human readable. The procedures `print` and `?` produce a "text" representation of an object that is output to a file or device. The text shows the *numerical form* of the object. If the object `x` is a sequence then it uses `{ , , }` braces and commas to show the structure.

### Arguments:

1. `fn` : an integer, the handle to a file or device to output to
2. `x` : the object to print

### Errors:

The target file or device must be open and able to be written to.

### Comments:

This is not used to write to *binary* files as it only outputs text.

### Example 1:

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

print(STDOUT, "65")
--> output is:  "65"

print(STDOUT, 65.1234)
--> output is:  "65.1234"

include std/io.e
```

```
puts (STDOUT, "ABC")
--> output is: "ABC"

puts (STDOUT, 65)
--> output is: "A"
-- ASCII-65 is 'A'

puts (STDOUT, 65.1234)
--> output is: "A"
-- converts to integer first, then outputs
```

## Example 2:

```
include std/io.e

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

## See Also:

? | puts

## printf

```
<built-in> procedure printf(integer fn, sequence format, object values)
```

outputs format as a string sequence with *format specifiers* replaced by corresponding formatted values from the values argument.

## Arguments:

1. fn : an integer, the handle to a file or device to output to
2. format : a string sequence that may contain format specifiers.
3. values : usually a list of values enclosed by{} . It should have as many values as format specifiers in the format string.

## Errors:

If there are fewer value items than format specifiers then a run time error will occur. Having more value items than format specifiers results in the extra values being quietly ignored.

The target file or device must be open.

## Comments:

The printf procedure follows standard formatting conventions as used by many programming languages.

A **format string**, the format argument, is a "string sequence that may contain embedded *format specifiers*."

A **format specifier** is "a string of characters starting with % percent sign, optional field descriptor, and ending in a specifier character."

```
integer_width := integer
float_width := integer.integer
width := integer_width | float_width
zeros_fill := 0
left_justify := -
positive_sign := +
field_descriptor = [zeros_fill][left_justify][positive_sign][width]
format_specifier := %[field_descriptor](d|e|f|g|x|o|s|%)
```

The **values argument** is "a list of items to be matched, in the order they are written, with format specifiers found in the format string."

It helps if you always start writing the values argument as an {} empty list. If you have just one item to match with one format specifier then write it as { item } rather than just item.

- Convenient but deceptive notation.

An single atom, **a**, will be automatically promoted to { a } to match one format specifier.

- A common misuse of the printf procedure.

A single string sequence *str* *will not be promoted* to { str } but will be viewed as a list of individual characters. If there is only one format specifier the result is only the first character is matched--not the entire string.

- This is result you probably want.

A single string sequence written as part of a list { str } will be one item that is matched, *completely*, with one format specifier.

This procedure outputs the format text to the output file *fn*, replacing format specifiers with the corresponding data from the values argument. Whenever a format specifier is found in format, the *n*-th item in values will be turned into a string according to the format specifier. The resulting string will the format specifier. This means that the first format specifier uses the first item in values, the second format specifier the second item, and so on.

You must have at least as many items in values as there are format specifiers in format. This means that if there is only one format specifier then values can be either an atom, integer or a non-empty sequence. And when there are more than one format specifier in format then values must be a sequence with a length that is greater than or equal to the number of format specifiers present.

This way, printf always takes exactly three arguments no matter how many values are to be printed.

The basic format specifiers are:

Specifier Displays Using Base				Formatted As	Example 126.01
%d	atom	decimal	integer		"126"
%e	atom	decimal	floating-point with exponential notation		"1.12601e+01"
%f	atom	decimal	floating-point number with a decimal point but no exponent		"126.01"
%g	atom	decimal	as a floating-point number using whichever format seems appropriate, given the magnitude of the number		"126.01"
%x	atom	hexadecimal	integer		"7E"
			Negative numbers are printed in two's complement, so -1 will print as FFFFFFFF		
%o	atom	octal	integer		"176"
%s	atom		as character		"~"
	sequence		as string		"~"
%%	literal		'%' character itself		"%"

Field widths can be added to the basic formats (for example: %5d, %8.2f, %10.4s). The number before the decimal point is the minimum field width to be used. The number after the decimal point is the precision to be used for numeric values.

If the field width is negative (for example %-5d) then the value will be left-justified within

the field. Normally it will be right-justified, even strings. If the field width starts with a leading 0 (for example %08d) then leading zeros will be supplied to fill up the field. If the field width starts with a '+' (for example %+7d ) then a plus sign will be printed for positive values.

Watch out for the following common mistake. The intention is to output all the characters in the third argument but actually only outputs the first character:

```
include std/io.e
sequence name="John Smith"
printf(STDOUT, "My name is %s", name )
--> My name is J
```

The output of this will be My name is J because each format specifier uses exactly *one* item from the values parameter. In this case we have only one specifier so it uses the first item in the values parameter, which is the character 'J'. To fix this situation, you must ensure that the first item in the values argument is the entire text string and not just a character, so you need code this instead:

```
include std/io.e
name="John Smith"
printf(STDOUT, "My name is %s", {name} )
--> My name is John Smith
```

Now, the third argument of printf is a one item sequence containing all the text to be formatted.

Also note that if there is only one format specifier then values can simply be an atom or integer.

### Note:

printf cannot use an item in values that contains nested sequences. Thus this is an error,

```
include std/io.e
sequence name = {"John", "Smith"}
printf(STDOUT, "%s", {name} )
```

because the item that is used from the values parameter contains two subsequences (strings in this case). To get the correct output you would need to do this instead,

```
include std/io.e
sequence name = {"John", "Smith"}
printf(STDOUT, "%s %s", {name[1], name[2]} )
```

### Example 1:

```
include std/io.e
atom rate
rate = 7.875
printf(STDOUT, "The interest rate is: %8.2f\n", rate)
--> output is: "The interest rate is:      7.88"
```

### Example 2:

```
include std/io.e
sequence name="John Smith"
integer score=97
printf(STDOUT, "%15s, %5d\n", {name, score})
--> output is: "      John Smith,    97"
```

### Example 3:

```
include std/io.e
```



```
printf(STDOUT, "%-10.4s $ %s", {"ABCDEFGHIJKLMNOP", "XXX"})
--> output is: "ABCD      $ XXX"
```

#### Example 4:

```
include std/io.e

printf(STDOUT, "%d %e %f %g", repeat(7.75, 4))
--> output is: "7 7.750000e+000 7.750000 7.75"
-- same value in four different formats
```

#### See Also:

[sprintf](#) | [sprint](#) | [print](#)

## puts

```
<built-in> procedure puts(integer fn, object text)
```

outputs text characters to a screen or file.

#### Arguments:

1. *fn* : an integer, the handle to an opened file or device
2. *text* : an object, either a single character or a sequence of characters.

#### Errors:

The target file or device must be open.

#### Comments:

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

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 zeros to the screen or to standard output. Your output might get truncated.

Remember that if the output file was opened in text mode, *windows* will change '\n' (10) to "\r\n" ({13,10}). Open the file in binary mode if this is not what you want.

#### Example 1:

```
puts(1, "Enter your first name: ")
--> output is: Enter your first name:
```

#### Example 2:

```
integer output = 1

puts(output, 'A')
--> output is: A
-- the single byte 65 is displayed as a character
```

#### Example 3:

```
sequence s

s = "Hello Euphoria"
puts(1, s )
```

```

--> output is: Hello Euphoria
-- argument is a flat string sequence

s = { "Hello", "Euphoria" }
puts(1, s)
--> error: sequence found inside character string
-- argument is a nested sequence

```

### Example 4:

A Linux terminal will display a UTF-8 encoded string sequence.

```

sequence s
s = "Hello ΣûρhφЯiÄ"
? s
--> s is {72, 101, 108, 108, 111, 32, 226,
--      136, 145, 197, 175, 207, 129, 209,
--      155, 209, 132, 208, 175, 196, 171,
--      196, 130}
puts(1,s)
--> s is: Hello ΣûρhφЯiÄ

```

### See Also:

[print](#)

## getc

```
<built-in> function getc(integer fn)
```

gets the next character (byte) from a file or device fn.

### Arguments:

1. fn : an integer, the handle of the file or device to read from.

### Returns:

An **integer**, the character read from the file, in the 0..255 range. If no character is left to read, **EOF** is returned instead.

### Errors:

The target file or device must be open.

### Comments:

File input using `getc` is buffered, that means `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" returning **EOF**.

### See Also:

[gets](#) | [get\\_key](#)

## gets

```
<built-in> function gets(integer fn)
```

gets a sequence of characters.

### Arguments:

1. `fn` : an integer, the handle of the file or device to read from.

### Returns:

An **object**, either `EOF` on end of file, or the next line of text from the file.

### Errors:

The file or device must be open.

### Comments:

This function gets the next sequence (one line, including '\n') of characters from a file or device. The characters will have values from 0 to 255 .

If the line had an end of line marker, a '\n' terminates the line. The last line of a file needs not have an end of line marker.

After reading a line of text from the keyboard, you should normally output a '\n' character, (for example `puts(1, '\n')` ), before printing something. Only on the last line of the screen does the operating system automatically scroll the screen and advance to the next line.

When your program reads from the keyboard, the user can type Control+Z, which the operating system treats as "end of file". `EOF` will be returned.

### Example 1:

```
sequence buffer
object line
integer fn

-- open a file for reading
fn = open("my_file.txt", "r")
if fn = -1 then
    puts(1, "Couldn't open my_file.txt\n")
    abort(1)
end if

-- read a text file into a sequence
buffer = {}
while 1 do
    line = gets(fn)
    if atom(line) then
        exit -- EOF is returned at end of file
    end if
    buffer = append(buffer, line)
end while

-- display file contents
for i=1 to length(buffer) do
    puts(1,buffer[i])
end for
```

### Example 2:

```
object line

-- read standard input (keyboard)
puts(1, "What is your name? \n")
line = gets(0)
-- line contains '\n' return character
```

```
puts(1, line & " is a nice name.\n")

line = line[1..$-1]
-- get rid of \n character at end
puts(1, line & " is a nice name.\n")
```

### See Also:

[getc](#) | [read\\_lines](#)

## get\_bytes

```
include std/io.e
namespace io
public function get_bytes(integer fn, integer n)
```

reads the next bytes from a file.

### Arguments:

1. `fn` : an integer, the handle to an open file to read from.
2. `n` : a positive integer, the number of bytes to read.

### Returns:

A **sequence**, of length at most `n`, made of the bytes that could be read from the file.

### Comments:

When `n > 0` and the function returns a sequence of length less than `n` you know you have reached the end of file. Eventually, an empty sequence will be returned.

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

### Example 1:

```
integer fn
fn = open("temp", "rb") -- an existing file

sequence whole_file
whole_file = {}

sequence chunk

while 1 do
  chunk = get_bytes(fn, 100) -- read 100 bytes at a time
  whole_file &= chunk        -- chunk might be empty, that's ok
  if length(chunk) < 100 then
    exit
  end if
end while

close(fn)
? length(whole_file) -- should match DIR size of "temp"
```

### See Also:

[getc](#) | [gets](#) | [get\\_integer32](#) | [get\\_dstring](#)

## get\_integer32

```
include std/io.e
namespace io
public function get_integer32(integer fh)
```

reads the next four bytes from a file and returns them as a single integer.

### Arguments:

1. fh : an integer, the handle to an open file to read from.

### Returns:

An **atom**, between -1 and  $\text{power}(2,32)-1$ , made of the bytes that could be read from the file. When an end of file is encountered, it returns -1.

### Comments:

- This function is normally used with files opened in binary mode,"rb".

### Example 1:

```
integer fn
fn = open("temp", "rb") -- an existing file

atom file_type_code
file_type_code = get_integer32(fn)
```

### See Also:

[getc](#) | [gets](#) | [get\\_bytes](#) | [get\\_dstring](#)

## get\_integer16

```
include std/io.e
namespace io
public function get_integer16(integer fh)
```

reads the next two bytes from a file and returns them as a single integer.

### Arguments:

1. fh : an integer, the handle to an open file to read from.

### Returns:

An **integer**, made of the bytes that could be read from the file. When an end of file is encountered, it returns -1.

### Comments:

- This function is normally used with files opened in binary mode,"rb".

### Example 1:

```
integer fn
fn = open("temp", "rb") -- an existing file

atom file_type_code
file_type_code = get_integer16(fn)
```

### See Also:

[getc](#) | [gets](#) | [get\\_bytes](#) | [get\\_dstring](#)

## put\_integer32

```
include std/io.e
namespace io
public procedure put_integer32(integer fh, atom val)
```

writes the supplied integer as four bytes to a file.

### Arguments:

1. fh : an integer, the handle to an open file to write to.
2. val : an integer

### Comments:

- This function is normally used with files opened in binary mode,"wb".

### Example 1:

```
integer fn
fn = open("temp", "wb")

put_integer32(fn, 1234)
```

### See Also:

[getc](#) | [gets](#) | [get\\_bytes](#) | [get\\_dstring](#)

## put\_integer16

```
include std/io.e
namespace io
public procedure put_integer16(integer fh, atom val)
```

writes the supplied integer as two bytes to a file.

### Arguments:

1. fh : an integer, the handle to an open file to write to.
2. val : an integer

### Comments:

- This function is normally used with files opened in binary mode,"wb".

### Example 1:

```
integer fn
fn = open("temp", "wb")

put_integer16(fn, 1234)
```

### See Also:

[getc](#) | [gets](#) | [get\\_bytes](#) | [get\\_dstring](#)

## get\_dstring

```
include std/io.e
namespace io
public function get_dstring(integer fh, integer delim = 0)
```

read a delimited byte string from an opened file.

### Arguments:

1. `fh` : an integer, the handle to an open file to read from.
2. `delim` : an integer, the delimiter that marks the end of a byte string. If omitted, a zero is assumed.

### Returns:

An **sequence**, made of the bytes that could be read from the file.

### Comments:

- If the end-of-file is found before the delimiter, the delimiter is appended to the returned string.

### Example 1:

```
integer fn
fn = open("temp", "rb") -- an existing file

sequence text
text = get_dstring(fn) -- Get a zero-delimited string
text = get_dstring(fn, '$') -- Get a '$'-delimited string
```

### See Also:

[getc](#) | [gets](#) | [get\\_bytes](#) | [get\\_integer32](#)

## Low Level File and Device Handling

### enum

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

`LOCK_SHARED` | `LOCK_EXCLUSIVE`

### file\_number

```
include std/io.e
namespace io
public type file_number(object f)
```

File number type

### file\_position

```
include std/io.e
namespace io
public type file_position(object p)
```

File position type

## lock\_type

```
include std/io.e
namespace io
public type lock_type(object t)
```

Lock Type

## byte\_range

```
include std/io.e
namespace io
public type byte_range(object r)
```

Byte Range Type

## open

```
<built-in> function open(sequence path, sequence mode, integer cleanup = 0)
```

opens a file or device, to get the file number.

### Arguments:

1. path : a string, the path to the file or device to open.
2. mode : a string, the mode being used o open the file.
3. cleanup : an integer, if 0 zero, then the file must be manually closed by the coder. If 1 one, then the file will be closed when either the file handle's references goes to 0 zero, or if called as a parameter to delete.

### Returns:

A small **integer**, -1 on failure, else 0 or more.

### Errors:

There is a limit on the number of files that can be simultaneously opened, currently fourty. After this limit is reached the next call to open will produce an error.

The length of path should not exceed 1\_024 characters.

### Comments:

Possible *windows* modes are:

Mode	Use
"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

Possible *unix* modes are:

Mode	Use	Redundant
------	-----	-----------



"r"	open file file for reading	"rb"
"w"	create file file for writing	"wb"
"u"	open file file for update (reading and writing)	"ub"
"a"	open text file for appending	"ab"

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 zero bytes. Output to a file opened for append will start at the end of file.

On *windows*, output to text files will have carriage-return characters automatically added before linefeed characters. On input, these carriage-return characters are removed. A Control+Z character (ASCII 26) will signal an immediate end of file.

I/O to binary files is not modified in any way. Any byte values from 0 to 255 can be read or written. On *unix*, all files are binary files, so "r" mode and "rb" mode are equivalent, as are "w" and "wb", "u" and "ub", and "a" and "ab".

Some typical devices that you can open on *windows* are:

- "CON" -- the console (screen)
- "AUX" -- the serial auxiliary port
- "COM1" -- serial port 1
- "COM2" -- serial port 2
- "PRN" -- the printer on the parallel port
- "NUL" -- a non-existent device that accepts and discards output

Close a file or device when done with it, flushing out any still-buffered characters prior.

*Windows* and *unix*: Long filenames are fully supported for reading and writing and creating.

*Windows*: Be careful not to use the special device names in a file name, even if you add an extension. For example: CON.TXT, CON.DAT, CON.JPG all refer to the CON device and *not* to a file.

### Example 1:

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

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

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

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

## close

```
<built-in> procedure close(atom fn)
```

closes a file or device and flushes out any still-buffered characters.

## Arguments:

1. `fn` : an integer, the handle to the file or device to query.

## Errors:

The target file or device must be open.

## Comments:

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

## seek

```
include std/io.e
namespace io
public function seek(file_number fn, file_position pos)
```

Seek (move) to any byte position in a file.

## Arguments:

1. `fn` : an integer, the handle to the file or device to seek
2. `pos` : an atom, either an absolute 0-based position or -1 to seek to end of file.

## Returns:

An **integer**, 0 on success, 1 on failure.

## Errors:

The target file or device must be open.

## Comments:

For each open file, there is a current byte position that is updated as a result of I/O operations on the file. The initial file position is 0 for files opened for read, write or update. The initial position is the end of file for files opened for append. It is possible to seek past the end of a file. If you seek past the end of the file, and write some data, undefined bytes will be inserted into the gap between the original end of file and your new data.

After seeking and reading (writing) a series of bytes, you may need to call `seek` explicitly before you switch to writing (reading) bytes, even though the file position should already be what you want.

This function is normally used with files opened in binary mode. In text mode, *windows* converts CR LF to LF on input, and LF to CR LF on output, which can cause great confusion when you are trying to count bytes because `seek` counts the *Windows* end of line sequences as two bytes, even if the file has been opened in text mode.

## Example 1:

```
include std/io.e

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

## See Also:

[get\\_bytes](#) | [puts](#) | [where](#)

## where

```
include std/io.e
namespace io
public function where(file_number fn)
```

retrieves the current file position for an opened file or device.

### Arguments:

1. `fn` : an integer, the handle to the file or device to query.

### Returns:

An **atom**, the current byte position in the file.

### Errors:

The target file or device must be open.

### Comments:

The file position is the place in the file where the next byte will be read from, or written to. It is updated by reads, writes and seeks on the file. This procedure always counts *Windows* end of line sequences (CR LF) as two bytes even when the file number has been opened in text mode.

## flush

```
include std/io.e
namespace io
public procedure flush(file_number fn)
```

forces writing any buffered data to an open file or device.

### Arguments:

1. `fn` : an integer, the handle to the file or device to close.

### Errors:

The target file or device must be open.

### Comments:

When you write data to a file, Euphoria normally stores the data in a memory buffer until a large enough chunk of data has accumulated. This large chunk can then be written to disk very efficiently. Sometimes you may want to force, or flush, all data out immediately, even if the memory buffer is not full. To do this you must call `flush(fn)`, where `fn` is the file number of a file open for writing or appending.

When a file is closed, (see [close](#)), all buffered data is flushed out. When a program terminates, all open files are flushed and closed automatically. Use `flush` when another process may need to see all of the data written so far, but you are not ready to close the file yet. `flush` is also used in crash routines, where files may not be closed in the cleanest possible way.

## Example 1:

```
f = open("file.log", "w")
puts(f, "Record#1\n")
puts(STDOUT, "Press Enter when ready\n")

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

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

## See Also:

[close](#) | [crash\\_routine](#)

## lock\_file

```
include std/io.e
namespace io
public function lock_file(file_number fn, lock_type t, byte_range r = {})
```

locks a file so access is restricted.

## Arguments:

1. *fn* : an integer, the handle to the file or device to (partially) lock.
2. *t* : an integer which defines the kind of lock to apply.
3. *r* : a sequence, defining a section of the file to be locked, or {} for the whole file (the default).

## Returns:

An **integer**, 0 on failure, 1 on success.

## Errors:

The target file or device must be open.

## Comments:

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.

There are two types of locks that you can request using the *t* parameter. 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 is ok for many processes to simultaneously have shared locks on the same file, but only one process can have an exclusive lock, and that can happen only when no other process has any kind of lock on the file. `io.e` contains the following declarations:

```
public enum
    LOCK_SHARED,
    LOCK_EXCLUSIVE
```

On */Windows* you can lock a specified portion of a file using the *r* parameter. *r* is a sequence of the form: {first\_byte, last\_byte}. It indicates the first byte and last byte in the file, that the lock applies to. Specify the empty sequence {}, if you want to lock the whole

file, or don't specify it at all, as this is the default. In the current release for *Unix*, locks always apply to the whole file, and you should use this default value.

`lock_file` does not wait for other processes to relinquish their locks. You may have to call it repeatedly, before the lock request is granted.

On *Unix*, these locks are called advisory locks, which means they are not 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 *Windows* locks are enforced by the operating system.

### Example 1:

```
include std/io.e
integer v
atom t
v = open("visitor_log", "a") -- open for append
t = time()
while not lock_file(v, LOCK_EXCLUSIVE, {}) do
  if time() > t + 60 then
    puts(STDOUT, "One minute already ... I can't wait forever!\n")
    abort(1)
  end if
  sleep(5) -- let other processes run
end while
puts(v, "Yet another visitor\n")
unlock_file(v, {})
close(v)
```

### See Also:

[unlock\\_file](#)

## unlock\_file

```
include std/io.e
namespace io
public procedure unlock_file(file_number fn, byte_range r = {})
```

unlock (a portion of) an open file.

### Arguments:

1. `fn` : an integer, the handle to the file or device to (partially) lock.
2. `r` : a sequence, defining a section of the file to be locked, or `{}` for the whole file (the default).

### Errors:

The target file or device must be open.

### Comments:

You must have previously locked the file using `lock_file`. On *Windows* you can unlock a range of bytes within a file by specifying the `r` as `{first_byte, last_byte}`. The same range of bytes must have been locked by a previous call to [lock\\_file](#). On *Unix* you can currently only lock or unlock an entire file. `r` should be `{}` when you want to unlock an entire file. On *Unix*, `r` must always be `{}`, which is the default.

You should unlock a file as soon as possible so other processes can use it.

Any files that you have locked, will automatically be unlocked when your program terminates.

See Also:

[lock\\_file](#)

## File Reading and Writing

### read\_lines

```
include std/io.e
namespace io
public function read_lines(object file)
```

reads the contents of a file as a sequence of lines.

#### Arguments:

file : an object, either a file path or the handle to an open file. If this is an empty string, STDIN (the console) is used.

#### Returns:

-1 on error or a **sequence**, made of lines from the file, as [gets](#) could read them.

#### Comments:

If file was a sequence, the file will be closed on completion. Otherwise, it will remain open, but at end of file.

#### Example 1:

```
data = read_lines("my_file.txt")
-- data contains the entire contents of ##my_file.txt##, 1 sequence per line:
-- {"Line 1", "Line 2", "Line 3"}
```

#### Example 2:

```
fh = open("my_file.txt", "r")
data = read_lines(fh)
close(fh)

-- data contains the entire contents of ##my_file.txt##, 1 sequence per line:
-- {"Line 1", "Line 2", "Line 3"}
```

#### See Also:

[gets](#) | [write\\_lines](#) | [read\\_file](#)

### process\_lines

```
include std/io.e
namespace io
public function process_lines(object file, integer proc, object user_data = 0)
```

processes the contents of a file, one line at a time.

#### Arguments:

1. file : an object. Either a file path or the handle to an open file. An empty string signifies STDIN -- the console keyboard.
2. proc : an integer. The routine\_id of a function that will process the line.

3. `user_data` : on object. This is passed untouched to `proc` for each line.

### Returns:

An object. If 0 then all the file was processed successfully. Anything else means that something went wrong and this is whatever value was returned by `proc`.

### Comments:

- The function `proc` must accept three parameters:
  - A sequence: The line to process. It will **not** contain an end-of-line character.
  - An integer: The line number.
  - An object : This is the `user_data` that was passed to `process_lines`.
- If file was a sequence, the file will be closed on completion. Otherwise, it will remain open, and be positioned where ever reading stopped.

### Example 1:

```
-- Format each supplied line according to the format pattern supplied as well.
function show(sequence aLine, integer line_no, object data)
  writeln( data[1], {line_no, aLine})
  if data[2] > 0 and line_no = data[2] then
    return 1
  else
    return 0
  end if
end function
-- Show the first 20 lines.
process_lines("sample.txt", routine_id("show"), {"[1z:4] : [2]", 20})
```

### See Also:

[gets](#) | [read\\_lines](#) | [read\\_file](#)

## write\_lines

```
include std/io.e
namespace io
public function write_lines(object file, sequence lines)
```

write a sequence of lines to a file.

### Arguments:

1. `file` : an object, either a file path or the handle to an open file.
2. `lines` : the sequence of lines to write

### Returns:

An **integer**, 1 on success, -1 on failure.

### Errors:

If [puts](#) cannot write some line of text, a runtime error will occur.

### Comments:

If file was a sequence, the file will be closed on completion. Otherwise, it will remain open, but at end of file.

Whatever integer the lines in `lines` holds will be truncated to its 8 lowest bits so as to fall in the 0..255 range.

### Example 1:

```
if write_lines("data.txt", {"This is important data", "Goodbye"}) != -1 then
    puts(STDERR, "Failed to write data\n")
end if
```

### See Also:

[read\\_lines](#) | [write\\_file](#) | [puts](#)

## append\_lines

```
include std/io.e
namespace io
public function append_lines(sequence file, sequence lines)
```

appends a sequence of lines to a file.

### Arguments:

1. file : an object, either a file path or the handle to an open file.
2. lines : the sequence of lines to write

### Returns:

An **integer**, 1 on success, -1 on failure.

### Errors:

If [puts](#) cannot write some line of text, a runtime error will occur.

### Comments:

file is opened, written to and then closed.

### Example 1:

```
if append_lines("data.txt", {"This is important data", "Goodbye"}) != -1 then
    puts(STDERR, "Failed to append data\n")
end if
```

### See Also:

[write\\_lines](#) | [puts](#)

## enum

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

## read\_file

```
include std/io.e
namespace io
public function read_file(object file, integer as_text = BINARY_MODE)
```

reads the contents of a file as a single sequence of bytes.



## Arguments:

1. `file` : an object, either a file path or the handle to an open file.
2. `as_text` : integer, `BINARY_MODE` (the default) assumes *binary mode* that causes every byte to be read in, and `TEXT_MODE` assumes *text mode* that ensures that lines end with just a Control+J (NewLine) character, and the first byte value of 26 (Control+Z) is interpreted as End-Of-File.

## Returns:

A **sequence**, holding the entire file.

### Comments

- When using `BINARY_MODE`, each byte in the file is returned as an element in the return sequence.
- When not using `BINARY_MODE`, the file will be interpreted as a text file. This means that all line endings will be transformed to a single 0x0A character and the first 0x1A character (Control+Z) will indicate the end of file (all data after this will not be returned to the caller.)

## Example 1:

```
data = read_file("my_file.txt")
-- data contains the entire contents of ##my_file.txt##
```

## Example 2:

```
fh = open("my_file.txt", "r")
data = read_file(fh)
close(fh)

-- data contains the entire contents of ##my_file.txt##
```

## See Also:

[write\\_file](#) | [read\\_lines](#)

## write\_file

```
include std/io.e
namespace io
public function write_file(object file, sequence data, integer as_text = BINARY_MODE)
```

write a sequence of bytes to a file.

## Arguments:

1. `file` : an object, either a file path or the handle to an open file.
2. `data` : the sequence of bytes to write
3. `as_text` : integer
  - `BINARY_MODE` (the default) assumes *binary mode* that causes every byte to be written out as is,
  - `TEXT_MODE` assumes *text mode* that causes a NewLine to be written out according to the operating system's end of line convention. On *Unix* this is Control+J and on *Windows* this is the pair {Ctrl-L, Ctrl-J}.
  - `UNIX_TEXT` ensures that lines are written out with *Unix* style line endings (Control+J).
  - `DOS_TEXT` ensures that lines are written out with *Windows* style line endings {Ctrl-L, Ctrl-J}.

## Returns:

An **integer**, 1 on success, -1 on failure.

### Errors:

If **puts** cannot write data, a runtime error will occur.

### Comments:

- When file is a file handle, the file is not closed after writing is finished. When file is a file name, it is opened, written to and then closed.
- Note that when writing the file in any of the text modes, the file is truncated at the first Control+Z character in the input data.

### Example 1:

```
if write_file("data.txt", "This is important data\nGoodbye") = -1 then
  puts(STDERR, "Failed to write data\n")
end if
```

### See Also:

[read\\_file](#) | [write\\_lines](#)

## writeln

```
include std/io.e
namespace io
public procedure writeln(object fm, object data = {}, object fn = 1, object data_not_string = 0)
```

writes formatted text to a file.

### Arguments:

There are two ways to pass arguments to this function:

1. Traditional way with first arg being a file handle.
  1. : integer, The file handle.
  2. : sequence, The format pattern.
  3. : object, The data that will be formatted.
  4. data\_not\_string: object, If not 0 then the data is not a string.  
By default this is 0 meaning that data could be a single string.
1. Alternative way with first argument being the format pattern.
  1. : sequence, Format pattern.
  2. : sequence, The data that will be formatted,
  3. : object, The file to receive the formatted output. Default is to the STDOUT device (console).
  4. data\_not\_string: object, If not 0 then the data is not a string.  
By default this is 0 meaning that data could be a single string.

### Comments:

- With the traditional arguments, the first argument must be an integer file handle.
- With the alternative arguments, the third argument can be a file name string, in which case it is opened for output, written to and then closed.
- With the alternative arguments, the third argument can be a two-element sequence containing a file name string and an output type ("a" for append, "w" for write), in which case it is opened accordingly, written to and then closed.
- With the alternative arguments, the third argument can be a file handle, in which case it is written to only
- The format pattern uses the formatting codes defined in [text:format](#).
- When the data to be formatted is a single text string, it does not have to be enclosed in braces,

## Example 1:

```
-- To console
writef("Today is [4], [u2:3] [3:02], [1:4].",
      {Year, MonthName, Day, DayName})
-- To "sample.txt"
writef("Today is [4], [u2:3] [3:02], [1:4].",
      {Year, MonthName, Day, DayName}, "sample.txt")
-- To "sample.dat"
integer dat = open("sample.dat", "w")
writef("Today is [4], [u2:3] [3:02], [1:4].",
      {Year, MonthName, Day, DayName}, dat)
-- Appended to "sample.log"
writef("Today is [4], [u2:3] [3:02], [1:4].",
      {Year, MonthName, Day, DayName}, {"sample.log", "a"})
-- Simple message to console
writef("A message")
-- Another console message
writef(STDERR, "This is a []", "message")
-- Outputs two numbers
writef(STDERR, "First [], second []", {65, 100}, 1)
-- Note that {65, 100} is also "Ad"
```

## See Also:

[text:format](#) | [writefln](#) | [write\\_lines](#)

## writefln

```
include std/io.e
namespace io
public procedure writefln(object fm, object data = {}, object fn = 1,
                        object data_not_string = 0)
```

writes formatted text to a file, ensuring that a new line is also output.

## Arguments:

1. fm : sequence, Format pattern.
2. data : sequence, The data that will be formatted,
3. fn : object, The file to receive the formatted output. Default is to the `STDOUT` device (console).
4. data\_not\_string: object, If not 0 then the data is not a string. By default this is 0 meaning that data could be a single string.

## Comments:

- This is the same as `writef`, except that it always adds a New Line to the output.
- When fn is a file name string, it is opened for output, written to and then closed.
- When fn is a two-element sequence containing a file name string and an output type ("a" for append, "w" for write), it is opened accordingly, written to and then closed.
- When fn is a file handle, it is written to only
- The fm uses the formatting codes defined in [text:format](#).

## Example 1:

```
-- To console
writefln("Today is [4], [u2:3] [3:02], [1:4].",
      {Year, MonthName, Day, DayName})
-- To "sample.txt"
writefln("Today is [4], [u2:3] [3:02], [1:4].",
      {Year, MonthName, Day, DayName}, "sample.txt")
-- Appended to "sample.log"
writefln("Today is [4], [u2:3] [3:02], [1:4].",
      {Year, MonthName, Day, DayName}, {"sample.log", "a"})
```

---

### See Also:

[text:format](#) | [writef](#) | [write\\_lines](#)

# Operating System Helpers

## CMD\_SWITCHES

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

## Operating System Constants

### enum

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

These constants are returned by the platform function.

The enums are self descriptive: WIN32 | WINDOWS | LINUX | FREEBSD | OSX | OPENBSD | NETBSD

#### Note:

In most situations you are better off to test the host platform by using the [ifdef statement](#). It is faster.

#### See Also:

[platform](#) | [ifdef statement](#)

## Environment

### instance

```
include std/os.e
namespace os
public function instance()
```

returns hInstance on *windows* and Process ID (pid) on *unix*.

#### Returns:

An **atom**

#### Comments:

Under *windows* the hInstance can be passed around to various *Windows* routines.

#### Example 1:

```
include std/os.e

? instance()
--> 3055
```

```
? gets(0) -- pause so you can examine the system monitor
```

### See Also:

[get\\_pid](#)

## get\_pid

```
include std/os.e
namespace os
public function get_pid()
```

returns the ID of the current Process (pid).

### Returns:

An **atom**, the current id for a process.

### Example 1:

```
include std/os.e

? get_pid()
--> 3284

? gets(0)
```

### See Also:

[instance](#)

## uname

```
include std/os.e
namespace os
public function uname()
```

retrieves the name of the host OS.

### Returns:

A **sequence**, starting with the OS name. If identification fails, returns an OS name of UNKNOWN. Extra information depends on the OS.

Under *unix* returns the same information as the `uname` syscall in the same order as the struct `utsname`. This information is:

- OS Name/Kernel Name
- Local Hostname
- Kernel Version/Kernel Release
- Kernel Specific Version information (This is usually the date that the kernel was compiled on and the name of the host that performed the compiling.)
- Architecture Name (Usually a string of i386 vs x86\_64 vs ARM vs etc)

Under *windows* returns the following in order:

- Windows Platform (out of WinCE, Win9x, WinNT, Win32s, or Unknown Windows)
- Name of Windows OS (Windows 3.1, Win95, WinXP, etc)
- Platform Number
- Build Number
- Minor OS version number

- Major OS version number }}}}

On UNKNOWN returns an OS name of "UNKNOWN". No other information is returned.

Returns an empty string of "" if an internal error has occurred.

### Comments:

Under *unix* M\_UNAME is defined as a machine\_func and this is passed to the C backend. If the M\_UNAME call fails, the raw machine\_func returns -1 minus one. On non-*unix* platforms, calling the machine\_func directly returns 0 zero.

### Example 1:

```

include std/os.e
include std/console.e

display( uname() )

/*
output under Linux
{
  "Linux",
  "eu",
  "3.13.0-24-generic",
  "#46-Ubuntu SMP Thu Apr 10 19:08:14 UTC 2014",
  "i686"
}

output under Windows7
{
  "WinNT",
  "Windows7",
  "Service Pack 1",
  2,
  7601,
  1,
  6
}
*/

```

## is\_win\_nt

```

include std/os.e
namespace os
public function is_win_nt()

```

reports whether the host system is a newer Windows version (NT/2K/XP/Vista).

### Returns:

An **integer**, 1 if host system is a newer Windows (NT/2K/XP/Vista), else 0.

### Example 1:

```

include std/os.e

? is_win_nt()

--> -1
    -- under Linux

--> 1
    -- under Windows7

```

## getenv

```
<built-in> function getenv(sequence var_name)
```

returns the value of an environment variable.

### Arguments:

1. `var_name` : a string, the name of the variable being queried.

### Returns:

An **object**, -1 if the variable does not exist, else a sequence holding its value.

### Comments:

Both the argument and the return value, may, or may not be, case sensitive. You might need to test this on your own system.

### Example 1:

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

### Example 2:

```
include std/os.e
include std/console.e

display( getenv( "EUDIR" ) )

--> -1
-- under Linux, Windows7
-- using eu.cfg and no EUDIR

? gets(0)
```

### See Also:

[setenv](#) | [command\\_line](#)

## setenv

```
include std/os.e
namespace os
public function setenv(sequence name, sequence val, integer overwrite = 1)
```

sets an environment variable.

### Arguments:

1. `name` : a string, the environment variable name
2. `val` : a string, the value to set to
3. `overwrite` : an integer, nonzero to overwrite an existing variable, 0 to disallow this.

### Example 1:

```
include std/os.e
include std/console.e

? setenv( "NAME", "John Doe" )
-->1

display( getenv( "NAME" ) )
```



```

--> "John Doe"

? setenv( "NAME", "Jane Doe" )
--> 1

display( getenv( "NAME" ) )
--> "Jane Doe"

? setenv( "NAME", "Jim Doe", 0 )
display( getenv( "NAME" ) )
--> "Jane Doe"

```

### See Also:

[getenv](#) | [unsetenv](#)

## unsetenv

```

include std/os.e
namespace os
public function unsetenv(sequence env)

```

unsets an environment variable.

### Arguments:

1. name : name of environment variable to unset

### Example 1:

```

include std/os.e
include std/console.e

display( setenv( "NAME", "Euphoria" )
         getenv( "NAME" ) )
--> Euphoria

display( unsetenv( "NAME" )
         getenv( "NAME" ) )
--> -1

```

### See Also:

[setenv](#) | [getenv](#)

## platform

```

<built-in> function platform()

```

indicates the platform that the program is being executed on.

### Returns:

An **integer**

### Comments:

Constants from the enum: WIN32 | WINDOWS | LINUX | FREEBSD | OSX | OPENBSD | NETBSD | FREEBSD

The `ifdef` statement is much more versatile and in most cases supersedes `platform`.

`platform` used to be the way to execute different code depending on which platform the program is running on. Additional platforms will be added as Euphoria is ported to new machines and operating environments.

### Example 1:

```

                                include std/os.e

? platform()

if platform() = WIN32 then
    puts(1, "This is Windows! \n" )
elseif platform() = LINUX then
    puts(1, "All is well \n" )
end if

ifdef WINDOWS then
    puts(1, "This is Windows! \n" )
elsedef
    -- Linux, FreeBSD
    puts(1, "All is well" )
end ifdef

```

### See Also:

[Platform-Specific Issues | ifdef statement](#)

## Interacting with the OS

### system

```
<built-in> procedure system(sequence command, integer mode=0)
```

passes a command string to the operating system command interpreter.

### Arguments:

1. `command` : a string to be passed to the shell
2. `mode` : an integer, indicating the manner in which to return from the call.

### Errors:

`command` should not exceed 1\_024 characters.

### Comments:

Allowable values for `mode` are:

#### Mode

0	the previous graphics mode is restored and the screen is cleared.
1	a beep sound will be made and the program will wait for the user to press a key before the previous graphics mode is restored.
2	the graphics mode is not restored and the screen is not cleared.

`mode = 2` should only be used when it is known that the command executed by `system` will not change the graphics mode.

You can use Euphoria as a sophisticated "batch" (.bat) language by making calls to `system` and `system_exec`.

`system` will start a new command shell.

system allows you to use command-line redirection of standard input and output in command.

### Example 1:

```
system("copy temp.txt a:\\temp.bak", 2)
-- note use of double backslash in literal string to get
-- single backslash
```

### Example 2:

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

### See Also:

[system\\_exec](#) | [command\\_line](#) | [current\\_dir](#) | [getenv](#)

## system\_exec

```
<built-in> function system_exec(sequence command, integer mode=0)
```

tries to run the a shell executable command.

### Arguments:

1. **command** : a string to be passed to the shell, representing an executable command
2. **mode** : an integer, indicating the manner in which to return from the call.

### Returns:

An **integer**, basically the exit or return code from the called process.

### Errors:

command should not exceed 1\_024 characters.

### Comments:

Allowable values for mode are:

#### Mode

0	The previous graphics mode is restored and the screen is cleared.
1	A beep sound will be made and the program will wait for the user to press a key before the previous graphics mode is restored.
2	The graphics mode is not restored and the screen is not cleared.

If it is not possible to run the program, system\_exec will return -1.

On *Windows* system\_exec will only run .exe and .com programs. To run .bat files, or built-in shell commands, you need [system](#). Some commands, such as DEL, are not programs, they are actually built-in to the command interpreter.

On *Windows* system\_exec does not allow the use of command-line redirection in command. Nor does it allow you to quote strings that contain blanks, such as file names.

exit codes from *Windows* programs are normally in the range 0 to 255, with 0 indicating "success".

You can run a Euphoria program using system\_exec. A Euphoria program can return an exit code using [abort](#).

`system_exec` does not start a new command shell.

### 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")
elseif exit_code = 0 then
  puts(2, "\n xcopy succeeded\n")
else
  printf(2, "\n xcopy failed with code %d\n", exit_code)
end if
```

### Example 2:

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

### See Also:

[system](#) | [abort](#)

## Miscellaneous

### sleep

```
include std/os.e
namespace os
public procedure sleep(atom t)
```

suspend thread execution for t seconds.

### Arguments:

1. t : an atom, the number of seconds for which to sleep.

### Comments:

The operating system will suspend your process and schedule other processes.

With multiple tasks, the whole program sleeps, not just the current task. To make just the current task sleep, you can

- Call `task_schedule(task_self(), {i, i})` and then execute `task_yield`.
- Another option is to call `task_delay`.

### Example 1:

```
puts(1, "Waiting 15 seconds and a quarter...\n")
sleep(15.25)
puts(1, "Done.\n")
```

### See Also:

[task\\_schedule](#) | [task\\_yield](#) | [task\\_delay](#)

# Pipe Input and Output

## Notes

Due to a bug, Euphoria does not handle STDERR properly. STDERR cannot be captured for Euphoria programs (other programs will work fully). The IO functions currently work with file handles, a future version might wrap them in streams so that they can be used directly alongside other file/socket/other-streams with a `stream_select` function.

## Accessor Constants

### enum

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

### STDIN

```
include std/pipeio.e
namespace pipeio
STDIN
```

### STDOUT

```
include std/pipeio.e
namespace pipeio
STDOUT
```

### STDERR

```
include std/pipeio.e
namespace pipeio
STDERR
```

### PID

```
include std/pipeio.e
namespace pipeio
PID
```

### enum

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

## PARENT

```
include std/pipeio.e
namespace pipeio
PARENT
```

## CHILD

```
include std/pipeio.e
namespace pipeio
CHILD
```

## Opening and Closing

### process

```
include std/pipeio.e
namespace pipeio
public type process(object o)
```

Process Type

### close

```
include std/pipeio.e
namespace pipeio
public function close(atom fd)
```

closes handle fd.

#### Returns:

An **integer**, 0 on success, -1 on failure

#### Example 1:

```
integer status = pipeio:close(p[STDIN])
```

### kill

```
include std/pipeio.e
namespace pipeio
public procedure kill(process p, atom signal = 15)
```

closes pipes and kills process p with signal signal (default 15).

#### Comments:

Signal is ignored on *Windows*.

#### Example 1:

```
kill(p)
```

## Read and Write Process

### read

```
include std/pipeio.e
namespace pipeio
public function read(atom fd, integer bytes)
```

reads bytes bytes from handle fd.

#### Returns:

A **sequence**, containing data, an empty sequence on EOF or an error code. Similar to `get_bytes`.

#### Example 1:

```
sequence data=read(p[STDOUT],256)
```

### write

```
include std/pipeio.e
namespace pipeio
public function write(atom fd, sequence str)
```

writes bytes to handle fd.

#### Returns:

An **integer**, number of bytes written, or -1 on error

#### Example 1:

```
integer bytes_written = write(p[STDIN],"Hello World!")
```

### error\_no

```
include std/pipeio.e
namespace pipeio
public function error_no()
```

gets error no from last call to a pipe function.

#### Comments:

Value returned will be OS-specific, and is not always set on *Windows* at least

#### Example 1:

```
integer error = error_no()
```

### create

```
include std/pipeio.e
```

```
namespace pipeio
public function create()
```

creates pipes for inter-process communication.

#### Returns:

A **handle**, process handles { {parent side pipes},{child side pipes} }

#### Example 1:

```
object p = exec("dir", create())
```

## exec

```
include std/pipeio.e
namespace pipeio
public function exec(sequence cmd, sequence pipe)
```

opens process with command line cmd.

#### Returns:

A **handle**, process handles { **PID**, **STDIN**, **STDOUT**, **STDERR** }

#### Example 1:

```
object p = exec("dir", create())
```



# Pretty Printing

## PRETTY\_DEFAULT

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

## enum

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

## Subroutines

## pretty\_print

```
include std/pretty.e
namespace pretty
public procedure pretty_print(integer fn, object x, sequence options = PRETTY_DEFAULT)
```

prints an object to a file or device using braces { , , , }, indentation, and multiple lines to show the structure.

### Arguments:

1. `fn` : an integer, the file or device number to write to
2. `x` : the object to display or convert to printable form
3. `options` : is an (up to) 10-element options sequence.

### Comments:

Pass {} in options to select the defaults, or set options as below:

1. display ASCII characters:
  - 0 -- never
  - 1 -- alongside any integers in printable ASCII range (default)
  - 2 -- display as "string" when all integers of a sequence are in ASCII range
  - 3 -- show strings, and quoted characters (only) for any integers in ASCII range as well as the characters: `\t \r \n`
2. amount to indent for each level of sequence nesting -- default: 2
3. column we are starting at -- default: 1
4. approximate column to wrap at -- default: 78
5. format to use for integers -- default: "%d"
6. format to use for floating-point numbers -- default: "%.10g"
7. minimum value for printable ASCII -- default 32
8. maximum value for printable ASCII -- default 127
9. maximum number of lines to output
10. line breaks between elements -- default 1 (0 = no line breaks, -1 = line breaks to wrap only)

If the length is less than ten, unspecified options at the end of the sequence will keep the default values. For example: {0, 5} will choose "never display ASCII", plus 5-character

indentation, with defaults for everything else.

The default options can be applied using the public constant `PRETTY_DEFAULT`, and the elements may be accessed using the following public enum:

1. `DISPLAY_ASCII`
2. `INDENT`
3. `START_COLUMN`
4. `WRAP`
5. `INT_FORMAT`
6. `FP_FORMAT`
7. `MIN_ASCII`
8. `MAX_ASCII`
9. `MAX_LINES`
10. `LINE_BREAKS`

The display will start at the current cursor position. Normally you will want to call `pretty_print` when the cursor is in column 1 (after printing a `\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.

When specifying the format to use for integers and floating-point numbers, you can add some decoration. For example: `"(%d)"` or `"$ %.2f"`.

### Example 1:

```
pretty_print(1, "ABC", {})  
  
{65'A',66'B',67'C'}
```

### Example 2:

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

### Example 3:

```
pretty_print(1, {"Euphoria", "Programming", "Language"}, {2})  
  
{  
  "Euphoria",  
  "Programming",  
  "Language"  
}
```

### Example 4:

```
puts(1, "word_list = ") -- moves cursor to column 13  
pretty_print(1,  
  {{{"Euphoria", 8, 5.3},  
    {"Programming", 11, -2.9},  
    {"Language", 8, 9.8}},  
  {2, 4, 13, 78, "%03d", "%.3f"}) -- first 6 of 8 options  
  
word_list = {  
  {  
    "Euphoria",  
    008,  
    5.300  
  },  
  {  
    "Programming",
```

```
    011,  
    -2.900  
  },  
  {  
    "Language",  
    008,  
    9.800  
  }  
}
```

### See Also:

[print](#) | [sprint](#) | [printf](#) | [sprintf](#) | [pretty\\_sprint](#)

## pretty\_sprint

```
include std/pretty.e  
namespace pretty  
public function pretty_sprint(object x, sequence options = PRETTY_DEFAULT)
```

formats an object using braces { , , }, indentation, and multiple lines to show the structure.

### Arguments:

1. `x` : the object to display
2. `options` : is an (up to) 10-element options sequence: Pass {} to select the defaults, or set options

### Returns:

A **sequence**, of printable characters, representing `x` in an human-readable form.

### Comments:

This function formats objects the same as [pretty\\_print](#) but returns the sequence obtained instead of sending it to some file..

### See Also:

[pretty\\_print](#) | [sprint](#)

# Multi-Tasking

## General Notes

For a complete overview of the task system, please see the mini-guide [Multitasking in Euphoria](#).

## Warning

The task system does not yet function in a shared library. Task routine calls that are compiled into a shared library are emitted as a NOP (no operation) and will therefore have no effect.

It is planned to allow the task system to function in shared libraries in future versions of OpenEuphoria.

## Suboutines

### task\_delay

```
include std/task.e
namespace task
public procedure task_delay(atom delaytime)
```

suspends a task for a short period, allowing other tasks to run in the meantime.

#### Arguments:

1. delaytime : an atom, the duration of the delay in seconds.

#### Comments:

This procedure is similar to [sleep](#) but allows for other tasks to run by yielding on a regular basis. Like [sleep](#) its argument needs not being an integer.

#### See Also:

[sleep](#)

### task\_clock\_start

```
<built-in> procedure task_clock_start()
```

restarts the clock used for scheduling real-time tasks.

#### Comments:

Call this routine, some time after calling [task\\_clock\\_stop](#), when you want scheduling of real-time tasks to continue.

[task\\_clock\\_stop](#) and [task\\_clock\\_start](#) can be used to freeze the scheduling of real-time tasks.

[task\\_clock\\_start](#) causes the scheduled times of all real-time tasks to be incremented by the amount of time since [task\\_clock\\_stop](#) was called. This allows a game, simulation, or other program to continue smoothly.

Time-shared tasks are not affected.

### Example 1:

```
-- freeze the game while the player answers the phone
task_clock_stop()
while get_key() = -1 do
end while
task_clock_start()
```

### See Also:

[task\\_clock\\_stop](#) | [task\\_schedule](#) | [task\\_yield](#) | [task\\_suspend](#) | [task\\_delay](#)

## task\_clock\_stop

```
<built-in> procedure task_clock_stop()
```

stops the scheduling of real-time tasks.

### Comments:

Call `task_clock_stop` when you want to take time out from scheduling real-time tasks. For instance, you want to temporarily suspend a game or simulation for a period of time.

Scheduling will resume when `task_clock_start` is called.

Time-shared tasks can continue. The current task can also continue, unless it is a real-time task and it yields.

The `time` function is not affected by this.

### See Also:

[task\\_clock\\_start](#) | [task\\_schedule](#) | [task\\_yield](#) | [task\\_suspend](#) | [task\\_delay](#)

## task\_create

```
<built-in> function task_create(integer rid, sequence args)
```

creates a new task, given a home procedure and the arguments passed to it.

### Arguments:

1. `rid` : an integer, the `routine_id` of a user-defined Euphoria procedure.
2. `args` : a sequence, the list of arguments that will be passed to this procedure when the task starts executing.

### Returns:

An **atom**, a task identifier, created by the system. It can be used to identify this task to the other Euphoria multitasking routines.

### Errors:

There must be at most 12 parameters in `args`.

### Comments:

`task_create` creates a new task, but does not start it executing. You must call `task_schedule` for this purpose.

Each task has its own set of private variables and its own call stack. Global and local variables are shared between all tasks.

If a run-time error is detected, the traceback will include information on all tasks, with the offending task listed first.

Many tasks can be created that all run the same procedure, possibly with different parameters.

A task cannot be based on a function, since there would be no way of using the function result.

Each task id is unique. `task_create` never returns the same task id as it did before. Task id's are integer-valued atoms and can be as large as the largest integer-valued atom (15 digits).

### Example 1:

```
mytask = task_create(routine_id("myproc"), {5, 9, "ABC"})
```

### See Also:

[task\\_schedule](#) | [task\\_yield](#) | [task\\_suspend](#) | [task\\_self](#)

## task\_list

```
<built-in> function task_list()
```

gets a sequence containing the task id's for all active or suspended tasks.

### Returns:

A **sequence**, of atoms, the list of all task that are or may be scheduled.

### Comments:

This function lets you find out which tasks currently exist. Tasks that have terminated are not included. You can pass a task id to [task\\_status](#) to find out more about a particular task.

### Example 1:

```
sequence tasks

tasks = task_list()
for i = 1 to length(tasks) do
  if task_status(tasks[i]) > 0 then
    printf(1, "task %d is active\n", tasks[i])
  end if
end for
```

### See Also:

[task\\_status](#) | [task\\_create](#) | [task\\_schedule](#) | [task\\_yield](#) | [task\\_suspend](#)

## task\_schedule

```
<built-in> procedure task_schedule(atom task_id, object schedule)
```

schedules a task to run using a scheduling parameter.

## Arguments:

1. `task_id` : an atom, the identifier of a task that did not terminate yet.
2. `schedule` : an object, describing when and how often to run the task.

## Comments:

`task_id` must have been returned by `task_create`.

The task scheduler, which is built-in to the Euphoria run-time system, will use `schedule` as a guide when scheduling this task. It may not always be possible to achieve the desired number of consecutive runs, or the desired time frame. For instance, a task might take so long before yielding control, that another task misses its desired time window.

`schedule` is being interpreted as follows:

`schedule` is an integer:

This defines `task_id` as time shared, and tells the task scheduler how many times it should the task in one burst before it considers running other tasks. `schedule` must be greater than zero then.

Increasing this count will increase the percentage of CPU time given to the selected task, while decreasing the percentage given to other time-shared tasks. Use trial and error to find the optimal trade off. It will also increase the efficiency of the program, since each actual task switch wastes a bit of time.

`schedule` is a sequence:

In this case, it must be a pair of positive atoms, the first one not being less than the second one. This defines `task_id` as a real time task. The pair states the minimum and maximum times, in seconds, to wait before running the task. The pair also sets the time interval for subsequent runs of the task, until the next call to `task_schedule` or `task_suspend`.

Real-time tasks have a higher priority. Time-shared tasks are run when no real-time task is ready to execute.

A task can switch back and forth between real-time and time-shared. It all depends on the last call to `task_schedule` for that task. The scheduler never runs a real-time task before the start of its time frame (min value in the {min, max} pair), and it tries to avoid missing the task's deadline (max value).

For precise timing, you can specify the same value for min and max. However, by specifying a range of times, you give the scheduler some flexibility. This allows it to schedule tasks more efficiently, and avoid non-productive delays. When the scheduler must delay, it calls `sleep`, unless the required delay is very short. `sleep` lets the operating system run other programs.

The min and max values can be fractional. If the min value is smaller than the resolution of the scheduler's clock (currently 0.01 seconds on *Windows* or *Unix*) then accurate time scheduling cannot be performed, but the scheduler will try to run the task several times in a row to approximate what is desired.

For example, if you ask for a min time of 0.002 seconds, then the scheduler will try to run your task  $0.01/0.002 = 5$  times in a row before waiting for the clock to "click" ahead by 0.01. During the next 0.01 seconds it will run your task (up to) another 5 times etc. provided your task can be completed 5 times in one clock period.

At program start-up there is a single task running. Its task id is 0, and initially it is a time-shared task allowed 1 run per `task_yield`. No other task can run until task 0 executes a `task_yield`.

If task 0 (top-level) runs off the end of the main file, the whole program terminates, regardless of what other tasks may still be active.

If the scheduler finds that no task is active, i.e. no task will ever run again (not even task 0), it terminates the program with a 0 exit code, similar to `abort(0)`.

### Example 1:

```
-- Task t1 will be executed up to 10 times in a row before
-- other time-shared tasks are given control. If a real-time
-- task needs control, t1 will lose control to the real-time task.
task_schedule(t1, 10)

-- Task t2 will be scheduled to run some time between 4 and 5 seconds
-- from now. Barring any rescheduling of t2, it will continue to
-- execute every 4 to 5 seconds thereafter.
task_schedule(t2, {4, 5})
```

### See Also:

`task_create` | `task_yield` | `task_suspend`

## task\_self

```
<built-in> function task_self()
```

returns the task id of the current task.

### Comments:

This value may be needed, if a task wants to schedule or suspend itself.

### Example 1:

```
-- schedule self
task_schedule(task_self(), {5.9, 6.0})
```

### See Also:

`task_create` | `task_schedule` | `task_yield` | `task_suspend`

## task\_status

```
<built-in> function task_status(atom task_id)
```

returns the status of a task.

### Arguments:

1. `task_id` : an atom, the id of the task being queried.

### Returns:

An **integer**,

- -1 -- task does not exist, or terminated
- 0 -- task is suspended
- 1 -- task is active

### Comments:

A task might want to know the status of one or more other tasks when deciding whether to proceed with some processing.



### Example 1:

```
integer s

s = task_status(tid)
if s = 1 then
    puts(1, "ACTIVE\n")
elseif s = 0 then
    puts(1, "SUSPENDED\n")
else
    puts(1, "DOESN'T EXIST\n")
end if
```

### See Also:

[task\\_list](#) | [task\\_create](#) | [task\\_schedule](#) | [task\\_suspend](#)

## task\_suspend

```
<built-in> procedure task_suspend(atom task_id)
```

suspends execution of a task.

### Arguments:

1. `task_id` : an atom, the id of the task to suspend.

### Comments:

A suspended task will not be executed again unless there is a call to [task\\_schedule](#) for the task.

`task_id` is a task id returned from [task\\_create](#). - Any task can suspend any other task. If a task suspends itself, the suspension will start as soon as the task calls [task\\_yield](#).

Suspending a task and never scheduling it again is how to kill a task. There is no `task_kill` primitives because undead tasks were creating too much trouble and confusion. As a general fact, nothing that impacts a running task can be effective as long as the task has not yielded.

### Example 1:

```
-- suspend task 15
task_suspend(15)

-- suspend current task
task_suspend(task_self())
```

### See Also:

[task\\_create](#) | [task\\_schedule](#) | [task\\_self](#) | [task\\_yield](#)

## task\_yield

```
<built-in> procedure task_yield()
```

yields control to the scheduler. The scheduler can then choose another task to run, or perhaps let the current task continue running.

### Comments:

Tasks should call `task_yield` periodically so other tasks will have a chance to run. Only when `task_yield` is called, is there a way for the scheduler to take back control from a task. This is what is known as cooperative multitasking.

A task can have calls to `task_yield` in many different places in its code, and at any depth of subroutine call.

The scheduler will use the current scheduling parameter (see [task\\_schedule](#)), in determining when to return to the current task.

When control returns, execution will continue with the statement that follows `task_yield`. The call-stack and all private variables will remain as they were when `task_yield` was called. Global and local variables may have changed, due to the execution of other tasks.

Tasks should try to call `task_yield` often enough to avoid causing real-time tasks to miss their time window, and to avoid blocking time-shared tasks for an excessive period of time. On the other hand, there is a bit of overhead in calling `task_yield`, and this overhead is slightly larger when an actual switch to a different task takes place. A `task_yield` where the same task continues executing takes less time.

A task should avoid calling `task_yield` when it is in the middle of a delicate operation that requires exclusive access to some data. Otherwise a race condition could occur, where one task might interfere with an operation being carried out by another task. In some cases a task might need to mark some data as "locked" or "unlocked" in order to prevent this possibility. With cooperative multitasking, these concurrency issues are much less of a problem than with the preemptive multitasking that other languages support.

### Example 1:

```
-- From Language war game.
-- This small task deducts life support energy from either the
-- large Euphoria ship or the small shuttle.
-- It seems to run "forever" in an infinite loop,
-- but it's actually a real-time task that is called
-- every 1.7 to 1.8 seconds throughout the game.
-- It deducts either 3 units or 13 units of life support energy each time.

procedure task_life()
-- independent task: subtract life support energy
  while TRUE do
    if shuttle then
      p_energy(-3)
    else
      p_energy(-13)
    end if
    task_yield()
  end while
end procedure
```

### See Also:

[task\\_create](#) | [task\\_schedule](#) | [task\\_suspend](#)

# Types - Extended

## OBJ\_UNASSIGNED

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

Object not assigned

## OBJ\_INTEGER

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

Object is integer

## OBJ\_ATOM

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

Object is atom

## OBJ\_SEQUENCE

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

Object is sequence

## object

```
<built-in> type object(object x)
```

returns information about the object type of the supplied argument x.

### Returns:

1. An **integer**,
  - OBJ\_UNASSIGNED if x has not been assigned anything yet.
  - OBJ\_INTEGER if x holds an integer value.
  - OBJ\_ATOM if x holds a number that is not an integer.
  - OBJ\_SEQUENCE if x holds a sequence value.

### Example 1:

```
? object(1) --> OBJ_INTEGER
? object(1.1) --> OBJ_ATOM
? object("1") --> OBJ_SEQUENCE
object x
? object(x) --> OBJ_UNASSIGNED
```

### See Also:

[sequence](#) | [integer](#) | [atom](#)

## integer

```
<built-in> type integer(object x)
```

tests the supplied argument x to see if it is an integer or not.

### Returns:

1. An **integer**.
  - 1 if x is an integer.
  - 0 if x is not an integer.

### Example 1:

```
? integer(1) --> 1
? integer(1.1) --> 0
? integer("1") --> 0
```

### See Also:

[sequence](#) | [object](#) | [atom](#)

## atom

```
<built-in> type atom(object x)
```

tests the supplied argument x to see if it is an atom or not.

### Returns:

1. An **integer**,
  - 1 if x is an atom.
  - 0 if x is not an atom.

### Example 1:

```
? atom(1) --> 1
? atom(1.1) --> 1
? atom("1") --> 0
```

### See Also:

[sequence](#) | [object](#) | [integer](#)

## sequence

```
<built-in> type sequence( object x)
```

tests the supplied argument *x* to see if it is a sequence or not.

### Returns:

1. An
  - 1 if *x* is a sequence.
  - 0 if *x* is not an sequence.

### Example 1:

```
? sequence(1) --> 0
? sequence({1}) --> 1
? sequence("1") --> 1
```

### See Also:

[integer](#) | [object](#) | [atom](#)

## FALSE

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

Boolean FALSE value

## TRUE

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

Boolean TRUE value

## Predefined Character Sets

### enum

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

## Support Functions

### char\_test

```
include std/types.e
namespace types
public function char_test(object test_data, sequence char_set)
```

determines whether one or more characters are in a given character set.

### Arguments:

1. *test\_data* : an object to test, either a character or a string

2. `char_set` : a sequence, either a list of allowable characters, or a list of pairs representing allowable ranges.

### Returns:

An **integer**, 1 if all characters are allowed, else 0.

### Comments:

`pCharset` is either a simple sequence of characters ( such as "qwertyuiop[]" ) or a sequence of character pairs, which represent allowable ranges of characters. For example Alphabetic is defined as { {'a','z'}, {'A', 'Z'} } .

To add an isolated character to a character set which is defined using ranges, present it as a range of length one, like in {%,%}.

### Example 1:

```
char_test("ABCD", {{'A', 'D'}})
-- TRUE, every char is in the range 'A' to 'D'

char_test("ABCD", {{'A', 'C'}})
-- FALSE, not every char is in the range 'A' to 'C'

char_test("Harry", {{'a', 'z'}, {'D', 'J'}})
-- TRUE, every char is either in the range 'a' to 'z',
--      or in the range 'D' to 'J'

char_test("Potter", "novel")
-- FALSE, not every character is in the set 'n', 'o', 'v', 'e', 'l'
```

## set\_default\_charsets

```
include std/types.e
namespace types
public procedure set_default_charsets()
```

sets all the defined character sets to their default definitions.

### Example 1:

```
set_default_charsets()
```

## get\_charsets

```
include std/types.e
namespace types
public function get_charsets()
```

gets the definition for each of the defined character sets.

### Returns:

A **sequence**, of pairs. The first element of each pair is the character set id ( such as `CS_Whitespace` ) and the second is the definition of that character set.

### Comments:

This is the same format required for the `set_charsets` routine.

### Example 1:

```
sequence sets
sets = get_charsets()
```

### See Also:

[set\\_charsets](#) | [set\\_default\\_charsets](#)

## set\_charsets

```
include std/types.e
namespace types
public procedure set_charsets(sequence charset_list)
```

sets the definition for one or more defined character sets.

### Arguments:

1. `charset_list` : a sequence of zero or more character set definitions.

### Comments:

`charset_list` must be a sequence of pairs. The first element of each pair is the character set id (such as `CS_Whitespace`) and the second is the definition of that character set.

This is the same format returned by the [get\\_charsets](#) routine.

You cannot create new character sets using this routine.

### Example 1:

```
set_charsets({{CS_Whitespace, " \t"}})
t_space('\n') --> FALSE

t_specword('$') --> FALSE
set_charsets({{CS_SpecWord, "_-#%"}})
t_specword('$') --> TRUE
```

### See Also:

[get\\_charsets](#)

## Types

## boolean

```
include std/types.e
namespace types
public type boolean(object test_data)
```

test for an integer boolean.

### Returns:

Returns TRUE if argument is 1 or 0

Returns FALSE if the argument is anything else other than 1 or 0.

### Example 1:

```

boolean(-1)      -- FALSE
boolean(0)       -- TRUE
boolean(1)       -- TRUE
boolean(1.234)   -- FALSE
boolean('A')    -- FALSE
boolean('9')    -- FALSE
boolean('?')    -- FALSE
boolean("abc")  -- FALSE
boolean("ab3")  -- FALSE
boolean({1,2,"abc"}) -- FALSE
boolean({1, 2, 9.7}) -- FALSE
boolean({})     -- FALSE (empty sequence)

```

## t\_boolean

```

include std/types.e
namespace types
public type t_boolean(object test_data)

```

tests elements for boolean.

### Returns:

Returns TRUE if argument is boolean (1 or 0) or if every element of the argument is boolean.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-boolean elements

### Example 1:

```

t_boolean(-1)      -- FALSE
t_boolean(0)       -- TRUE
t_boolean(1)       -- TRUE
t_boolean({1, 1, 0}) -- TRUE
t_boolean({1, 1, 9.7}) -- FALSE
t_boolean({})     -- FALSE (empty sequence)

```

## t\_alnum

```

include std/types.e
namespace types
public type t_alnum(object test_data)

```

tests for alphanumeric character.

### Returns:

Returns TRUE if argument is an alphanumeric character or if every element of the argument is an alphanumeric character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-alphanumeric elements

### Example 1:

```

t_alnum(-1)      -- FALSE
t_alnum(0)       -- FALSE
t_alnum(1)       -- FALSE
t_alnum(1.234)   -- FALSE
t_alnum('A')    -- TRUE

```



```

t_alnum('9')      -- TRUE
t_alnum('?')      -- FALSE
t_alnum("abc")    -- TRUE (every element is alphabetic or a digit)
t_alnum("ab3")    -- TRUE
t_alnum({1, 2, "abc"}) -- FALSE (contains a sequence)
t_alnum({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_alnum({})       -- FALSE (empty sequence)

```

## t\_identifier

```

include std/types.e
namespace types
public type t_identifier(object test_data)

```

tests string if it is an valid identifier.

### Returns:

Returns TRUE if argument is an alphanumeric character or if every element of the argument is an alphanumeric character and that the first character is not numeric and the whole group of characters are not all numeric.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-alphanumeric elements

### Example 1:

```

t_identifier(-1)      -- FALSE
t_identifier(0)       -- FALSE
t_identifier(1)       -- FALSE
t_identifier(1.234)   -- FALSE
t_identifier('A')     -- TRUE
t_identifier('9')     -- FALSE
t_identifier('?')     -- FALSE
t_identifier("abc")   -- TRUE (every element is alphabetic or a digit)
t_identifier("ab3")   -- TRUE
t_identifier("ab_3")  -- TRUE (underscore is allowed)
t_identifier("1abc")  -- FALSE (identifier cannot start with a number)
t_identifier("102")   -- FALSE (identifier cannot be all numeric)
t_identifier({1, 2, "abc"}) -- FALSE (contains a sequence)
t_identifier({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_identifier({})      -- FALSE (empty sequence)

```

## t\_alpha

```

include std/types.e
namespace types
public type t_alpha(object test_data)

```

tests for alphabetic characters.

### Returns:

Returns TRUE if argument is an alphabetic character or if every element of the argument is an alphabetic character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-alphabetic elements

### Example 1:

```

t_alpha(-1)      -- FALSE
t_alpha(0)       -- FALSE
t_alpha(1)       -- FALSE
t_alpha(1.234)   -- FALSE
t_alpha('A')     -- TRUE
t_alpha('9')     -- FALSE
t_alpha('?')     -- FALSE
t_alpha("abc")   -- TRUE (every element is alphabetic)
t_alpha("ab3")   -- FALSE
t_alpha({1, 2, "abc"}) -- FALSE (contains a sequence)
t_alpha({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_alpha({})      -- FALSE (empty sequence)

```

## t\_ascii

```

include std/types.e
namespace types
public type t_ascii(object test_data)

```

tests for ASCII characters.

### Returns:

Returns TRUE if argument is an ASCII character or if every element of the argument is an ASCII character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-ASCII elements

### Example 1:

```

t_ascii(-1)      -- FALSE
t_ascii(0)       -- TRUE
t_ascii(1)       -- TRUE
t_ascii(1.234)   -- FALSE
t_ascii('A')     -- TRUE
t_ascii('9')     -- TRUE
t_ascii('?')     -- TRUE
t_ascii("abc")   -- TRUE (every element is ascii)
t_ascii("ab3")   -- TRUE
t_ascii({1, 2, "abc"}) -- FALSE (contains a sequence)
t_ascii({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_ascii({})      -- FALSE (empty sequence)

```

## t\_cntrl

```

include std/types.e
namespace types
public type t_cntrl(object test_data)

```

tests for control characters.

### Returns:

Returns TRUE if argument is an Control character or if every element of the argument is an Control character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-Control elements

### Example 1:

```

t_cntrl(-1)      -- FALSE
t_cntrl(0)       -- TRUE
t_cntrl(1)       -- TRUE
t_cntrl(1.234)   -- FALSE
t_cntrl('A')     -- FALSE
t_cntrl('9')     -- FALSE
t_cntrl('?')     -- FALSE
t_cntrl("abc")   -- FALSE (every element is ascii)
t_cntrl("ab3")   -- FALSE
t_cntrl({1, 2, "abc"}) -- FALSE (contains a sequence)
t_cntrl({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_cntrl({1, 2, 'a'}) -- FALSE (contains a non-control)
t_cntrl({})      -- FALSE (empty sequence)

```

## t\_digit

```

include std/types.e
namespace types
public type t_digit(object test_data)

```

tests for digits.

### Returns:

Returns TRUE if argument is an digit character or if every element of the argument is an digit character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-digits

### Example 1:

```

t_digit(-1)      -- FALSE
t_digit(0)       -- FALSE
t_digit(1)       -- FALSE
t_digit(1.234)   -- FALSE
t_digit('A')     -- FALSE
t_digit('9')     -- TRUE
t_digit('?')     -- FALSE
t_digit("abc")   -- FALSE
t_digit("ab3")   -- FALSE
t_digit("123")   -- TRUE
t_digit({1, 2, "abc"}) -- FALSE (contains a sequence)
t_digit({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_digit({1, 2, 'a'}) -- FALSE (contains a non-digit)
t_digit({})      -- FALSE (empty sequence)

```

## t\_graph

```

include std/types.e
namespace types
public type t_graph(object test_data)

```

test for glyphs (printable) characters.

### Returns:

Returns TRUE if argument is a glyph character or if every element of the argument is a glyph character. (One that is visible when displayed)

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-glyph

### Example 1:

```

t_graph(-1)           -- FALSE
t_graph(0)            -- FALSE
t_graph(1)            -- FALSE
t_graph(1.234)         -- FALSE
t_graph('A')          -- TRUE
t_graph('9')          -- TRUE
t_graph('?')          -- TRUE
t_graph(' ')          -- FALSE
t_graph("abc")        -- TRUE
t_graph("ab3")         -- TRUE
t_graph("123")         -- TRUE
t_graph({1, 2, "abc"}) -- FALSE (contains a sequence)
t_graph({1, 2, 9.7})   -- FALSE (contains a non-integer)
t_graph({1, 2, 'a'})   -- FALSE (control chars (1,2) don't have glyphs)
t_graph({})            -- FALSE (empty sequence)

```

## t\_specword

```

include std/types.e
namespace types
public type t_specword(object test_data)

```

tests for a special word character.

### Returns:

Returns TRUE if argument is a special word character or if every element of the argument is a special word character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-special-word characters.

### Comments:

A *special word character* is any character that is not normally part of a word but in certain cases may be considered. This is most commonly used when looking for words in programming source code which allows an underscore as a word character.

### Example 1:

```

t_specword(-1)           -- FALSE
t_specword(0)            -- FALSE
t_specword(1)            -- FALSE
t_specword(1.234)         -- FALSE
t_specword('A')          -- FALSE
t_specword('9')          -- FALSE
t_specword('?')          -- FALSE
t_specword('_')          -- TRUE
t_specword("abc")        -- FALSE
t_specword("ab3")         -- FALSE
t_specword("123")         -- FALSE
t_specword({1, 2, "abc"}) -- FALSE (contains a sequence)
t_specword({1, 2, 9.7})   -- FALSE (contains a non-integer)
t_specword({1, 2, 'a'})   -- FALSE (control chars (1,2) don't have glyphs)
t_specword({})            -- FALSE (empty sequence)

```

## t\_bytearray

```

include std/types.e
namespace types

```

```
public type t_bytearray(object test_data)
```

tests for bytes.

### Returns:

Returns TRUE if argument is a byte or if every element of the argument is a byte. (Integers from 0 to 255)

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-byte

### Example 1:

```
t_bytearray(-1)      -- FALSE (contains value less than zero)
t_bytearray(0)       -- TRUE
t_bytearray(1)       -- TRUE
t_bytearray(10)      -- TRUE
t_bytearray(100)     -- TRUE
t_bytearray(1000)    -- FALSE (greater than 255)
t_bytearray(1.234)   -- FALSE (contains a floating number)
t_bytearray('A')     -- TRUE
t_bytearray('9')     -- TRUE
t_bytearray('?')     -- TRUE
t_bytearray(' ')     -- TRUE
t_bytearray("abc")   -- TRUE
t_bytearray("ab3")   -- TRUE
t_bytearray("123")   -- TRUE
t_bytearray({1, 2, "abc"}) -- FALSE (contains a sequence)
t_bytearray({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_bytearray({1, 2, 'a'}) -- TRUE
t_bytearray({})      -- FALSE (empty sequence)
```

## t\_lower

```
include std/types.e
namespace types
public type t_lower(object test_data)
```

tests for lowercase characters.

### Returns:

Returns TRUE if argument is a lowercase character or if every element of the argument is an lowercase character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-lowercase

### Example 1:

```
t_lower(-1)      -- FALSE
t_lower(0)       -- FALSE
t_lower(1)       -- FALSE
t_lower(1.234)   -- FALSE
t_lower('A')     -- FALSE
t_lower('9')     -- FALSE
t_lower('?')     -- FALSE
t_lower("abc")   -- TRUE
t_lower("ab3")   -- FALSE
t_lower("123")   -- TRUE
t_lower({1, 2, "abc"}) -- FALSE (contains a sequence)
t_lower({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_lower({1, 2, 'a'}) -- FALSE (contains a non-digit)
t_lower({})      -- FALSE (empty sequence)
```

## t\_print

```
include std/types.e
namespace types
public type t_print(object test_data)
```

tests for ASCII glyph characters.

### Returns:

Returns TRUE if argument is a character that has an ASCII glyph or if every element of the argument is a character that has an ASCII glyph.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains characters that do not have an ASCII glyph.

### Example 1:

```
t_print(-1)           -- FALSE
t_print(0)            -- FALSE
t_print(1)            -- FALSE
t_print(1.234)         -- FALSE
t_print('A')          -- TRUE
t_print('9')          -- TRUE
t_print('?')          -- TRUE
t_print("abc")        -- TRUE
t_print("ab3")        -- TRUE
t_print("123")        -- TRUE
t_print("123 ")       -- FALSE (contains a space)
t_print("123\n")      -- FALSE (contains a new-line)
t_print({1, 2, "abc"}) -- FALSE (contains a sequence)
t_print({1, 2, 9.7})  -- FALSE (contains a non-integer)
t_print({1, 2, 'a'})  -- FALSE
t_print({})           -- FALSE (empty sequence)
```

## t\_display

```
include std/types.e
namespace types
public type t_display(object test_data)
```

tests for printable characters.

### Returns:

Returns TRUE if argument is a character that can be displayed or if every element of the argument is a character that can be displayed.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains characters that cannot be displayed.

### Example 1:

```
t_display(-1)         -- FALSE
t_display(0)          -- FALSE
t_display(1)          -- FALSE
t_display(1.234)       -- FALSE
t_display('A')        -- TRUE
t_display('9')        -- TRUE
t_display('?')        -- TRUE
t_display("abc")      -- TRUE
```

```

t_display("ab3")      -- TRUE
t_display("123")      -- TRUE
t_display("123 ")     -- TRUE
t_display("123\n")    -- TRUE
t_display({1, 2, "abc"}) -- FALSE (contains a sequence)
t_display({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_display({1, 2, 'a'}) -- FALSE
t_display({})         -- FALSE (empty sequence)

```

## t\_punct

```

include std/types.e
namespace types
public type t_punct(object test_data)

```

tests for punctuation characters.

### Returns:

Returns TRUE if argument is an punctuation character or if every element of the argument is an punctuation character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-punctuation symbols.

### Example 1:

```

t_punct(-1)      -- FALSE
t_punct(0)       -- FALSE
t_punct(1)       -- FALSE
t_punct(1.234)   -- FALSE
t_punct('A')     -- FALSE
t_punct('9')     -- FALSE
t_punct('?')     -- TRUE
t_punct("abc")   -- FALSE
t_punct("(-)")   -- TRUE
t_punct("123")   -- TRUE
t_punct({1, 2, "abc"}) -- FALSE (contains a sequence)
t_punct({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_punct({1, 2, 'a'}) -- FALSE (contains a non-digit)
t_punct({})      -- FALSE (empty sequence)

```

## t\_space

```

include std/types.e
namespace types
public type t_space(object test_data)

```

tests for whitespace characters.

### Returns:

Returns TRUE if argument is a whitespace character or if every element of the argument is an whitespace character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-whitespace character.

### Example 1:

```

t_space(-1)      -- FALSE
t_space(0)       -- FALSE

```

```

t_space(1)           -- FALSE
t_space(1.234)        -- FALSE
t_space('A')         -- FALSE
t_space('9')         -- FALSE
t_space('\t')        -- TRUE
t_space("abc")       -- FALSE
t_space("123")       -- FALSE
t_space({1, 2, "abc"}) -- FALSE (contains a sequence)
t_space({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_space({1, 2, 'a'}) -- FALSE (contains a non-digit)
t_space({})          -- FALSE (empty sequence)

```

## t\_upper

```

include std/types.e
namespace types
public type t_upper(object test_data)

```

tests for uppercase characters.

### Returns:

Returns TRUE if argument is an uppercase character or if every element of the argument is an uppercase character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-uppercase characters.

### Example 1:

```

t_upper(-1)           -- FALSE
t_upper(0)            -- FALSE
t_upper(1)            -- FALSE
t_upper(1.234)        -- FALSE
t_upper('A')          -- TRUE
t_upper('9')          -- FALSE
t_upper('?')          -- FALSE
t_upper("abc")        -- FALSE
t_upper("ABC")        -- TRUE
t_upper("123")        -- FALSE
t_upper({1, 2, "abc"}) -- FALSE (contains a sequence)
t_upper({1, 2, 9.7})  -- FALSE (contains a non-integer)
t_upper({1, 2, 'a'})  -- FALSE (contains a non-digit)
t_upper({})           -- FALSE (empty sequence)

```

## t\_xdigit

```

include std/types.e
namespace types
public type t_xdigit(object test_data)

```

tests for hexadecimal characters.

### Returns:

Returns TRUE if argument is an hexadecimal digit character or if every element of the argument is an hexadecimal digit character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-hexadecimal character.

### Example 1:



```

t_xdigit(-1)      -- FALSE
t_xdigit(0)       -- FALSE
t_xdigit(1)       -- FALSE
t_xdigit(1.234)   -- FALSE
t_xdigit('A')     -- TRUE
t_xdigit('9')     -- TRUE
t_xdigit('?')     -- FALSE
t_xdigit("abc")   -- TRUE
t_xdigit("fgh")   -- FALSE
t_xdigit("123")   -- TRUE
t_xdigit({1, 2, "abc"}) -- FALSE (contains a sequence)
t_xdigit({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_xdigit({1, 2, 'a'}) -- FALSE (contains a non-digit)
t_xdigit({})      -- FALSE (empty sequence)

```

## t\_vowel

```

include std/types.e
namespace types
public type t_vowel(object test_data)

```

tests for vowel characters.

### Returns:

Returns TRUE if argument is a vowel or if every element of the argument is a vowel character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-vowels

### Example 1:

```

t_vowel(-1)      -- FALSE
t_vowel(0)       -- FALSE
t_vowel(1)       -- FALSE
t_vowel(1.234)   -- FALSE
t_vowel('A')     -- TRUE
t_vowel('9')     -- FALSE
t_vowel('?')     -- FALSE
t_vowel("abc")   -- FALSE
t_vowel("aiu")   -- TRUE
t_vowel("123")   -- FALSE
t_vowel({1, 2, "abc"}) -- FALSE (contains a sequence)
t_vowel({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_vowel({1, 2, 'a'}) -- FALSE (contains a non-digit)
t_vowel({})      -- FALSE (empty sequence)

```

## t\_consonant

```

include std/types.e
namespace types
public type t_consonant(object test_data)

```

tests for consonant characters.

### Returns:

Returns TRUE if argument is a consonant character or if every element of the argument is an consonant character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or

contains non-consonant character.

### Example 1:

```
t_consonant(-1)      -- FALSE
t_consonant(0)       -- FALSE
t_consonant(1)       -- FALSE
t_consonant(1.234)   -- FALSE
t_consonant('A')     -- FALSE
t_consonant('9')     -- FALSE
t_consonant('?')     -- FALSE
t_consonant("abc")   -- FALSE
t_consonant("rTfM")  -- TRUE
t_consonant("123")   -- FALSE
t_consonant({1, 2, "abc"}) -- FALSE (contains a sequence)
t_consonant({1, 2, 9.7}) -- FALSE (contains a non-integer)
t_consonant({1, 2, 'a'}) -- FALSE (contains a non-digit)
t_consonant({})      -- FALSE (empty sequence)
```

## integer\_array

```
include std/types.e
namespace types
public type integer_array(object x)
```

tests for integer elements.

### Returns:

TRUE if argument is a sequence that only contains zero or more integers.

### Example 1:

```
integer_array(-1)      -- FALSE (not a sequence)
integer_array("abc")   -- TRUE (all single characters)
integer_array({1, 2, "abc"}) -- FALSE (contains a sequence)
integer_array({1, 2, 9.7}) -- FALSE (contains a non-integer)
integer_array({1, 2, 'a'}) -- TRUE
integer_array({})      -- TRUE
```

## t\_text

```
include std/types.e
namespace types
public type t_text(object x)
```

tests for text characters.

### Returns:

TRUE if argument is a sequence that only contains zero or more characters.

### Comments:

A **character** is defined as a positive integer or zero. This is a broad definition that may be refined once proper UNICODE support is implemented.

### Example 1:

```
t_text(-1)      -- FALSE (not a sequence)
t_text("abc")   -- TRUE (all single characters)
```

```
t_text({1, 2, "abc"}) -- FALSE (contains a sequence)
t_text({1, 2, 9.7})   -- FALSE (contains a non-integer)
t_text({1, 2, 'a'})   -- TRUE
t_text({1, -2, 'a'})  -- FALSE (contains a negative integer)
t_text({})            -- TRUE
```

## number\_array

```
include std/types.e
namespace types
public type number_array(object x)
```

tests for atom elements.

### Returns:

TRUE if argument is a sequence that only contains zero or more numbers.

### Example 1:

```
number_array(-1)      -- FALSE (not a sequence)
number_array("abc")   -- TRUE (all single characters)
number_array({1, 2, "abc"}) -- FALSE (contains a sequence)
number_array(1, 2, 9.7) -- TRUE
number_array(1, 2, 'a') -- TRUE
number_array({})      -- TRUE
```

## sequence\_array

```
include std/types.e
namespace types
public type sequence_array(object x)
```

tests for sequence with possible nested sequences.

### Returns:

TRUE if argument is a sequence that only contains zero or more sequences.

### Example 1:

```
sequence_array(-1)      -- FALSE (not a sequence)
sequence_array("abc")   -- FALSE (all single characters)
sequence_array({1, 2, "abc"}) -- FALSE (contains some atoms)
sequence_array({1, 2, 9.7}) -- FALSE
sequence_array({1, 2, 'a'}) -- FALSE
sequence_array({"abc", {3.4, 99182.78737}}) -- TRUE
sequence_array({})      -- TRUE
```

## ascii\_string

```
include std/types.e
namespace types
public type ascii_string(object x)
```

tests for ASCII elements.

### Returns:

TRUE if argument is a sequence that only contains zero or more ASCII characters.

### Comments:

An ASCII 'character' is defined as a integer in the range [0 to 127].

### Example 1:

```
ascii_string(-1)           -- FALSE (not a sequence)
ascii_string("abc")        -- TRUE (all single ASCII characters)
ascii_string({1, 2, "abc"}) -- FALSE (contains a sequence)
ascii_string({1, 2, 9.7})   -- FALSE (contains a non-integer)
ascii_string({1, 2, 'a'})   -- TRUE
ascii_string({1, -2, 'a'})  -- FALSE (contains a negative integer)
ascii_string({})           -- TRUE
```

## string

```
include std/types.e
namespace types
public type string(object x)
```

tests for a string sequence.

### Returns:

TRUE if argument is a sequence that only contains zero or more byte characters.

### Comments:

A byte 'character' is defined as a integer in the range [0 to 255].

### Example 1:

```
string(-1)           -- FALSE (not a sequence)
string("abc'6")       -- TRUE (all single byte characters)
string({1, 2, "abc'6"}) -- FALSE (contains a sequence)
string({1, 2, 9.7})    -- FALSE (contains a non-integer)
string({1, 2, 'a'})    -- TRUE
string({1, 2, 'a', 0}) -- TRUE (even though it contains a null byte)
string({1, -2, 'a'})   -- FALSE (contains a negative integer)
string({})            -- TRUE
```

## cstring

```
include std/types.e
namespace types
public type cstring(object x)
```

tests for a string sequence (that has no null character).

### Returns:

TRUE if argument is a sequence that only contains zero or more non-null byte characters.

### Comments:

A non-null byte 'character' is defined as a integer in the range [1 to 255].

### Example 1:

```

cstring(-1)           -- FALSE (not a sequence)
cstring("abc'6")       -- TRUE (all single byte characters)
cstring({1, 2, "abc'6"}) -- FALSE (contains a sequence)
cstring({1, 2, 9.7})   -- FALSE (contains a non-integer)
cstring({1, 2, 'a'})   -- TRUE
cstring({1, 2, 'a', 0}) -- FALSE (contains a null byte)
cstring({1, -2, 'a'})  -- FALSE (contains a negative integer)
cstring({})            -- TRUE

```

## INVALID\_ROUTINE\_ID

```

include std/types.e
namespace types
public constant INVALID_ROUTINE_ID

```

Value returned from `routine_id` when the routine does not exist or is out of scope. This is typically seen as -1 in legacy code.

## NO\_ROUTINE\_ID

```

include std/types.e
namespace types
public constant NO_ROUTINE_ID

```

To be used as a flag for no `routine_id` supplied.

## t\_integer32

```

include std/types.e
namespace types
public type t_integer32(object o)

```

tests for Euphoria integer.

### Returns:

TRUE if the argument is a valid 31-bit Euphoria integer.

### Comments:

This function is the same as `integer(o)` on 32-bit Euphoria, but is portable to 64-bit architectures.

# Utilities

## Subroutines

### iif

```
include std/utls.e
namespace utls
public function iif(atom test, object ifTrue, object ifFalse)
```

Used to embed an 'if' test inside an expression. iif stands for inline if or immediate if.

#### Arguments:

1. test : an atom, the result of a boolean expression
2. ifTrue : an object, returned if test is **non-zero**
3. ifFalse : an object, returned if test is zero

#### Returns:

An object. Either ifTrue or ifFalse is returned depending on the value of test.

#### Warning Note:

You must take care when using this function because just like all other Euphoria routines, this does not do any *lazy evaluation*. All parameter expressions are evaluated **before** the function is called, thus, it cannot be used when one of the parameters could fail to evaluate correctly. For example, this is an **improper** use of the iif statement:

```
first = iif(sequence(var), var[1], var)
```

The reason for this is that both var[1] and var will be evaluated. Therefore if var happens to be an atom, the var[1] statement will fail.

In situations like this, it is better to use the *long* style.

```
if sequence(var) then
    first = var[1]
else
    first = var
end if
```

#### Example 1:

```
msg = sprintf("%s: %s", {
    iif(ErrType = 'E', "Fatal error", "Warning"),
    errortext
})
```

# SEQUENCE

- `convert.e`
- `get.e`
- `search.e`
- `sequence.e`
- `serialize.e`
- `sort.e`

# Data Type Conversion

## Subroutines

### int\_to\_bytes

```
include std/convert.e
namespace convert
public function int_to_bytes(atom x, integer size = 4)
```

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

#### Arguments:

1. *x* : an atom, the value to convert.

#### Returns:

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

#### Comments:

If the atom does not fit into a 32-bit integer, things may still work right:

- If there is a fractional part, the first element in the returned value will carry it. If you poke the sequence to RAM, that fraction will be discarded anyway.
- If *x* is simply too big, the first three bytes will still be correct, and the 4th element will be  $\text{floor}(x/\text{power}(2,24))$ . If this is not a byte sized integer, some truncation may occur, but usually no error.

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

#### Example 1:

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

#### Example 2:

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

#### See Also:

[bytes\\_to\\_int](#) | [int\\_to\\_bits](#) | [atom\\_to\\_float64](#) | [poke4](#)

### bytes\_to\_int

```
include std/convert.e
namespace convert
public function bytes_to_int(sequence s)
```

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

#### Arguments:



1. `s` : the sequence to convert

### Returns:

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

### Comments:

This performs the reverse operation from [int\\_to\\_bytes](#)

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

### Example 1:

```
atom int32

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

### See Also:

[bits\\_to\\_int](#) | [float64\\_to\\_atom](#) | [int\\_to\\_bytes](#) | [peek](#) | [peek4s](#) | [peek4u](#) | [poke4](#)

## int\_to\_bits

```
include std/convert.e
namespace convert
public function int_to_bits(atom x, integer nbits = 32)
```

extracts the lower bits from an integer.

### Arguments:

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

### Returns:

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

### Comments:

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

The bits are returned lowest first.

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

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

### Example 1:

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

### See Also:

[bits\\_to\\_int](#) | [int\\_to\\_bytes](#) | [Relational operators](#) | [operations on sequences](#)

## bits\_to\_int

```
include std/convert.e
namespace convert
public function bits_to_int(sequence bits)
```

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

### Arguments:

1. bits : the sequence to convert.

### Returns:

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

### Comments:

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

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

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

### Example 1:

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

### See Also:

[bytes\\_to\\_int](#) | [int\\_to\\_bits](#) | [operations on sequences](#)

## atom\_to\_float64

```
include std/convert.e
namespace convert
public function atom_to_float64(atom a)
```

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

### Arguments:

1. a : the atom to convert:

### Returns:

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

### Comments:

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

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

### Example 1:

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

[See Also:](#)

[float64\\_to\\_atom](#) | [int\\_to\\_bytes](#) | [atom\\_to\\_float32](#)

## atom\_to\_float80

```
include std/convert.e
namespace convert
public function atom_to_float80(atom a)
```

## float80\_to\_atom

```
include std/convert.e
namespace convert
public function float80_to_atom(sequence bytes)
```

## atom\_to\_float32

```
include std/convert.e
namespace convert
public function atom_to_float32(atom a)
```

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

### Arguments:

1. `a` : the atom to convert:

### Returns:

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

### Comments:

Euphoria atoms can have values which are 64-bit IEEE floating-point numbers, so you may lose precision when you convert to 32-bits (16 significant digits versus 7). The range of exponents is much larger in 64-bit format (10 to the 308, versus 10 to the 38), so some atoms may be too large or too small to represent in 32-bit format. In this case you will get one of the special 32-bit values: `inf` or `-inf` (infinity or -infinity). To avoid this, you can use [atom\\_to\\_float64](#).

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

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

### Example 1:

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

[See Also:](#)

[float32\\_to\\_atom](#) | [int\\_to\\_bytes](#) | [atom\\_to\\_float64](#)

## float64\_to\_atom

```
include std/convert.e
namespace convert
public function float64_to_atom(sequence_8 ieee64)
```

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

### Arguments:

1. `ieee64` : the sequence to convert.

### Returns:

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

### Comments:

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

### Example 1:

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

### See Also:

[float32\\_to\\_atom](#) | [bytes\\_to\\_int](#) | [atom\\_to\\_float64](#)

## float32\_to\_atom

```
include std/convert.e
namespace convert
public function float32_to_atom(sequence_4 ieee32)
```

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

### Arguments:

1. `ieee32` : the sequence to convert.

### Returns:

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

### Comments:

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

### Example 1:

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

## See Also:

[float64\\_to\\_atom](#) | [bytes\\_to\\_int](#) | [atom\\_to\\_float32](#)

## hex\_text

```
include std/convert.e
namespace convert
public function hex_text(sequence text)
```

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

## Arguments:

1. text : the text to convert.

## Returns:

An **atom**, the numeric equivalent to text

## Comments:

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

## Example 1:

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

## See Also:

[value](#)

## set\_decimal\_mark

```
include std/convert.e
namespace convert
public function set_decimal_mark(integer new_mark)
```

gets, and possibly sets, the decimal mark that [to\\_number](#) uses.

## Arguments:

1. new\_mark : An integer: Either a comma (,), a period (.) or any other integer.

## Returns:

An **integer**, The current value, before new\_mark changes it.

## Comments:

- When new\_mark is a *period* it will cause to\_number to interpret a dot (.) as the decimal point symbol. The pre-changed value is returned.
- When new\_mark is a *comma* it will cause to\_number to interpret a comma (,) as the decimal point symbol. The pre-changed value is returned.

- Any other value does not change the current setting. Instead it just returns the current value.
- The initial value of the decimal marker is a period.

## to\_number

```
include std/convert.e
namespace convert
public function to_number(sequence text_in, integer return_bad_pos = 0)
```

converts the text into a number.

### Arguments:

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

### Returns:

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

### Comments:

1. You can supply **Hexadecimal** values if the value is preceded by a '#' character, **Octal** values if the value is preceded by a '@' character, and **Binary** values if the value is preceded by a '!' character. With hexadecimal values, the case of the digits 'A' - 'F' is not important. Also, any decimal marker embedded in the number is used with the correct base.
2. Any underscore characters or thousands separators, that are embedded in the text number are ignored. These can be used to help visual clarity for long numbers. The thousands separator is a ',' when the decimal mark is '.' (the default), or '.' if the decimal mark is ','. You inspect and set it using `set_decimal_mark()`.
3. You can supply a single leading or trailing sign. Either a minus (-) or plus (+).
4. You can supply one or more trailing adjacent percentage signs. The first one causes the resulting value to be divided by 100, and each subsequent one divides the result by a further 10. Thus 3845% gives a value of (3845 / 100) ==> 38.45, and 3845%% gives a value of (3845 / 1000) ==> 3.845.
5. You can have single currency symbol before the first digit or after the last digit. A currency symbol is any character of the string: "\$€£¥¢".
6. You can have any number of whitespace characters before the first digit and after the last digit.
7. The currency, sign and base symbols can appear in any order. Thus "\$ -21.10" is the same as " -\$21.10 ", which is also the same as "21.10\$-", and so on.
8. This function can optionally return information about invalid numbers. If `return_bad_pos` is not zero, a two-element sequence is returned. The first element is the converted number value, and the second is the position in the text where conversion stopped. If no errors were found then the second element is zero.
9. When converting floating point text numbers to atoms, you need to be aware that many numbers cannot be accurately converted to the exact value expected due to the limitations of the 64-bit IEEE Floating point format.

### Example 1:

```

object val
val = to_number("12.34")      ---> 12.34 -- No errors and no error return needed.
val = to_number("12.34", 1)   ---> {12.34, 0} -- No errors.
val = to_number("12.34", -1)  ---> 12.34 -- No errors.
val = to_number("12.34a", 1)  ---> {12.34, 6} -- Error at position 6
val = to_number("12.34a", -1) ---> {6} -- Error at position 6
val = to_number("12.34a")     ---> 0 because its not a valid number

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

```

## to\_integer

```

include std/convert.e
namespace convert
public function to_integer(object data_in, integer def_value = 0)

```

converts an object into a integer.

### Arguments:

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

### Returns:

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

### Comments:

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

### Example 1:

```

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

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

```

## to\_string

```

include std/convert.e
namespace convert
public function to_string(object data_in, integer string_quote = 0,
    integer embed_string_quote = '')

```

converts an object into a text string.

## Arguments:

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

## Returns:

A **sequence**. This is the string representation of `data_in`.

## Comments:

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

## Example 1:

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



# Input Routines

## Error Status Constants

These are returned from `get` and `value`.

### GET\_SUCCESS

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

### GET\_EOF

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

### GET\_FAIL

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

### GET\_NOTHING

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

## Answer Types

### GET\_SHORT\_ANSWER

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

### GET\_LONG\_ANSWER

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

## Subroutines

## get

```
include std/get.e
namespace stdget
public function get(integer file, integer offset = 0, integer answer = GET_SHORT_ANSWER)
```

reads from an open file a human-readable string of characters representing a Euphoria object. Converts the string into the numeric value of that object.

### Arguments:

1. file : an integer, the handle to an open file from which to read
2. offset : an integer, an offset to apply to file position before reading. Defaults to 0 zero.
3. answer : an integer, either GET\_SHORT\_ANSWER (the default) or GET\_LONG\_ANSWER.

### Returns:

A **sequence**, of length two (GET\_SHORT\_ANSWER) or four (GET\_LONG\_ANSWER) consisting of:

- an integer, the return status. This is any of:
  - GET\_SUCCESS -- object was read successfully
  - GET\_EOF -- end of file before object was read completely
  - GET\_FAIL -- object is not syntactically correct
  - GET\_NOTHING -- nothing was read, even a partial object string, before end of input
- an object, the value that was read. This is valid only if return status is GET\_SUCCESS.
- an integer, the number of characters read. On an error, this is the point at which the error was detected.
- an integer, the amount of initial whitespace read before the first active character was found

### Comments:

When answer is not specified, or explicitly GET\_SHORT\_ANSWER, only the first two items in the returned sequence are actually returned.

The GET\_NOTHING return status will not be returned if answer is GET\_SHORT\_ANSWER.

get can read arbitrarily complicated Euphoria objects. You could have a long sequence of values in braces and separated by commas and comments. For example: {23, {49, 57}, 0.5, -1, 99, 'A', "john"}. A single call to get will read in this entire sequence, return its value as a result, and return complementary information.

If a nonzero offset is supplied, it is interpreted as an offset to the current file position; the file will start seek from there first.

get returns a two or four item sequence; similar to what **value** returns:

- a status code ( success, error, end of file, no value at all )
- the value just read (meaningful only when the status code is GET\_SUCCESS) (optionally)
- the total number of characters read
- the amount of initial whitespace read.

Using the default value for answer, or setting it to GET\_SHORT\_ANSWER, returns two items. Setting it to GET\_LONG\_ANSWER causes four items to be returned.

Each call to get picks up where the previous call left off. For instance: a series of five calls to get would be needed to read in this sequence: `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.

After seeing -- something\nBut no value and the input stream stops right there, you will receive a status code of GET\_NOTHING, because nothing but whitespace or comments

was read. If you had opted for a short answer, you would get GET\_EOF instead.

Multiple "top-level" objects in the input stream must be separated from each other with one or more "whitespace" characters (blank, tab, \r, or \n). At the very least, a top level number must be followed by a white space from the following object. Whitespace is not necessary *within* a top-level object. Comments, terminated by either '\n' or '\r', are allowed anywhere inside sequences, and ignored if at the top level. A call to get will read one entire top-level object, plus possibly one additional (whitespace) character, after a top level number, even though the next object may have an identifiable starting point.

The combination of `print` and `get` can be used to save a Euphoria object to disk and later read it back. This technique could be used to implement a database as one or more large Euphoria sequences stored in disk files. The sequences could be read into memory, updated and then written back to disk after each series of transactions is complete. Remember to write out a whitespace character (using `puts`) after each call to `print`, at least when a top level number was just printed.

The value returned is not meaningful unless you have a GET\_SUCCESS status.

### Example 1:

```

                                include std/get.e
? get(0)
  -- At the prompt you type 77.5
  --> output is {0,77.5}
  -- that is {GET_SUCCESS, 77.5}

-- whereas gets(0) would return:
-- "77.5\n"
-- {55,55,46,53,10}

? gets(0)
  -- At the prompt you type 77.5
  --> output is {55,55,46,53,10}

puts(1, gets(0) )
  -- At the prompt you type 77.5
  --> you see 77.5
  -- that is "77.5\n"
```

### Example 2:

See `.../euphoria/demo/mydata.ex`

### See Also:

`value`

## value

```

include std/get.e
namespace stdget
public function value(sequence st, integer start_point = 1, integer answer = GET_SHORT_ANSWER)
```

reads, from a string, a human-readable string of characters representing a Euphoria object. Converts the string into the numeric value of that object.

### Arguments:

1. `st` : a sequence, from which to read text
2. `offset` : an integer, the position at which to start reading. Defaults to 1.
3. `answer` : an integer, either GET\_SHORT\_ANSWER (the default) or GET\_LONG\_ANSWER.

### Returns:

A **sequence**, of length two (GET\_SHORT\_ANSWER) or four (GET\_LONG\_ANSWER) made of:

- an integer, the return status. This is any of
  - GET\_SUCCESS -- object was read successfully
  - GET\_EOF -- end of file before object was read completely
  - GET\_FAIL -- object is not syntactically correct
  - GET\_NOTHING -- nothing was read, even a partial object string, before end of input
- an object, the value that was read. This is valid only if return status is GET\_SUCCESS.
- an integer, the number of characters read. On an error, this is the point at which the error was detected.
- an integer, the amount of initial whitespace read before the first active character was found

### Comments:

When answer is not specified, or explicitly GET\_SHORT\_ANSWER, only the first two items in the returned sequence are actually returned.

This works the same as `get` but it reads from a string that you supply, rather than from a file or device.

After reading one valid representation of a Euphoria object value will stop reading and ignore any additional characters in the string. For example: "36" and "36P" will both give you {GET\_SUCCESS, 36}.

The function returns {return\_status, value} if the answer type is not passed or set to GET\_SHORT\_ANSWER. If set to GET\_LONG\_ANSWER, the number of characters read and the amount of leading whitespace are returned in 3rd and 4th position. The GET\_NOTHING return status can occur only on a long answer.

### Example 1:

```

                                include std/get.e
                                sequence s

    s = value("12345")
    ? s
--> s is {0,          12345}
--      {GET_SUCCESS, 12345}

```

### Example 2:

```

                                include std/get.e
                                sequence s

    s = value("{0, 1, -99.9}")
    ? s
--> s is {
--      0,
--      {0,1,-99.9}
--      }
-- that is {GET_SUCCESS, {0, 1, -99.9}}

```

### Example 3:

```

                                include std/get.e
                                sequence s

    s = value("+++")
    ? s
--> s is {1,0}
-- that is {GET_FAIL, 0}

```

### See Also:

[get](#) | [defaulted\\_value](#)

## defaulted\_value

```
include std/get.e
namespace stdget
public function defaulted_value(object st, object def, integer start_point = 1)
```

calls the value function and returns the resulting value on success or the default default on failure.

### Arguments:

1. st : object to retrieve value from.
2. def : the value returned if st is an atom or value(st) fails.
3. start\_point : an integer, the position inst at which to start getting the value from. Defaults to 1

### Returns:

- If st, is an atom then def is returned.
- If calling value(st) is a success. then value()[2], otherwise it will return the parameter def.

### Example 1:

```
include std/get.e
object x

x = defaulted_value("10", 0)
? x
--> x is 10

x = defaulted_value("abc", 39)
? x
--> x is 39

x = defaulted_value(12, 42)
? x
--> x is 42

x = defaulted_value("{1,2}", 42)
? x
--> x is {1,2}
```

### See Also:

value

# Searching

## Equality

### compare

```
<built-in> function compare(object compared, object reference)
```

compares two items returning less than, equal or greater than.

#### Arguments:

1. compared : the compared object
2. reference : the reference object

#### Returns:

An **integer**,

- 0 -- if objects are identical
- 1 -- if compared is greater than reference
- -1 -- if compared is less than reference

#### Comments:

Atoms are considered to be less than sequences. Sequences are compared alphabetically starting with the first element until a difference is found or one of the sequences is exhausted. Atoms are compared as ordinary reals.

#### Example 1:

```
x = compare({1,2,{3,{4}}},5}, {2-1,1+1,{3,{4}},6-1})
-- identical, x is 0
```

#### Example 2:

```
if compare("ABC", "ABCD") < 0 then -- -1
  -- will be true: ABC is "less" because it is shorter
end if
```

#### Example 3:

```
x = compare('a', "a")
-- x will be -1 because 'a' is an atom
-- while "a" is a sequence
```

#### See Also:

[equal](#) | [relational operators](#) | [operations on sequences](#) | [sort](#)

### equal

```
<built-in> function equal(object left, object right)
```

compares two Euphoria objects to see if they are the same.

#### Arguments:

1. left : one of the objects to test
2. right : the other object

### Returns:

An **integer**, 1 if the two objects are identical, else 0.

### Comments:

This is equivalent to the expression: `compare(left, right) = 0`.

This routine, like most other built-in routines, is very fast. It does not have any subroutine call overhead.

### Example 1:

```
if equal(PI, 3.14) then
    puts(1, "give me a better value for PI!\n")
end if
```

### Example 2:

```
if equal(name, "George") or equal(name, "GEORGE") then
    puts(1, "name is George\n")
end if
```

### See Also:

[compare](#)

## Finding

### find

```
<built-in> function find(object needle, sequence haystack, integer start)
```

finds the first occurrence of a "needle" as an element of a "haystack", starting from position "start".

### Arguments:

1. needle : an object whose presence is being queried
2. haystack : a sequence, which is being looked up for needle
3. start : an integer, the position at which to start searching. Defaults to 1.

### Returns:

An **integer**, 0 if needle is not on haystack, else the smallest index of an element of haystack that equals needle.

### Example 1:

```
location = find(11, {5, 8, 11, 2, 3})
-- location is set to 3
```

### Example 2:

```
names = {"fred", "rob", "george", "mary", ""}
location = find("mary", names)
-- location is set to 4
```

### See Also:

[find](#) | [match](#) | [compare](#)

## find\_from

```
<built-in> function find_from(object needle, object haystack, integer start)
```

### Deprecated:

Deprecated since version 4.0.0

In Euphoria 4.0.0 we have the ability to default parameters to procedures and functions. The built-in [find](#) therefore now has a start parameter that is defaulted to the beginning of the sequence. Thus, [find](#) can perform the identical functionality provided by `find_from`. In an undetermined future release of Euphoria, `find_from` will be removed.

### See Also:

[find](#)

## find\_any

```
include std/search.e
namespace search
public function find_any(object needles, sequence haystack, integer start = 1)
```

finds any element from a list inside a sequence. Returns the location of the first hit.

### Arguments:

1. `needles` : a sequence, the list of items to look for
2. `haystack` : a sequence, in which "needles" are looked for
3. `start` : an integer, the starting point of the search. Defaults to 1.

### Returns:

An **integer**, the smallest index in haystack of an element of needles, or 0 if no needle is found.

### Comments:

This function may be applied to a string sequence or a complex sequence.

### Example 1:

```
location = find_any("aeiou", "John Smith", 3)
-- location is 8
```

### Example 2:

```
location = find_any("aeiou", "John Doe")
-- location is 2
```

### See Also:

[find](#)

## match\_any



```
include std/search.e
namespace search
public function match_any(sequence needles, sequence haystack, integer start = 1)
```

determines if any element from needles is in haystack.

### Arguments:

1. needles : a sequence, the list of items to look for
2. haystack : a sequence, in which "needles" are looked for
3. start : an integer, the starting point of the search. Defaults to 1.

### Returns:

An **integer**, 0 if no matches, 1 if any matches.

### Comments:

This function may be applied to a string sequence or a complex sequence. An empty needles sequence will always result in 0.

### Example 1:

```
ok = match_any("aeiou", "John Smith")
-- okay is 1
ok = match_any("xyz", "John Smith" )
-- okay is 0
```

### See Also:

[find\\_any](#)

## find\_each

```
include std/search.e
namespace search
public function find_each(sequence needles, sequence haystack, integer start = 1)
```

finds all instances of any element from the needle sequence that occur in the haystack sequence. Returns a list of indexes.

### Arguments:

1. needles : a sequence, the list of items to look for
2. haystack : a sequence, in which "needles" are looked for
3. start : an integer, the starting point of the search. Defaults to 1.

### Returns:

A **sequence**, the list of indexes into haystack that point to an element that is also in needles.

### Comments:

This function may be applied to a string sequence or a complex sequence.

### Example 1:

```
location = find_each("aeiou", "John Smith", 3)
-- location is {8}
```

### Example 2:

```
location = find_each("aeiou", "John Doe")  
-- location is {2,7,8}
```

### See Also:

[find](#) | [find\\_any](#)

## find\_all

```
include std/search.e  
namespace search  
public function find_all(object needle, sequence haystack, integer start = 1)
```

finds all occurrences of an object inside a sequence, starting at some specified point.

### Arguments:

1. needle : an object, what to look for
2. haystack : a sequence to search in
3. start : an integer, the starting index position (defaults to 1)

### Returns:

A **sequence**, the list of all indexes no less than start of elements of haystack that equal needle. This sequence is empty if no match found.

### Example 1:

```
s = find_all('A', "ABCABAB")  
-- s is {1,4,6}
```

### See Also:

[find](#) | [match](#) | [match\\_all](#)

## find\_all\_but

```
include std/search.e  
namespace search  
public function find_all_but(object needle, sequence haystack, integer start = 1)
```

finds all non-occurrences of an object inside a sequence, starting at some specified point.

### Arguments:

1. needle : an object, what to look for
2. haystack : a sequence to search in
3. start : an integer, the starting index position (defaults to 1)

### Returns:

A **sequence**, the list of all indexes no less than start of elements of haystack that not equal to needle. This sequence is empty if haystack only consists of needle.

### Example 1:

```
s = find_all_but('A', "ABCABAB")  
-- s is {2,3,5,7}
```

## See Also:

[find\\_all](#) | [match](#) | [match\\_all](#)

### NESTED\_ANY

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

### NESTED\_ALL

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

### NESTED\_INDEX

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

### NESTED\_BACKWARD

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

### find\_nested

```
include std/search.e
namespace search
public function find_nested(object needle, sequence haystack, integer flags = 0,
                           integer rtn_id = types:NO_ROUTINE_ID)
```

finds any object (among a list) in a sequence of arbitrary shape at arbitrary nesting.

## Arguments:

1. `needle` : an object, either what to look up, or a list of items to look up
2. `haystack` : a sequence, where to look up
3. `flags` : options to the function, see Comments section. Defaults to 0.
4. `routine` : an integer, the `routine_id` of an user supplied `equal/find` function. Defaults to `types:NO_ROUTINE_ID`.

## Returns:

A possibly empty **sequence**, of results, one for each hit.

## Comments:

Each item in the returned sequence is either a sequence of indexes, or a pair {sequence of indexes, index in needle}.

The following flags are available to fine tune the search:

- `NESTED_BACKWARD` -- if on flags, search is performed backward. Default is forward.
- `NESTED_ALL` -- if on flags, all occurrences are looked for. Default is one hit only.
- `NESTED_ANY` -- if present on flags, needle is a list of items to look for. Not the default.
- `NESTED_INDEXES` -- if present on flags, an individual result is a pair {position, index in needle}. Default is just return the position.

If `s` is a single index list, or position, from the returned sequence, then `fetch(haystack, s) = needle`.

If a routine id is supplied, the routine must behave like `equal` if the `NESTED_ANY` flag is not supplied, and like `find` if it is. The routine is being passed the current haystack item and needle. The returned integer is interpreted as if returned by `equal` or `find`.

If the `NESTED_ANY` flag is specified, and needle is an atom, then the flag is removed.

### Example 1:

```
sequence s = find_nested(3, {5, {4, {3, {2}}}})
-- s is {2 ,2 ,1}
```

### Example 2:

```
sequence s = find_nested({3, 2}, {1, 3, {2,3}},
                        NESTED_ANY + NESTED_BACKWARD + NESTED_ALL)
-- s is {{3,2}, {3,1}, {2}}
```

### Example 3:

```
sequence s = find_nested({3, 2}, {1, 3, {2,3}},
                        NESTED_ANY + NESTED_INDEXES + NESTED_ALL)
-- s is {{2}, 1}, {{3, 1}, 2}, {{3, 2}, 1}}
```

### See Also:

`find` | `rfind` | `find_any` | `fetch`

## rfind

```
include std/search.e
namespace search
public function rfind(object needle, sequence haystack, integer start = length(haystack))
```

finds a needle in a haystack in reverse order.

### Arguments:

1. `needle` : an object to search for
2. `haystack` : a sequence to search in
3. `start` : an integer, the starting index position (defaults to `length(haystack)`)

### Returns:

An **integer**, 0 if no instance of `needle` can be found on `haystack` before index `start`, or the highest such index otherwise.

### Comments:

If `start` is less than 1, it will be added once to `length(haystack)` to designate a position counted backwards. Thus, if `start` is -1, the first element to be queried in `haystack` will be `haystack[$-1]`, then `haystack[$-2]` and so on.

### Example 1:

```
location = rfind(11, {5, 8, 11, 2, 11, 3})
-- location is set to 5
```

### Example 2:

```
names = {"fred", "rob", "rob", "george", "mary"}
location = rfind("rob", names)
-- location is set to 3
location = rfind("rob", names, -4)
-- location is set to 2
```

### See Also:

[find](#) | [rmatch](#)

## find\_replace

```
include std/search.h
namespace search
public function find_replace(object needle, sequence haystack, object replacement,
                           integer max = 0)
```

finds a needle in the haystack, and replaces all or upto max occurrences with replacement.

### Arguments:

1. needle : an object to search and perhaps replace
2. haystack : a sequence to be inspected
3. replacement : an object to substitute for any (first) instance of needle
4. max : an integer, 0 to replace all occurrences

### Returns:

A **sequence**, the modified haystack.

### Comments:

Replacements will not be made recursively on the part of haystack that was already changed.

If max is 0 or less, any occurrence of needle in haystack will be replaced by replacement. Otherwise, only the first max occurrences are.

### Example 1:

```
s = find_replace('b', "The batty book was all but in Canada.", 'c', 0)
-- s is "The catty cook was all cut in Canada."
```

### Example 2:

```
s = find_replace('/', "/euphoria/demo/unix", '\\', 2)
-- s is "\\euphoria\\demo/unix"
```

### Example 3:

```
s = find_replace("theater", { "the", "theater", "theif" }, "theatre")
-- s is { "the", "theatre", "theif" }
```

### See Also:

[find](#) | [replace](#) | [match\\_replace](#)

## match\_replace

```
include std/search.e
namespace search
public function match_replace(object needle, sequence haystack, object replacement,
                             integer max = 0)
```

finds a "needle" in a "haystack", and replace any, or only the first few, occurrences with a replacement.

### Arguments:

1. needle : an non-empty sequence or atom to search and perhaps replace
2. haystack : a sequence to be inspected
3. replacement : an object to substitute for any (first) instance of needle
4. max : an integer, 0 to replace all occurrences

### Returns:

A **sequence**, the modified haystack.

### Comments:

Replacements will not be made recursively on the part of haystack that was already changed.

If max is 0 or less, any occurrence of needle in haystack will be replaced by replacement. Otherwise, only the first max occurrences are.

If either needle or replacement are atoms they will be treated as if you had passed in a length-1 sequence containing the said atom.

If needle is an empty sequence, an error will be raised and your program will exit.

### Example 1:

```
s = match_replace("the", "the cat ate the food under the table", "THE", 0)
-- s is "THE cat ate THE food under THE table"
```

### Example 2:

```
s = match_replace("the", "the cat ate the food under the table", "THE", 2)
-- s is "THE cat ate THE food under the table"
```

### Example 3:

```
s = match_replace('/', "/euphoria/demo/unix", '\\', 2)
-- s is "\\euphoria\\demo/unix"
```

### Example 4:

```
s = match_replace('a', "abracadabra", 'X')
-- s is now "XbrXcXdXbrX"
s = match_replace("ra", "abracadabra", 'X')
-- s is now "abXcadabX"
s = match_replace("a", "abracadabra", "aa")
-- s is now "aabraacaadaabraa"
s = match_replace("a", "abracadabra", "")
-- s is now "brcdbr"
```

### See Also:

[find](#) | [replace](#) | [regex:find\\_replace](#) | [find\\_replace](#)

## binary\_search

```
include std/search.e
namespace search
public function binary_search(object needle, sequence haystack, integer start_point = 1,
    integer end_point = 0)
```

finds a "needle" in an ordered "haystack". Start and end point can be given for the search.

### Arguments:

1. needle : an object to look for
2. haystack : a sequence to search in
3. start\_point : an integer, the index at which to start searching. Defaults to 1.
4. end\_point : an integer, the end point of the search. Defaults to 0, ie search to end.

### Returns:

An **integer**, either:

1. a positive integer *i*, which means haystack[*i*] equals needle.
2. a negative integer, -*i*, with *i* between adjusted start and end points. This means that needle is not in the searched slice of haystack, but would be at index *i* if it were there.
3. a negative integer -*i* with *i* out of the searched range. This means that needle might be either below the start point if *i* is below the start point, or above the end point if *i* is.

### Comments:

- If end\_point is not greater than zero, it is added to length(haystack) once only. Then, the end point of the search is adjusted to length(haystack) if out of bounds.
- The start point is adjusted to 1 if below 1.
- The way this function returns is very similar to what [db\\_find\\_key](#) does. They use variants of the same algorithm. The latter is all the more efficient as haystack is long.
- haystack is assumed to be in ascending order. Results are undefined if it is not.
- If duplicate copies of needle exist in the range searched on haystack, any of the possible contiguous indexes may be returned.

### See Also:

[find](#) | [db\\_find\\_key](#)

## Matching

### match

```
<built-in> function match(sequence needle, sequence haystack, integer start)
```

tries to match a "needle" against some slice of a "haystack", starting at position "start".

### Arguments:

1. needle : a sequence whose presence as a "substring" is being queried
2. haystack : a sequence, which is being looked up for needle as a sub-sequence
3. start : an integer, the point from which matching is attempted. Defaults to 1.

### Returns:

An **integer**, 0 if no slice of haystack is needle, else the smallest index at which such a slice

starts.

### Comments:

If needle is an empty sequence, an error is raised and your program will exit.

### Example 1:

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

### See Also:

[find](#) | [compare](#) | [wildcard:is\\_match](#)

## match\_from

```
<built-in> function match_from(sequence needle, sequence haystack, integer start)
```

### Deprecated:

Deprecated since version 4.0.0

In Euphoria 4.0.0 we have the ability to default parameters to procedures and functions. The built-in `match` therefore now has a start parameter that is defaulted to the beginning of the sequence. Thus, `match` can perform the identical functionality provided by `match_from`. In an undetermined future release of Euphoria, `match_from` will be removed.

### Comments:

If needle is an empty sequence, an error is raised and your program will exit.

### See Also:

[match](#)

## match\_all

```
include std/search.e
namespace search
public function match_all(sequence needle, sequence haystack, integer start = 1)
```

matches all items of haystack in needle.

### Arguments:

1. needle : a non-empty sequence, what to look for
2. haystack : a sequence to search in
3. start : an integer, the starting index position (defaults to 1)

### Returns:

A **sequence**, of integers, the list of all lower indexes, not less than `start`, of all slices in haystack that equal needle. The list may be empty.

### Comments:

If needle is an empty sequence, an error will be raised and your program will exit.

### Example 1:



```
s = match_all("the", "the dog chased the cat under the table.")
-- s is {1,16,30}
```

### See Also:

[match](#) | [regex::find\\_all](#) | [find](#) | [find\\_all](#)

## rmatch

```
include std/search.e
namespace search
public function rmatch(sequence needle, sequence haystack, integer start = length(haystack))
```

tries to match a needle against some slice of a haystack in reverse order.

### Arguments:

1. `needle` : a sequence to search for
2. `haystack` : a sequence to search in
3. `start` : an integer, the starting index position (defaults to `length(haystack)`)

### Returns:

An **integer**, either 0 if no slice of `haystack` starting before `start` equals `needle`, else the highest lower index of such a slice.

### Comments:

If `start` is less than 1, it will be added once to `length(haystack)` to designate a position counted backwards. Thus, if `start` is -1, the first element to be queried in `haystack` will be `haystack[$-1]`, then `haystack[$-2]` and so on.

If a `needle` is an empty sequence this will return 0.

### Example 1:

```
location = rmatch("the", "the dog ate the steak from the table.")
-- location is set to 28 (3rd 'the')
location = rmatch("the", "the dog ate the steak from the table.", -11)
-- location is set to 13 (2nd 'the')
```

### See Also:

[rfind](#) | [match](#)

## begins

```
include std/search.e
namespace search
public function begins(object sub_text, sequence full_text)
```

tests whether a sequence is the head of another one.

### Arguments:

1. `sub_text` : an object to be looked for
2. `full_text` : a sequence, the head of which is being inspected.

### Returns:

An **integer**, 1 if sub\_text begins full\_text, else 0.

### Comments:

If sub\_text is an empty sequence, this returns 1 unless full\_text is also an empty sequence. When they are both empty sequences this returns 0.

### Example 1:

```
s = begins("abc", "abcdef")
-- s is 1
s = begins("bcd", "abcdef")
-- s is 0
```

### See Also:

[ends](#) | [head](#)

## ends

```
include std/search.e
namespace search
public function ends(object sub_text, sequence full_text)
```

tests whether a sequence ends another one.

### Arguments:

1. sub\_text : an object to be looked for
2. full\_text : a sequence, the tail of which is being inspected.

### Returns:

An **integer**, 1 if sub\_text ends full\_text, else 0.

### Comments:

If sub\_text is an empty sequence, this returns 1 unless full\_text is also an empty sequence. When they are both empty sequences this returns 0.

### Example 1:

```
s = ends("def", "abcdef")
-- s is 1
s = begins("bcd", "abcdef")
-- s is 0
```

### See Also:

[begins](#) | [tail](#)

## is\_in\_range

```
include std/search.e
namespace search
public function is_in_range(object item, sequence range_limits, sequence boundries = "[ ]")
```

tests to see if the item is in a range of values supplied by range\_limits.

### Arguments:

1. `item` : The object to test for.
2. `range_limits` : A sequence of two or more elements. The first is assumed to be the smallest value and the last is assumed to be the highest value.
3. `boundries`: a sequence. This determines if the range limits are inclusive or not. Must be one of `[]` (the default), `[]`, `[]`, or `[]`.

### Returns:

An **integer**, 0 if `item` is not in the `range_limits` otherwise it returns 1.

### Comments:

- In `boundries`, square brackets mean *inclusive* and round brackets mean *exclusive*. Thus `[]` includes both limits in the range, while `[]` excludes both limits. And, `[]` includes the lower limit and excludes the upper limits while `[]` does the reverse.

### Example 1:

```
if is_in_range(2, {2, 75}) then
    procA(user_data) -- Gets run (both limits included)
end if
if is_in_range(2, {2, 75}, "[]") then
    procA(user_data) -- Does not get run
end if
```

## is\_in\_list

```
include std/search.e
namespace search
public function is_in_list(object item, sequence list)
```

tests to see if the `item` is in a list of values supplied by `list`.

### Arguments:

1. `item` : The object to test for.
2. `list` : A sequence of elements that `item` could be a member of.

### Returns:

An **integer**, 0 if `item` is not in the `list`, otherwise it returns 1.

### Example 1:

```
if is_in_list(user_data, {100, 45, 2, 75, 121}) then
    procA(user_data)
end if
```

## lookup

```
include std/search.e
namespace search
public function lookup(object find_item, sequence source_list, sequence target_list,
    object def_value = 0)
```

returns the corresponding element from the `target_list` if the supplied `item` is in the `source_list`.

### Arguments:

1. `find_item`: an object that might exist in `source_list`.

2. `source_list`: a sequence that might contain `plTem`.
3. `target_list`: a sequence from which the corresponding item will be returned.
4. `def_value`: an object (defaults to zero). This is returned when `find_item` is not in `source_list` **and** `target_list` is not longer than `source_list`.

### Returns:

#### An **object**

- If `find_item` is found in `source_list` then this is the corresponding element from `target_list`
- If `find_item` is not in `source_list` then if `target_list` is longer than `source_list` then the last item in `target_list` is returned otherwise `def_value` is returned.

### Example 1:

```
lookup('a', "cat", "dog") --> 'o'
lookup('d', "cat", "dogx") --> 'x'
lookup('d', "cat", "dog") --> 0
lookup('d', "cat", "dog", -1) --> -1
lookup("ant", {"ant","bear","cat"}, {"spider","seal","dog","unknown"})
    --> "spider"
lookup("dog", {"ant","bear","cat"}, {"spider","seal","dog","unknown"})
    --> "unknown"
```

## vlookup

```
include std/search.e
namespace search
public function vlookup(object find_item, sequence grid_data, integer source_col,
    integer target_col, object def_value = 0)
```

returns the corresponding element from the target column if the supplied item is in a source grid column.

### Arguments:

1. `find_item`: an object that might exist in `source_col`.
2. `grid_data`: a 2D grid sequence that might contain `plTem`.
3. `source_col`: an integer. The column number to look for `find_item`.
4. `target_col`: an integer. The column number from which the corresponding item will be returned.
5. `def_value`: an object (defaults to zero). This is returned when `find_item` is not found in the `source_col` column, or if found but the target column does not exist.

### Comments:

- If a row in the grid is actually a single atom, the row is ignored.
- If a row's length is less than the `source_col`, the row is ignored.

### Returns:

#### An **object**,

- If `find_item` is found in the `source_col` column then this is the corresponding element from the `target_col` column.

### Example 1:

```
sequence grid
grid = {
    {"ant", "spider", "mortein"},
    {"bear", "seal", "gun"},
    {"cat", "dog", "ranger"},
    $
}
```

```
}  
vlookup("ant", grid, 1, 2, "?") --> "spider"  
vlookup("ant", grid, 1, 3, "?") --> "mortein"  
vlookup("seal", grid, 2, 3, "?") --> "gun"  
vlookup("seal", grid, 2, 1, "?") --> "bear"  
vlookup("mouse", grid, 2, 3, "?") --> "?"
```

# Sequence Manipulation

```


```

```

binop_ok }
length  } stats
sim_index }
```

```

2
```

```

binop_ok }
length  } stats —
sim_index }
```

```

fetch
store
valid_index
```

## Constants

### enum

```

include std/sequence.e
namespace stdseq
public enum
```

```

| ADD_PREPEND | ADD_APPEND | ADD_SORT_UP | ADD_SORT_DOWN |
```

### ROTATE\_LEFT

```

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

### ROTATE\_RIGHT

```

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

### See Also:

[rotate](#)

## Basic Subroutines

### binop\_ok

```

include std/sequence.e
namespace stdseq
public function binop_ok(object a, object b)
```

checks whether two objects can perform a sequence operation together.

### Arguments:

1. a : one of the objects to test for compatible shape
2. b : the other object

### Returns:

An **integer**,

- 1 (*true*)  
if a sequence operation is valid between a and b
- 0 (*false*)  
not valid

**Example 1:**

```
include std/sequence.e
integer i

i = binop_ok({1,2,3},{4,5})
--> is 0 (false)

i = binop_ok({1,2,3},4)
--> is 1 (true)

i = binop_ok({1,2,3},{4,{5,6},7})
-- is 1 (true)
```

**See Also:**

[series](#)

**fetch**

```
include std/sequence.e
namespace stdseq
public function fetch(sequence source, sequence indexes)
```

retrieves an item or slice using index values instead of subscript notation.

**Arguments:**

1. source : the sequence from which to fetch
2. indexes : a sequence of integers, the path to follow to reach the item or slice to return.

**Returns:**

An **object**,

item	which is source[indexes[1]][indexes[2]]...[indexes[\$]]
slice	source[indexes[1]][indexes[2]]...[indexes[\$]][1..\$]

**Errors:**

If the path cannot be followed to its end an error about reading a nonexistent element or subscripting an atom will occur.

**Comments:**

The last element of indexes may be a pair {lower,upper}, in which case a slice of the innermost referenced sequence is returned.

**Example 1:**

```
x = fetch({0,1,2,3,{"abc","def","ghi"}},6},{5,2,3})
-- x is 'f', or 102.
```

**Example 2:**

```
include std/sequence.e
include std/console.e
object x

sequence s = {0,1,2,3,{"abc","def","ghi"},6}

x = fetch(s, {5,1,2})
--> 98 or 'b'

x = s[5][1][2]
--> 98 or 'b'
```

**Example 3:**

```
include std/sequence.e
include std/console.e
object x

sequence s = {0,1,2,3,{"abc","def","ghi"},6}
x = fetch(s, {5,1,{1,3}})

display( x )
--> x is "abc"
```

## See Also:

[store](#) | [Subscripting of Sequences](#)

## store

```
include std/sequence.e
namespace stdseq
public function store(sequence target, sequence indexes, object x)
```

stores something at a location using index values instead of subscript notation.

## Arguments:

1. target : the sequence in which to store something
2. indexes : a sequence of integers, the path to follow to reach the place where to store
3. x : the object to store.

## Returns:

A **sequence**,

a copy of target

with the specified location indexes modified by storing x

## Errors:

If the path to storage location cannot be followed to its end, or an index is not what one would expect or is not valid, an error about illegal sequence operations will occur.

## Comments:

If the last item of indexes is a pair of integers then x will be stored as a slice; the bounding indexes being given in the pair as {lower,upper}.

In Euphoria you can never modify an object by passing it to a subroutine. You have to get a modified copy and then assign it back to the original.

## Example 1:

```
include std/sequence.e
include std/console.e
sequence s

s = {0,1,2,3,{"abc","def","ghi"},6}
s = store(s, {5,2,3}, 108)
display( s )

-- s is {0,1,2,3,{"abc","del","ghi"},6}
```

## Example 2:

```
include std/sequence.e
include std/console.e
sequence s

s = {0,1,2,3,{"abc","def","ghi"},6}
s = store(s, {5,2,{1,3}}, "XXX" )
display( s )

-- s is {0,1,2,3,{"abc","XXX","ghi"},6}
```

## See Also:

[fetch](#) | [Subscripting of Sequences](#)

## valid\_index

```
include std/sequence.e
namespace stdseq
public function valid_index(sequence st, object x)
```

checks whether an index exists in a sequence.

## Arguments:



1. `s` : the sequence for which to check
2. `x` : an object, the index to check.

**Returns:**An **integer**,

- 1 (*true*)  
if `s[x]` makes sense
- 0 (*false*)  
invalid index

**Example 1:**

```
include std/sequence.e
integer i

i = valid_index({51,27,33,14},2)
? i
--> i is 1
```

**See Also:**[Subscripting of Sequences](#)**rotate**

```
include std/sequence.e
namespace stdseq
public function rotate(sequence source, integer shift, integer start = 1,
    integer stop = length(source))
```

rotates items in a sequence or slice.

**Arguments:**

1. `source` : sequence to be rotated
2. `shift` : direction and count to be shifted (`ROTATE_LEFT` or `ROTATE_RIGHT`)
3. `start` : starting position for shift, defaults to 1
4. `stop` : stopping position for shift, defaults to `length(source)`

**Returns:**A **sequence**,

- `source`  
re-arranged by the rotation

**Comments:**

Use `amount * direction` to specify the shift; direction is either `ROTATE_LEFT` or `ROTATE_RIGHT`. This enables to shift multiple places in a single call. For instance, use `ROTATE_LEFT * 5` to rotate left, 5 positions.

A null shift does nothing and returns source unchanged.

**Example 1:**

```
include std/sequence.e
sequence s

s = rotate({1, 2, 3, 4, 5}, ROTATE_LEFT )
? s
-- s is {2, 3, 4, 5, 1}
```

**Example 2:**

```
include std/sequence.e
sequence s

s = rotate({1, 2, 3, 4, 5}, ROTATE_RIGHT * 2)
? s
-- s is {4, 5, 1, 2, 3}
```

**Example 3:**

```
include std/sequence.e
sequence s

s = rotate({11,13,15,17,19,23}, ROTATE_LEFT, 2, 5)
```

```
? s
-- s is {11,15,17,19,13,23}
```

#### Example 4:

```
include std/sequence.e
sequence s

s = rotate({11,13,15,17,19,23}, ROTATE_RIGHT, 2, 5)
? s
-- s is {11,19,13,15,17,23}
```

#### See Also:

[slice](#) | [head](#) | [tail](#)

## columnize

```
include std/sequence.e
namespace stdseq
public function columnize(sequence source, object cols = {}, object defval = 0)
```

converts a set of sub sequences into a set of "columns."

#### Arguments:

1. `source` : sequence containing the sub-sequences
2. `cols` : either a specific column number or a set of column numbers. Default is 0 which returns the maximum number of columns.
3. `defval` : an object. Used when a column value is not available. Default is 0

#### Returns:

A **sequence**,

column view  
 taken from source

#### Errors:

Crashes for an invalid `cols` argument.

#### Comments:

Any atoms found in `source` are treated as if they are a one-item sequence.

If `cols` is greater than the length of all items in `source` then a *blank column* such as {0,0,0} will be returned.

#### Example 1:

```
include std/sequence.e
sequence s

s = {
  {1, 2},
  {3, 4},
  {5, 6}
}

s = columnize( s )
? s
-- s is {
--   {1,3,5},
--   {2,4,6}
-- }
```

#### Example 2:

```
include std/sequence.e
sequence s

s = { {1, 2},
      {3, 4},
      {5, 6, 7} }

? columnize( s )
-- default 'filler' is '0'
```

```

--> s is { {1,3,5},
--         {2,4,6},
--         {0,0,7} }

? columnize( s, ,-999 )
-- custom 'filler' is '-999'
--> s is { {1,3,5},
--         {2,4,6},
--         {-999,-999,7} }

```

### Example 3:

```

include std/sequence.e
sequence s

s = { {1, 2},
      {3, 4},
      {5, 6, 7} }

? columnize( s, 2)
--> s is { {2,4,6} }
-- column 2 only

```

### Example 4:

```

include std/sequence.e
sequence s

s = { {1, 2},
      {3, 4},
      {5, 6, 7} }

? columnize( s, {2,1} )
--> s is { {2,4,6},
--         {1,3,4} }
-- column 2 then column 1

```

### Example 5:

```

include std/sequence.e
include std/console.e
sequence s

s = {"abc", "def", "ghi"}
console:display( columnize(s) )

--> s is { "adg",
--         "beh",
--         "cfi" }

```

### See Also:

[vslice](#)

## apply

```

include std/sequence.e
namespace stdseq
public function apply(sequence source, integer rid, object userdata = {})

```

applies a function to every item of a sequence returning a new sequence of the same size.

### Arguments:

- `source` : the sequence to map
- `rid` : the routine\_id of the function to use as converter
- `userdata` : an object passed to each invocation of `rid`. If omitted, `{}` is used.

### Returns:

A **sequence**,

with mapped items

Returned sequence is the same length as `source`. Each item corresponds to an item in `source` mapped after the function named by `rid` returns its value.

### Comments:

The supplied function must take two arguments. The data-type of the first argument must be compatible with every item in `source`. The second argument is an object containing `userdata` that is used by the mapping function.

**Example 1:**

```
include std/sequence.e
include std/console.e
sequence s

function greeter(object o, object d)
    return o[1] & ", " & o[2] & d
end function

s = {{"Hello", "John"}, {"Goodbye", "John"}}
-- length is 2

s = apply(s, routine_id("greeter"), "!" )
console:display( s )
--> s is
-- {"Hello, John!", "Goodbye, John!"}
-- length is 2
```

**See Also:**

[filter](#) | [routine\\_id](#)

**mapping**

```
include std/sequence.e
namespace stdseq
public function mapping(object source_arg, sequence from_set, sequence to_set,
    integer one_level = 0)
```

changes each item from `source_arg` found in `from_set` into the corresponding item in `to_set`

**Arguments:**

1. `source_arg` : Any Euphoria object to be transformed.
2. `from_set` : A sequence of objects representing the only items from `source_arg` that are actually transformed.
3. `to_set` : A sequence of objects representing the transformed equivalents of those found in `from_set`.
4. `one_level` : An integer. 0 (the default) means that mapping applies to every atom in every level of sub-sequences. 1 means that mapping only applies to the items at the first level in `source_arg`.

**Returns:**

An **object**, The transformed version of `source_arg`.

**Comments:**

- When `one_level` is zero or omitted, for each item in `source_arg`,
  - if it is an atom then it may be transformed
  - if it is a sequence, then the mapping is performed recursively on the sequence.
  - This option required `from_set` to only contain atoms and contain no sub-sequences.
- When `one_level` is not zero, for each item in `source_arg`,
  - regardless of whether it is an atom or sequence, if it is found in `from_set` then it is mapped to the corresponding object in `to_set`.
- Mapping occurs when an item in `source_arg` is found in `from_set`, then it is replaced by the corresponding object in `to_set`.

**Example 1:**

```
include std/sequence.e
include std/console.e
sequence result

result = mapping("The Cat in the Hat", "aeiou",
    "AEIOU" )

console:display( result )
--> res is now "ThE CAT In thE HAt"
```

**length**

```
<built-in> function length(object target)
```

returns the length of an object.

**Arguments:**

1. `target` : the object being queried

**Returns:**

An **integer**, the number of item involved with `target`.

**Comments:**

Think of a sequence as a list of items. The length of a list is the number of *top level* items in that list.

If a list item is sequence--empty, flat, or nested--that sequence counts as *one item*.

The length of an "" or {} empty sequence is 0 zero. When an empty sequence is a list item it counts as *one* top level item.

An atom has a length of 1 one. While a real number has no length, an atom *will contribute one item* to a list when it is spliced or inserted.

The length of each sequence is stored internally by the interpreter for fast access. In some other languages this operation requires a search through memory for an end marker.

**Example 1:**

```
? length({{1,2}, {3,4}, {5,6}})  -- 3
? length("")                    -- 0
? length({})                   -- 0
? length( 7 )                  -- 1
? length( 3.14 )               -- 1
```

**See Also:**

[append](#) | [prepend](#) | [&](#)

**reverse**

```
include std/sequence.e
namespace stdseq
public function reverse(object target, integer pFrom = 1, integer pTo = 0)
```

reverses the order of items in a sequence.

**Arguments:**

1. `target` : the sequence to reverse.
2. `pFrom` : an integer, the starting point. Defaults to 1 one.
3. `pTo` : an integer, the end point. Defaults to 0 zero.

**Returns:**

An **atom**,

`target`

an atom is returned unchanged.

A **sequence**,

a modified `target`

with the same length and items in the original `target`; however the items between index `pFrom` and index `pTo` appear in reverse order.

**Comments:**

In the resultant sequence--*some or all*--top-level items appear in reverse order compared to the original sequence.

This function does not reverse any sub-sequences found in the original sequence.

The `pTo` argument is the *offset* from the end of the sequence. Thus 0 zero means last item in the sequence. A negative value indicates an offset from the last element; thus -1 means the second-last element.

**Example 1:**

```
include std/sequence.e

? reverse({1,3,5,7})           -- {7,5,3,1}
? reverse({1,3,5,7,9}, 2, -1)  -- {1,7,5,3,9}
? reverse({1,3,5,7,9}, 2)      -- {1,9,7,5,3}
? reverse({{1,2,3}, {4,5,6}}) -- {{4,5,6}, {1,2,3}}
? reverse({99})                -- {99}
? reverse({})                  -- {}
? reverse(42)                  -- 42
```

## shuffle

```
include std/sequence.e
namespace stdseq
public function shuffle(object seq)
```

shuffles the items of a sequence.

### Arguments:

1. seq: the sequence to shuffle.

### Returns:

A **sequence**

### Comments:

The items in the returned sequence can appear in any order.

The returned sequence may contain duplicated items; not all items from the original sequence must be returned.

The items and order of items in returned sequence is unpredictable and could even be identical to the original sequence.

### Example 1:

```
include std/sequence.e

? shuffle({1,2,3,3}) -- {3,1,3,2}
? shuffle({1,2,3,3}) -- {2,3,1,3}
? shuffle({1,2,3,3}) -- {1,2,3,3}
-- the output will differ each time the example is run
```

## Building Sequences

## series

```
include std/sequence.e
namespace stdseq
public function series(object start, object increment, integer count = 2, integer op = '+')
```

returns a sequence built as a series from a given object.

### Arguments:

1. start : the initial value from which to start
2. increment : the value to recursively add to start to get new elements
3. count : an integer, the number of items in the returned sequence. The default is 2.
4. operation : an integer, the type of operation used to build the series. Can be either '+' for a linear series or '\*' for a geometric series. The default is '+'.

### Returns:

An **object**,

0 zero

on failure

a sequence

containing the constructed series of items.

### Comments:

The first item in the returned series is always start.

A **linear series** is "formed by *adding* increment to start."

A **geometric series** is "formed by *multiplying* increment by start."

A sequence, of length count+1, starting with start and whose adjacent elements differ by increment, is returned.

### Error:

If count is negative or if start op increment is invalid then 0 zero is returned.

### Example 1:

```
include std/sequence.e
```

```

object x

x = series( 1, 4, 5)
? x
-- x is {1, 5, 9, 13, 17}

x = series( 1, 4, 0 )
? x
--> x is {}

x = series( 1, 4, -1 )
? x
--> x is 0

```

### Example 2:

```

include std/sequence.e

? series( 1, 2, 6, '*' )
--> is {1, 2, 4, 8, 16, 32}

? series({1,2,3}, 4, 2)
--> is {{1,2,3}, {5,6,7}}

? series({1,2,3}, {4,-1,10}, 2)
--> is {{1,2,3}, {5,1,13}}

```

### See Also:

[repeat\\_pattern](#)

## repeat\_pattern

```

include std/sequence.e
namespace stdseq
public function repeat_pattern(object pattern, integer count)

```

returns a periodic sequence based on a given pattern and a count.

### Arguments:

1. pattern : the sequence whose items are to be repeated
2. count : an integer, the number of times the pattern is to be repeated.

### Returns:

A **sequence**,

```

{}
  on failure
a sequence
  with length count*length(pattern)

```

### Comments:

The first items of the returned sequence are those of pattern. So are those that follow, on to the end.

The items in a *repeat\_pattern* are spliced (concatenated) to each other.

### Example 1:

```

include std/sequence.e

s = repeat_pattern({1,2,5},3)
--> s is {1,2,5,1,2,5,1,2,5}
--      ^ ^ ^      ^ ^ ^

```

### See Also:

[repeat](#) | [series](#)

## repeat

```

<built-in> function repeat(object item, atom count)

```

creates a sequence with identical items of the specified length.

Arguments:

1. item : an object, which become a repeated item in the resultant sequence.
2. count : an atom, the requested length of the resultant sequence. This must be an integer from 0 zero to 0x3FFFFFFF; real values are floored to an integer.

Returns:

A **sequence**, of length count where every item is equal to the argument item.

Errors:

If the argument count is <0 negative you get a crash.

As the argument count approaches the limit 1\_073\_741\_823 you may run out of memory.

Comments:

When you repeat a item--*sequence or atom*--the interpreter does not actually make multiple copies in memory. Rather, a single copy is *pointed to* a number of times.

Example 1:

```
include std/sequence.e

? repeat(0, 10)
--> {0,0,0,0,0,0,0,0,0,0}

? repeat("JOHN", 4)
--> {"JOHN", "JOHN", "JOHN", "JOHN"}

-- The interpreter creates only one copy of "JOHN".
-- The items in the returned sequence all point to the
-- same item; no extra memory is consumed.
```

See Also:

[repeat\\_pattern](#) | [series](#)

Adding Items

append

<built-in> function append(sequence target, object x)

inserts an item after the last item in a sequence.

Arguments:

1. source : the sequence to add to

2. x : the object to add

Returns:

A **sequence**, whose first items are those of target and whose last item is x.

Comments:

The new sequence is longer by *just one*. The length of the resulting sequence will be length(target) + 1, no matter what x is.

When x is an atom then append and & concatenating will have the same result.

When x is a sequence then append and & concatenation will have different results.

object	append	& concatenate
atom	<pre>? append( {1,2,3}, 99) --&gt; {1,2,3,99}</pre>	<pre>? {1,2,3} &amp; 99 --&gt; {1,2,3,99}</pre>
sequence	<pre>? append( {1,2,3}, {99} ) --&gt; {1,2,3,{99}}</pre>	<pre>{1,2,3} &amp; {99} --&gt; {1,2,3,99}</pre>



sequence	? append( {1,2,3}, {44,55} ) --> {1,2,3,{44,55}}	{1,2,3} & {44,55} --> {1,2,3,44,55}

Note:

You may consider append and prepend to be special cases of the insert function.

Note:

The extra storage is allocated automatically and very efficiently with Euphoria's dynamic storage allocation.

The special case where target is appended and then assigned to itself is highly optimized; the code in Example 1 is executed very efficiently.

Example 1:

```
sequence x

x = {}
for i = 1 to 10 do
  x = append(x, i)
end for

? x
--> x is now {1,2,3,4,5,6,7,8,9,10}
```

Example 2:

```
include std/console.e
sequence x, y, z

x = {"fred", "barney"}

y = append(x, "wilma")
console:display( y )
--> y is now {"fred", "barney", "wilma"}

z = append( x, {"bam", "bam"})
console:display(z)
--> z is now {"fred", "barney", {"bam", "bam"}}
```

See Also:

prepend | & | splice | insert

prepend

<built-in> function prepend(sequence target, object x)

inserts an object before the first item of a sequence.

Arguments:

1. source : the sequence to add to  
2. x : the object to add

Returns:

A **sequence**, whose last elements are those of target and whose first element is x.

Comments:

The new sequence is longer by *just one*. The length of the returned sequence will be length(target) + 1, no matter what x is.

When x is an atom then prepend and & concatenating will have the same result.

When x is a sequence then prepend and & concatenating will have different results.

object	prepend	& concatenate
atom	? prepend( {1,2,3}, 99 ) --> {99,1,2,3}	? 99 & {1,2,3} --> {99,1,2,3}

<b>sequence</b>	<pre>? prepend( {1,2,3}, {99} ) --&gt; {{99},1,2,3}</pre>	<pre>{99} &amp; {1,2,3} --&gt; {1,2,3,99}</pre>
<b>sequence</b>	<pre>? prepend( {1,2,3}, {44,55} ) --&gt; {{44,55},1,2,3}</pre>	<pre>{44,55} &amp; {1,2,3} --&gt; {44,55,1,2,3}</pre>

**Note:**

You may consider append and prepend to be special cases of the insert function.

**Note:**

The extra storage is allocated automatically and very efficiently with Euphoria's dynamic storage allocation.

The case where target itself is prepended to is handled very efficiently; the code in Example 1 is executed very efficiently.

**Example 1:**

```
sequence x

x = {}
for i = 1 to 10 do
    x = prepend(x, i )
end for

? x
--> x is now {10,9,8,7,6,5,4,3,2,1}
```

**Example 2:**

```
include std/console.e
sequence x, y, z

x = {"fred", "barney"}

y = prepend(x, "wilma")
console:display( y )
--> y is now { "wilma", "fred", "barney" }

z = prepend( x, {"bam", "bam"})
console:display(z)
--> z is now { {"bam", "bam"}, "fred", "barney" }
```

**See Also:**

[append](#) | [&](#) | [insert](#) | [splice](#)

**insert**

```
<built-in> function insert(sequence target, object what, integer index)
```

*individually inserts* an object into a sequence as *distinct item* at a given location.

**Arguments:**

1. target : the sequence to insert into
2. what : the object to insert
3. index : an integer, the position in target where argument what should appear

**Returns:**

A **sequence**, which is target with one more item at index, which is what.

**Comments:**

The new sequence is longer by *just one*. The length of the returned sequence is always `length(target) + 1`, no matter what x is.

target can be a sequence of any shape, and what any kind of object.

When x is an atom then insert and splice will have the same result.

When x is a sequence then insert and splice will have different results.

**object****insert****splice**

<b>atom</b>	<pre>? insert( {1,2,3}, 99, 2 ) --&gt; {1,99,2,3}</pre>	<pre>? splice( {1,2,3}, 99, 2 ) --&gt; {1,99,2,3}</pre>
<b>sequence</b>	<pre>? insert( {1,2,3}, {99}, 2 ) --&gt; {1,{99},2,3}</pre>	<pre>? splice({1,2,3},{99},2) --&gt; {1,99,2,3}</pre>
<b>sequence</b>	<pre>? insert( {1,2,3}, {44,55} ) --&gt; {1,{44,55},2,3}</pre>	<pre>? splice({1,2,3},{44,55},2) --&gt; {1,44,55,2,3}</pre>

**Error:**

Inserting a string into an existing string results in a nested sequence which is no longer a valid string sequence. This is a common mistake and puts will no longer display this sequence.

**Note:**

You may consider append and prepend to be special cases of the insert function.

**Example 1:**

```
include std/console.e
include std/pretty.e
sequence s

s = insert("John Doe", " Middle", 5)
console:display( s )

--> s is
-- { 74,111,104,110, " Middle", 32,68,111,101}

pretty_print(1, s )
--> s is
-- {'J','o','h','n'," Middle",' ','D','o','e'}
```

**Example 2:**

```
sequence s

s = insert({10,30,40}, 20, 2)
? s
--> s is {10,20,30,40}
```

**See Also:**

[remove](#) | [splice](#) | [append](#) | [prepend](#) | [:](#) &

**splice**

```
<built-in> function splice(sequence target, object what, integer index)
```

*smoothly splices* an object as a new slice *blended* into a sequence at a given position.

**Arguments:**

1. `target` : the sequence to insert into
2. `what` : the object to insert
3. `index` : an integer, the position in `target` where `what` should appear

**Returns:**

A **sequence**, which is target with one or more elements, those of `what`, inserted at locations starting at `index`.

**Comments:**

The length of this new sequence is the sum of the lengths of `target` and `what`.

`target` can be a sequence of any shape, and `what` any kind of object.

When `x` is an atom then `insert` and `splice` will have the same result.

When `x` is a sequence then `insert` and `splice` will have different results.

**object**

**insert**

**splice**

<b>atom</b>	<pre>? insert( {1,2,3}, 99, 2 ) --&gt; {1,99,2,3}</pre>	<pre>? splice( {1,2,3}, 99, 2 ) --&gt; {1,99,2,3}</pre>
<b>sequence</b>	<pre>? insert( {1,2,3}, {99}, 2 ) --&gt; {1,{99},2,3}</pre>	<pre>? splice({1,2,3},{99},2) --&gt; {1,99,2,3}</pre>
<b>sequence</b>	<pre>? insert( {1,2,3}, {44,55} ) --&gt; {1,{44,55},2,3}</pre>	<pre>? splice({1,2,3},{44,55},2) --&gt; {1,44,55,2,3}</pre>

**Note:**

You may consider the & concatenation operator to be a special case of the splice function.

**Note:**

Splicing a string into a string results into a new valid string sequence. The puts function will show this string correctly.

**Example 1:**

```
sequence s

s = splice("John Doe", " Middle", 5)
puts(1, s )
--> s is "John Middle Doe"
```

**Example 2:**

```
sequence s

s = splice({10,30,40}, 20, 2)
? s
--> is {10,20,30,40}
```

**See Also:**

[insert](#) | [remove](#) | [replace](#) | [&](#)

**pad\_head**

```
include std/sequence.e
namespace stdseq
public function pad_head(object target, integer size, object ch = ' ')
```

pads the beginning of a sequence with repeated objects to satisfy a minimum length requirement.

**Arguments:**

1. target : the sequence to pad.
2. size : an integer, the target minimum size for target
3. padding : an object, usually the character to pad to (defaults to ' ').

**Returns:**

A **sequence**, either target if it was long enough, or a sequence of length size whose last elements are those of target and whose first few head elements all equal padding.

**Comments:**

The pad\_head function will not remove characters. If length(target) is greater than size, this function simply returns target. See [head](#) if you wish to truncate long sequences.

**Example 1:**

```
include std/sequence.e
include std/console.e
sequence s

s = pad_head("ABC", 6)
console:display( s )

--> s is "   ABC"

s = pad_head("ABC", 6, '-')
```

```
console:display( s )
--> s is "---ABC"
```

### Example 2:

```
include std/sequence.e
include std/console.e
sequence s

s = pad_head("ABC", 6, `*x`)
console:display( s )
-- the result is no longer a string sequence
--> s is { "*x","*x","*x",65,66,67 }
```

### See Also:

[trim\\_head](#) | [pad\\_tail](#) | [head](#)

## pad\_tail

```
include std/sequence.e
namespace stdseq
public function pad_tail(object target, integer size, object ch = ' ')
```

pads the end of a sequence with repeated objects to satisfy a minimum length requirement.

### Arguments:

1. `target` : the sequence to pad.
2. `size` : an integer, the target minimum size for `target`.
3. `padding` : an object, usually the character to pad to (defaults to ' ').

### Returns:

A **sequence**, either `target` if it was long enough, or a sequence of length `size` whose first elements are those of `target` and whose last few head elements all equal `padding`.

### Comments:

The function `pad_tail` will not remove characters. If `length(str)` is greater than `size` this function simply returns `target`. See `tail` if you wish to truncate long sequences.

### Example 1:

```
s = pad_tail("ABC", 6)
-- s is "ABC  "

s = pad_tail("ABC", 6, '-')
-- s is "ABC---"
```

### See Also:

[trim\\_tail](#) | [pad\\_head](#) | [tail](#)

## add\_item

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

if an item is *unique* to the sequence it is inserted otherwise original is unchanged.

### Arguments:

1. `needle` : object to insert.
2. `haystack` : sequence to add it to.
3. `order` : an integer; determines how the `needle` affects the `haystack`. It can be inserted to the front (prepended), to the back (appended), or sorted after adding. The default is to prepend it.

### Returns:

A **sequence**,

a new sequence

if `needle` is unique to the sequence

original sequence

if needle already exists in the sequence

### Comments:

An error occurs if an invalid order argument is supplied.

The following enum is provided for specifying order:

- `ADD_PREPEND` -- prepend needle to haystack. This is the default option.
- `ADD_APPEND` -- append needle to haystack.
- `ADD_SORT_UP` -- sort haystack in ascending order after inserting needle
- `ADD_SORT_DOWN` -- sort haystack in descending order after inserting needle

### Example 1:

```
include std/sequence.e
sequence s

s = add_item( 1, {3,4,2}, ADD_PREPEND ) -- prepend
? s
--> s is {1,3,4,2}
```

### Example 2:

```
include std/sequence.e
sequence s

s = add_item( 1, {3,4,2}, ADD_APPEND ) -- append
? s
--> s is {3,4,2,1}
```

### Example 3:

```
include std/sequence.e
sequence s

s = add_item( 1, {3,4,2}, ADD_SORT_UP ) -- ascending
? s
--> s is {1,2,3,4}
```

### Example 4:

```
include std/sequence.e
sequence s

s = add_item( 1, {3,4,2}, ADD_SORT_DOWN ) -- descending
? s
--> s is {4,3,2,1}
```

### Example 5:

```
include std/sequence.e
sequence s

s = add_item( 1, {3,1,4,2} )
? s
--> s is {3,1,4,2}
-- item already in list so no change.
```

### See Also:

[remove\\_item](#) | [prepend](#) | [append](#)

## remove\_item

```
include std/sequence.e
namespace stdseq
public function remove_item(object needle, sequence haystack)
```

removes an item from the sequence.

### Arguments:

1. `needle` : object to remove.
2. `haystack` : sequence to remove it from.

### Returns:

A **sequence**, which is haystack with needle removed from it.

### Comments:

If needle is not in haystack then haystack is returned unchanged.

### Example 1:

```
include std/sequence.e
sequence s

s = remove_item( 1, {3,4,2,1} )
? s
--> s is {3,4,2}

s = remove_item( 5, {3,4,2,1} )
? s
--> s is {3,4,2,1}
```

### See Also:

[add\\_item](#)

## Extracting, Removing, Replacing

### head

```
<built-in> function head(sequence source, atom size=1)
```

returns the first size item or items of a sequence.

### Arguments:

1. source : the sequence from which items will be returned
2. size : an integer; how many items, at most, will be returned. Defaults to one.

### Returns:

A **sequence**,

original source	
if size is longer than length(source)	
a slice	
the first size item or items from source sequence	

### Example 1:

```
sequence s = head("John Doe", 4)
puts(1, s )
--> s is "John"
```

### Example 2:

```
sequence s = head("John Doe", 50)
puts(1, s )
--> s is "John Doe"
```

### Example 3:

```
include std/console.e
sequence s

s = head({1, 5.4, "John", 30}, 3)
console:display( s )
--> s is {1, 5.4, "John"}
```

### See Also:

[tail](#) | [mid](#) | [slice](#)

### tail

```
<built-in> function tail(sequence source, atom size=length(source) - 1)
```

returns the last size item or items of a sequence.

**Arguments:**

1. `source` : the sequence to get the tail of.
2. `size` : an integer, the number of items to return. (defaults to `length(source) - 1`)

**Returns:**

A **sequence**,

original source  
 if size is greater than `length(source)`  
 a slice  
 the last size item or items from source sequence

**Comments:**

`source` can be any type of sequence, including nested sequences.

**Example 1:**

```
sequence s

s = tail("John Doe", 3)
puts(1, s )
--> s is "Doe"
```

**Example 2:**

```
sequence s

s = tail("John Doe", 50)
puts(1, s )
--> s is "John Doe"
```

**Example 3:**

```
include std/console.e
sequence s

s = tail({1, 5.4, "John", 30}, 3)
console:display( s )
--> s is {5.4, "John", 30}
```

**See Also:**

[head](#) | [mid](#) | [slice](#)

**mid**

```
include std/sequence.e
namespace stdseq
public function mid(sequence source, atom start, atom len)
```

returns a slice of a sequence chosen by a starting index and a length.

**Arguments:**

1. `source` : the sequence some items of which will be returned
2. `start` : an integer, the lower index of the slice to return
3. `len` : an integer, the length of the slice to return

**Returns:**

A **sequence**, made of at most `len` elements of `source`. These elements are at contiguous positions in `source` starting at `start`.

**Errors:**

If `len` is less than `-length(source)` then an error occurs.

**Comments:**

Values for `len` less than 1 one are permitted; they represent an offset from the end of the sequence.

If `len` is 0 zero then the slice length equals the length of the sequence.

If `len` is -1 for example, the slice is one less than the length of the sequence.

**Note:**



The slice function also returns a *slice* but it is based on a starting index and a final index.

### Example 1:

```
include std/sequence.e
sequence s

s = mid("John Middle Doe", 6, 6)
puts(1, s )
--> s is "Middle"
```

### Example 2:

```
include std/sequence.e
sequence s

s = mid("John Middle Doe", 6, 50)
puts(1, s )
--> s is "Middle Doe"
```

### Example 3:

```
include std/sequence.e
include std/console.e
sequence s

s = mid({1, 5.4, "John", 30}, 2, 2)
console:display( s )
--> s is {5.4, "John"}
```

### Example 4:

```
include std/sequence.e
include std/console.e
sequence s

s = mid({1, 5.4, "John", 30}, 2, -1)
console:display( s )
--> s is {5.4, "John", 30}
```

### See Also:

[head](#) | [tail](#) | [slice](#)

## slice

```
include std/sequence.e
namespace stdseq
public function slice(sequence source, atom start = 1, atom stop = 0)
```

returns a portion of the supplied sequence.

### Arguments:

1. *source* : the sequence from which to get a portion
2. *start* : an integer, the starting point of the portion. Default is 1.
3. *stop* : an integer, the ending point of the portion. Default is length(*source*).

### Returns:

A **sequence**.

### Comments:

A **slice** is "a sequence of consecutive items."

The slice function returns a *slice* of the original sequence whose length can be from zero or upto the original sequence.

If the start index is less than 1 then it set to 1 one.

If the stop index is less than 1 then length(*source*) is added to it.

In this way, 0 zero represents the end of *source*, and -1 represents one item in from the end of *source* and so on.

If the stop index is greater than length(*source*) then it is set to the end.

After these adjustments, and if *source*[*start*..*stop*] makes sense, it is returned, otherwise, {} is returned.

**Note:**

The `mid` function also returns a *slice* but is based on starting index and a length.

**Example 1:**

```
include std/sequence.e
include std/console.e
sequence s

s = slice("John Doe", 6, 8)
console:display(s) --> "Doe"

s = slice("John Doe", 6, 50)
console:display(s) --> "Doe"

s = slice({1, 5.4, "John", 30}, 2, 3)
console:display(s) --> {5.4, "John"}

s = slice({1,2,3,4,5}, 2, -1)
? s --> {2,3,4}

s = slice({1,2,3,4,5}, 2)
? s --> {2,3,4,5}

s = slice({1,2,3,4,5}, , 4)
? s --> {1,2,3,4}
```

**See Also:**

[head](#) | [mid](#) | [tail](#)

**vslice**

```
include std/sequence.e
namespace stdseq
public function vslice(sequence source, atom colno, object error_control = 0)
```

performs a vertical slice on a nested sequence.

**Arguments:**

1. `source` : the sequence to take a vertical slice from
2. `colno` : an atom, the column number to extract (rounded down)
3. `error_control` : an object which says what to do if some element does not exist. Defaults to 0 (crash in such a circumstance).

**Returns:**

A **sequence**, usually of the same length as `source`, made of all the `source[x][colno]`.

**Errors:**

If an element is not defined and `error_control` is 0, an error occurs. If `colno` is less than 1, it cannot be any valid column, and an error occurs.

**Comments:**

If it is not possible to return the sequence of all `source[x][colno]` for all available `x`, the outcome is decided by `error_control`:

- If 0 (the default), program is aborted.
- If a nonzero atom, the short vertical slice is returned.
- Otherwise, elements of `error_control` will be taken to make for any missing element. The elements are selected from the first to the last, as needed and this cycles again from the first.

**Example 1:**

```
s = vslice({{5,1}, {5,2}, {5,3}}, 2)
-- s is {1,2,3}

s = vslice({{5,1}, {5,2}, {5,3}}, 1)
-- s is {5,5,5}
```

**See Also:**

[slice](#) | [project](#) | [columnize](#)

## remove

```
<built-in> function remove(sequence target, atom start, atom stop=start)
```

removes an item or a slice from a sequence.

### Arguments:

1. target : the sequence to remove from.
2. start : an atom, the (starting) index at which to remove
3. stop : an atom, the index at which to stop removing (defaults to start)

### Returns:

A **sequence**, obtained from target by carving the start..stop slice out of it.

### Comments:

A new sequence is created. target can be a string or complex sequence.

### Example 1:

```
sequence s
s = remove("Johnn Doe", 4)
--           ^
puts(1, s ) --> s is "John Doe"
```

### Example 2:

```
sequence s
s = remove({1,2,3,3,4}, 4)
--           ^
? s --> s is {1,2,3,4}
```

### Example 3:

```
sequence s
s = remove("John Middle Doe", 6, 12)
--           ^^^^^^^
puts(1, s ) -- s is "John Doe"
```

### Example 4:

```
sequence s
s = remove({1,2,3,3,4,4}, 4, 5)
--           ^ ^
? s --> s is {1,2,3,4}
```

### See Also:

[replace](#) | [insert](#) | [splice](#) | [remove\\_all](#)

## patch

```
include std/sequence.e
namespace stdseq
public function patch(sequence target, sequence source, integer start, object filler = ' ')
```

changes a sequence slice, possibly with padding.

### Arguments:

1. target : a sequence, a modified copy of which will be returned
2. source : a sequence, to be patched inside or outside target
3. start : an integer, the position at which to patch
4. filler : an object, used for filling gaps. Defaults to ' '

### Returns:

A **sequence**, which looks like target, but a slice starting at start equals source.

### Comments:

In some cases, this call will result in the same result as [replace](#).

If source does not fit into target because of the lengths and the supplied start value, gaps will be created, and filler is used to fill them in.

Notionally, target has an infinite amount of filler on both sides, and start counts position relative to where target actually starts. Then, notionally, a `replace` operation is performed.

#### Example 1:

```
sequence source = "abc", target = "John Doe"
sequence s = patch(target, source, 11, '0')
-- s is now "John Doe00abc"
```

#### Example 2:

```
sequence source = "abc", target = "John Doe"
sequence s = patch(target, source, -1)
-- s is now "abcohn Doe"
Note that there was no gap to fill.
Since -1 = 1 - 2, the patching started 2 positions before the initial 'J'.
```

#### Example 3:

```
sequence source = "abc", target = "John Doe"
sequence s = patch(target, source, 6)
-- s is now "John Dabc"
```

#### See Also:

`mid` | `replace`

## remove\_all

```
include std/sequence.e
namespace stdseq
public function remove_all(object needle, sequence haystack)
```

removes all occurrences of some object from a sequence.

#### Arguments:

1. `needle` : the object to remove.
2. `haystack` : the sequence to remove from.

#### Returns:

A **sequence**, of length at most `length(haystack)`, and which has the same elements, without any copy of `needle` left

#### Comments:

This function weeds elements out, not sub-sequences.

#### Example 1:

```
include std/sequence.e
sequence s

s = remove_all( 1, {1,2,4,1,3,2,4,1,2,3} )
? s -->      s is { 2,4, 3,2,4, 2,3}
```

#### Example 2:

```
include std/sequence.e
sequence s

s = remove_all('x', "I'm toox secxksy for my shixrt.")
puts(1,s) --> s is "I'm too secxksy for my shirt."
```

#### See Also:

`remove` | `replace` | `[[: retain_all ]]`

## retain\_all

```
include std/sequence.e
namespace stdseq
public function retain_all(object needles, sequence haystack)
```

keeps all occurrences of a set of objects from a sequence and removes all others.

Arguments:

- 1. needles : the set of objects to retain.
- 2. haystack : the sequence to remove items not in needles.

Returns:

A **sequence** containing only those objects from haystack that are also in needles.

Example 1:

```
include std/sequence.e
sequence s

s = retain_all( {1,3,5}, {1,2,4,1,3,2,4,1,2,3} )
? s          --> s is {1, 1,3, 1, 3}

s = retain_all("0123456789", "+34 (04) 555-44392")
puts(1,s)    --> s is "340455544392"
```

See Also:

[remove](#) | [replace](#) | [remove\\_all](#)

filter

```
include std/sequence.e
namespace stdseq
public function filter(sequence source, object rid, object userdata = {},
    object rangetype = "")
```

filters a sequence based on a user supplied comparator function.

Arguments:

- source : sequence to filter
- rid : Either a [routine id](#) of function to use as comparator or one of the predefined comparitors.
- userdata : an object passed to each invocation of rid. If omitted, {} is used.
- rangetype: A sequence. Only used when rid is "in" or "out". This is used to let the function know how to interpret userdata. When rangetype is an empty string (which is the default), then userdata is treated as a set of zero or more discrete items such that "in" will only return items from source that are in the set of item inuserdata and "out" returns those not in userdata. The other values for rangetype mean that userdata must be a set of exactly two items, that represent the lower and upper limits of a range of values.

Returns:

A **sequence**, made of the elements in source which passed the comparator test.

Comments:

- The only items from source that are returned are those that pass the test.
- When rid is a routine id, that user defined routine must be a function. Each item in source, along with the userdata is passed to the function. The function must return a non-zero atom if the item is to be included in the result sequence, otherwise it should return zero to exclude it from the result.

The predefined comparitors are:

Comparator	Return Items in source that are...	
"<"	"lt"	less than userdata
"<="	"le"	less than or equal to userdata
"=" or "=="	"eq"	equal to userdata
"!="	"ne"	not equal to userdata
">"	"gt"	greater than userdata
">="	"ge"	greater than or equal to userdata
	"in"	in userdata
	"out"	not in userdata

Range Type Usage

Range Type	Range	Meaning
"[]"	Inclusive range.	Lower and upper are in the range.
"[]"	Low Inclusive range.	Lower is in the range but upper is not.
"()]"	High Inclusive range.	Lower is not in the range but upper is.

"()"	Exclusive range.	Lower and upper are not in the range.
------	------------------	---------------------------------------

**Example 1:**

```
include std/sequence.e

function mask_nums(atom a, object t)
  if sequence(t) then
    return 0
  end if
  return and_bits(a, t) != 0
end function

function even_nums(atom a, atom t)
  return and_bits(a,1) = 0
end function

constant data = {5,8,20,19,3,2,10}
? filter(data, routine_id("mask_nums"), 1) --> {5,19,3 }
? filter(data, routine_id("mask_nums"), 2) --> {      19,3,2,10}
? filter(data, routine_id("even_nums"), 1)  --> { 8,20,      2,10}

-- Using 'in' and 'out' with sets.
? filter(data, "in", {3,4,5,6,7,8})        --> {5,8,      3      }
? filter(data, "out", {3,4,5,6,7,8})        --> {      20,19,  2,10}

-- Using 'in' and 'out' with ranges.
? filter(data, "in", {3,8}, "[ ]")          --> {5,8,      3      }
? filter(data, "in", {3,8}, "[ ]")          --> {5,      3      }
? filter(data, "in", {3,8}, "[ ]")          --> {5,8      }
? filter(data, "in", {3,8}, "[ ]")          --> {5      }
? filter(data, "out", {3,8}, "[ ]")          --> {      20,19,  2,10}
? filter(data, "out", {3,8}, "[ ]")          --> { 8,20,19,  2,10}
? filter(data, "out", {3,8}, "[ ]")          --> {      20,19,3,2,10}
? filter(data, "out", {3,8}, "[ ]")          --> { 8,20,19,3,2,10}
```

**Example 2:**

```
include std/sequence.e

function quiksort(sequence s)
  if length(s) < 2 then
    return s
  end if
  return quiksort(
    filter( s[2..$], "<=", s[1] ) &
    s[1] &
    quiksort(filter(s[2..$], ">", s[1] )
  )
end function

? quiksort( {5,4,7,2,4,9,1,0,4,32,7,54,2,5,8,445,67} )
--> {0,1,2,2,4,4,4,5,5,7,7,8,9,32,54,67,445}
```

**See Also:**[apply](#)**STDFLTR\_ALPHA**

```
public constant STDFLTR_ALPHA
```

Predefined routine\_id for use with [filter](#).

**Comments:**

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

**Example 1:**

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

**replace**

```
<built-in> function replace(sequence target, object replacement, integer start, integer stop=start)
```

replaces a slice in a sequence by a new object.

### Arguments:

1. target : the sequence in which replacement will be done.
2. replacement : an object, the item to replace with.
3. start : an integer, the starting index of the slice to replace.
4. stop : an integer, the stopping index of the slice to replace.

### Returns:

A **sequence**, which is made of target with the start..stop slice removed and replaced by replacement, which is spliced in.

### Comments:

A new sequence is created. The target can be a string or complex sequence of any shape.

The replacement is smoothly spliced into the target.

To replace by just one item enclose the replacement in {} braces; the braces will be removed at replacement time. Note that an atom and a one item sequence will both be spliced in the same way.

### Example 1:

```
include std/console.e
sequence s

      s = replace("John Middle Doe", "Smith", 6, 11)
      ^      ^
--
console:display(s)
--> s is "John Smith Doe"

      s = replace({45.3, "John", 5, {10, 20}}, 25, 2, 3)
      ^      ^
--
console:display(s)
--> s is {45.3, 25, , {10, 20}}
```

### Example 2:

```
include std/console.e
sequence s

      s = replace("John Middle Doe", {"Smith"}, 6, 11)
      ^      ^
--
console:display(s)
--> s is {74,111,104,110,32,"Smith",32,68,111,101}

      s = replace({45.3, "John", 5, {10, 20}}, {25}, 2, 3)
      ^      ^
--
console:display(s)
--> s is {45.3, 25, , {10, 20}}
```

### See Also:

[splice](#) | [remove](#) | [remove\\_all](#)

## extract

```
include std/sequence.e
namespace stdseq
public function extract(sequence source, sequence indexes)
```

picks items from the source sequence (based on a list of indexes) and returns a new sequence.

### Arguments:

1. source : the sequence from which to extract elements
2. indexes : a sequence of atoms, the indexes of the elements to be fetched in source.

### Returns:

A **sequence**, of the same length as indexes.

### Comments:

The list of indexes can have repeated index values.

**Example 1:**

```
include std/sequence.e
sequence s

s = extract( {11,13,15,17}, { 3, 1, 2, 1, 4} )
? s
--> s is {15,11,13,11,17}
```

**See Also:**

[slice](#) | [fetch](#) | [store](#)

**project**

```
include std/sequence.e
namespace stdseq
public function project(sequence source, sequence coords)
```

creates a list of sequences based on selected items from sequences in the source.

**Arguments:**

1. source : a list of sequences.
2. coords : a list of index lists.

**Returns:**

A **sequence**, with the same length as source. Each of its elements is a sequence, the length of coords. Each innermost sequence is made of the elements from the corresponding source sub-sequence.

**Comments:**

For each sequence in source, a set of sub-sequences is created; one for each index list in coords. An index list is just a sequence containing indexes for items in a sequence.

**Example 1:**

```
include std/sequence.e
include std/console.e
sequence s

s = project( {"789"}, { {1,2}, {3,1}, {2} } )
console:display(s)
--> s is { { "78", "97", "8" } }

s = project( {"ABCD"}, { {1,2}, {3,1}, {2} } )
console:display(s)
--> s is { { "AB", "CA", "B" } }

s = project( {"ABCD", "789"}, { {1,2}, {3,1}, {2} } )
console:display(s)
--> s is { {"AB", "CA", "B"}, {"78", "97", "8"} }
```

**See Also:**

[vslice](#) | [extract](#)

**Changing the Shape of a Sequence****split**

```
include std/sequence.e
namespace stdseq
public function split(sequence st, object delim = ' ', integer no_empty = 0,
integer limit = 0)
```

splits a string sequence by *one* delimiter into a number of sub-sequences.

**Arguments:**

1. source : the sequence to split.
2. delim : an object (default is ' '). The delimiter that separates items in source.
3. no\_empty : an integer (default is 0 zero). If not zero then all zero-length sub-sequences are removed from the returned sequence. Use this when leading, trailing and duplicated delimiters are not significant.
4. limit : an integer (default is 0 zero). The maximum number of sub-sequences to create. If zero, there is no limit.



**Returns:**

A **sequence**, of sub-sequences of source. Delimiters are removed.

**Comments:**

This function may be applied to a string sequence or a complex sequence.

If limit is > 0, this is the maximum number of sub-sequences that will be created, otherwise there is no limit.

**Example 1:**

```
result = split("John Middle Doe")
-- result is {"John", "Middle", "Doe"}
```

**Example 2:**

```
result = split("John,Middle,Doe", ",", 2) -- Only want 2 sub-sequences.
-- result is {"John", "Middle,Doe"}
```

**Example 3:**

```
result = split("John|Middle|Doe", '|') -- Each '|' is significant by default
-- result is {"John","","Middle","","Doe",""}
result = split("John|Middle|Doe", '|', 1) -- Adjacent '|' are just a single delim,
-- and leading/trailing '|' ignored.
-- result is {"John","Middle","Doe"}
```

**See Also:**

[split\\_any](#) | [breakup](#) | [join](#)

**split\_any**

```
include std/sequence.e
namespace stdseq
public function split_any(sequence source, object delim = ", \t|", integer limit = 0,
integer no_empty = 0)
```

splits a string sequence by *any* of the separators in the list of delimiters into a number of sub-sequences.

If limit is > 0 then limit the number of tokens that will be split to limit.

**Arguments:**

1. source : the sequence to split.
2. delim : a list of delimiters to split by. The default set is comma, space, tab and bar.
3. limit : an integer (default is 0). The maximum number of sub-sequences to create. If zero, there is no limit.
4. no\_empty : an integer (default is 0). If not zero then all zero-length sub-sequences removed from the returned sequence. Use this when leading, trailing and duplicated delimiters are not significant.

**Comments:**

- This function may be applied to a string sequence or a complex sequence.
- It works like split, but in this case delim is a set of potential delimiters rather than a single delimiter.
- If delim is an empty set, the source is returned in a sequence.

**Example 1:**

```
include std/sequence.e
include std/console.e
sequence result

-- Default delims
result = split_any("One,Two|Three Four")
console::display(result)
--> result is {"One", "Two", "Three", "Four"}

-- Using dot and colon
result = split_any("192.168.1.103:8080", ".:")
console::display(result)
--> result is {"192","168","1","103","8080"}

-- Limited to two splits
result = split_any("One,Two|Three Four", , 2)
console::display(result)
--> result is {"One", "Two", "Three Four"}

-- Allow Empty option
result = split_any(",One,,Two| Three|| Four," )
console::display(result)
```

```
-- result is {"","One","","Two","","Three","","","Four",""}

-- No Empty option
result = split_any(",One,,Two| Three|| Four",,,1)
--> result is {"One", "Two", "Three", "Four"}

-- Empty delimiters
result = split_any(",One,,Two| Three|| Four",, "")
--> result is {"One,,Two| Three|| Four,"}
```

### See Also:

[split](#) | [breakup](#) | [join](#)

## join

```
include std/sequence.e
namespace stdseq
public function join(sequence items, object delim = " ")
```

splices sequences together placing a delimiter between each joint.

### Arguments:

1. items : the sequence of items to join.
2. delim : an object, the delimiter to join by. Defaults to " ".

### Comments:

This function may be applied to a string sequence or a complex sequence

The delimiter could be an empty sequence.

### Example 1:

```
include std/sequence.e
include std/console.e
sequence result

result = join({"John", "Middle", "Doe"})
console:display(result)
--> result is "John Middle Doe"
```

### Example 2:

```
include std/sequence.e
include std/console.e
sequence result

result = join({"John", "Middle", "Doe"}, ",")
console:display(result)
--> result is "John,Middle,Doe"
```

### See Also:

[split](#) | [split\\_any](#) | [breakup](#)

## public enum

```
include std/sequence.e
namespace stdseq
public enum
```

Style options for breakup

BK\_LEN | BK\_PIECES

## BK\_LEN

```
include std/sequence.e
namespace stdseq
BK_LEN
```

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

**BK\_PIECES**

```
include std/sequence.e
namespace stdseq
BK_PIECES
```

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

**breakup**

```
include std/sequence.e
namespace stdseq
public function breakup(sequence source, object size, integer style = BK_LEN)
```

breaks up a sequence into multiple sequences of a given length.

**Arguments:**

1. **source** : the sequence to be broken up into sub-sequences.
2. **size** : an object, if an integer it is either the maximum length of each resulting sub-sequence or the maximum number of sub-sequences to break source into.  
If size is a sequence, it is a list of element counts for the sub-sequences it creates.
3. **style** : an integer, Either `BK_LEN` if size integer represents the sub-sequences' maximum length, or `BK_PIECES` if the size integer represents the maximum number of sub-sequences (pieces) to break source into.

**Returns:**

A **sequence**, of sequences.

**Comments:**

When size is an integer and style is `BK_LEN`

The sub-sequences have length size, except possibly the last one, which may be shorter. For example if source has 11 items and size is 3, then the first three sub-sequences will get 3 items each and the remaining 2 items will go into the last sub-sequence. If size is less than 1 or greater than the length of the source, the source is returned as the only sub-sequence.

When size is an integer and style is `BK_PIECES`

There is exactly size sub-sequences created. If the source is not evenly divisible into that many pieces, then the lefthand sub-sequences will contain one more element than the right-hand sub-sequences. For example, if source contains 10 items and we break it into 3 pieces, piece #1 gets 4 elements, piece #2 gets 3 items and piece #3 gets 3 items - a total of 10. If source had 11 elements then the pieces will have 4,4, and 3 respectively.

When size is a sequence

The style parameter is ignored in this case. The source will be broken up according to the counts contained in the size parameter. For example, if size was {3,4,0,1} then piece #1 gets 3 items, #2 gets 4 items, #3 gets 0 items, and #4 gets 1 item. Note that if not all items from source are placed into the sub-sequences defined by size, and *extra* sub-sequence is appended that contains the remaining items from source.

In all cases, when concatenated these sub-sequences will be identical to the original source.

**Example 1:**

```
include std/sequence.e
include std/console.e
sequence s

s = breakup("5545112133234454", 4)
console:display(s)
--> s is {"5545", "1121", "3323", "4454"}
```

**Example 2:**

```
include std/sequence.e
include std/console.e
sequence s

s = breakup("12345", 2)
console:display(s)
--> s is {"12", "34", "5"}
```

**Example 3:**

```
include std/sequence.e
sequence s
```

```
s = breakup( {1,2,3,4,5,6}, 3)
? s
--> s is { {1,2,3}, {4,5,6} }
```

#### Example 4:

```
include std/sequence.e
include std/console.e
sequence s

s = breakup( "ABCDEF", 0)
console:display(s)
--> s is { "ABCDEF" }
```

#### See Also:

[split](#) | [flatten](#)

## flatten

```
include std/sequence.e
namespace stdseq
public function flatten(sequence s, object delim = "")
```

removes all nesting from a sequence.

#### Arguments:

1. `s` : the sequence to flatten out.
2. `delim` : An optional delimiter to place after each flattened sub-sequence (except the last one).

#### Returns:

A **sequence**, of atoms, all the atoms in `s` enumerated.

#### Comments:

If you consider a sequence as a tree, then the enumeration is performed by left-right reading of the tree. The elements are simply read left to right, without any care for braces.

Empty sub-sequences are stripped out entirely.

#### Example 1:

```
include std/sequence.e
sequence s

s = flatten({{18, 19}, 45, {18.4, 29.3}})
? s
--> s is {18, 19, 45, 18.4, 29.3}
```

#### Example 2:

```
include std/sequence.e
sequence s

s = flatten({18,{ 19, {45}}, {18.4, {}, 29.3}})
? s
--> s is {18, 19, 45, 18.4, 29.3}
```

#### Example 3:

```
include std/sequence.e
include std/console.e
sequence s

-- Using the delimiter argument

s = flatten({"abc", "def", "ghi"}, ", ")
console:display(s)
--> s is      "abc, def, ghi"
```

## pivot

```
include std/sequence.e
```

```
namespace stdseq
public function pivot(object data_p, object pivot_p = 0)
```

returns a sequence of three sub-sequences. The sub-sequences contain all the elements less than the supplied pivot value, equal to the pivot, and greater than the pivot.

### Arguments:

1. data\_p : Either an atom or a list. An atom is treated as if it is one-element sequence.
2. pivot\_p : An object. Default is zero.

### Returns:

A **sequence**, { {less than pivot}, {equal to pivot}, {greater than pivot} }

### Comments:

pivot is used as a split up a sequence relative to a specific value.

### Example 1:

```
include std/sequence.e
include std/console.e
sequence s

s = {7, 2, 8.5, 6, 6, -4.8, 6, 6, 3.341, -8, "text"}
s = pivot(s,6)
console:display(s)
--> Ans: {{2, -4.8, 3.341, -8}, {6, 6, 6, 6}, {7, 8.5, "text"}}

? pivot( {4, 1, -4, 6, -1, -7, 9, 10} )
--> Ans: {{-4, -1, -7}, {}, {4, 1, 6, 9, 10}}

? pivot( 5 )
--> Ans: {{}, {}, {5}}
```

### Example 2:

```
include std/sequence.e
include std/console.e
sequence t2

function quicksort(sequence s)
  if length(s) < 2 then
    return s
  end if

  sequence k = pivot(s, s[rand(length(s))])

  return quicksort(k[1]) & k[2] & quicksort(k[3])
end function

t2 = {5,4,7,2,4,9,1,0,4,32,7,54,2,5,8,445,67}
? quicksort(t2)
--> {0,1,2,2,4,4,4,5,5,7,7,8,9,32,54,67,445}
```

## build\_list

```
include std/sequence.e
namespace stdseq
public function build_list(sequence source, object transformer, integer singleton = 1,
  object user_data = {})
```

implements *List Comprehension* or building a list based on the contents of another list.

### Arguments:

1. source : A sequence. The list of items to base the new list upon.
2. transformer : One or more routine\_ids. These are [routine ids](#) of functions that must receive three parameters (object x, sequence i, object u) where 'x' is an item in the source list, 'i' contains the position that 'x' is found in the source list and the length of source, and 'u' is the user\_data value. Each transformer must return a two-element sequence. If the first element is zero, then build\_list continues on with the next transformer function for the same 'x'. If the first element is not zero, the second element is added to the new list being built (other elements are ignored) and build\_list skips the rest of the transformers and processes the next element in source.
3. singleton : An integer. If zero then the transformer functions return multiple list elements. If not zero then the transformer functions return a single item (which might be a sequence).
4. user\_data : Any object. This is passed unchanged to each transformer function.

### Returns:

A **sequence**, The new list of items.

### Comments:

If the transformer is -1 then the source item is just copied.

### Example 1:

```

                                include std/sequence.e
                                sequence s

function remitem(object x, sequence i, object q)
  if (x < q) then
    return {0} -- no output
  else
    return {1,x} -- copy 'x'
  end if
end function

-- Remove negative elements (x < 0)
s = {-3, 0, 1.1, -2, 2, 3, -1.5}
s = build_list( s, routine_id("remitem"), , 0)
? s
--> s is {0, 1.1, 2, 3}

```

## transform

```

include std/sequence.e
namespace stdseq
public function transform(sequence source_data, object transformer_rids)

```

transforms the input sequence by using one or more user-supplied transformers.

### Arguments:

1. `source_data` : A sequence to be transformed.
2. `transformer_rids` : An object. One or more `routine_ids` used to transform the input.

### Returns:

A **sequence**, transformed from the original.

### Comments:

- This works by calling each transformer in order, passing to it the result of the previous transformation. Of course, the first transformer gets the original sequence as passed to this routine.
- Each transformer routine takes one or more parameters. The first is a source sequence to be transformed and others are any user data that may have been supplied to the transform routine.
- Each transformer routine returns a transformed sequence.
- The `transformer_rids` parameters is either a single `routine_id` or a sequence of `routine_ids`. In this second case, the `routine_id` may actually be a multi-element sequence containing the real `routine_id` and some user data to pass to the transformer routine. If there is no user data then the transformer is called with only one parameter.

### Example 1:

```

                                include std/sequence.e
                                include std/text.e
                                sequence res

res = " hello  "

res = transform( res, {
                                { routine_id("trim"), " ", 0 },
                                routine_id("upper")
                              }
                )

puts(1, res )
--> "HELLO"

```

### See Also:

[transmute](#)

## transmute

```

include std/sequence.e
namespace stdseq
public function transmute(sequence source_data, sequence current_items, sequence new_items,

```

```
integer start = 1, integer limit = length(source_data))
```

replaces all instances of any element from the `current_items` sequence that occur in the `source_data` sequence with the corresponding item from the `new_items` sequence.

### Arguments:

1. `source_data` : a sequence, the data that might contain elements from `current_items`
2. `current_items` : a sequence, the set of items to look for in `source_data`. Matching data is replaced with the corresponding data from `new_items`.
3. `new_items` : a sequence, the set of replacement data for any matches found.
4. `start` : an integer, the starting point of the search. Defaults to 1.
5. `limit` : an integer, the maximum number of replacements to be made. Defaults to `length(source_data)`.

### Returns:

A **sequence**, an updated version of `source_data`.

### Comments:

By default, this routine operates on single elements from each of the arguments. That is to say, it scans `source_data` for elements that match any single element in `current_items` and when matched, replaces that with a single element from `new_items`.

For example, you can find all occurrences of 'h', 's', and 't' in a string and replace them with '1', '2', and '3' respectively.

```
transmute(SomeString, "hts", "123")
```

However, the routine can also be used to scan for sub-sequences and/or replace matches with sequences rather than single elements. This is done by making the first element in `current_items` and/or `new_items` an empty sequence.

For example, to find all occurrences of "sh", "th", and "sch" you have the `current_items` as `{}, "sh", "th", "sch"`. Note that for the purposes of determine the corresponding replacement data, the leading empty sequence is not counted, so in this example "th" is the second item.

```
res = transmute("the school shoes", {}, "sh", "th", "sch", "123")
-- res becomes "2e 3ool 1oes"
```

The similar syntax is used to indicate that replacements are sequences and not single elements.

```
res = transmute("the school shoes", {}, "sh", "th", "sch", {}, "SH", "TH", "SCH")
-- res becomes "The SHool SHoes"
```

Using this option also allows you to remove matching data.

```
res = transmute("the school shoes", {}, "sh", "th", "sch", {}, "", "", "")
-- res becomes "e ool oes"
```

Another thing to note is that when using this syntax, you can still mix together atoms and sequences.

```
res = transmute("the school shoes", {}, "sh", 't', "sch", {}, 'x', "TH", "SCH")
-- res becomes "THhe SHool xoes"
```

### Example 1:

```
include std/sequence.e
sequence res

res = transmute("John Smith enjoys uncooked apples.", "aeiouy", "YUOIEA")
puts(1,res)
--> res is "JIhn Sm0th UnjIAs EncIIkUd YpplUs."
```

### See Also:

[find](#) | [match](#) | [replace](#) | [mapping](#)

## sim\_index

```
include std/sequence.e
namespace stdseq
public function sim_index(sequence A, sequence B)
```

calculates the similarity between two sequences.

### Arguments:

1. A : A sequence.
2. B : A sequence.

**Returns:**

An **atom**, the closer to zero, the more the two sequences are alike.

**Comments:**

The calculation is weighted to give mismatched elements towards the front of the sequences larger scores. This means that sequences that differ near the beginning are considered more un-alike than mismatches towards the end of the sequences. Also, unmatched elements from the first sequence are weighted more than unmatched elements from the second sequence.

Two identical sequences return zero. A non-zero means that they are not the same and larger values indicate a larger differences.

**Example 1:**

```
include std/sequence.e

? sim_index("sit", "sin") --> 0.08784
? sim_index("sit", "sat") --> 0.32394
? sim_index("sit", "skit") --> 0.34324
? sim_index("sit", "its") --> 0.68293
? sim_index("sit", "kit") --> 0.86603

? sim_index("knitting", "knitting") --> 0.00000
? sim_index("kitting", "kitten") --> 0.09068
? sim_index("knitting", "knotting") --> 0.27717
? sim_index("knitting", "kitten") --> 0.35332
? sim_index("abacus", "zoological") --> 0.76304
```

**SEQ\_NOALT**

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

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

**remove\_subseq**

```
include std/sequence.e
namespace stdseq
public function remove_subseq(sequence source_list, object alt_value = SEQ_NOALT)
```

removes all sub-sequences from the supplied sequence, optionally replacing them with a supplied alternative value. One common use is to remove all strings from a mixed set of numbers and strings.

**Arguments:**

1. `source_list` : A sequence from which sub-sequences are removed.
2. `alt_value` : An object. The default is `SEQ_NOALT`, which causes sub-sequences to be physically removed, otherwise any other value will be used to replace the sub-sequence.

**Returns:**

A **sequence**, which contains only the atoms from `source_list` and optionally the `alt_value` where sub-sequences used to be.

**Example 1:**

```
include std/sequence.e
sequence s

s = {4,6,"Apple",0.1, {1,2,3}, 4 }
s = remove_subseq(s)
? s
--> s is now {4,6, 0.1, 4 }
-- length now 4

s = {4,6,"Apple",0.1, {1,2,3}, 4}
s = remove_subseq(s,-1)
? s
--> s is now {4,6, -1, 0.1, -1, 4}
-- length unchanged.
```



**enum**

```
include std/sequence.e
namespace stdseq
public enum
```

RD\_INPLACE | RD\_PRESORTED | RD\_SORT

**RD\_INPLACE**

```
include std/sequence.e
namespace stdseq
RD_INPLACE
```

Remove items while preserving the original order of the unique items.

**See Also:**

[remove\\_dups](#)

**RD\_PRESORTED**

```
include std/sequence.e
namespace stdseq
RD_PRESORTED
```

Assume that the elements in `source_data` are already sorted. If they are not already sorted, this option merely removed adjacent duplicate elements.

**See Also:**

[remove\\_dups](#)

**RD\_SORT**

```
include std/sequence.e
namespace stdseq
RD_SORT
```

Will return the unique elements in ascending sorted order.

**See Also:**

[remove\\_dups](#)

**remove\_dups**

```
include std/sequence.e
namespace stdseq
public function remove_dups(sequence source_data, integer proc_option = RD_PRESORTED)
```

removes duplicate itmes.

**Arguments:**

1. `source_data` : A sequence that may contain duplicated items
2. `proc_option` : One of RD\_INPLACE, RD\_PRESORTED, or RD\_SORT.
  - RD\_INPLACE removes items while preserving the original order of the unique items.
  - RD\_PRESORTED assumes that the items in `source_data` are already sorted. If they are not already sorted, this option merely removed adjacent duplicate items.
  - RD\_SORT will return the unique items in ascending sorted order.

**Returns:**

A **sequence**, that contains only the unique items from `source_data`.

**Example 1:**

```
include std/sequence.e
include std/sort.e
sequence s
```

```

s = { 4,7,9,7,2,5,5,9,0,4,4,5,6,5 }
? remove_dups(s, RD_INPLACE)
--> s is { 4,7,9 ,2,5, 0, 6 }

? remove_dups(s, RD_SORT)
--> s is { 0,2,4,5,6,7,9 }

? remove_dups(s, RD_PRESORTED)
--> s is { 4,7,9,7,2,5, 9,0,4, 5,6,5 }

? remove_dups(sort(s), RD_PRESORTED)
--> s is { 0,2,4,5,6,7,9 }

```

### See Also:

[sort](#)

## enum

```

include std/sequence.e
namespace stdseq
public enum

```

COMBINE\_UNSORTED | COMBINE\_SORTED

## combine

```

include std/sequence.e
namespace stdseq
public function combine(sequence source_data, integer proc_option = COMBINE_SORTED)

```

combines all the sub-sequences into a single, optionally sorted, list.

### Arguments:

1. `source_data` : A sequence that contains sub-sequences to be combined.
2. `proc_option` : An integer; COMBINE\_UNSORTED to return a non-sorted list and COMBINE\_SORTED (the default) to return a sorted list.

### Returns:

A **sequence**, that contains all the itmes from all the first-level of sub-sequences from `source_data`.

### Comments:

The itmes in the sub-sequences do not have to be pre-sorted.

Only one level of sub-sequence is combined.

### Example 1:

```

include std/sequence.e
sequence s

s = { {4,7,9}, {7,2,5,9}, {0,4}, {5}, {6,5} }
? combine(s, COMBINE_UNSORTED)
--> s is { 4,7,9, 7,2,5,9, 0,4, 5, 6,5 }

? combine(s, COMBINE_SORTED)
--> s is {0,2,4,4,5,5,5,6,7,7,9,9}

```

### Example 2:

```

include std/sequence.e
include std/console.e
sequence s, s1, s2

s = { {"cat","dog"}, {"fish","whale"}, {"wolf"}, {"snail", "worm"} }

s1 = combine(s)
console:display(s1)
--> s1 is { "cat","dog","fish","snail","whale","wolf","worm" }

s2 = combine(s, COMBINE_UNSORTED)
console:display(s2)

```

```
--> s2 is { "cat","dog", "fish","whale", "wolf", "snail","worm" }
```

### Example 3:

```
include std/sequence.e
include std/console.e
sequence s, s1, s2

s = {"cat", "dog", "fish", "whale", "wolf", "snail", "worm"}

s1 = combine(s)
console:display(s1)
--> s1 is "aaacdeffghhiilllmnoorsstwww"

s2 = combine(s, COMBINE_UNSORTED)
console:display(s2)
--> s2 is "catdogfishwhalewolfsnailworm"
```

## minsize

```
include std/sequence.e
namespace stdseq
public function minsize(object source_data,
    integer min_size = floor(length(source_data)* 1.5),
    object new_data = 0)
```

ensures that the supplied sequence is at least the supplied minimum length.

### Arguments:

1. `source_data` : An object that might need extending.
2. `min_size`: An integer. The minimum length that `source_data` must be. The default is to increase the length of `source_data` by 50 percent.
3. `new_data`: An object. This used to when `source_data` needs to be extended, in which case it is appended as many times as required to make the length equal to `min_size`. The default is 0 zero.

### Returns:

A **sequence**,

```
unchanged
    if original length is less than min_size
padded sequence
    now min_size long.
```

### Comments:

Pads `source_data` to the right until its length reaches `min_size` using `new_data` as filler.

### Example 1:

```
include std/sequence.e
sequence s

? minsize( {4,3,6,2,7,1,2} )
--> {4,3,6,2,7,1,2, 0, 0, 0}
-- default is 1.5 times longer than original length

? minsize( {4,3,6,2,7,1,2}, 10, -1)
--> {4,3,6,2,7,1,2,-1,-1,-1}
-- padded to achieve length 10

? minsize( {4,3,6,2,7,1,2}, 5, -1)
--> {4,3,6,2,7,1,2}
-- unaltered since length exceeds 5
```

### See Also:

[pad\\_head](#) | [pad\\_tail](#)

# Serialization of Euphoria Objects

**Serialization** "converts a *human friendly* Euphoria data-object into a *computer friendly* flat sequence of byte values (encoding values and structure) into a form suitable for saving in a file." To **serialize** is "to convert a native Euphoria data-object into an encoded flat sequence of byte values." To **deserialize** is "to convert an encoded sequence of byte values into a native Euphoria data-object."

A **serialized object** is "the value returned from the `serialize` function."

## Subroutines

### deserialize

```
include std/serialize.e
namespace serialize
public function deserialize(object sdata, integer pos = 1)
```

converts an encoded sequence of binary values into a native Euphoria data-object.

#### Arguments:

1. `sdata` : either a sequence containing one or more concatenated serialized objects or an open file handle. If this is a file handle, the current position in the file is assumed to be at a serialized object in the file.
2. `pos` : optional index into `sdata`. If omitted 1 is assumed. The index must point to the start of a serialized object.

#### Returns:

The return **value**, depends on the input type.

- If `sdata` is a file handle then this function returns a Euphoria object that had been stored in the file, and moves the current file to the first byte after the stored object.
- If `sdata` is a sequence then this returns a two-element sequence. The *first* element is the Euphoria object that corresponds to the serialized object that begins at index `pos`, and the *second* element is the index position in the input parameter just after the serialized object.

#### Comments:

A *serialized object* is one that has been returned from the `serialize` function.

#### Example 1:

```
sequence objcache
objcache = serialize(FirstName) &
           serialize(LastName) &
           serialize(PhoneNumber) &
           serialize(Address)

sequence res
integer pos = 1
res = deserialize( objcache , pos)
FirstName = res[1] pos = res[2]
res = deserialize( objcache , pos)
LastName = res[1] pos = res[2]
res = deserialize( objcache , pos)
PhoneNumber = res[1] pos = res[2]
res = deserialize( objcache , pos)
Address = res[1] pos = res[2]
```

### Example 2:

```
sequence objcache
objcache = serialize({FirstName,
                      LastName,
                      PhoneNumber,
                      Address})

sequence res
res = deserialize( objcache )
FirstName = res[1][1]
LastName = res[1][2]
PhoneNumber = res[1][3]
Address = res[1][4]
```

### Example 3:

```
integer fh
fh = open("cust.dat", "wb")
puts(fh, serialize(FirstName))
puts(fh, serialize(LastName))
puts(fh, serialize(PhoneNumber))
puts(fh, serialize(Address))
close(fh)

fh = open("cust.dat", "rb")
FirstName = deserialize(fh)
LastName = deserialize(fh)
PhoneNumber = deserialize(fh)
Address = deserialize(fh)
close(fh)
```

### Example 4:

```
integer fh
fh = open("cust.dat", "wb")
puts(fh, serialize({FirstName,
                      LastName,
                      PhoneNumber,
                      Address}))
close(fh)

sequence res
fh = open("cust.dat", "rb")
res = deserialize(fh)
close(fh)
FirstName = res[1]
LastName = res[2]
PhoneNumber = res[3]
Address = res[4]
```

## serialize

```
include std/serialize.e
namespace serialize
public function serialize(object x)
```

converts a native Euphoria data-object into an encoded sequence of binary values.

### Arguments:

1. x : any Euphoria data-object.

### Returns:

A **sequence**, this is the serialized version of the input object.

### Comments:

A *serialized object* is one that has been converted to a set of byte values. This can then be written directly out to a file for storage.

You can use the `deserialize` function to convert it back into a native Euphoria object.

### Example 1:

```
integer fh
fh = open("cust.dat", "wb")
puts(fh, serialize(FirstName))
puts(fh, serialize(LastName))
puts(fh, serialize(PhoneNumber))
puts(fh, serialize(Address))
close(fh)

fh = open("cust.dat", "rb")
FirstName = deserialize(fh)
LastName = deserialize(fh)
PhoneNumber = deserialize(fh)
Address = deserialize(fh)
close(fh)
```

### Example 2:

```
integer fh
fh = open("cust.dat", "wb")
puts(fh, serialize({FirstName,
                    LastName,
                    PhoneNumber,
                    Address}))
close(fh)

sequence res
fh = open("cust.dat", "rb")
res = deserialize(fh)
close(fh)
FirstName = res[1]
LastName = res[2]
PhoneNumber = res[3]
Address = res[4]
```

## dump

```
include std/serialize.e
namespace serialize
public function dump(sequence data, sequence filename)
```

saves a Euphoria object to disk in a binary format.

### Arguments:

1. `data` : any Euphoria object.
2. `filename` : the name of the file to save it to.

### Returns:

An **integer**, 0 if the function fails, otherwise the number of bytes in the created file.

### Comments:

If the named file does not exist it is created, otherwise it is overwritten.

You can use the `load` function to recover the data from the file.

### Example 1:

```
include std/serialize.e
integer size = dump(myData, theFileName)
if size = 0 then
    puts(1, "Failed to save data to file\n")
else
    printf(1, "Saved file is %d bytes long\n", size)
end if
```

## load

```
include std/serialize.e
namespace serialize
public function load(sequence filename)
```

restores a Euphoria object that has been saved to disk by `dump`.

### Arguments:

1. filename : the name of the file to restore it from.

### Returns:

A **sequence**, the first element is the result code. If the result code is 0 then it means that the function failed, otherwise the restored data is in the second element.

### Comments:

This is used to load back data from a file created by the `dump` function.

### Example 1:

```
include std/serialize.e
sequence mydata = load(theFileName)
if mydata[1] = 0 then
    puts(1, "Failed to load data from file\n")
else
    mydata = mydata[2] -- Restored data is in second element.
end if
```

# Sorting

- stability
- sort order

## Comparator Function

- default is compare(a,b)
- optional user defined

## Shell Sort Algorithm

`sort` | `custom_sort` | `sort_columns`

## Insertion Sort Algorithm

`insertion_sort`

## Utility

`merge`

See Also:

`demo/allsorts.ex`

## Constants

### ASCENDING

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

Ascending sort order, always the default.

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

### NORMAL\_ORDER

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

The normal sort order used by the custom comparison routine.

### DESCENDING

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



Descending sort order, which is the reverse of ASCENDING.

## REVERSE\_ORDER

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

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

## Subroutines

### sort

```
include std/sort.e
namespace stdsort
public function sort(sequence x, integer order = ASCENDING)
```

sorts the items of a sequence into ascending order.

#### Arguments:

1. *x* : The sequence to be sorted.
2. *order* : the sort order. Default is ASCENDING.

#### Returns:

A **sequence**, a copy of the original sequence in ascending order

#### Comments:

The items can be atoms or sequences.

The standard compare routine is used to compare items. This means that "y is greater than x" is defined by compare(y, x)=1.

This function uses the *shell* sort algorithm. This sort is not "stable" which means items that are considered equal might change position relative to each other.

#### Example 1:

```
include std/sort.e
sequence student_ages
sequence sorted_ages

student_ages = {16,21,23,17,20,20,19,18,16}
sorted_ages = sort( student_ages )
? sorted_ages
--> sorted_ages is {16,16,17,18,19,20,20,21,23}
```

#### Example 2:

```
include std/sort.e
include std/console.e
sequence students

students = {"Freya",16}, {"Bob",21}, {"Diane",23},
           {"Eddy",17}, {"Ian",20}, {"George",20},
           {"Heidi",19}, {"Anne",18}, {"Chris",16}}
```

```
console:display( sort(students) )
--> students is [{"Anne",18}, {"Bob",21}, {"Chris",16},
--             {"Diane",23}, {"Eddy",17}, {"Freya",16},
--             {"George",20}, {"Heidi",19}, {"Ian",20}]
```

## See Also:

[compare](#) | [custom\\_sort](#) | [shuffle](#) | [reverse](#)

## custom\_sort

```
include std/sort.e
namespace stdsort
public function custom_sort(integer custom_compare, sequence x, object data = {},
                           integer order = NORMAL_ORDER)
```

sorts the items of a sequence according to a user-defined order.

## Arguments:

1. `custom_compare` : an integer, the routine-id of the user defined routine that compares two items which appear in the sequence to sort.
2. `x` : the sequence of items to be sorted.
3. `data` : an object, either {} (no custom data, the default), an atom or a non-empty sequence.
4. `order` : an integer, either `NORMAL_ORDER` (the default) or `REVERSE_ORDER`.

## Returns:

A **sequence**, a copy of the original sequence in sorted order

## Errors:

If the user defined routine does not return according to the specifications in the *Comments* section below, an error will occur.

## Comments:

- If some user data is being provided, that data must be either an atom or a sequence with at least one item. **NOTE** only the first item is passed to the user defined comparison routine, any other items are just ignored. The user data is not used or inspected in any way other than passing it to the user defined routine.
- The user defined routine must return an integer *comparison result*
  - a **negative** value if object A must appear before object B
  - a **positive** value if object B must appear before object A
  - 0 if the order does not matter

**NOTE:** The meaning of the value returned by the user-defined routine is reversed when `order = REVERSE_ORDER`. The default is `order = NORMAL_ORDER`, which sorts in order returned by the custom comparison routine.

- When no user data is provided, the user defined routine must accept two objects (A, B) and return just the *comparison result*.
- When some user data is provided, the user defined routine must take three objects (A, B, data). It must return either...
  - an integer, which is a *comparison result*
  - a two-item sequence, in which the first item is a *comparison result* and the second item is the updated user data that is to be used for the next call to the user defined routine.
- The items of `x` can be atoms or sequences. Each time that the sort needs to compare two items in the sequence, it calls the user-defined function to determine the order.
- This function uses the "Shell" sort algorithm. This sort is not "stable" which means the

items that are considered equal might change position relative to each other.

### Example 1:

```

include std/sort.e
include std/console.e
include std/text.e
sequence students
sequence sorted_byage

students = {"Freya",16}, {"Bob",21}, {"Diane",23},
           {"Eddy",17}, {"Ian",20}, {"George",20},
           {"Heidi",19}, {"Anne",18}, {"Chris",16}}

function byage(object a, object b)
  -- If the ages are the same, compare the names otherwise just compare ages.
  if equal(a[2], b[2]) then
    return compare(upper(a[1]), upper(b[1]))
  end if
  return compare(a[2], b[2])
end function

sorted_byage = custom_sort( routine_id("byage"), students )
console:display( sorted_byage )

--> sorted_byage is {"Chris",16}, {"Freya",16}, {"Eddy",17},
--                  {"Anne",18}, {"Heidi",19}, {"George",20},
--                  {"Ian",20}, {"Bob",21}, {"Diane",23}}

sorted_byage = custom_sort( routine_id("byage"), students, , REVERSE_ORDER )
console:display( sorted_byage )

--> sorted_byage is {"Diane",23}, {"Bob",21}, {"Ian",20},
--                  {"George",20}, {"Heidi",19}, {"Anne",18},
--                  {"Eddy",17}, {"Freya",16}, {"Chris",16} }
```

### Example 2:

```

include std/sort.e
include std/text.e
include std/console.e
sequence students
sequence sorted

students =
  {"Freya","Brash",16}, {"Bob","Palmer",21}, {"Diane","Fry",23},
  {"Eddy","Brash",17}, {"Ian","Gungle",20}, {"George","Gungle",20},
  {"Heidi","Smith",19}, {"Anne","Baxter",18}, {"Chris","duPont",16}}

function colsort(object a, object b, sequence cols)
  integer sign
  for i = 1 to length(cols) do
    if cols[i] < 0 then
      sign = -1
      cols[i] = -cols[i]
    else
      sign = 1
    end if
    if not equal(a[cols[i]], b[cols[i]]) then
      return sign * compare(upper(a[cols[i]]), upper(b[cols[i]]))
    end if
  end for

  return 0
end function
```

```

-- Order is age:descending, Surname, Given Name
sequence column_order = {-3,2,1}
sorted = custom_sort( routine_id("colsort"), students, {column_order} )
console:display(sorted)

--> sorted is
-- {"Diane","Fry",23}, {"Bob","Palmer",21}, {"George","Gungle",20},
-- {"Ian","Gungle",20}, {"Heidi","Smith",19}, {"Anne","Baxter",18},
-- {"Eddy","Brash",17}, {"Freya","Brash",16}, {"Chris","duPont",16}}

sorted = custom_sort( routine_id("colsort"), students, {column_order}, REVERSE_ORDER )
console:display(sorted)

--> sorted is
-- {"Chris","duPont",16}, {"Freya","Brash",16}, {"Eddy","Brash",17},
-- {"Anne","Baxter",18}, {"Heidi","Smith",19}, {"Ian","Gungle",20},
-- {"George","Gungle",20}, {"Bob","Palmer",21}, {"Diane","Fry",23}}

```

## See Also:

[compare](#) | [sort](#)

## sort\_columns

```

include std/sort.e
namespace stdsort
public function sort_columns(sequence x, sequence column_list)

```

sorts the rows in a sequence according to a user-defined column order.

## Arguments:

1. `x` : a sequence, holding the sequences to be sorted.
2. `column_list` : a list of columns indexes `x` is to be sorted by.

## Returns:

A **sequence**, a copy of the original sequence in sorted order.

## Comments:

`x` must be a sequence of sequences.

A non-existent column is treated as coming before an existing column. This allows sorting of records that are shorter than the columns in the column list.

By default columns are sorted in ascending order. To sort in descending order make the column number negative.

This function uses the "Shell" sort algorithm. This sort is not "stable" which means items that are considered equal might change position relative to each other.

## Example 1:

```

include std/sort.e
include std/console.e
sequence students
sequence sorted

students =
  {"Freya","Brash",16}, {"Bob","Palmer",21}, {"Diane","Fry",23},
  {"Eddy","Brash",17}, {"Ian","Gungle",20}, {"George","Gungle",20},
  {"Heidi","Smith",19}, {"Anne","Baxter",18}, {"Chris","duPont",16}}

```

```
sorted = sort_columns( students, {3} )
console:display(sorted)

--> sorted is
--  {"Freya","Brash",16}, {"Chris","duPont",16}, {"Eddy","Brash",17},
--  {"Anne","Baxter",18}, {"Heidi","Smith",19}, {"Ian","Gungle",20},
--  {"George","Gungle",20}, {"Bob","Palmer",21}, {"Diane","Fry",23}}
```

## Example 2:

```
include std/sort.e
include std/filesys.e
include std/console.e
sequence dirlist
sequence sorted

-- Change to this line if on windows
-- dirlist = dir("c:\\temp")

-- Use this line if on unix
dirlist = dir( "/home/mint17/temp" )
console:display( dirlist )
--> dirlist is
--  {"example4.ex","",145,2014,11,6,12,25,8,0,0},
--  {"example3.ex","",0,2014,11,6,12,25,43,0,0},
--  {"example1.ex","",783,2014,11,6,12,24,54,0,0},
--  {"..","d",24576,2014,11,6,12,22,27,0,0},
--  {"example2.ex","",145,2014,11,6,12,25,8,0,0},
--  {"..","d",4096,2014,11,6,12,38,36,0,0}}

-- Order is Size:descending, Name

sorted = sort_columns( dirlist, {-D_SIZE, D_NAME} )
console:display( sorted )
--> sorted is
--  {"..","d",4096,2014,11,6,12,38,36,0,0},
--  {"..","d",24576,2014,11,6,12,22,27,0,0},
--  {"example1.ex","",783,2014,11,6,12,24,54,0,0},
--  {"example2.ex","",145,2014,11,6,12,25,8,0,0},
--  {"example4.ex","",145,2014,11,6,12,25,8,0,0},
--  {"example3.ex","",0,2014,11,6,12,25,43,0,0}}
```

## See Also:

[compare](#) | [sort](#)

## merge

```
include std/sort.e
namespace stdsort
public function merge(sequence a, sequence b, integer compfunc = - 1, object userdata = "")
```

merges two pre-sorted sequences into a single sequence.

## Arguments:

1. a : a sequence, holding pre-sorted data.
2. b : a sequence, holding pre-sorted data.
3. compfunc : an integer, either -1 or the routine id of a user-defined comparison function.

## Returns:

A **sequence**, consisting of a and b merged together.

## Comments:

- If a or b is not already sorted, the resulting sequence might not be sorted either.
- The input sequences do not have to be the same size.
- The user-defined comparison function must accept two objects and return an integer. It returns -1 if the first object must appear before the second one, and 1 if the first object must after before the second one, and 0 if the order doesn't matter.

## Example 1:

```

                                include std/sort.e
                                sequence X,Y

X = sort( {5,3,7,1,9,0} )
  ? X --> {0,1,3,5,7,9}

Y = sort( {6,8,10,2} )
  ? Y --> {2,6,8,10}

? merge(X,Y)
  --> {0,1,2,3,5,6,7,8,9,10}

```

## Example 2:

```

                                include std/sort.e
                                include std/console.e
                                sequence X, Y

X = { {"Freya",16}, {"Bob",21}, {"Diane",23}, {"Eddy",17} }
Y = { {"Ian",20}, {"George",20}, {"Heidi",19}, {"Anne",18}, {"Chris",16} }

X = sort(X)
console:display( merge(X,Y) )

-- {"Bob",21}, {"Diane",23}, {"Eddy",17}, {"Freya",16},
-- {"Ian",20}, {"George",20}, {"Heidi",19}, {"Anne",18}, {"Chris",16}}

Y = sort(Y)
console:display( merge(X,Y) )

-- {"Anne",18}, {"Bob",21}, {"Chris",16}, {"Diane",23},
-- {"Eddy",17}, {"Freya",16}, {"George",20}, {"Heidi",19}, {"Ian",20}}

```

## See Also:

[compare](#) | [sort](#)

## insertion\_sort

```

include std/sort.e
namespace stdsort
public function insertion_sort(sequence s, object e = "", integer compfunc = - 1,
                             object userdata = "")

```

sorts a sequence and optionally another object together.

## Arguments:

1. s : a sequence, holding data to be sorted.
2. e : an object. If this is an atom, it is sorted in withs. If this is a non-empty sequence then s and e are both sorted independantly using thisinsertion\_sort function and then the results are merged and returned.
3. compfunc : an integer, either -1 or the routine id of a user-defined comparison function.

## Returns:

A **sequence**, consisting of s and e sorted together.

### Comments:

- This routine is usually a lot faster than the standard sort whens and e are (mostly) sorted before calling the function. For example, you can use this routine to quickly add to a sorted list.
- The input sequences do not have to be the same size.
- The user-defined comparison function must accept two objects and return an integer. It returns -1 if the first object must appear before the second one, and 1 if the first object must after before the second one, and 0 if the order does not matter.

### Example 1:

```
-- code fragment only
sequence X = {}
while true do
  newdata = get_data()
  if compare(-1, newdata) then
    exit
  end if
  X = insertion_sort(X, newdata)
  process(new_data)
end while
```

### Example 2:

```
include std/sort.e
include std/console.e
sequence students
sequence newstudents

students = {"Anne",18}, {"Bob",21}, {"Chris",16},
           {"Diane",23}, {"Eddy",17}, {"Freya",16},
           {"George",20}, {"Heidi",19}, {"Ian",20}}      --sorted

newstudents = { {"Fred",18}, {"Debbie",20} }             --unsorted

console:display( insertion_sort( students, newstudents ) )
-->      {"Anne",18}, {"Bob",21}, {"Chris",16},          --sorted
--      {"Debbie",20}, {"Diane",23},
--      {"Eddy",17}, {"Fred",18}, {"Freya",16},
--      {"George",20}, {"Heidi",19}, {"Ian",20}}
```

### See Also:

[compare](#) | [sort](#) | [merge](#)

# STRING

- `locale.e`
- `localeconv.e`
- `regex.e`
- `text.e`

1. \* `unicode.e`
2. \* `ucstypes.e`

- `wildcard.e`
- `base64.e`



# Locale Routines

## Message Translation Functions

### set\_lang\_path

```
include std/locale.e
namespace locale
public procedure set_lang_path(object pp)
```

sets the language path.

#### Arguments:

1. pp : an object, either an actual path or an atom.

#### Comments:

When the language path is not set, and it is unset by default, `set` does not load any language file.

#### See Also:

`set`

### get\_lang\_path

```
include std/locale.e
namespace locale
public function get_lang_path()
```

gets the language path.

#### Returns:

An **object**, the current language path.

#### See Also:

`get_lang_path`

### lang\_load

```
include std/locale.e
namespace locale
public function lang_load(sequence filename)
```

loads a language file.

#### Arguments:

1. filename : a sequence, the name of the file to load. If no file extension is supplied, then ".lng" is used.

#### Returns:

A language **map**, if successful. This is to be used when calling [translate](#).

If the load fails it returns a zero.

### Comments:

The language file must be made of lines which are either comments, empty lines or translations. Note that leading whitespace is ignored on all lines except continuation lines.

- *Comments* are lines that begin with a # character and extend to the end of the line.
- *Empty Lines* are ignored.
- *Translations* have two forms.

```
keyword translation_text
```

In which the 'keyword' is a word that must not have any spaces in it.

```
keyphrase = translation_text
```

In which the 'keyphrase' is anything up to the first = equals symbol.

It is possible to have the translation text span multiple lines. You do this by having an & ampersand as the last character of the line. These are placed by newline characters when loading.

### Example 1:

```
# Example translation file
#

hello Hola
world Mundo
greeting %s, %s!

help text = &
This is an example of some &
translation text that spans &
multiple lines.

# End of example P0 #2
```

### See Also:

[translate](#)

## set\_def\_lang

```
include std/locale.e
namespace locale
public procedure set_def_lang(object langmap)
```

sets the default language (translation) map.

### Arguments:

1. langmap : A value returned by [lang\\_load](#), or zero to remove any default map.

### Example 1:

```
set_def_lang( lang_load("appmsgs") )
```

## get\_def\_lang

```
include std/locale.e
namespace locale
public function get_def_lang()
```

gets the default language (translation) map.

### Arguments:

none.

### Returns:

An **object**, a language map, or zero if there is no default language map yet.

### Example 1:

```
object langmap = get_def_lang()
```

## translate

```
include std/locale.e
namespace locale
public function translate(sequence word, object langmap = 0, object defval = "",
    integer mode = 0)
```

translates a word, using the current language file.

### Arguments:

1. word : a sequence, the word to translate.
2. langmap : Either a value returned by `lang_load` or zero to use the default language map
3. defval : a object. The value to return if the word cannot be translated. Default is "". If defval is PINF then the word is returned if it can not be translated.
4. mode : an integer. If zero (the default) it uses word as the keyword and returns the translation text. If not zero it uses word as the translation and returns the keyword.

### Returns:

A **sequence**, the value associated with word, or defval if there is no association.

### Example 1:

```
sequence newword
newword = translate(msgtext)
if length(msgtext) = 0 then
    error_message(msgtext)
else
    error_message(newword)
end if
```

### Example 2:

```
error_message(translate(msgtext, , PINF))
```

### See Also:

`set` | `lang_load`

## trsprintf

```
include std/locale.e
namespace locale
public function trsprintf(sequence fmt, sequence data, object langmap = 0)
```

returns a formatted string with automatic translation performed on the parameters.

### Arguments:

1. `fmt` : A sequence. Contains the formatting string. See `printf` for details.
2. `data` : A sequence. Contains the data that goes into the formatted result. see `printf` for details.
3. `langmap` : An object. Either 0 (the default) to use the default language maps, or the result returned from `lang_load` to specify a particular language map.

### Returns:

A **sequence**, the formatted result.

### Comments:

This works very much like the `sprintf` function. The difference is that the `fmt` sequence and sequences contained in the `data` parameter are **translated** before passing them to `sprintf`. If an item has no translation, it remains unchanged.

Further more, after the translation pass, if the result text begins with "`__`", the "`__`" is removed. This function can be used when you do not want an item to be translated.

### Example 1:

```
-- Assuming a language has been loaded and
-- "greeting" translates as '%s %s, %s'
-- "hello" translates as "G'day"
-- "how are you today" translates as "How's the family?"
sequence UserName = "Bob"
sequence result = trsprintf( "greeting", {"hello", "__" & UserName, "how are you today"})
--> "G'day Bob, How's the family?"
```

## Time and Number Translation

### set

```
include std/locale.e
namespace locale
public function set(sequence new_locale)
```

sets the computer locale, and possibly loads an appropriate translation file.

### Arguments:

1. `new_locale` : a sequence representing a new locale.

### Returns:

An **integer**, either 0 on failure or 1 on success.

### Comments:

Locale strings have the following format: `xx_YY` or `xx_YY.xyz` . The `xx` part refers to a culture, or main language or script. For instance, "en" refers to English, "de" refers to German, and so on. For some languages, a script may be specified, like "mn\_Cyrl\_MN" (Mongolian in cyrillic transcription).

The YY part refers to a subculture, or variant, of the main language. For instance, "fr\_FR" refers to metropolitan France, while "fr\_BE" refers to the variant spoken in Wallonie, the French speaking region of Belgium.

The optional .xyz part specifies an encoding, like .utf8 or .1252 . This is required in some cases.

## get

```
include std/locale.e
namespace locale
public function get()
```

gets the current locale string.

### Returns:

A **sequence**, a locale string.

### See Also:

[set](#)

## money

```
include std/locale.e
namespace locale
public function money(object amount)
```

converts an amount of currency into a string representing that amount.

### Arguments:

1. amount : an atom, the value to write out.

### Returns:

A **sequence**, a string that writes out amount of current currency.

### Example 1:

```
-- Assuming an en_US locale
money(1020.5) -- returns"$1,020.50"
```

### See Also:

[set](#) | [number](#)

## number

```
include std/locale.e
namespace locale
public function number(object num)
```

converts a number into a string representing that number.

### Arguments:

1. `num` : an atom, the value to write out.

### Returns:

A **sequence**, a string that writes out `num`.

### Example 1:

```
-- Assuming an en_US locale
number(1020.5) -- returns "1,020.50"
```

### See Also:

[set](#) | [money](#)

## datetime

```
include std/locale.e
namespace locale
public function datetime(sequence fmt, datetime :datetime dtm)
```

formats a date according to current locale.

### Arguments:

1. `fmt` : A format string, as described in [datetime:format](#)
2. `dtm` : the datetime to write out.

### Returns:

A **sequence**, representing the formatted date.

### Example 1:

```
include std/datetime.e

datetime("Today is a %A", datetime:now())
```

### See Also:

[datetime:format](#)

## get\_text

```
include std/locale.e
namespace locale
public function get_text(integer MsgNum, sequence LocalQuals = {}, sequence DBBase = "teksto")
```

gets the text associated with the message number in the requested locale.

### Arguments:

1. `MsgNum` : An integer. The message number whose text you are trying to get.
2. `LocalQuals` : A sequence. Zero or more locale codes. Default is {}.
3. `DBBase`: A sequence. The base name for the database files containing the locale text strings. The default is "teksto".

### Returns:

A string **sequence**, the text associated with the message number and locale.  
The **integer** zero, if associated text can not be found for any reason.

## Comments:

- This first scans the database or databases linked to the locale codes supplied.
- The database name for each locale takes the format of "<DBBase>\_<Locale>.edb" so if the default DBBase is used, and the locales supplied are{"enus", "enau"} the databases scanned are "teksto\_enus.edb" and "teksto\_enau.edb". The database table name searched is "1" with the key being the message number, and the text is the record data.
- If the message is not found in these databases (or the databases do not exist) a database called "<DBBase>.edb" is searched. Again the table name is "1" but it first looks for keys with the format {<locale>,msgnum} and failing that it looks for keys in the format{"", msgnum}, and if that fails it looks for a key of just themsgnum.

# Locale Names

## Constants

*Windows* locale names:

af-ZA	sq-AL	gsw-FR	am-ET	ar-DZ	ar-BH	ar-EG	ar-IQ
ar-JO	ar-KW	ar-LB	ar-LY	ar-MA	ar-OM	ar-QA	ar-SA
ar-SY	ar-TN	ar-AE	ar-YE	hy-AM	as-IN	az-Cyrl-AZ	az-Latn-AZ
ba-RU	eu-ES	be-BY	bn-IN	bs-Cyrl-BA	bs-Latn-BA	br-FR	bg-BG
ca-ES	zh-HK	zh-MO	zh-CN	zh-SG	zh-TW	co-FR	hr-BA
hr-HR	cs-CZ	da-DK	prs-AF	dv-MV	nl-BE	nl-NL	en-AU
en-BZ	en-CA	en-029	en-IN	en-IE	en-JM	en-MY	en-NZ
en-PH	en-SG	en-ZA	en-TT	en-GB	en-US	en-ZW	et-EE
fo-FO	fil-PH	fi-FI	fr-BE	fr-CA	fr-FR	fr-LU	fr-MC
fr-CH	fy-NL	gl-ES	ka-GE	de-AT	de-DE	de-LI	de-LU
de-CH	el-GR	kl-GL	gu-IN	ha-Latn-NG	he-IL	hi-IN	hu-HU
is-IS	ig-NG	id-ID	iu-Latn-CA	iu-Cans-CA	ga-IE	it-IT	it-CH
ja-JP	kn-IN	kk-KZ	kh-KH	qut-GT	rw-RW	kok-IN	ko-KR
ky-KG	lo-LA	lv-LV	lt-LT	dsb-DE	lb-LU	mk-MK	ms-BN
ms-MY	ml-IN	mt-MT	mi-NZ	arn-CL	mr-IN	moh-CA	mn-Cyrl-MN
mn-Mong-CN	ne-IN	ne-NP	nb-NO	nn-NO	oc-FR	or-IN	ps-AF
fa-IR	pl-PL	pt-BR	pt-PT	pa-IN	quz-BO	quz-EC	quz-PE
ro-RO	rm-CH	ru-RU	smn-FI	smj-NO	smj-SE	se-FI	se-NO
se-SE	sms-FI	sma-NO	sma-SE	sa-IN	sr-Cyrl-BA	sr-Latn-BA	sr-Cyrl-CS
sr-Latn-CS	ns-ZA	tn-ZA	si-LK	sk-SK	sl-SI	es-AR	es-BO
es-CL	es-CO	es-CR	es-DO	es-EC	es-SV	es-GT	es-HN
es-MX	es-NI	es-PA	es-PY	es-PE	es-PR	es-ES	es-ES_tradnl
es-US	es-UY	es-VE	sw-KE	sv-FI	sv-SE	syr-SY	tg-Cyrl-TJ
tmz-Latn-DZ	ta-IN	tt-RU	te-IN	th-TH	bo-BT	bo-CN	tr-TR
tk-TM	ug-CN	uk-UA	wen-DE	tr-IN	ur-PK	uz-Cyrl-UZ	uz-Latn-UZ
vi-VN	cy-GB	wo-SN	xh-ZA	sah-RU	ii-CN	yo-NG	zu-ZA

## w32\_names

```
include std/localeconv.e
namespace localconv
public constant w32_names
```

## w32\_name\_canonical

```
include std/localeconv.e
namespace localconv
public constant w32_name_canonical
```

Canonical locale names for *Windows*:

Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa



Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Basque_Spain	Basque_Spain	Belarusian_Belarus
Belarusian_Belarus	Belarusian_Belarus	Belarusian_Belarus
Belarusian_Belarus	Belarusian_Belarus	Catalan_Spain
Catalan_Spain	Catalan_Spain	Catalan_Spain
Catalan_Spain	Catalan_Spain	Catalan_Spain
Catalan_Spain	Catalan_Spain	Catalan_Spain
Danish_Denmark	Danish_Denmark	Danish_Denmark
Danish_Denmark	Danish_Denmark	English_Australia
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
Finnish_Finland	French_France	French_France
French_France	French_France	French_France
French_France	French_France	French_France
French_France	French_France	French_France
French_France	French_France	French_France
French_France	French_France	French_France
Hungarian_Hungary	Hungarian_Hungary	Hungarian_Hungary
Hungarian_Hungary	Hungarian_Hungary	Hungarian_Hungary
Hungarian_Hungary	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Italian_Italy	Italian_Italy
Italian_Italy	Romanian_Romania	Romanian_Romania
Russian_Russia	Russian_Russia	Russian_Russia
Russian_Russia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia
Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia
Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia
Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia
Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia
Serbian (Cyrillic)_Serbia	Slovak_Slovakia	Estonian_Estonia
Estonian_Estonia	Estonian_Estonia	Estonian_Estonia
Estonian_Estonia	Estonian_Estonia	Estonian_Estonia
Estonian_Estonia	Estonian_Estonia	Estonian_Estonia

Estonian_Estonia	Estonian_Estonia	Estonian_Estonia
Estonian_Estonia	Estonian_Estonia	Estonian_Estonia
Estonian_Estonia	Estonian_Estonia	Estonian_Estonia
Estonian_Estonia	Estonian_Estonia	Swedish_Sweden
Swedish_Sweden	Swedish_Sweden	Swedish_Sweden
Swedish_Sweden	Swedish_Sweden	Swedish_Sweden
Swedish_Sweden	Swedish_Sweden	Swedish_Sweden
Swedish_Sweden	Swedish_Sweden	Swedish_Sweden
Swedish_Sweden	Swedish_Sweden	Ukrainian_Ukraine
Ukrainian_Ukraine	Ukrainian_Ukraine	Ukrainian_Ukraine
Ukrainian_Ukraine	Ukrainian_Ukraine	Ukrainian_Ukraine
Ukrainian_Ukraine	Ukrainian_Ukraine	Ukrainian_Ukraine
Ukrainian_Ukraine	Ukrainian_Ukraine	Ukrainian_Ukraine
Ukrainian_Ukraine		

## posix\_names

```
include std/localeconv.e
namespace localconv
public constant posix_names
```

POSIX locale names:

af_ZA	sq_AL	gsw_FR	am_ET	ar_DZ	ar_BH	ar_EG	ar_IQ
ar_JO	ar_KW	ar_LB	ar_LY	ar_MA	ar_OM	ar_QA	ar_SA
ar_SY	ar_TN	ar_AE	ar_YE	hy_AM	as_IN	az_Cyrl_AZ	az_Latn_AZ
ba_RU	eu_ES	be_BY	bn_IN	bs_Cyrl_BA	bs_Latn_BA	br_FR	bg_BG
ca_ES	zh_HK	zh_MO	zh_CN	zh_SG	zh_TW	co_FR	hr_BA
hr_HR	cs_CZ	da_DK	prs_AF	dv_MV	nl_BE	nl_NL	en_AU
en_BZ	en_CA	en_029	en_IN	en_IE	en_JM	en_MY	en_NZ
en_PH	en_SG	en_ZA	en_TT	en_GB	en_US	en_ZW	et_EE
fo_FO	fil_PH	fi_FI	fr_BE	fr_CA	fr_FR	fr_LU	fr_MC
fr_CH	fy_NL	gl_ES	ka_GE	de_AT	de_DE	de_LI	de_LU
de_CH	el_GR	kl_GL	gu_IN	ha_Latn_NG	he_IL	hi_IN	hu_HU
is_IS	ig_NG	id_ID	iu_Latn_CA	iu_Cans_CA	ga_IE	it_IT	it_CH
ja_JP	kn_IN	kk_KZ	kh_KH	qut_GT	rw_RW	kok_IN	ko_KR
ky_KG	lo_LA	lv_LV	lt_LT	dsb_DE	lb_LU	mk_MK	ms_BN
ms_MY	ml_IN	mt_MT	mi_NZ	arn_CL	mr_IN	moh_CA	mn_Cyrl_MN
mn_Mong_CN	ne_IN	ne_NP	nb_NO	nn_NO	oc_FR	or_IN	ps_AF
fa_IR	pl_PL	pt_BR	pt_PT	pa_IN	quz_BO	quz_EC	quz_PE
ro_RO	rm_CH	ru_RU	smn_FI	smj_NO	smj_SE	se_FI	se_NO
se_SE	sms_FI	sma_NO	sma_SE	sa_IN	sr_Cyrl_BA	sr_Latn_BA	sr_Cyrl_CS
sr_Latn_CS	ns_ZA	tn_ZA	si_LK	sk_SK	sl_SI	es_AR	es_BO
es_CL	es_CO	es_CR	es_DO	es_EC	es_SV	es_GT	es_HN
es_MX	es_NI	es_PA	es_PY	es_PE	es_PR	es_ES	es_ES_tradnl
es_US	es_UY	es_VE	sw_KE	sv_FI	sv_SE	syr_SY	tg_Cyrl_TJ
tmz_Latn_DZ	ta_IN	tt_RU	te_IN	th_TH	bo_BT	bo_CN	tr_TR
tk_TM	ug_CN	uk_UA	wen_DE	tr_IN	ur_PK	uz_Cyrl_UZ	uz_Latn_UZ
vi_VN	cy_GB	wo_SN	xh_ZA	sah_RU	ii_CN	yo_NG	zu_ZA

## locale\_canonical

```
include std/localeconv.e
namespace localconv
```

```
public constant locale_canonical
```

## platform\_locale

```
include std/localeconv.e  
namespace localconv  
public constant platform_locale
```

## Locale Name Translation

### canonical

```
include std/localeconv.e  
namespace localconv  
public function canonical(sequence new_locale)
```

Get canonical name for a locale.

#### Arguments:

1. `new_locale` : a sequence, the string for the locale.

#### Returns:

A **sequence**, either the translated locale on success or `new_locale` on failure.

#### See Also:

[get](#) | [set](#) | [decanonical](#)

### decanonical

```
include std/localeconv.e  
namespace localconv  
public function decanonical(sequence new_locale)
```

gets the translation of a locale string for current platform.

#### Arguments:

1. `new_locale`: a sequence, the string for the locale.

#### Returns:

A **sequence**, either the translated locale on success or `new_locale` on failure.

#### See Also:

[get](#) | [set](#) | [canonical](#)

### canon2win

```
include std/localeconv.e  
namespace localconv  
public function canon2win(sequence new_locale)
```

gets the translation of a canonical locale string for the *Windows* platform.

### Arguments:

1. `new_locale`: a sequence, the string for the locale.

### Returns:

A **sequence**, either the Windows native locale name on success or "C" on failure.

### See Also:

[get](#) | [set](#) | [canonical](#) | [decanonical](#)

# Regular Expressions

## Introduction

Regular expressions in Euphoria are based on the PCRE (Perl Compatible Regular Expressions) library created by Philip Hazel.

This document will detail the Euphoria interface to Regular Expressions, not really regular expression syntax. It is a very complex subject that many books have been written on. Here are a few good resources online that can help while learning regular expressions.

- [EUFORUM Article](#)
- [Perl Regular Expressions Man Page](#)
- [Regular Expression Library](#) (user supplied regular expressions for just about any task).
- [Wikipedia Regular Expression Article](#)
- [Man page of PCRE in HTML](#)

## General Use

Many functions take an optional options argument. This argument can be either a single option constant (see [Option Constants](#)), multiple option constants or'ed together into a single atom or a sequence of options, in which the function will take care of ensuring the are or'ed together correctly. Options are like their C equivalents with the 'PCRE\_' prefix stripped off. Name spaces disambiguate symbols so we do not need this prefix.

All strings passed into this library must be either 8-bit per character strings or UTF which uses multiple bytes to encode UNICODE characters. You can use UTF8 encoded UNICODE strings when you pass the UTF8 option.

## Option Constants

### Compile Time and Match Time

When a regular expression object is created via `new` we call also say it gets "compiled." The options you may use for this are called "compile time" option constants. Once the regular expression is created you can use the other functions that take this regular expression and a string. These routines' options are called "match time" option constants. To not set any options at all, do not supply the options argument or supply `DEFAULT`.

#### Compile Time Option Constants

The only options that may set at "compile time" (that is to pass to `new`) are:

`ANCHORED` | `AUTO_CALLOUT` | `BSR_ANYCRLF` | `BSR_UNICODE` | `CASELESS` | `DEFAULT` | `DOLLAR_ENDONLY` | `DOTALL` | `DUPNAMES` | `EXTENDED` | `EXTRA` | `FIRSTLINE` | `MULTILINE` | `NEWLINE_CR` | `NEWLINE_LF` | `NEWLINE_CRLF` | `NEWLINE_ANY` | `NEWLINE_ANYCRLF` | `NO_AUTO_CAPTURE` | `NO_UTF8_CHECK` | `UNGREEDY` | `UTF8`

#### Match Time Option Constants

Options that may be set at "match time" are:

`ANCHORED` | `NEWLINE_CR` | `NEWLINE_LF` | `NEWLINE_CRLF` | `NEWLINE_ANY` | `NEWLINE_ANYCRLF` | `NOTBOL` | `NOTEOL` | `NOTEMPTY` | `NO_UTF8_CHECK`

Routines that take match time option constants: match, split, or replace a regular expression against some string.

## ANCHORED

```
public constant ANCHORED
```

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

## AUTO\_CALLOUT

```
public constant AUTO_CALLOUT
```

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

## BSR\_ANYCRLF

```
public constant BSR_ANYCRLF
```

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

## BSR\_UNICODE

```
public constant BSR_UNICODE
```

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

## CASELESS

```
public constant CASELESS
```

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

## DEFAULT

```
public constant DEFAULT
```

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

## DFA\_SHORTEST

```
public constant DFA_SHORTEST
```

This is NOT used by any standard library routine.

## DFA\_RESTART

```
public constant DFA_RESTART
```

This is NOT used by any standard library routine.

## DOLLAR\_ENDONLY

```
public constant DOLLAR_ENDONLY
```

If this bit is set, a dollar sign metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar sign also matches immediately before a newline at the end of the string (but not before any other newlines). Thus you must include the newline character in the pattern before the dollar sign if you want to match a line that contains a newline character. The DOLLAR\_ENDONLY option is ignored if MULTILINE is set. There is no way to set this option within a pattern. This is passed to `new`.

## DOTALL

```
public constant DOTALL
```

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

## DUPNAMES

```
public constant DUPNAMES
```

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

## EXTENDED

```
public constant EXTENDED
```

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

## EXTRA

```
public constant EXTRA
```

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

## FIRSTLINE

```
public constant FIRSTLINE
```

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

## MULTILINE

```
public constant MULTILINE
```

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

## NEWLINE\_CR

```
public constant NEWLINE_CR
```

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

## NEWLINE\_LF

```
public constant NEWLINE_LF
```

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

## NEWLINE\_CRLF

```
public constant NEWLINE_CRLF
```

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

## NEWLINE\_ANY

```
public constant NEWLINE_ANY
```

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



## NEWLINE\_ANYCRLF

```
public constant NEWLINE_ANYCRLF
```

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

## NOTBOL

```
public constant NOTBOL
```

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

## NOTEOL

```
public constant NOTEOL
```

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

## NO\_AUTO\_CAPTURE

```
public constant NO_AUTO_CAPTURE
```

Disables capturing subpatterns except when the subpatterns are named. This is passed to *new*.

## NO\_UTF8\_CHECK

```
public constant NO_UTF8_CHECK
```

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

## NOTEMPTY

```
public constant NOTEMPTY
```

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

## PARTIAL

```
public constant PARTIAL
```

This option has no effect on whether a match will occur or not. However, it does affect the error code generated by `find` in the event of a failure: If for some pattern `re`, and two strings `s1` and `s2`, `find( re, s1 & s2 )` would return a match but both `find( re, s1 )` and `find( re, s2 )` would not, then `find( re, s1, 1, PCRE_PARTIAL )` will return `ERROR_PARTIAL` rather than `ERROR_NOMATCH`. We say `s1` has a *partial match* of `re`.

Note that `find( re, s2, 1, PCRE_PARTIAL )` will `ERROR_NOMATCH`. In C, this constant is called `PCRE_PARTIAL`.

## STRING\_OFFSETS

```
public constant STRING_OFFSETS
```

This is used by `matches` and `all_matches`.

## UNGREEDY

```
public constant UNGREEDY
```

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

## UTF8

```
public constant UTF8
```

Makes strings passed in to be interpreted as a UTF8 encoded string. This is passed to `new`.

## Error Constants

Error constants differ from their C equivalents as they do not have `PCRE_` prepended to each name.

## ERROR\_NOMATCH

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

There was no match found.

## ERROR\_NULL

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

There was an internal error in the EUPHORIA wrapper (`std/regex.e` in the standard include directory or `be_regex.c` in the EUPHORIA source).

## ERROR\_BADOPTION

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

There was an internal error in the EUPHORIA wrapper (std/regex.e in the standard include directory or be\_regex.c in the EUPHORIA source).

## ERROR\_BADMAGIC

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

The pattern passed is not a value returned from `new`.

## ERROR\_UNKNOWN\_OPCODE

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

An internal error either in the pcre library EUPHORIA uses or its wrapper occurred.

## ERROR\_UNKNOWN\_NODE

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

An internal error either in the pcre library EUPHORIA uses or its wrapper occurred.

## ERROR\_NOMEMORY

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

Out of memory.

## ERROR\_NOSUBSTRING

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

The wrapper or the PCRE backend did not preallocate enough capturing groups for this pattern.

## ERROR\_MATCHLIMIT

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

Too many matches encountered.

## ERROR\_CALLOUT

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

Not applicable to our implementation.

## ERROR\_BADUTF8

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

The subject or pattern is not valid UTF8 but it was specified as such with [UTF8](#).

## ERROR\_BADUTF8\_OFFSET

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

The offset specified does not start on a UTF8 character boundary but it was specified as UTF8 with [UTF8](#).

## ERROR\_PARTIAL

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

Pattern didn't match, but there is a *partial match*. See [PARTIAL](#).

## ERROR\_BADPARTIAL

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

PCRE backend doesn't support partial matching for this pattern.

## ERROR\_INTERNAL

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

## ERROR\_BADCOUNT

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

size parameter to find is less than minus 1.

## ERROR\_DFA\_UITEM

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

Not applicable to our implementation: The PCRE wrapper doesn't use DFA routines

## ERROR\_DFA\_UCOND

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

Not applicable to our implementation: The PCRE wrapper doesn't use DFA routines

## ERROR\_DFA\_UMLIMIT

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

Not applicable to our implementation: The PCRE wrapper doesn't use DFA routines

## ERROR\_DFA\_WSSIZE

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

Not applicable to our implementation: The PCRE wrapper doesn't use DFA routines

## ERROR\_DFA\_RECURSE

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

Not applicable to our implementation: The PCRE wrapper doesn't use DFA routines

## ERROR\_RECURSIONLIMIT

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

Too much recursion used for match.

## ERROR\_NULLWSLIMIT

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

This error isn't in the source code.

## ERROR\_BADNEWLINE

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

Both BSR\_UNICODE and BSR\_ANY options were specified. These options are contradictory.

## error\_names

```
include std/regex.e
namespace regex
public constant error_names
```

## Create and Destroy

### regex

```
include std/regex.e
namespace regex
public type regex(object o)
```

Regular expression type

### option\_spec

```
include std/regex.e
namespace regex
public type option_spec(object o)
```

Regular expression option specification type

Although the functions do not use this type (they return an error instead), you can use this to check if your routine is receiving something sane.

## option\_spec\_to\_string

```
include std/regex.e
namespace regex
public function option_spec_to_string(option_spec o)
```

converts an option spec to a string.

This can be useful for debugging what options were passed in. Without it you have to convert a number to hex and lookup the constants in the source code.

## error\_to\_string

```
include std/regex.e
namespace regex
public function error_to_string(integer i)
```

converts an regex error to a string.

This can be useful for debugging and even something rough to give to the user in case of a regex failure. It is preferable to a number.

### See Also:

[error\\_message](#)

## new

```
include std/regex.e
namespace regex
public function new(string pattern, option_spec options = DEFAULT)
```

returns an allocated regular expression.

### Arguments:

1. pattern : a sequence representing a human readable regular expression
2. options : defaults to **DEFAULT**. See [Compile Time Option Constants](#).

### Returns:

A **regex**, which other regular expression routines can work on or an atom to indicate an error. If an error, you can call [error\\_message](#) to get a detailed error message.

### Comments:

This is the only routine that accepts a human readable regular expression. The string is compiled and a **regex** is returned. Analyzing and compiling a regular expression is a costly operation and should not be done more than necessary. For instance, if your application looks for an email address among text frequently, you should create the regular expression as a constant accessible to your source code and any files that may use it, thus, the regular expression is analyzed and compiled only once per run of your application.

```
-- Bad Example
include std/regex.e as re
```

```
while sequence(line) do
  re:regex proper_name = re:new("[A-Z][a-z]+ [A-Z][a-z]+")
  if re:find(proper_name, line) then
    -- code
  end if
end while
```

```
-- Good Example
include std/regex.e as re
constant re_proper_name = re:new("[A-Z][a-z]+ [A-Z][a-z]+")
while sequence(line) do
  if re:find(re_proper_name, line) then
    -- code
  end if
end while
```

### Example 1:

```
include std/regex.e as re
re:regex number = re:new("[0-9]+")
```

### Note:

For simple matches, the built-in Euphoria routine `eu:match` and the library routine `wildcard:is_match` are often times easier to use and a little faster. Regular expressions are faster for complex searching/matching.

### See Also:

`error_message` | `find` | `find_all`

## error\_message

```
include std/regex.e
namespace regex
public function error_message(object re)
```

returns a text based error message.

### Arguments:

1. re: Regular expression to get the error message from

### Returns:

An atom (0) when no error message exists, otherwise a sequence describing the error.

### Comments:

If `new` returns an atom, this function will return a text error message as to the reason.

### Example 1:

```
include std/regex.e
object r = regex:new("[A-Z[a-z]*")
if atom(r) then
  printf(1, "Regex failed to compile: %s\n", { regex:error_message(r) })
end if
```

## Utility Routines



## escape

```
include std/regex.e
namespace regex
public function escape(string s)
```

escapes special regular expression characters that may be entered into a search string from user input.

### Arguments:

1. s: string sequence to escape

### Returns:

An escaped sequence representing s.

### Note:

Special regex characters are:

```
. \ + * ? [ ^ ] $ ( ) { } = ! < > | : -
```

### Example 1:

```
include std/regex.e as re
sequence search_s = re:escape("Payroll is $***15.00")
-- search_s = "Payroll is \\$\\*\\*\\*15\\.00"
```

## get\_ovector\_size

```
include std/regex.e
namespace regex
public function get_ovector_size(regex ex, integer maxsize = 0)
```

returns the number of capturing subpatterns (the ovector size) for a regex.

### Arguments:

1. ex : a regex
2. maxsize : optional maximum number of named groups to get data from

### Returns:

An **integer**

## Match

## find

```
include std/regex.e
namespace regex
public function find(regex re, string haystack, integer from = 1,
    option_spec options = DEFAULT,
    integer size = get_ovector_size(re, 30))
```

returns the first match of re in haystack. You can optionally start at the position from.

### Arguments:

1. `re` : a regex for a subject to be matched against
2. `haystack` : a string in which to searched
3. `from` : an integer setting the starting position to begin searching from. Defaults to 1
4. `options` : defaults to `DEFAULT`. See [Match Time Option Constants](#). The only options that may be set when calling `find` are:  
`ANCHORED` | `NEWLINE_CR` | `NEWLINE_LF` | `NEWLINE_CRLF` | `NEWLINE_ANY` | `NEWLINE_ANYCRLF` | `NOTBOL` | `NOTEOL` | `NOTEMPTY` | `NO_UTF8_CHECK`  
 options can be any match time option or a sequence of valid options or it can be a value that comes from using `or_bits` on any two valid option values.
5. `size` : internal (how large an array the C backend should allocate). Defaults to 90, in rare cases this number may need to be increased in order to accomodate complex regex expressions.

### Returns:

An **object**, which is either an atom of 0, meaning nothing matched or a sequence of index pairs. These index pairs may be fewer than the number of groups specified. These index pairs may be the invalid index pair {0,0}.

The first pair is the starting and ending indices of the sub-string that matches the expression. This pair may be followed by indices of the groups. The groups are subexpressions in the regular expression surrounded by parenthesis ().

Now, it is possible to get a match without having all of the groups match. This can happen when there is a quantifier after a group. For example: `'([01])'` or `'([01])?'`. In this case, the returned sequence of pairs will be missing the last group indices for which there is no match. However, if the missing group is followed by a group that *does* match, {0,0} will be used as a place holder. You can ensure your groups match when your expression matches by keeping quantifiers inside your groups: For example use: `'([01]?')` instead of `'([01])?'`

### Example 1:

```
include std/regex.e as re
r = re:new("([A-Za-z]+) ([0-9]+)") -- John 20 or Jane 45
object result = re:find(r, "John 20")

-- The return value will be:
-- {
--   { 1, 7 }, -- Total match
--   { 1, 4 }, -- First grouping "John" ([A-Za-z]+)
--   { 6, 7 } -- Second grouping "20" ([0-9]+)
-- }
```

## find\_all

```
include std/regex.e
namespace regex
public function find_all(regex re, string haystack, integer from = 1,
    option_spec options = DEFAULT,
    integer size = get_ovector_size(re, 30))
```

returns all matches of `re` in `haystack` optionally starting at the sequence position `from`.

### Arguments:

1. `re` : a regex for a subject to be matched against
2. `haystack` : a string in which to searched
3. `from` : an integer setting the starting position to begin searching from. Defaults to 1
4. `options` : defaults to `DEFAULT`. See [Match Time Option Constants](#).

### Returns:

A **sequence** of **sequences** that were returned by `find` and in the case of no matches this

returns an empty **sequence**.

### Comments:

Please see [find](#) for a detailed description of each member of the return sequence.

### Example 1:

```
include std/regex.e as re
constant re_number = re:new("[0-9]+")
object matches = re:find_all(re_number, "10 20 30")

-- matches is:
-- {
--     {{1, 2}},
--     {{4, 5}},
--     {{7, 8}}
-- }
```

## has\_match

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

determines if re matches any portion of haystack.

### Arguments:

1. re : a regex for a subject to be matched against
2. haystack : a string in which to searched
3. from : an integer setting the starting position to begin searching from. Defaults to 1
4. options : defaults to **DEFAULT**. See [Match Time Option Constants](#). options can be any match time option or a sequence of valid options or it can be a value that comes from using `or_bits` on any two valid option values.

### Returns:

An **atom**, 1 if re matches any portion of haystack or 0 if not.

## is\_match

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

determines if the entire haystack matches re.

### Arguments:

1. re : a regex for a subject to be matched against
2. haystack : a string in which to searched
3. from : an integer setting the starting position to begin searching from. Defaults to 1
4. options : defaults to **DEFAULT**. See [Match Time Option Constants](#). options can be any match time option or a sequence of valid options or it can be a value that comes from using `or_bits` on any two valid option values.

### Returns:

An **atom**, 1 if re matches the entire haystack or 0 if not.

## matches

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

gets the matched text only.

### Arguments:

1. re : a regex for a subject to be matched against
2. haystack : a string in which to searched
3. from : an integer setting the starting position to begin searching from. Defaults to 1
4. options : defaults to **DEFAULT**. See [Match Time Option Constants](#). options can be any match time option or **STRING\_OFFSETS** or a sequence of valid options or it can be a value that comes from using `or_bits` on any two valid option values.

### Returns:

Returns a **sequence** of strings, the first being the entire match and subsequent items being each of the captured groups or **ERROR\_NOMATCH** if there is no match. The size of the sequence is the number of groups in the expression plus one (for the entire match).

If options contains the bit **STRING\_OFFSETS**, then the result is different. For each item, a sequence is returned containing the matched text, the starting index in haystack and the ending index in haystack.

### Example 1:

```
include std/regex.e as re
constant re_name = re:new("([A-Z][a-z]+) ([A-Z][a-z]+)")

object matches = re:matches(re_name, "John Doe and Jane Doe")
-- matches is:
-- {
--   "John Doe", -- full match data
--   "John",     -- first group
--   "Doe"       -- second group
-- }

matches = re:matches(re_name, "John Doe and Jane Doe", 1, re:STRING_OFFSETS)
-- matches is:
-- {
--   { "John Doe", 1, 8 }, -- full match data
--   { "John",     1, 4 }, -- first group
--   { "Doe",      6, 8 }  -- second group
-- }
```

### See Also:

[all\\_matches](#)

## all\_matches

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

gets the text of all matches.

## Arguments:

1. `re` : a regex for a subject to be matched against
2. `haystack` : a string in which to searched
3. `from` : an integer setting the starting position to begin searching from. Defaults to 1
4. `options` : options, defaults to **DEFAULT**. See [Match Time Option Constants](#). options can be any match time option or a sequence of valid options or it can be a value that comes from using `or_bits` on any two valid option values.

## Returns:

Returns **ERROR\_NOMATCH** if there are no matches, or a **sequence** of **sequences** of **strings** if there is at least one match. In each member sequence of the returned sequence, the first string is the entire match and subsequent items being each of the captured groups. The size of the sequence is the number of groups in the expression plus one (for the entire match). In other words, each member of the return value will be of the same structure of that is returned by [matches](#).

If options contains the bit **STRING\_OFFSETS**, then the result is different. In each member sequence, instead of each member being a string each member is itself a sequence containing the matched text, the starting index in haystack and the ending index in haystack.

## Example 1:

```
include std/regex.e as re
constant re_name = re:new("([A-Z][a-z]+) ([A-Z][a-z]+)")

object matches = re:all_matches(re_name, "John Doe and Jane Doe")
-- matches is:
-- {
--   {
--     "John Doe", -- first match
--     "John",     -- full match data
--     "John",     -- first group
--     "Doe"       -- second group
--   },
--   {
--     "Jane Doe", -- second match
--     "Jane",     -- full match data
--     "Jane",     -- first group
--     "Doe"       -- second group
--   }
-- }

matches = re:all_matches(re_name, "John Doe and Jane Doe", , re:STRING_OFFSETS)
-- matches is:
-- {
--   {
--     { "John Doe", 1, 8 }, -- first match
--     { "John",     1, 4 }, -- full match data
--     { "John",     1, 4 }, -- first group
--     { "Doe",      6, 8 } -- second group
--   },
--   {
--     { "Jane Doe", 14, 21 }, -- second match
--     { "Jane",     14, 17 }, -- full match data
--     { "Jane",     14, 17 }, -- first group
--     { "Doe",      19, 21 } -- second group
--   }
-- }
```

## See Also:

[matches](#)

## Splitting

## split

```
include std/regex.e
namespace regex
public function split(regex re, string text, integer from = 1, option_spec options = DEFAULT)
```

splits a string based on a regex as a delimiter.

### Arguments:

1. `re` : a regex which will be used for matching
2. `text` : a string on which search and replace will apply
3. `from` : optional start position
4. `options` : options, defaults to **DEFAULT**. See [Match Time Option Constants](#). options can be any match time option or a sequence of valid options or it can be a value that comes from using `or_bits` on any two valid option values.

### Returns:

A **sequence** of string values split at the delimiter and if no delimiters were matched this **sequence** will be a one member sequence equal to `{text}`.

### Example 1:

```
include std/regex.e as re
regex comma_space_re = re:new(`,\s`)
sequence data = re:split(comma_space_re,
                        "euphoria programming, source code, reference data")
-- data is
-- {
--   "euphoria programming",
--   "source code",
--   "reference data"
-- }
```

## split\_limit

```
include std/regex.e
namespace regex
public function split_limit(regex re, string text, integer limit = 0, integer from = 1,
    option_spec options = DEFAULT)
```

## Replacement

## find\_replace

```
include std/regex.e
namespace regex
public function find_replace(regex ex, string text, sequence replacement, integer from = 1,
    option_spec options = DEFAULT)
```

replaces all matches of a regex with the replacement text.

### Arguments:

1. `re` : a regex which will be used for matching
2. `text` : a string on which search and replace will apply
3. `replacement` : a string, used to replace each of the full matches
4. `from` : optional start position
5. `options` : options, defaults to **DEFAULT**. See [Match Time Option Constants](#). options can be

any match time option or a sequence of valid options or it can be a value that comes from using `or_bits` on any two valid option values.

### Returns:

A **sequence**, the modified text. If there is no match with `re` the return value will be the same as `text` when it was passed in.

### Comments:

Special replacement operators:

- `\` -- Causes the next character to lose its special meaning.
- `\n` ~ -- Inserts a 0x0A (LF) character.
- `\r` -- Inserts a 0x0D (CR) character.
- `\t` -- Inserts a 0x09 (TAB) character.
- `\1` to `\9` -- Recalls stored substrings from registers (`\1`, `\2`, `\3`, to `\9`).
- `\0` -- Recalls entire matched pattern.
- `\u` -- Convert next character to uppercase
- `\l` -- Convert next character to lowercase
- `\U` -- Convert to uppercase till `\E` or `\e`
- `\L` -- Convert to lowercase till `\E` or `\e`
- `\E` or `\e` -- Terminate a `\U` or `\L` conversion

### Example 1:

```
include std/regex.e
regex r = new(`([A-Za-z]+)\.([A-Za-z]+)` )
sequence details = find_replace(r, "hello.txt",
                                `Filename: \U\1\e Extension: \U\2\e`)
-- details = "Filename: HELLO Extension: TXT"
```

## find\_replace\_limit

```
include std/regex.e
namespace regex
public function find_replace_limit(regex ex, string text, sequence replacement,
    integer limit, integer from = 1, option_spec options = DEFAULT)
```

replaces up to `limit` matches of `ex` in `text` except when `limit` is 0. When `limit` is 0, this routine replaces all of the matches.

### Arguments:

1. `re` : a regex which will be used for matching
2. `text` : a string on which search and replace will apply
3. `replacement` : a string, used to replace each of the full matches
4. `limit` : the number of matches to process
5. `from` : optional start position
6. `options` : options, defaults to **DEFAULT**. See **Match Time Option Constants**. options can be any match time option or a sequence of valid options or it can be a value that comes from using `or_bits` on any two valid option values.

### Comments:

This function is identical to `find_replace` except it allows you to limit the number of replacements to perform. Please see the documentation for `find_replace` for all the details.

### Returns:

A **sequence**, the modified text.

See Also:

[find\\_replace](#)

## find\_replace\_callback

```
include std/regex.e
namespace regex
public function find_replace_callback(regex ex, string text, integer rid, integer limit = 0,
    integer from = 1, option_spec options = DEFAULT)
```

finds and then replaces text that is processed by a call back function.

### Arguments:

1. `re` : a regex which will be used for matching
2. `text` : a string on which search and replace will apply
3. `rid` : routine id to execute for each match
4. `limit` : the number of matches to process
5. `from` : optional start position
6. `options` : options, defaults to `DEFAULT`. See [Match Time Option Constants](#). options can be any match time option or a sequence of valid options or it can be a value that comes from using `or_bits` on any two valid option values.

### Returns:

A **sequence**, the modified text.

### Comments:

When `limit` is positive, this routine replaces up to `limit` matches of `ex` in `text` with the result of the user defined callback, `rid`, and when `limit` is 0, replaces all matches of `ex` in `text` with the result of this user defined callback, `rid`.

The callback should take one sequence. The first member of this sequence will be a string representing the entire match and the subsequent members, if they exist, will be a strings for the captured groups within the regular expression.

The function `rid`. Must take one sequence parameter. The function needs to accept a sequence of strings and return a string. For each match, the function will be passed a sequence of strings. The first string is the entire match the subsequent strings are for the capturing groups. If a match succeeds with groups that don't exist, that place will contain a 0. If the sub-group does exist, the palce will contain the matching group string. for that group.

### Example 1:

```
include std/text.e
function my_convert(sequence params)
  switch params[1] do
    case "1" then
      return "one "
    case "2" then
      return "two "
    case else
      return "unknown "
  end switch
end function

regex r = re:new(`\d`)
sequence result = re:find_replace_callback(r, "125", routine_id("my_convert"))
-- result = "one two unknown "
```



```
integer missing_data_flag = 0
regex r2 = re:new(`[A-Z][a-z]+ ([A-Z][a-z]+)?`)
function my_toupper( sequence params)
    -- here params[2] may be 0.
    return upper( params[1] )
end function

result = find_replace_callback(r2, "John Doe", routine_id("my_toupper"))
-- params[2] is "Doe"
-- result = "JOHN DOE"
printf(1, "result=%s\n", {result} )
result = find_replace_callback(r2, "Mary", routine_id("my_toupper"))
-- result = "MARY"
```

# Text Manipulation

## Subroutines

### sprintf

```
<built-in> function sprintf(sequence format, object values)
```

returns the representation of any Euphoria object as a string of characters with formatting.

#### Arguments:

1. **format** : a sequence, the text to print. This text may contain format specifiers.
2. **values** : usually, a sequence of values. It should have as many elements as format specifiers in format, as these values will be substituted to the specifiers.

#### Returns:

A **sequence**, a string sequence of printable characters, representing format with the values in values spliced in.

#### Comments:

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.

`printf(fn, st, x)` is equivalent to `puts(fn, sprintf(st, x))`.

Some typical uses of `sprintf` are:

1. Converting numbers to strings.
2. Creating strings to pass to system.
3. Creating formatted error messages that can be passed to a common error message handler.

#### Example 1:

```
sequence s
s = sprintf("%08d", 12345)
puts(1,s)
--> s is          "00012345"
```

#### See Also:

`printf` | `sprint` | `format`

### sprint

```
include std/text.e
namespace text
public function sprint(object x)
```

returns the representation of any Euphoria object as a string of characters.

#### Arguments:

1. **x** : Any Euphoria object.

### Returns:

A **sequence**, a string sequence representation of *x*.

### Comments:

This corresponds to `print(fn, x)`; except that the output is returned as a sequence of characters rather than being sent to a file or device; *x* can be any Euphoria object.

The atoms contained within *x* will be displayed to a maximum of ten significant digits just as with `print`.

### Example 1:

```

                                include std/text.e
                                sequence s

    s = sprintf(12345)
    puts(1,s)
    --> s is      "12345"

```

### Example 2:

```

                                include std/text.e
                                sequence s

    s = sprintf( {10,20,30}+5 )
    puts(1,s)
    --> s is      "{15,25,35}"

```

### See Also:

`sprintf` | `printf`

## trim\_head

```

include std/text.e
namespace text
public function trim_head(sequence source, object what = " \t\r\n", integer ret_index = 0)

```

trims all items in the supplied set from the leftmost (start or head) of a sequence.

### Arguments:

1. *source* : the sequence to trim.
2. *what* : the set of item to trim from *source* (defaults to " \t\r\n").
3. *ret\_index* : If zero (the default) returns the trimmed sequence, otherwise it returns the index of the leftmost item **not** in *what*.

### Returns:

An **object**,

A **sequence**, if *ret\_index* is 0 zero (*false*), which is the trimmed version of *source*

A **integer**, if *ret\_index* is !0 not zero (*true*), which is index of the leftmost element in *source* that is not in *what*.

### Example 1:

```

                                include std/text.e
                                object x
                                integer TRUE = 1

    x = trim_head("\r\nSentence read from a file\r\n", "\r\n")
    puts(1,x)

```

```
--> x is          "Sentence read from a file\r\n"

    x = trim_head("\r\nSentence read from a file\r\n", "\r\n", TRUE)
    ? x
--> x is 3
```

## See Also:

[trim\\_tail](#) | [trim](#) | [pad\\_head](#)

## trim\_tail

```
include std/text.e
namespace text
public function trim_tail(sequence source, object what = " \t\r\n", integer ret_index = 0)
```

trims all items in the supplied set from the rightmost (end or tail) of a sequence.

## Arguments:

1. source : the sequence to trim.
2. what : the set of item to trim from source (defaults to " \t\r\n").
3. ret\_index : If 0 zero (the default) returns the trimmed sequence, otherwise it returns the index of the rightmost item **not** in what.

## Returns:

An **object**,

A **sequence**, if ret\_index is 0 zero (*false*), which is the trimmed version of source

A **integer**, if ret\_index is !0 not zero (*true*), which is index of the rightmost item in source that is not in what.

## Example 1:

```
include std/text.e
object x
integer TRUE = 1

    x = trim_tail("\r\nSentence read from a file\r\n", "\r\n")
puts(1,x)
--> x is          "\r\nSentence read from a file"

    x = trim_tail("\r\nSentence read from a file\r\n", "\r\n", TRUE)
    ? x
--> x is 27
```

## See Also:

[trim\\_head](#) | [trim](#) | [pad\\_tail](#)

## trim

```
include std/text.e
namespace text
public function trim(sequence source, object what = " \t\r\n", integer ret_index = 0)
```

trims all items in the supplied set from both the left end (head start) and right end (tail end) of a sequence.

## Arguments:

1. `source` : the sequence to trim.
2. `what` : the set of item to trim from source (defaults to "`\t\r\n`").
3. `ret_index` : If zero (the default) returns the trimmed sequence, otherwise it returns a 2-element sequence containing the index of the leftmost item and rightmost item **not** in what.

### Returns:

An **object**,

A **sequence**, a string sequence, if `ret_index` is 0 zero (*false*), which is the trimmed version of source

A **sequence**, a two-item sequence, if `ret_index` is !0 not zero (*true*), in the form {`left_index`, `right_index`} .

### Example 1:

```

include std/text.e
object x
integer TRUE = 1

x = trim("\r\nSentence read from a file\r\n", "\r\n")
puts(1,x)
--> x is      "Sentence read from a file"

x = trim("\r\nSentence read from a file\r\n", "\r\n", TRUE)
? x
--> x is {3,27}

x = trim(" This is a sentence.\n")
puts(1,x)
--> x is      "This is a sentence."
-- Default is to trim off all " \t\r\n"
```

### See Also:

[trim\\_head](#) | [trim\\_tail](#)

## set\_encoding\_properties

```

include std/text.e
namespace text
public procedure set_encoding_properties(sequence en = "", sequence lc = "", sequence uc = "")
```

sets the table of lowercase and uppercase characters that is used by [lower](#) and [upper](#)

### Arguments:

1. `en` : The name of the encoding represented by these character sets
2. `lc` : The set of lowercase characters
3. `uc` : The set of upper case characters

### Comments:

- `lc` and `uc` must be the same length.
- If no arguments are given, the default ASCII table is set.

### Example 1:

```

include std/text.e
include std/console.e
sequence s

set_encoding_properties( "Elvish", "aeiouy", "AEIOUY")
```

```

s = "the hobbit, the fellowship of the ring, ..."
display(upper(s))
--> s is "tHE hObbIt, thE fElloWshIp Of thE rInG, ..."
display(lower(s))
--> s is "the hobbit, the fellowship of the ring, ..."

s = "THE HOBBIT, THE FELLOWSHIP OF THE RING, ..."
display(upper(s))
--> s is "THE HOBBIT, THE FELLOWSHIP OF THE RING, ..."
display(lower(s))
--> s is "The HoBBiT, The FeLLoWShiP oF The RiNG, ..."

```

### Example 2:

```
set_encoding_properties( "1251") -- Loads a predefined code page.
```

### See Also:

[lower](#) | [upper](#) | [get\\_encoding\\_properties](#)

## get\_encoding\_properties

```

include std/text.e
namespace text
public function get_encoding_properties()

```

gets the table of lowercase and uppercase characters that is used by [lower](#) and [upper](#).

### Arguments:

none

### Returns:

A **sequence**, containing three items.  
{Encoding\_Name, LowerCase\_Set, UpperCase\_Set}

### Example 1:

```

include std/text.e
include std/console.e
sequence encode_sets

encode_sets = get_encoding_properties()
display(encode_sets)
--> {"ASCII", "", ""}
-- same for windows console or linux terminal

```

### See Also:

[lower](#) | [upper](#) | [set\\_encoding\\_properties](#)

## lower

```

include std/text.e
namespace text
public function lower(object x)

```

converts an atom or sequence to lower case.

## Arguments:

1. *x* : Any Euphoria object.

## Returns:

A **sequence**, the lowercase version of *x*

## Comments:

- For *windows* systems, this uses the current code page for conversion
- For *unix* this only works on ASCII characters. It alters characters in the 'a'..'z' range. If you need to do case conversion with other encodings use the [set\\_encoding\\_properties](#) first.
- *x* may be a sequence of any shape, all atoms of which will be acted upon.

## Warning:

When using ASCII encoding, this can also affect floating point numbers in the range 65 to 90.

## Example 1:

```

include std/text.e
include std/console.e
sequence s
integer a

s = lower("Euphoria")
display(s)
--> s is "euphoria"

a = lower('B')
display(a)
--> a is 98
-- that is 'b'

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

## See Also:

[upper](#) | [proper](#) | [set\\_encoding\\_properties](#) | [get\\_encoding\\_properties](#)

## upper

```

include std/text.e
namespace text
public function upper(object x)
```

converts an atom or sequence to upper case.

## Arguments:

1. *x* : Any Euphoria object.

## Returns:

A **sequence**, the uppercase version of *x*

## Comments:

- For *windows* systems, this uses the current code page for conversion
- For *unix* this only works on ASCII characters. It alters characters in the 'a'..'z' range. If

you need to do case conversion with other encodings use the [set\\_encoding\\_properties](#) first.

- x may be a sequence of any shape, all atoms of which will be acted upon.

### Warning:

When using ASCII encoding, this can also affects floating point numbers in the range 97 to 122.

### Example 1:

```

                                include std/text.e
                                include std/console.e
                                sequence s
                                integer a

    s = upper("Euphoria")
    display(s)
    --> s is "EUPHORIA"

    a = upper('b')
    display(a)
    --> a is 66
    -- that is 'B'

    s = upper({"Euphoria", "Programming"})
    display(s)
    --> s is {"EUPHORIA", "PROGRAMMING"}

```

### See Also:

[lower](#) | [proper](#) | [set\\_encoding\\_properties](#) | [get\\_encoding\\_properties](#)

## proper

```

include std/text.e
namespace text
public function proper(sequence x)

```

converts a text sequence to capitalized words.

### Arguments:

1. x : A text sequence.

### Returns:

A **sequence**, the Capitalized Version of x

### Comments:

A text sequence is one in which all elements are either characters or text sequences. This means that if a non-character is found in the input, it is not converted. However this rule only applies to elements on the same level, meaning that sub-sequences could be converted if they are actually text sequences.

### Example 1:

```

                                include std/text.e
                                include std/console.e
                                sequence s

    s = proper("euphoria programming language")
    display(s)
    --> s is      "Euphoria Programming Language"

```



```

s = proper("EUPHORIA PROGRAMMING LANGUAGE")
display(s)
--> s is      "Euphoria Programming Language"

s = proper({"EUPHORIA PROGRAMMING", "language", "rapid dEPLOYMENT", "sOfTwArE"})
display(s)
--> s is {"Euphoria Programming", "Language", "Rapid Deployment", "Software"}

s = proper({'a', 'b', 'c'})
display(s)
--> s is      "Abc"
-- that is    {'A', 'b', 'c'}

s = proper({'a', 'b', 'c', 3.1472})
display(s)
--> s is      { 97,   98,  99, 3.1472}
-- that is    {'a',  'b',  c', 3.1472}
--> Unchanged because it contains a non-character.

s = proper({"abc", 3.1472})
display(s)
--> s is      {"Abc", 3.1472}
--> The embedded text sequence is converted.

```

### See Also:

[lower](#) | [upper](#)

## keyvalues

```

include std/text.e
namespace text
public function keyvalues(sequence source, object pair_delim = ";;", object kv_delim = ":",
    object quotes = "\"'`", object whitespace = " \t\n\r", integer haskeys = 1)

```

converts a string containing Key/Value pairs into a set of sequences, one per K/V pair.

### Arguments:

1. `source` : a text sequence, containing the representation of the key/values.
2. `pair_delim` : an object containing a list of elements that delimit one key/value pair from the next. The defaults are semi-colon (;) and comma (,).
3. `kv_delim` : an object containing a list of elements that delimit the key from its value. The defaults are colon (:) and equal (=).
4. `quotes` : an object containing a list of elements that can be used to enclose either keys or values that contain delimiters or whitespace. The defaults are double-quote ("), single-quote (') and back-quote (`).
5. `whitespace` : an object containing a list of elements that are regarded as whitespace characters. The defaults are space, tab, new-line, and carriage-return.
6. `haskeys` : an integer containing true or false. The default is true. When true, the `kv_delim` values are used to separate keys from values, but when false it is assumed that each 'pair' is actually just a value.

### Returns:

A **sequence**, of pairs. Each pair is in the form {key, value}.

### Comments:

String representations of atoms are not converted, either in the key or value part, but returned as any regular string instead.

If `haskeys` is true, but a substring only holds what appears to be a value, the key is synthesized as `p[n]`, where `n` is the number of the pair. See Example 2.

By default, pairs can be delimited by either a comma or semi-colon ";;" and a key is delimited from its value by either an equal or a colon "=:". Whitespace between pairs, and between delimiters is ignored.

If you need to have one of the delimiters in the value data, enclose it in quotation marks. You can use any of single, double and back quotes, which also means you can quote quotation marks themselves. See Example 3.

It is possible that the value data itself is a nested set of pairs. To do this enclose the value in parentheses. Nested sets can nested to any level. See Example 4.

If a sub-list has only data values and not keys, enclose it in either braces or square brackets. See Example 5. If you need to have a bracket as the first character in a data value, prefix it with a tilde. Actually a leading tilde will always just be stripped off regardless of what it prefixes. See Example 6.

### Example 1:

```
include std/text.e
include std/console.e
sequence s

s=keyvalues("foo=bar,qwe=1234,asdf='contains space, comma, and equal(=)'"")
display(s)
--> s is
-- {
--   {"foo", "bar"},
--   {"qwe", "1234"},
--   {"asdf", "contains space, comma, and equal(=)"}
-- }
```

### Example 2:

```
include std/text.e
include std/console.e
sequence s

s = keyvalues("abc fgh=ijk def")
display(s)
--> s is { {"p[1]", "abc"}, {"fgh", "ijk"}, {"p[3]", "def"} }
```

### Example 3:

```
include std/text.e
include std/console.e
sequence s

s = keyvalues("abc=`'quoted'`")
display(s)
--> s is { {"abc", "'quoted'"} }
```

### Example 4:

```
include std/text.e
include std/console.e
sequence s

s = keyvalues("colors=(a=black, b=blue, c=red)")
display(s)
--> s is { {"colors", {{"a", "black"}, {"b", "blue"}, {"c", "red"}} } }

s = keyvalues("colors=(black=[0,0,0], blue=[0,0,FF], red=[FF,0,0])")
display(s)
--> s is
-- { {"colors",
--   {{"black", {"0", "0", "0"}},
--   {"blue", {"0", "0", "FF"}},
--   {"red", {"FF", "0", "0"}}}} }
```

### Example 5:

```

                                include std/text.e
                                include std/console.e
                                sequence s

    s = keyvalues("colors=[black, blue, red]")
display(s)
--> s is      { {"colors", { "black", "blue", "red"} } }

```

### Example 6:

```

                                include std/text.e
                                include std/console.e
                                sequence s

-- two identical results

    s = keyvalues("colors=~[black, blue, red]")
display(s)
--> s is { {"colors", "[black, blue, red]"} } }

    s = keyvalues("colors=`[black, blue, red]`")
display(s)
--> s is { {"colors", "[black, blue, red]"} } }

```

## escape

```

include std/text.e
namespace text
public function escape(sequence s, sequence what = "\"")

```

escapes special characters in a string.

### Arguments:

1. s: string to escape
2. what: sequence of characters to escape defaults to escaping a double quote.

### Returns:

An escaped sequence representing s.

### Example 1:

```

                                include std/text.e
                                include std/console.e
                                sequence s

    s = "John \"Mc\" Doe"
    ? s
--? s is {74,111,104,110,32,34,77,99,34,32,68,111,101}
display(s)
--> s is `John "Mc" Doe`

    s = escape(s)
    ? s
--> s is {74,111,104,110,32,92,34,77,99,92,34,32,68,111,101}
display(s)
--> s is `John \"Mc\" Doe`

```

### See Also:

[quote](#)

## quote

```
include std/text.e
namespace text
public function quote(sequence text_in, object quote_pair = {"\"", "\""}, integer esc = - 1,
                      t_text sp = "")
```

returns a quoted version of the first argument.

### Arguments:

1. `text_in` : The string or set of strings to quote.
2. `quote_pair` : A sequence of two strings. The first string is the opening quote to use, and the second string is the closing quote to use. The default is `{"\"", "\""}` which means that the output will be enclosed by double-quotation marks.
3. `esc` : A single escape character. If this is not negative (the default), then this is used to 'escape' any embedded quote characters and 'esc' characters already in the `text_in` string.
4. `sp` : A list of zero or more special characters. The `text_in` is only quoted if it contains any of the special characters. The default is `""` which means that the `text_in` is always quoted.

### Returns:

A **sequence**, the quoted version of `text_in`.

### Example 1:

```
include std/text.e
include std/console.e
sequence s

-- Using the defaults; output enclosed in double-quotes; no escapes and no specials.

s = quote("The small man")
display(s)
--> s is now '"the small man"'
-- includes the double-quote characters
```

### Example 2:

```
include std/text.e
include std/console.e
sequence s

s = quote("The small man", {"(", ")"})
display(s)
--> s is      '(the small man)'
```

### Example 3:

```
include std/text.e
include std/console.e
sequence s

s = quote("The (small) man", {"(", ")"}, '~' )
display(s)
--> s is      '(The ~(small~) man)'
```

### Example 4:

```
include std/text.e
include std/console.e
sequence s

s = quote("The (small) man", {"(", ")"}, '~', "#" )
display(s)
--> s is      `The (small) man`
-- because the input did not contain a '#' character.
```

### Example 5:

```

include std/text.e
include std/console.e
sequence s

s = quote("The #1 (small) man", {"(", ")"}, '~', "#" )
display(s)
--> s is      '(the #1 ~(small~) man)'
-- because the input did contain a '#' character.

```

### Example 6:

```

include std/text.e
include std/console.e
sequence s

-- input is a set of strings...

s = quote( { "a b c", "def", "g hi" }, )
display(s)
--> s is
--      { "\"a b c\"", "\"def\"", "\"g hi\"" }
-- that is three quoted strings:
--      "'a b c'  'def'  'g hi'"

```

### See Also:

[escape](#)

## dequote

```

include std/text.e
namespace text
public function dequote(sequence text_in, object quote_pairs = {"\"", "\""},
integer esc = - 1)

```

removes *quotation* text from the argument.

### Arguments:

1. `text_in` : The string or set of strings to de-quote.
2. `quote_pairs` : A set of one or more sub-sequences of two strings, or an atom representing a single character to be used as both the open and close quotes. The first string in each sub-sequence is the opening quote to look for, and the second string is the closing quote. The default is `"\""`, `"\""` which means that the output is *quoted* if it is enclosed by double-quotation marks.
3. `esc` : A single escape character. If this is not negative (the default), then this is used to 'escape' any embedded occurrences of the quote characters. In which case the 'escape' character is also removed.

### Returns:

A **sequence**, the original text but with *quote* strings stripped of quotes.

### Example 1:

```

include std/text.e
include std/console.e
sequence s

-- Using the defaults.
s = dequote("\"The small man\"")
display(s)
--> s is  "The small man"

```

### Example 2:

```

include std/text.e
include std/console.e
sequence s

-- Using the defaults.
s = dequote("(The small ?(?) man)", {"(", ","}), '?')
display(s)
--> s is "The small () man"

```

## format

```

include std/text.e
namespace text
public function format(sequence format_pattern, object arg_list = {})

```

formats a set of arguments in to a string based on a supplied pattern.

### Arguments:

1. `format_pattern` : A sequence: the pattern string that contains zero or more tokens.
2. `arg_list` : An object: Zero or more arguments used in token replacement.

### Returns:

A string **sequence**, the original `format_pattern` but with tokens replaced by corresponding arguments.

### Comments:

The `format_pattern` string contains text and argument tokens. The resulting string is the same as the format string except that each token is replaced by an item from the argument list.

A token has the form [**<Q>**], where **<Q>** is are optional qualifier codes.

The qualifier. **<Q>** is a set of zero or more codes that modify the default way that the argument is used to replace the token. The default replacement way is to convert the argument to its shortest string representation and use that to replace the token. This may be modified by the following codes, which can occur in any order.

Qualifier	Usage
N	('N' is an integer) The index of the argument to use
{id}	Uses the argument that begins with "id=" where "id" is an identifier name.
%envvar%	Uses the Environment Symbol 'envvar' as an argument
w	For string arguments, if capitalizes the first letter in each word
u	For string arguments, it converts it to upper case.
l	For string arguments, it converts it to lower case.
<	For numeric arguments, it left justifies it.
>	For string arguments, it right justifies it.
c	Centers the argument.
z	For numbers, it zero fills the left side.
:S	('S' is an integer) The maximum size of the resulting field. Also, if 'S' begins with '0' the field will be zero-filled if the argument is an integer
.N	('N' is an integer) The number of digits after the decimal point
+	For positive numbers, show a leading plus sign
(	For negative numbers, enclose them in parentheses
b	For numbers, causes zero to be all blanks

s	If the resulting field would otherwise be zero length, this ensures that at least one space occurs between this token's field
t	After token replacement, the resulting string up to this point is trimmed.
X	Outputs integer arguments using hexadecimal digits.
B	Outputs integer arguments using binary digits.
?	The corresponding argument is a set of two strings. This uses the first string if the previous token's argument is not the value 1 or a zero-length string, otherwise it uses the second string.
[	Does not use any argument. Outputs a left-square-bracket symbol
,X	Insert thousands separators. The <X> is the character to use. If this is a dot "." then the decimal point is rendered using a comma. Does not apply to zero-filled fields. N.B. if hex or binary output was specified, the separators are every 4 digits otherwise they are every three digits.
T	If the argument is a number it is output as a text character, otherwise it is output as text string

Clearly, certain combinations of these qualifier codes do not make sense and in those situations, the rightmost clashing code is used and the others are ignored.

Any tokens in the format that have no corresponding argument are simply removed from the result. Any arguments that are not used in the result are ignored.

Any sequence argument that is not a string will be converted to its *pretty* format before being used in token replacement.

If a token is going to be replaced by a zero-length argument, all white space following the token until the next non-whitespace character is not copied to the result string.

### Example 1:

```

                                include std/text.e
                                include std/console.e
                                sequence s

    s = format("Cannot open file '[' - code []", {"/usr/temp/work.dat", 32})
display(s)
--> s is      "Cannot open file '/usr/temp/work.dat' - code 32"

    s = format("Err-[2], Cannot open file '['", {"/usr/temp/work.dat", 32})
display(s)
--> s is      "Err-32, Cannot open file '/usr/temp/work.dat'"

    s = format("[4w] [3z:2] [6] [5l] [2z:2], [1:4]", {2009,4,21,"DAY","MONTH","of"})
display(s)
--> s is      "Day 21 of month 04, 2009"

    s = format("The answer is [:6.2]%", {35.22341})
display(s)
--> s is      "The answer is 35.22%"

    s = format("The answer is [.6]", {1.2345})
display(s)
--> s is      "The answer is 1.234500"

    s = format("The answer is [,.2]", {1234.56})
display(s)
--> s is      "The answer is 1,234.56"

    s = format("The answer is [,.2]", {1234.56})
display(s)
--> s is      "The answer is 1.234,56"

```

```

s = format("The answer is [,.2]", {1234.56})
display(s)
--> s is  "The answer is 1:234.56"

s = format("[ ] [?]", {5, {"cats", "cat"}})
display(s)
--> s is  "5 cats"

s = format("[ ] [?]", {1, {"cats", "cat"}})
display(s)
--> s is      "1 cat"

s = format("[<:4]", {"abcdef"})
display(s)
--> s is  "abcd"

s = format("[>:4]", {"abcdef"})
display(s)
--> s is      "cdef"

s = format("[>:8]", {"abcdef"})
display(s)
--> s is      " abcdef"

s = format("seq is [ ]", {{1.2, 5, "abcdef", {3}}})
display(s)
--> s is      `seq is {1.2,5,"abcdef",{3}}`

s = format("Today is [{day}], the [{date}]", {"date=10/Oct/2012", "day=Wednesday"})
display(s)
--> s is      "Today is Wednesday, the 10/Oct/2012"

s = format("'A' is [T]", 65)
display(s)
--> s is      ``A' is A`

```

## See Also:

[sprintf](#)

## wrap

```

include std/text.e
namespace text
public function wrap(sequence content, integer width = 78, sequence wrap_with = "\n",
    sequence wrap_at = " \t")

```

wraps text to a column width.

## Arguments:

- content -- sequence content to wrap
- width -- width to wrap at, defaults to 78
- wrap\_with -- sequence to wrap with, defaults to "\n"
- wrap\_at -- sequence of characters to wrap at, defaults to space and tab

## Returns:

Sequence containing wrapped text

## Example 1:

```

include std/text.e
include std/console.e

```



```

sequence s
s = wrap("Hello, World")
display(s)
--> s is      "Hello, World"

```

When output looks like:

```
Hello, World
```

### Example 2:

```

include std/text.e
include std/console.e
sequence s

s = "Hello, World. Today we are going to learn about apples."
s = wrap(s, 40)
display(s)
--> s is "Hello, World. Today we are going to\nlearn about apples."

```

When output looks like:

```
Hello, World. today we are going to
learn about apples.
```

### Example 3:

```

include std/text.e
include std/console.e
sequence s

s = "Hello, World. Today we are going to learn about apples."
s = wrap(s, 40, "\n    ")
display(s)
--> s is "Hello, World. today we are going to\n    learn about apples."

```

When output looks like:

```
Hello, World. Today we are going to
learn about apples.
```

### Example 4:

```

include std/text.e
include std/console.e
sequence s

s = "Hello, World. This, Is, A, Dummy, Sentence, Ok, World?"
s = wrap(s, 30, "\n", ",")
display(s)
--> s is "Hello, World. This, Is, A,\nDummy, Sentence, Ok, World?"

```

When output looks like:

```
Hello, World. This, Is, A,
Dummy, Sentence, Ok, World?
```

# Wildcard Matching

## Subroutines

### is\_match

```
include std/wildcard.e
namespace wildcard
public function is_match(sequence pattern, sequence string)
```

determines whether a string matches a pattern. The pattern may contain \* and ? wildcards.

#### Arguments:

1. pattern : a string, the pattern to match
2. string : the string to be matched against

#### Returns:

An **integer**, TRUE if string matches pattern, else FALSE.

#### Comments:

Character comparisons are case sensitive. If you want case insensitive comparisons, pass both pattern and string through **upper**, or both through **lower**, before calling **is\_match**.

If you want to detect a pattern anywhere within a string, add \* to each end of the pattern:

```
i = is_match('*' & pattern & '*', string)
```

There is currently no way to treat \* or ? literally in a pattern.

#### Example 1:

```
include std/wildcard.e
integer i

i = is_match( "A?B*", "AQBXXYY" )
? i
--> i is 1 (true)
```

#### Example 2:

```
i = is_match("*xyz*", "AAAbbbxyz")
? i
--> i is 1 (true)
```

#### Example 3:

```
include std/wildcard.e
integer i

i = is_match("A*B*C", "a111b222c")
? i
--> i is 0 (false)
-- because upper case does not match lower case
```

#### Example 4:

.../euphoria/demo/search.ex

**See Also:**

[upper](#) | [lower](#) | [Regular Expressions](#)

# Base 64 Encoding and Decoding

**Base64** is "used to encode binary data into an ASCII string." This allows binary data to be transmitted using media designed to transmit text data only. See [en.wikipedia.org/wiki/Base64](https://en.wikipedia.org/wiki/Base64) and the RFC 2045 standard for more information.

## Subroutines

### encode

```
include std/base64.e
namespace base64
public function encode(sequence in, integer wrap_column = 0)
```

encodes to base64.

#### Arguments:

1. in : a flat sequence
2. wrap\_column : column width to word-wrap the base64 encoded message to; defaults to 0 zero which is *do not wrap*.

#### Returns:

A **sequence**, a base64 encoded sequence representing in.

#### Example 1:

```
include std/base64.e

puts(1, encode( "Hello Euphoria!" ) )
--> SGVsbG8gRXVwaG9yaWEh
```

#### See Also:

[decode](#)

### decode

```
include std/base64.e
namespace base64
public function decode(sequence in)
```

decodes from base64.

#### Arguments:

1. in -- must be a flat sequence of length 4 to 76 .

#### Returns:

A **sequence**, base256 decode of passed sequence; the length of data to decode must be a multiple of 4 .

#### Comments:

The calling program is expected to strip newlines and so on before calling.

### Example 1:

```
include std/base64.e  
  
puts(1, decode( "SGVsbG8gRXVwaG9yaWEh" ) )  
--> Hello Euphoria!
```

### See Also:

[encode](#)

# MATH

- [math.e](#)
- [mathcons.e](#)
- [rand.e](#)
- [stats.e](#)

# Math

## Sign and Comparisons

### abs

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

returns the absolute value of numbers.

#### Arguments:

1. value : an object, each atom is processed, no matter how deeply nested.

#### Returns:

An **object**, the same shape as value. When value is an atom, the result is the same if not less than zero, and the opposite value otherwise.

#### Comments:

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

#### Example 1:

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

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

#### See Also:

[sign](#)

### sign

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

returns -1, 0 or 1 for each element according to it being negative, zero or positive.

#### Arguments:

1. value : an object, each atom of which will be acted upon, no matter how deeply nested.

#### Returns:

An **object**, the same shape as value. When value is an atom, the result is -1 if value is less than zero, 1 if greater and 0 if equal.

#### Comments:

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

For an atom, `sign(x)` is the same as `compare(x,0)`.

### Example 1:

```
i = sign(5)
i is 1

i = sign(0)
-- i is 0

i = sign(-2)
-- i is -1
```

### See Also:

[compare](#)

## larger\_of

```
include std/math.e
namespace math
public function larger_of(object objA, object objB)
```

returns the larger of two objects.

### Arguments:

1. objA : an object.
2. objB : an object.

### Returns:

Whichever of objA and objB is the larger one.

### Comments:

Introduced in v4.0.3

### Example 1:

```
? larger_of(10, 15.4) -- returns 15.4
? larger_of("cat", "dog") -- returns "dog"
? larger_of("apple", "apes") -- returns "apple"
? larger_of(10, 10) -- returns 10
```

### See Also:

[max](#) | [compare](#) | [smaller\\_of](#)

## smaller\_of

```
include std/math.e
namespace math
public function smaller_of(object objA, object objB)
```

returns the smaller of two objects.

### Arguments:

1. objA : an object.
2. objB : an object.



### Returns:

Whichever of objA and objB is the smaller one.

### Comments:

Introduced in v4.0.3

### Example 1:

```
? smaller_of(10, 15.4) -- returns 10
? smaller_of("cat", "dog") -- returns "cat"
? smaller_of("apple", "apes") -- returns "apes"
? smaller_of(10, 10) -- returns 10
```

### See Also:

[min](#) | [compare](#) | [larger\\_of](#)

## max

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

computes the maximum value among all the argument's elements.

### Arguments:

1. values : an object, all atoms of which will be inspected, no matter how deeply nested.

### Returns:

An **atom**, the maximum of all atoms in [flatten](#)(values).

### Comments:

This function may be applied to an atom or to a sequence of any shape.

### Example 1:

```
a = max({10,15.4,3})
-- a is 15.4
```

### See Also:

[min](#) | [compare](#) | [flatten](#)

## min

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

computes the minimum value among all the argument's elements.

### Arguments:

1. values : an object, all atoms of which will be inspected, no matter how deeply nested.

## Returns:

An **atom**, the minimum of all atoms in `flatten(values)`.

## Comments:

This function may be applied to an atom or to a sequence of any shape.

## Example 1:

```
a = min({10,15.4,3})
-- a is 3
```

## ensure\_in\_range

```
include std/math.e
namespace math
public function ensure_in_range(object item, sequence range_limits)
```

ensures that the item is in a range of values supplied by inclusive `range_limits`.

## Arguments:

1. `item` : The object to test for.
2. `range_limits` : A sequence of two or more elements. The first is assumed to be the smallest value and the last is assumed to be the highest value.

## Returns:

A **object**, If `item` is lower than the first item in the `range_limits` it returns the first item. If `item` is higher than the last element in the `range_limits` it returns the last item. Otherwise it returns `item`.

## Example 1:

```
object valid_data = ensure_in_range(user_data, {2, 75})
if not equal(valid_data, user_data) then
    errmsg("Invalid input supplied. Using %d instead.", valid_data)
end if
procA(valid_data)
```

## ensure\_in\_list

```
include std/math.e
namespace math
public function ensure_in_list(object item, sequence list, integer default = 1)
```

ensures that the item is in a list of values supplied by `list`.

## Arguments:

1. `item` : The object to test for.
2. `list` : A sequence of elements that `item` should be a member of.
3. `default` : an integer, the index of the list item to return if `item` is not found. Defaults to 1.

## Returns:

An **object**, if `item` is not in the list, it returns the list item of index `default`, otherwise it returns `item`.

## Comments:

If default is set to an invalid index, the first item on the list is returned instead when item is not on the list.

## Example 1:

```
object valid_data = ensure_in_list(user_data, {100, 45, 2, 75, 121})
if not equal(valid_data, user_data) then
  errmsg("Invalid input supplied. Using %d instead.", valid_data)
end if
procA(valid_data)
```

## Roundings and Remainders

### remainder

```
<built-in> function remainder(object dividend, object divisor)
```

computes the remainder of the division of two objects using truncated division.

## Arguments:

1. dividend : any Euphoria object.
2. divisor : any Euphoria object.

## Returns:

An **object**, the shape of which depends on dividend's and divisor's. For two atoms, this is the remainder of dividing dividend by divisor, with dividend's sign.

## Errors:

1. If any atom in divisor is 0, this is an error condition as it amounts to an attempt to divide by zero.
2. If both dividend and divisor are sequences, they must be the same length as each other.

## Comments:

- There is a integer  $N$  such that  $\text{dividend} = N * \text{divisor} + \text{result}$ .
- The result has the sign of dividend and lesser magnitude than divisor.
- The result has the same sign as the dividend.
- This differs from `mod` in that when the operands' signs are different this function rounds dividend/divisor towards zero whereas `mod` rounds away from zero.

The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply, and determine the shape of the returned object.

## Example 1:

```
a = remainder(9, 4)
-- a is 1
```

## Example 2:

```
s = remainder({81, -3.5, -9, 5.5}, {8, -1.7, 2, -4})
-- s is {1, -0.1, -1, 1.5}
```

## Example 3:

```
s = remainder({17, 12, 34}, 16)
-- s is {1, 12, 2}
```

### Example 4:

```
s = remainder(16, {2, 3, 5})
-- s is {0, 1, 1}
```

### See Also:

[mod](#) | [Relational operators](#) | [Operations on sequences](#)

## mod

```
include std/math.e
namespace math
public function mod(object x, object y)
```

computes the remainder of the division of two objects using floored division.

### Arguments:

1. dividend : any Euphoria object.
2. divisor : any Euphoria object.

### Returns:

An **object**, the shape of which depends on dividend's and divisor's. For two atoms, this is the remainder of dividing dividend by divisor, with divisor's sign.

### Comments:

- There is a integer  $N$  such that  $\text{dividend} = N * \text{divisor} + \text{result}$ .
- The result is non-negative and has lesser magnitude than divisor.  $n$  needs not fit in an Euphoria integer.
- The result has the same sign as the dividend.
- The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply, and determine the shape of the returned object.
- When both arguments have the same sign, `mod()` and `remainder` return the same result.
- This differs from `remainder` in that when the operands' signs are different this function rounds dividend/divisor away from zero whereas `remainder` rounds towards zero.

### Example 1:

```
a = mod(9, 4)
-- a is 1
```

### Example 2:

```
s = mod({81, -3.5, -9, 5.5}, {8, -1.7, 2, -4})
-- s is {1,-0.1,1,-2.5}
```

### Example 3:

```
s = mod({17, 12, 34}, 16)
-- s is {1, 12, 2}
```

### Example 4:

```
s = mod(16, {2, 3, 5})
-- s is {0, 1, 1}
```

### See Also:

[remainder](#) | [Relational operators](#) | [Operations on sequences](#)

## trunc

```
include std/math.e
namespace math
public function trunc(object x)
```

returns the integer portion of a number.

### Arguments:

1. value : any Euphoria object.

### Returns:

An **object**, the shape of which depends on values's. Each item in the returned object will be an integer. These are the same corresponding items in value except with any fractional portion removed.

### Comments:

- This is essentially done by always rounding towards zero. The `floor` function rounds towards negative infinity, which means it rounds towards zero for positive values and away from zero for negative values.
- Note that  $\text{trunc}(x) + \text{frac}(x) = x$

### Example 1:

```
a = trunc(9.4)
-- a is 9
```

### Example 2:

```
s = trunc({81, -3.5, -9.999, 5.5})
-- s is {81,-3, -9, 5}
```

### See Also:

[floor](#) [frac](#)

## frac

```
include std/math.e
namespace math
public function frac(object x)
```

returns the fractional portion of a number.

### Arguments:

1. value : any Euphoria object.

### Returns:

An **object**, the shape of which depends on values's. Each item in the returned object will be the same corresponding items in value except with the integer portion removed.

### Comments:

Note that  $\text{trunc}(x) + \text{frac}(x) = x$

### Example 1:

```
a = frac(9.4)
```

```
-- a is 0.4
```

### Example 2:

```
s = frac({81, -3.5, -9.999, 5.5})
-- s is {0, -0.5, -0.999, 0.5}
```

### See Also:

[trunc](#)

## intdiv

```
include std/math.e
namespace math
public function intdiv(object a, object b)
```

returns an integral division of two objects.

### Arguments:

1. `divided` : any Euphoria object.
2. `divisor` : any Euphoria object.

### Returns:

An **object**, which will be a sequence if either dividend or divisor is a sequence.

### Comments:

- This calculates how many non-empty sets when dividend is divided by divisor.
- The result's sign is the same as the dividend's sign.

### Example 1:

```
object Tokens = 101
object MaxPerEnvelope = 5
integer Envelopes = intdiv( Tokens, MaxPerEnvelope) --> 21
```

## floor

```
<built-in> function floor(object value)
```

Rounds value down to the next integer less than or equal to value.

### Arguments:

1. `value` : any Euphoria object; each atom in value will be acted upon.

### Comments:

It does not simply truncate the fractional part, but actually rounds towards negative infinity.

### Returns:

An **object**, the same shape as `value` but with each item guaranteed to be an integer less than or equal to the corresponding item in `value`.

### Example 1:

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

## See Also:

[ceil](#) | [round](#)

## ceil

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

computes the next integer equal or greater than the argument.

## Arguments:

1. `value` : an object, each atom of which processed, no matter how deeply nested.

## Returns:

An **object**, the same shape as `value`. Each atom in `value` is returned as an integer that is the smallest integer equal to or greater than the corresponding atom in `value`.

## Comments:

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

`ceil(X)` is 1 more than `floor(X)` for non-integers. For integers,  $X = \text{floor}(X) = \text{ceil}(X)$ .

## Example 1:

```
sequence nums
nums = {8, -5, 3.14, 4.89, -7.62, -4.3}
nums = ceil(nums) -- {8, -5, 4, 5, -7, -4}
```

## See Also:

[floor](#) | [round](#)

## round

```
include std/math.e
namespace math
public function round(object a, object precision = 1)
```

returns the argument's elements rounded to some precision.

## Arguments:

1. `value` : an object, each atom of which will be acted upon, no matter how deeply nested.
2. `precision` : an object, the rounding precision(s). If not passed, this defaults to 1.

## Returns:

An **object**, the same shape as `value`. When `value` is an atom, the result is that atom rounded to the nearest integer multiple of  $1/\text{precision}$ .

## Comments:

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

### Example 1:

```
round(5.2) -- 5
round({4.12, 4.67, -5.8, -5.21}, 10) -- {4.1, 4.7, -5.8, -5.2}
round(12.2512, 100) -- 12.25
```

### See Also:

[floor](#) | [ceil](#)

## Trigonometry

### arctan

```
<built-in> function arctan(object tangent)
```

returns an angle with given tangent.

### Arguments:

1. tangent : an object, each atom of which will be converted, no matter how deeply nested.

### Returns:

An **object**, of the same shape as tangent. For each atom in flatten(tangent), the angle with smallest magnitude that has this atom as tangent is computed.

### Comments:

All atoms in the returned value lie between  $-\pi/2$  and  $\pi/2$ , exclusive.

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

arctan is faster than [arcsin](#) or [arccos](#).

### Example 1:

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

### See Also:

[arcsin](#) | [arccos](#) | [tan](#) | [flatten](#)

### tan

```
<built-in> function tan(object angle)
```

returns the tangent of an angle, or a sequence of angles.

### Arguments:

1. angle : an object, each atom of which will be converted, no matter how deeply nested.

### Returns:

An **object**, of the same shape as angle. Each atom in the flattened angle is replaced by its



tangent.

### Errors:

If any atom in angle is an odd multiple of  $\pi/2$ , an error occurs, as its tangent would be infinite.

### Comments:

This function may be applied to an atom or to all elements of a sequence of arbitrary shape, recursively.

### Example 1:

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

### See Also:

[sin](#) | [cos](#) | [arctan](#)

## COS

```
<built-in> function cos(object angle)
```

returns the cosine of an angle expressed in radians.

### Arguments:

1. angle : an object, each atom of which will be converted, no matter how deeply nested.

### Returns:

An **object**, the same shape as angle. Each atom in angle is turned into its cosine.

### Comments:

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

The cosine of an angle is an atom between -1 and 1 inclusive. 0.0 is hit by odd multiples of  $\pi/2$  only.

### Example 1:

```
x = cos({.5, .6, .7})
-- x is {0.8775826, 0.8253356, 0.7648422}
```

### See Also:

[sin](#) | [tan](#) | [arccos](#) | [PI](#) | [deg2rad](#)

## SIN

```
<built-in> function sin(object angle)
```

returns the sine of an angle expressed in radians.

### Arguments:

1. angle : an object, each atom in which will be acted upon.

### Returns:

An **object**, the same shape as `angle`. When `angle` is an atom, the result is the sine of `angle`.

### Comments:

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

The sine of an angle is an atom between -1 and 1 inclusive. 0.0 is hit by integer multiples of `PI` only.

### Example 1:

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

### See Also:

`cos` | `arcsin` | `PI` | `deg2rad`

## arccos

```
include std/math.e
namespace math
public function arccos(trig_range x)
```

returns an angle given its cosine.

### Arguments:

1. `value` : an object, each atom in which will be acted upon.

### Returns:

An **object**, the same shape as `value`. When `value` is an atom, the result is an atom, an angle whose cosine is `value`.

### Errors:

If any atom in `value` is not in the -1..1 range, it cannot be the cosine of a real number, and an error occurs.

### Comments:

A value between 0 and `PI` radians will be returned.

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

`arccos` is not as fast as `arctan`.

### Example 1:

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

### See Also:

`cos` | `PI` | `arctan`

## arcsin

```
include std/math.e
namespace math
public function arcsin(trig_range x)
```

returns an angle given its sine.

### Arguments:

1. value : an object, each atom in which will be acted upon.

### Returns:

An **object**, the same shape as value. When value is an atom, the result is an atom, an angle whose sine is value.

### Errors:

If any atom in value is not in the -1..1 range, it cannot be the sine of a real number, and an error occurs.

### Comments:

A value between  $-\pi/2$  and  $+\pi/2$  (radians) inclusive will be returned.

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

arcsin is not as fast as [arctan](#).

### Example 1:

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

### See Also:

[arccos](#), [arccos](#), [sin](#)

## atan2

```
include std/math.e
namespace math
public function atan2(atom y, atom x)
```

calculate the arctangent of a ratio.

### Arguments:

1. y : an atom, the numerator of the ratio
2. x : an atom, the denominator of the ratio

### Returns:

An **atom**, which is equal to [arctan](#)(y/x), except that it can handle zero denominator and is more accurate.

### Example 1:

```
a = atan2(10.5, 3.1)
-- a is 1.283713958
```

### See Also:

[arctan](#)

## rad2deg

```
include std/math.e
namespace math
public function rad2deg(object x)
```

converts an angle measured in radians to an angle measured in degrees.

### Arguments:

1. angle : an object, all atoms of which will be converted, no matter how deeply nested.

### Returns:

An **object**, the same shape as `angle`, all atoms of which were multiplied by  $180/\pi$ .

### Comments:

This function may be applied to an atom or sequence. A flat angle is  $\pi$  radians and 180 degrees.

`arcsin`, `arccos` and `arctan` return angles in radians.

### Example 1:

```
x = rad2deg(3.385938749)
-- x is 194
```

### See Also:

`deg2rad`

## deg2rad

```
include std/math.e
namespace math
public function deg2rad(object x)
```

converts an angle measured in degrees to an angle measured in radians.

### Arguments:

1. angle : an object, all atoms of which will be converted, no matter how deeply nested.

### Returns:

An **object**, the same shape as `angle`, all atoms of which were multiplied by  $\pi/180$ .

### Comments:

This function may be applied to an atom or sequence. A flat angle is  $\pi$  radians and 180 degrees. `sin`, `cos` and `tan` expect angles in radians.

### Example 1:

```
x = deg2rad(194)
-- x is 3.385938749
```

### See Also:

`rad2deg`

# Logarithms and Powers

## log

```
<built-in> function log(object value)
```

returns the natural logarithm of a positive number.

### Arguments:

1. value : an object, any atom of which log acts upon.

### Returns:

An **object**, the same shape as value. For an atom, the returned atom is its logarithm of base E.

### Errors:

If any atom in value is not greater than zero, an error occurs as its logarithm is not defined.

### Comments:

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

To compute the inverse, you can use power(E, x) where E is 2.7182818284590452, or equivalently `exp(x)`. Beware that the logarithm grows very slowly with x, so that `exp` grows very fast.

### Example 1:

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

### See Also:

E, exp, log10

## log10

```
include std/math.e
namespace math
public function log10(object x1)
```

returns the base 10 logarithm of a number.

### Arguments:

1. value : an object, each atom of which will be converted, no matter how deeply nested.

### Returns:

An **object**, the same shape as value. When value is an atom, raising 10 to the returned atom yields value back.

### Errors:

If any atom in value is not greater than zero, its logarithm is not a real number and an error occurs.

## Comments:

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

$\log_{10}$  is proportional to  $\log$  by a factor of  $1/\log(10)$ , which is about 0.435 .

## Example 1:

```
a = log10(12)
-- a is 2.48490665
```

## See Also:

[log](#)

## exp

```
include std/math.e
namespace math
public function exp(atom x)
```

computes some power of E.

## Arguments:

1. **value** : an object, all atoms of which will be acted upon, no matter how deeply nested.

## Returns:

An **object**, the same shape as **value**. When **value** is an atom, its exponential is being returned.

## Comments:

This function can be applied to a single atom or to a sequence of any shape.

Due to its rapid growth, the returned values start losing accuracy as soon as values are greater than 10. Values above 710 will cause an overflow in hardware.

## Example 1:

```
x = exp(5.4)
-- x is 221.4064162
```

## See Also:

[log](#)

## power

```
<built-in> function power(object base, object exponent)
```

raises a base value to some power.

## Arguments:

1. **base** : an object, the value or values to raise to some power.
2. **exponent** : an object, the exponent or exponents to apply to base.

## Returns:

An **object**, the shape of which depends on base's and exponent's. For two atoms, this will be base raised to the power exponent.

### Errors:

If some atom in base is negative and is raised to a non integer exponent, an error will occur, as the result is undefined.

If 0 is raised to any negative power, this is the same as a zero divide and causes an error.

power(0,0) is illegal, because there is not an unique value that can be assigned to that quantity.

### Comments:

The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

Powers of 2 are calculated very efficiently.

Other languages have a `**` or `^` operator to perform the same action. But they do not have sequences.

### Example 1:

```
? power(5, 2)
-- 25 is printed
```

### Example 2:

```
? power({5, 4, 3.5}, {2, 1, -0.5})
-- {25, 4, 0.534522} is printed
```

### Example 3:

```
? power(2, {1, 2, 3, 4})
-- {2, 4, 8, 16}
```

### Example 4:

```
? power({1, 2, 3, 4}, 2)
-- {1, 4, 9, 16}
```

### See Also:

[log](#), [Operations on sequences](#)

## sqrt

```
<built-in> function sqrt(object value)
```

calculates the square root of a number.

### Arguments:

1. value : an object, each atom in which will be acted upon.

### Returns:

An **object**, the same shape as value. When value is an atom, the result is the positive atom whose square is value.

## Errors:

If any atom in value is less than zero, an error will occur, as no squared real can be less than zero.

## Comments:

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

## Example 1:

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

## See Also:

[power](#) | [Operations on sequences](#)

## fib

```
include std/math.e
namespace math
public function fib(integer i)
```

computes the nth Fibonacci Number.

## Arguments:

1. value : an integer. The starting value to compute a Fibonacci Number from.

## Returns:

An **atom**,

- The Fibonacci Number specified by value.

## Comments:

- Note that due to the limitations of the floating point implementation, only 'i' values less than 76 are accurate on *Windows* platforms, and 69 on other platforms (due to rounding differences in the native C runtime libraries).

## Example 1:

```
? fib(6)
-- output ...
-- 8
```

# Hyperbolic Trigonometry

## cosh

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

computes the hyperbolic cosine of an object.

## Arguments:

1. x : the object to process.



### Returns:

An **object**, the same shape as *x*, each atom of which was acted upon.

### Comments:

The hyperbolic cosine grows like the exponential function.

For all reals,  $\text{power}(\cosh(x), 2) - \text{power}(\sinh(x), 2) = 1$ . Compare with ordinary trigonometry.

### Example 1:

```
? cosh(LN2) -- prints out 1.25
```

### See Also:

[cos](#) | [sinh](#) | [arccosh](#)

## sinh

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

computes the hyperbolic sine of an object.

### Arguments:

1. *x* : the object to process.

### Returns:

An **object**, the same shape as *x*, each atom of which was acted upon.

### Comments:

The hyperbolic sine grows like the exponential function.

For all reals,  $\text{power}(\cosh(x), 2) - \text{power}(\sinh(x), 2) = 1$ . Compare with ordinary trigonometry.

### Example 1:

```
? sinh(LN2) -- prints out 0.75
```

### See Also:

[cosh](#) | [sin](#) | [arcsinh](#)

## tanh

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

computes the hyperbolic tangent of an object.

### Arguments:

1. *x* : the object to process.

### Returns:

An **object**, the same shape as *x*, each atom of which was acted upon.

### Comments:

The hyperbolic tangent takes values from -1 to +1.

tanh is the ratio sinh / cosh. Compare with ordinary trigonometry.

### Example 1:

```
? tanh(LN2) -- prints out 0.6
```

### See Also:

[cosh](#) | [sinh](#) | [tan](#) | [arctanh](#)

## arcsinh

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

computes the reverse hyperbolic sine of an object.

### Arguments:

1. *x* : the object to process.

### Returns:

An **object**, the same shape as *x*, each atom of which was acted upon.

### Comments:

The hyperbolic sine grows like the logarithm function.

### Example 1:

```
? arcsinh(1) -- prints out 0,4812118250596034
```

### See Also:

[arccosh](#) | [arcsin](#) | [sinh](#)

## arccosh

```
include std/math.e
namespace math
public function arccosh(not_below_1 a)
```

computes the reverse hyperbolic cosine of an object.

### Arguments:

1. *x* : the object to process.

### Returns:

An **object**, the same shape as `x`, each atom of which was acted upon.

### Errors:

Since `cosh` only takes values not below 1, an argument below 1 causes an error.

### Comments:

The hyperbolic cosine grows like the logarithm function.

### Example 1:

```
? arccosh(1) -- prints out 0
```

### See Also:

`arccos` | `arcsinh` | `cosh`

## arctanh

```
include std/math.e
namespace math
public function arctanh(abs_below_1 a)
```

computes the reverse hyperbolic tangent of an object.

### Arguments:

1. `x` : the object to process.

### Returns:

An **object**, the same shape as `x`, each atom of which was acted upon.

### Errors:

Since `tanh` only takes values between -1 and +1 excluded, an out of range argument causes an error.

### Comments:

The hyperbolic cosine grows like the logarithm function.

### Example 1:

```
? arctanh(1/2) -- prints out 0,5493061443340548456976
```

### See Also:

`arccos` | `arcsinh` | `cosh`

## Accumulation

## sum

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

computes the sum of all atoms in the argument, no matter how deeply nested.

## Arguments:

1. `values` : an object, all atoms of which will be added up, no matter how nested.

## Returns:

An **atom**, the sum of all atoms in `flatten(values)`.

## Comments:

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

## Example 1:

```
a = sum({10, 20, 30})
-- a is 60

a = sum({10.5, {11.2} , 8.1})
-- a is 29.8
```

## See Also:

`product` | `or_all`

## product

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

computes the product of all the atom in the argument, no matter how deeply nested.

## Arguments:

1. `values` : an object, all atoms of which will be multiplied up, no matter how nested.

## Returns:

An **atom**, the product of all atoms in `flatten(values)`.

## Comments:

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

## Example 1:

```
a = product({10, 20, 30})
-- a is 6000

a = product({10.5, {11.2} , 8.1})
-- a is 952.56
```

## See Also:

`sum` | `or_all`

## or\_all

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

or's together all atoms in the argument, no matter how deeply nested.

### Arguments:

1. **values** : an object, all atoms of which will be added up, no matter how nested.

### Returns:

An **atom**, the result of bitwise or of all atoms in `flatten(values)`.

### Comments:

This function may be applied to an atom or to all elements of a sequence. It performs `or_bits` operations repeatedly.

### Example 1:

```
a = or_all({10, 7, 35})
-- a is 47
-- To see why notice:
-- 10=0b1010, 7=0b111 and 35=0b100011.
-- combining these gives:
--           0b001010
-- (or_bits)0b000111
--           0b100011
--           -----
--           0b101111 = 47
```

### See Also:

`sum` | `product` | `or_bits`

## Bitwise Operations

### and\_bits

```
<built-in> function and_bits(object a, object b)
```

performs the bitwise AND operation on corresponding bits in two objects. A bit in the result will be 1 only if the corresponding bits in both arguments are 1.

### Arguments:

1. **a** : one of the objects involved
2. **b** : the second object

### Returns:

An **object**, whose shape depends on the shape of both arguments. Each atom in this object is obtained by bitwise AND between atoms on both objects.

### Comments:

The arguments to this function may be atoms or sequences. The rules for operations on sequences apply. The atoms in the arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as `atom`, rather than `integer`. Euphoria's `integer` type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

To understand the binary representation of a number you should display it in

hexadecimal notation. Use the %x format of `printf`. Using `int_to_bits` is an even more direct approach.

### Example 1:

```
a = and_bits(#0F0F0000, #12345678)
-- a is #02040000
```

### Example 2:

```
a = and_bits(#FF, {#123456, #876543, #2211})
-- a is {#56, #43, #11}
```

### Example 3:

```
a = and_bits(#FFFFFFFF, #FFFFFFFF)
-- a is -1
-- Note that #FFFFFFFF is a positive number,
-- but the result of a bitwise operation is interpreted
-- as a signed 32-bit number, so it's negative.
```

### See Also:

[or\\_bits](#) | [xor\\_bits](#) | [not\\_bits](#) | [int\\_to\\_bits](#)

## xor\_bits

```
<built-in> function xor_bits(object a, object b)
```

performs the bitwise XOR operation on corresponding bits in two objects. A bit in the result will be 1 only if the corresponding bits in both arguments are different.

### Arguments:

1. a : one of the objects involved
2. b : the second object

### Returns:

An **object**, whose shape depends on the shape of both arguments. Each atom in this object is obtained by bitwisel XOR between atoms on both objects.

### Comments:

The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as `atom`, rather than `integer`. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

### Example 1:

```
a = xor_bits(#0110, #1010)
-- a is #1100
```

### See Also:

[and\\_bits](#) | [or\\_bits](#) | [not\\_bits](#) | [int\\_to\\_bits](#)

## or\_bits

```
<built-in> function or_bits(object a, object b)
```

performs the bitwise OR operation on corresponding bits in two objects. A bit in the result will be 1 only if the corresponding bits in both arguments are both 0.

### Arguments:

1. a : one of the objects involved
2. b : the second object

### Returns:

An **object**, whose shape depends on the shape of both arguments. Each atom in this object is obtained by bitwise OR between atoms on both objects.

### Comments:

The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as atom, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

### 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](#)

## not\_bits

```
<built-in> function not_bits(object a)
```

performs the bitwise NOT operation on each bit in an object. A bit in the result will be 1 when the corresponding bit in x1 is 0, and will be 0 when the corresponding bit in x1 is 1.

### Arguments:

1. a : the object to invert the bits of.

### Returns:

An **object**, the same shape as a. Each bit in an atom of the result is the reverse of the corresponding bit inside a.

### Comments:

The argument to this function may be an atom or a sequence.

The argument must be representable as a 32-bit number, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as `atom`, rather than `integer`. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

A simple equality holds for an atom `a`: `a + not_bits(a) = -1`.

### Example 1:

```
a = not_bits(#000000F7)
-- a is -248 (i.e. FFFFFFF08 interpreted as a negative number)
```

### See Also:

[and\\_bits](#) | [or\\_bits](#) | [xor\\_bits](#) | [int\\_to\\_bits](#)

## shift\_bits

```
include std/math.e
namespace math
public function shift_bits(object source_number, integer shift_distance)
```

moves the bits in the input value by the specified distance.

### Arguments:

1. `source_number` : object: The value or values whose bits will be moved.
2. `shift_distance` : integer: number of bits to be moved by.

### Comments:

- If `source_number` is a sequence, each element is shifted.
- The value or values in `source_number` are first truncated to a 32-bit integer.
- The output is truncated to a 32-bit integer.
- Vacated bits are replaced with zero.
- If `shift_distance` is negative, the bits in `source_number` are moved left.
- If `shift_distance` is positive, the bits in `source_number` are moved right.
- If `shift_distance` is zero, the bits in `source_number` are not moved.

### Returns:

Atom or atoms containing a 32-bit integer. A single atom in `source_number` is an atom, or a sequence in the same form as `source_number` containing 32-bit integers.

### Example 1:

```
? shift_bits((7, -3) --> 56
? shift_bits((0, -9) --> 0
? shift_bits((4, -7) --> 512
? shift_bits((8, -4) --> 128
? shift_bits((0xFE427AAC, -7) --> 0x213D5600
? shift_bits((-7, -3) --> -56 which is 0xFFFFFC8
? shift_bits((131, 0) --> 131
? shift_bits((184.464, 0) --> 184
? shift_bits((999_999_999_999_999, 0) --> -1530494977 which is 0xA4C67FFF
? shift_bits((184, 3) --> 23
? shift_bits((48, 2) --> 12
? shift_bits((121, 3) --> 15
? shift_bits((0xFE427AAC, 7) --> 0x01FC84F5
? shift_bits((-7, 3) --> 0x1FFFFFFF
? shift_bits({48, 121}, 2) --> {12, 30}
```

### See Also:



[rotate\\_bits](#)**rotate\_bits**

```
include std/math.e
namespace math
public function rotate_bits(object source_number, integer shift_distance)
```

rotates the bits in the input value by the specified distance.

**Arguments:**

1. `source_number` : object: value or values whose bits will be rotated.
2. `shift_distance` : integer: number of bits to be moved by.

**Comments:**

- If `source_number` is a sequence, each element is rotated.
- The value(s) in `source_number` are first truncated to a 32-bit integer.
- The output is truncated to a 32-bit integer.
- If `shift_distance` is negative, the bits in `source_number` are rotated left.
- If `shift_distance` is positive, the bits in `source_number` are rotated right.
- If `shift_distance` is zero, the bits in `source_number` are not rotated.

**Returns:**

Atom or atoms containing a 32-bit integer. A single atom in `source_number` is an atom, or a sequence in the same form as `source_number` containing 32-bit integers.

**Example 1:**

```
? rotate_bits(7, -3) --> 56
? rotate_bits(0, -9) --> 0
? rotate_bits(4, -7) --> 512
? rotate_bits(8, -4) --> 128
? rotate_bits(0xFE427AAC, -7) --> 0x213D567F
? rotate_bits(-7, -3) --> -49 which is 0xFFFFFCF
? rotate_bits(131, 0) --> 131
? rotate_bits(184.464, 0) --> 184
? rotate_bits(999_999_999_999_999, 0) --> -1530494977 which is 0xA4C67FFF
? rotate_bits(184, 3) -- 23
? rotate_bits(48, 2) --> 12
? rotate_bits(121, 3) --> 536870927
? rotate_bits(0xFE427AAC, 7) --> 0x59FC84F5
? rotate_bits(-7, 3) --> 0x3FFFFFFF
? rotate_bits({48, 121}, 2) --> {12, 1073741854}
```

**See Also:**

[shift\\_bits](#)

**Arithmetic****gcd**

```
include std/math.e
namespace math
public function gcd(atom p, atom q)
```

returns the greater common divisor of to atoms.

### Arguments:

1.  $p$  : one of the atoms to consider
2.  $q$  : the other atom.

### Returns:

A positive **integer**, which is the largest value that evenly divides into both parameters.

### Comments:

- Signs are ignored. Atoms are rounded down to integers.
- If both parameters are zero, 0 is returned.
- If one parameter is zero, the other parameter is returned.

Parameters and return value are atoms so as to take mathematical integers up to `power(2,53)`.

### Example 1:

```
? gcd(76.3, -114) --> 38
? gcd(0, -114) --> 114
? gcd(0, 0) --> 0 (This is often regarded as an error condition)
```

## Floating Point

### approx

```
include std/math.e
namespace math
public function approx(object p, object q, atom epsilon = 0.005)
```

compares two (sets of) numbers based on approximate equality.

### Arguments:

1.  $p$  : an object, one of the sets to consider
2.  $q$  : an object, the other set.
3.  $\epsilon$  : an atom used to define the amount of inequality allowed. This must be a positive value. Default is 0.005

### Returns:

An **integer**,

- 1 when  $p > (q + \epsilon)$  :  $P$  is definitely greater than  $q$ .
- -1 when  $p < (q - \epsilon)$  :  $P$  is definitely less than  $q$ .
- 0 when  $p \geq (q - \epsilon)$  and  $p \leq (q + \epsilon)$  :  $p$  and  $q$  are approximately equal.

### Comments:

This can be used to see if two numbers are near enough to each other.

Also, because of the way floating point numbers are stored, it not always possible express every real number exactly, especially after a series of arithmetic operations. You can use `approx` to see if two floating point numbers are almost the same value.

If  $p$  and  $q$  are both sequences, they must be the same length as each other.

If  $p$  or  $q$  is a sequence, but the other is not, then the result is a sequence of results whose length is the same as the sequence argument.

### Example 1:

```
? approx(10, 33.33 * 30.01 / 100)
--> 0 because 10 and 10.002333 are within 0.005 of each other
? approx(10, 10.001)
--> 0 because 10 and 10.001 are within 0.005 of each other
? approx(10, {10.001, 9.999, 9.98, 10.04})
--> {0,0,1,-1}
? approx({10.001, 9.999, 9.98, 10.04}, 10)
--> {0,0,-1,1}
? approx({10.001, {9.999, 10.01}, 9.98, 10.04}, {10.01, 9.99, 9.8, 10.4})
--> {-1, {1,1}, 1, -1}
? approx(23, 32, 10)
--> 0 because 23 and 32 are within 10 of each other.
```

## powof2

```
include std/math.e
namespace math
public function powof2(object p)
```

tests for power of 2.

### Arguments:

1. *p* : an object. The item to test. This can be an integer, atom or sequence.

### Returns:

An **integer**,

- 1 for each item in *p* that is a power of two (like 2, 4, 8, 16, 32, ...)
- 0 for each item in *p* that is **not** a power of two (like 3, 54.322, -2)

### Example 1:

```
for i = 1 to 10 do
  ? {i, powof2(i)}
end for
-- output ...
-- {1,1}
-- {2,1}
-- {3,0}
-- {4,1}
-- {5,0}
-- {6,0}
-- {7,0}
-- {8,1}
-- {9,0}
-- {10,0}
```

## is\_even

```
include std/math.e
namespace math
public function is_even(integer test_integer)
```

tests if the supplied integer is a even or odd number.

### Arguments:

1. *test\_integer* : an integer. The item to test.

## Returns:

An **integer**,

- 1 if its even.
- 0 if its odd.

## Example 1:

```

for i = 1 to 10 do
  ? {i, is_even(i)}
end for
-- output ...
-- {1,0}
-- {2,1}
-- {3,0}
-- {4,1}
-- {5,0}
-- {6,1}
-- {7,0}
-- {8,1}
-- {9,0}
-- {10,1}

```

## is\_even\_obj

```

include std/math.e
namespace math
public function is_even_obj(object test_object)

```

tests if the supplied Euphoria object is even or odd.

## Arguments:

1. test\_object : any Euphoria object. The item to test.

## Returns:

An **object**,

- If test\_object is an integer...
  - 1 if its even.
  - 0 if its odd.
- Otherwise if test\_object is an atom this always returns 0
- otherwise if test\_object is a sequence it tests each element recursively, returning a sequence of the same structure containing ones and zeros for each element. A 1 means that the element at this position was even otherwise it was odd.

## Example 1:

```

for i = 1 to 5 do
  ? {i, is_even_obj(i)}
end for
-- output ...
-- {1,0}
-- {2,1}
-- {3,0}
-- {4,1}
-- {5,0}

```

## Example 2:

```

? is_even_obj(3.4) --> 0

```

### Example 3:

```
? is_even_obj({{1,2,3}, {{4,5},6,{7,8}}},9) --> {{0,1,0},{1,0},1,{0,1}},0}
```

# Math Constants

## Constants

### PI

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

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

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

### QUARTPI

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

Quarter of PI

### HALFPI

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

Half of PI

### TWOPI

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

Two times PI

### PISQR

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

$PI^2$

### INVSQ2PI

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

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

## PHI

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

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

## E

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

Euler (e)The base of the natural logarithm.

## LN2

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

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

## INVLN2

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

$1 / (\ln(2))$

## LN10

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

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

## INVLN10

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

$1 / \ln(10)$

## SQRT2

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

$\sqrt{2}$

## HALFSQRT2

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

$\sqrt{2} / 2$

## SQRT3

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

Square root of 3

## DEGREES\_TO\_RADIANS

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

Conversion factor: Degrees to Radians =  $\pi / 180$

## RADIANS\_TO\_DEGREES

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

Conversion factor: Radians to Degrees =  $180 / \pi$

## EULER\_GAMMA

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

Gamma (Euler Gamma)



## SQRTE

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

`sqrt(e)`

## PINF

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

Positive Infinity

## MINF

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

Negative Infinity

## SQRT5

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

`sqrt(5)`

# Random Numbers

## rand

```
<built-in> function rand(object maximum)
```

returns a random integral value.

### Arguments:

1. maximum : an atom, a cap on the value to return.

### Returns:

An **atom**, from 1 to maximum.

### Comments:

- The minimum value of maximum is 1.
- The maximum value that can possibly be returned is #FFFFFFFF (4\_294\_967\_295)
- This function may be applied to an atom or to all elements of a sequence.
- In order to get reproducible results from this function, you should call `set_rand` with a reproducible value prior.

### Example 1:

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

### See Also:

`set_rand` | `ceil`

## rand\_range

```
include std/rand.e  
namespace random  
public function rand_range(atom lo, atom hi)
```

returns a random integer from a specified inclusive integer range.

### Arguments:

1. lo : an atom, the lower bound of the range
2. hi : an atom, the upper bound of the range.

### Returns:

An **atom**, randomly drawn between lo and hi inclusive.

### Comments:

This function may be applied to an atom or to all elements of a sequence. In order to get reproducible results from this function, you should call `set_rand` with a reproducible value prior.

### Example 1:

```
s = rand_range(18, 24)
-- s could be any of: 18, 19, 20, 21, 22, 23 or 24
```

### See Also:

[rand](#) | [set\\_rand](#) | [rnd](#)

## **rnd**

```
include std/random
namespace random
public function rnd()
```

returns a random floating point number in the range 0 to 1.

### Arguments:

None.

### Returns:

An **atom**, randomly drawn between 0.0 and 1.0 inclusive.

### Comments:

In order to get reproducible results from this function, you should call `set_rand` with a reproducible value prior to calling this.

### Example 1:

```
set_rand(1001)
s = rnd()
-- s is 0.6277338201
```

### See Also:

[rand](#) | [set\\_rand](#) | [rand\\_range](#)

## **rnd\_1**

```
include std/random
namespace random
public function rnd_1()
```

returns a random floating point number in the range 0 to less than 1.

### Arguments:

None.

### Returns:

An **atom**, randomly drawn between 0.0 and a number less than 1.0

### Comments:

In order to get reproducible results from this function, you should call `set_rand` with a reproducible value prior to calling this.

### Example 1:

```
set_rand(1001)
s = rnd_1()
-- s is 0.6277338201
```

## See Also:

[rand](#) | [set\\_rand](#) | [rand\\_range](#)

## set\_rand

```
include std/rand.e
namespace random
public procedure set_rand(object seed)
```

resets the random number generator.

## Arguments:

1. **seed** : an object. The generator uses this initialize itself for the next random number generated. This can be a single integer or atom, or a sequence of two integers, or an empty sequence or any other sort of sequence.

## Comments:

- Starting from a seed, the values returned by `rand` are reproducible. This is useful for demos and stress tests based on random data. Normally the numbers returned by the `rand` function are totally unpredictable, and will be different each time you run your program. Sometimes however you may wish to repeat the same series of numbers, perhaps because you are trying to debug your program, or maybe you want the ability to generate the same output (for example random picture) for your user upon request.
- Internally there are actually two seed values.
  - When `set_rand` is called with a single integer or atom, the two internal seeds are derived from the parameter.
  - When `set_rand` is called with a sequence of exactly two integers or atoms the internal seeds are set to the parameter values.
  - When `set_rand` is called with an empty sequence, the internal seeds are set to random values and are unpredictable. This is how to reset the generator.
  - When `set_rand` is called with any other sequence, the internal seeds are set based on the length of the sequence and the hashed value of the sequence.
- Aside from an empty seed parameter, this sets the generator to a known state and the random numbers generated after come in a predicable order, though they still appear to be random.

## Example 1:

```
sequence s, t
s = repeat(0, 3)
t = s

set_rand(12345)
s[1] = rand(10)
s[2] = rand(100)
s[3] = rand(1000)

set_rand(12345) -- same value for set_rand()
t[1] = rand(10) -- same arguments to rand() as before
t[2] = rand(100)
t[3] = rand(1000)
-- at this point s and t will be identical
set_rand("") -- Reset the generator to an unknown seed.
t[1] = rand(10) -- Could be anything now, no way to predict it.
```

## See Also:

[rand](#)

## get\_rand

```
include std/rand.e
namespace random
public function get_rand()
```

retrieves the current values of the random generator's seeds.

### Returns:

a sequence. A 2-element sequence containing the values of the two internal seeds.

### Comments:

You can use this to save the current seed values so that you can later reset them back to a known state.

### Example 1:

```
sequence seeds
seeds = get_rand()
some_func() -- Which might set the seeds to anything.
set_rand(seeds) -- reset them back to whatever they were
                -- before calling 'some_func()'.
```

## See Also:

[set\\_rand](#)

## chance

```
include std/rand.e
namespace random
public function chance(atom my_limit, atom top_limit = 100)
```

simulates the probability of a desired outcome.

### Arguments:

1. `my_limit`: an atom. The desired chance of something happening.
2. `top_limit`: an atom. The maximum chance of something happening. The default is 100.

### Returns:

an integer. 1 if the desired chance happened otherwise 0.

### Comments:

This simulates the chance of something happening. For example, if you want something to happen with a probability of 25 times out of 100 times then you code `chance(25)` and if you want something to (most likely) occur 345 times out of 999 times, you code `chance(345, 999)`.

### Example 1:

```
-- 65% of the days are sunny, so ...
if chance(65) then
```

```

    puts(1, "Today will be a sunny day")
elseif chance(40) then
    -- And 40% of non-sunny days it will rain.
    puts(1, "It will rain today")
else
    puts(1, "Today will be a overcast day")
end if

```

## See Also:

[rnd](#) | [roll](#)

## roll

```

include std/rand.e
namespace random
public function roll(object desired, integer sides = 6)

```

simulates the probability of a dice throw.

## Arguments:

1. desired : an object. One or more desired outcomes.
2. sides: an integer. The number of sides on the dice. Default is 6.

## Returns:

an integer. 0 if none of the desired outcomes occurred, otherwise the face number that was rolled.

## Comments:

The minimum number of sides is two and there is no maximum.

## Example 1:

```

res = roll(1, 2)
    --> Simulate a coin toss.
res = roll({1,6})
    --> Try for a 1 or a 6 from a standard die toss.
res = roll({1,2,3,4}, 20)
    --> Looking for any number under 5 from a 20-sided die.

```

## See Also:

[rnd](#) | [chance](#)

## sample

```

include std/rand.e
namespace random
public function sample(sequence population, integer sample_size, integer sampling_method = 0)

```

selects a set of random samples from a population set.

## Arguments:

1. population : a sequence. The set of items from which to take a sample.
2. sample\_size: an integer. The number of samples to take.
3. sampling\_method: an integer.
  1. When < 0, "with-replacement" method used.

2. When = 0, "without-replacement" method used and a single set of samples returned.
3. When > 0, "without-replacement" method used and a sequence containing the set of samples (chosen items) and the set unchosen items, is returned.

## Returns:

A sequence. When sampling\_method less than or equal to 0 then this is the set of samples, otherwise it returns a two-element sequence; the first is the samples, and the second is the remainder of the population (in the original order).

## Comments:

Selects a set of random samples from a population set. This can be done with either the "with-replacement" or "without-replacement" methods. When using the "with-replacement" method, after each sample is taken it is returned to the population set so that it could possible be taken again. The "without-replacement" method does not return the sample so these items can only ever be chosen once.

- If sample\_size is less than 1 , an empty set is returned.
- When using "without-replacement" method, if sample\_size is greater than or equal to the population count, the entire population set is returned, but in a random order.
- When using "with-replacement" method, if sample\_size can be any positive integer, thus it is possible to return more samples than there are items in the population set as items can be chosen more than once.

## Example 1:

```
-- without replacement

set_rand("example")
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 1)})
--> "t"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 5)})
--> "flukq"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", -1)})
--> ""
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 26)})
--> "kghrsxmjoeubaywlfzftcpivqnd"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 25)})
--> "omntrqsbjguaikzywvxflpedc"
```

## Example 2:

```
-- with replacement

set_rand("example")
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 1, -1)})
--> "t"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 5, -1)})
--> "fzycn"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", -1, -1)})
--> ""
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 26, -1)})
--> "keeamenuvvfyelqapucergghgfa"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 45, -1)})
--> "orwpsaxuwyrbstqqwfkjukukzkuxvzvzniinnpxm"
```

## Example 3:

```
-- Deal 4 hands of 5 cards from a standard deck of cards.
sequence theDeck
sequence hands = {}
sequence rt
function new_deck(integer suits = 4, integer cards_per_suit = 13, integer wilds = 0)
sequence nd = {}
```

```
for i = 1 to suits do
  for j = 1 to cards_per_suit do
    nd = append(nd, {i,j})
  end for
end for
for i = 1 to wilds do
  nd = append(nd, {suits+1 , i})
end for
return nd
end function

theDeck = new_deck(4, 13, 2) -- Build the initial deck of cards
for i = 1 to 4 do
  -- Pick out 5 cards and also return the remaining cards.
  rt = sample(theDeck, 5, 1)
  theDeck = rt[2] -- replace the 'deck' with the remaining cards.
  hands = append(hands, rt[1])
end for
```



# Statistics

## Subroutines

### small

```
include std/stats.e
namespace stats
public function small(sequence data_set, integer ordinal_idx)
```

determines the k-th smallest value from the supplied set of numbers.

#### Arguments:

1. data\_set : The list of values from which the smallest value is chosen.
2. ordinal\_idx : The relative index of the desired smallest value.

#### Returns:

A **sequence**, {The k-th smallest value, its index in the set}.

#### Comments:

small is used to return a value based on its size relative to all the other elements in the sequence. When index is 1, the smallest index is returned. Use index = length(data\_set) to return the highest.

If ordinal\_idx is less than one, or greater than length of data\_set, an empty sequence is returned.

The set of values does not have to be in any particular order. The values may be any Euphoria object.

#### Example 1:

```
small( {4,5,6,8,5,4,3,"text"}, 3 )
--> Ans: {4,1} (The 3rd smallest value)
small( {4,5,6,8,5,4,3,"text"}, 1 )
--> Ans: {3,7} (The 1st smallest value)
small( {4,5,6,8,5,4,3,"text"}, 7 )
--> Ans: {8,4} (The 7th smallest value)
small( {"def", "qwe", "abc", "try"}, 2 )
--> Ans: {"def", 1} (The 2nd smallest value)
small( {1,2,3,4}, -1)
--> Ans: {} -- no-value
small( {1,2,3,4}, 10)
--> Ans: {} -- no-value
```

### largest

```
include std/stats.e
namespace stats
public function largest(object data_set)
```

returns the largest of the data points that are atoms.

#### Arguments:

1. `data_set` : a list of 1 or more numbers among which you want the largest.

### Returns:

An **object**, either of:

- an atom (the largest value) if there is at least one atom item in the set
- {} if there is no largest value.

### Comments:

Any `data_set` element which is not an atom is ignored.

### Example 1:

```
largest( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"} ) -- Ans: 8
largest( {"just","text"} ) -- Ans: {}
```

### See Also:

[range](#)

## smallest

```
include std/stats.e
namespace stats
public function smallest(object data_set)
```

returns the smallest of the data points.

### Arguments:

1. `data_set` : A list of 1 or more numbers for which you want the smallest. **Note:** only atom elements are included and any sub-sequences elements are ignored.

### Returns:

An **object**, either of:

- an atom (the smallest value) if there is at least one atom item in the set
- {} if there is no largest value.

### Comments:

Any `data_set` element which is not an atom is ignored.

### Example 1:

```
? smallest( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"} ) -- Ans: 1
? smallest( {"just","text"} ) -- Ans: {}
```

### See Also:

[range](#)

## range

```
include std/stats.e
namespace stats
public function range(object data_set)
```

determines a number of *range* statistics for the data set.

### Arguments:

1. `data_set` : a list of 1 or more numbers for which you want the range data.

### Returns:

A **sequence**, empty if no atoms were found, else like {Lowest, Highest, Range, Mid-range} ,

### Comments:

Any sequence element in `data_set` is ignored.

### Example 1:

```
? range( {7,2,8,5,6,6,4,8,6,16,3,3,4,1,8,"text"} ) -- Ans: {1, 16, 15, 8.5}
```

### See Also:

[smallest](#) | [largest](#)

Enums used to influence the results of some of these functions.

## enum

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

## ST\_FULLPOP

```
include std/stats.e
namespace stats
ST_FULLPOP
```

The supplied data is the entire population.

## ST\_SAMPLE

```
include std/stats.e
namespace stats
ST_SAMPLE
```

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

## enum

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

## ST\_ALLNUM

```
include std/stats.e
namespace stats
ST_ALLNUM
```

The supplied data consists of only atoms.

## ST\_IGNSTR

```
include std/stats.e
namespace stats
ST_IGNSTR
```

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

## ST\_ZEROSTR

```
include std/stats.e
namespace stats
ST_ZEROSTR
```

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

## stdev

```
include std/stats.e
namespace stats
public function stdev(sequence data_set, object subseq_opt = ST_ALLNUM,
    integer population_type = ST_SAMPLE)
```

returns the standard deviation based on the population.

### Arguments:

1. `data_set` : a list of 1 or more numbers for which you want the estimated standard deviation.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.
3. `population_type` : an integer. `ST_SAMPLE` (the default) assumes that `data_set` is a random sample of the total population. `ST_FULLPOP` means that `data_set` is the entire population.

### Returns:

An **atom**, the estimated standard deviation. An empty **sequence** means that there is no meaningful data to calculate from.

### Comments:

`stdev` is a measure of how values are different from the average.

The numbers in `data_set` can either be the entire population of values or just a random subset. You indicate which in the `population_type` parameter. By default `data_set` represents a sample and not the entire population. When using this function with sample data, the result is an *estimated* standard deviation.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not

the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

The equation for standard deviation is:

```
stdev(X) ==> SQRT(SUM(SQ(X{1..N} - MEAN)) / (N))
```

### Example 1:

```
? stdev( {4,5,6,7,5,4,3,7} )           -- Ans: 1.457737974
? stdev( {4,5,6,7,5,4,3,7} , , ST_FULLPOP ) -- Ans: 1.363589014
? stdev( {4,5,6,7,5,4,3,"text"} , ST_IGNSTR ) -- Ans: 1.345185418
? stdev( {4,5,6,7,5,4,3,"text"} , ST_IGNSTR, ST_FULLPOP ) -- Ans: 1.245399698
? stdev( {4,5,6,7,5,4,3,"text"} , 0 ) -- Ans: 2.121320344
? stdev( {4,5,6,7,5,4,3,"text"} , 0, ST_FULLPOP ) -- Ans: 1.984313483
```

### See Also:

[average](#) | [avedev](#)

## avedev

```
include std/stats.e
namespace stats
public function avedev(sequence data_set, object subseq_opt = ST_ALLNUM,
                      integer population_type = ST_SAMPLE)
```

returns the average of the absolute deviations of data points from their mean.

### Arguments:

1. `data_set` : a list of 1 or more numbers for which you want the mean of the absolute deviations.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.
3. `population_type` : an integer. `ST_SAMPLE` (the default) assumes that `data_set` is a random sample of the total population. `ST_FULLPOP` means that `data_set` is the entire population.

### Returns:

An **atom** , the deviation from the mean.

An empty **sequence**, means that there is no meaningful data to calculate from.

### Comments:

`avedev` is a measure of the variability in a data set. Its statistical properties are less well behaved than those of the standard deviation, which is why it is used less.

The numbers in `data_set` can either be the entire population of values or just a random subset. You indicate which in the `population_type` parameter. By default `data_set` represents a sample and not the entire population. When using this function with sample data, the result is an *estimated* deviation.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to

ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

The equation for absolute average deviation is:

```
avedev(X) ==> SUM( ABS(X{1..N} - MEAN(X)) ) / N
```

### Example 1:

```
? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,7} )
--> Ans: 1.966666667
? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,7},, ST_FULLPOP )
--> Ans: 1.84375
? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, ST_IGNSTR )
--> Ans: 1.99047619
? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, ST_IGNSTR,ST_FULLPOP )
--> Ans: 1.857777778
? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, 0 )
--> Ans: 2.225
? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, 0, ST_FULLPOP )
--> Ans: 2.0859375
```

### See Also:

[average](#) | [stdev](#)

## sum

```
include std/stats.e
namespace stats
public function sum(object data_set, object subseq_opt = ST_ALLNUM)
```

returns the sum of all the atoms in an object.

### Arguments:

1. `data_set` : Either an atom or a list of numbers to sum.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

### Returns:

An **atom**, the sum of the set.

### Comments:

`sum` is used as a measure of the magnitude of a sequence of positive values.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

The equation is:

```
sum(X) ==> SUM( X{1..N} )
```

### Example 1:

```
? sum( {7,2,8.5,6,6,-4.8,6,6,3.341,-8,"text"}, 0 ) -- Ans: 32.041
```

### See Also:

[average](#)

## count

```
include std/stats.e
namespace stats
public function count(object data_set, object subseq_opt = ST_ALLNUM)
```

returns the count of all the atoms in an object.

### Arguments:

1. `data_set` : either an atom or a list.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

### Comments:

This returns the number of numbers in `data_set`

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

### Returns:

An **integer**, the number of atoms in the set. When `data_set` is an atom, 1 is returned.

### Example 1:

```
? count( {7,2,8.5,6,6,-4.8,6,6,3.341,-8,"text"} ) -- Ans: 10
? count( {"cat", "dog", "lamb", "cow", "rabbit"} ) -- Ans: 0 (no atoms)
? count( 5 ) -- Ans: 1
```

### See Also:

[average](#) | [sum](#)

## average

```
include std/stats.e
namespace stats
public function average(object data_set, object subseq_opt = ST_ALLNUM)
```

returns the average (mean) of the data points.

## Arguments:

1. `data_set` : A list of 1 or more numbers for which you want the mean.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

## Returns:

An **object**,

- `{}` (the empty sequence) if there are no atoms in the set.
- an atom (the mean) if there are one or more atoms in the set.

## Comments:

average is the theoretical probable value of a randomly selected item from the set.

The equation for average is:

```
average(X) ==> SUM( X{1..N} ) / N
```

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

## Example 1:

```
? average( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, ST_IGNSTR ) -- Ans: 5.13333333
```

## See Also:

[geomean](#) | [harmean](#) | [movavg](#) | [emovavg](#)

## geomean

```
include std/stats.e
namespace stats
public function geomean(object data_set, object subseq_opt = ST_ALLNUM)
```

returns the geometric mean of the atoms in a sequence.

## Arguments:

1. `data_set` : the values to take the geometric mean of.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

## Returns:

An **atom**, the geometric mean of the atoms in `data_set`. If there is no atom to take the mean of, 1 is returned.

## Comments:

The geometric mean of N atoms is the n-th root of their product. Signs are ignored.



This is useful to compute average growth rates.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

### Example 1:

```
? geomean({3, "abc", -2, 6}, ST_IGNSTR) -- prints out power(36,1/3) = 3,30192724889462669
? geomean({1,2,3,4,5,6,7,8,9,10}) -- = 4.528728688
```

### See Also:

[average](#)

## harmean

```
include std/stats.e
namespace stats
public function harmean(sequence data_set, object subseq_opt = ST_ALLNUM)
```

returns the harmonic mean of the atoms in a sequence.

### Arguments:

1. `data_set` : the values to take the harmonic mean of.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

### Returns:

An **atom**, the harmonic mean of the atoms in `data_set`.

### Comments:

The harmonic mean is the inverse of the average of their inverses.

This is useful in engineering to compute equivalent capacities and resistances.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

### Example 1:

```
? harmean({3, "abc", -2, 6}, ST_IGNSTR) -- = 0.
? harmean({2, 3, 4}) -- 3 / (1/2 + 1/3 + 1/4) = 2.769230769
```

### See Also:

[average](#)**movavg**

```
include std/stats.e
namespace stats
public function movavg(object data_set, object period_delta)
```

returns the average (mean) of the data points for overlapping periods. This can be either a simple or weighted moving average.

**Arguments:**

1. `data_set` : a list of 1 or more numbers for which you want a moving average.
2. `period_delta` : an object, either
  - an integer representing the size of the period, or
  - a list of weightings to apply to the respective period positions.

**Returns:**

A **sequence**, either the requested averages or {} if the Data sequence is empty or the supplied period is less than one.

If a list of weights was supplied, the result is a weighted average; otherwise, it is a simple average.

**Comments:**

A moving average is used to smooth out a set of data points over a period. For example, given a period of 5:

1. the first returned element is the average of the first five data points[1..5],
  2. the second returned element is the average of the second five data points[2..6], and so on
- until the last returned value is the average of the last 5 data points [\$-4 .. \$].

When `period_delta` is an atom, it is rounded down to the width of the average. When it is a sequence, the width is its length. If there are not enough data points, zeroes are inserted.

Note that only atom elements are included and any sub-sequence elements are ignored.

**Example 1:**

```
? movavg( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8}, 10 )
-- Ans: {5.8, 5.4, 5.5, 5.1, 4.7, 4.9}
? movavg( {7,2,8,5,6}, 2 )
-- Ans: {4.5, 5, 6.5, 5.5}
? movavg( {7,2,8,5,6}, {0.5, 1.5} )
-- Ans: {3.25, 6.5, 5.75, 5.75}
```

**See Also:**[average](#)**emovavg**

```
include std/stats.e
namespace stats
public function emovavg(object data_set, atom smoothing_factor)
```

returns the exponential moving average of a set of data points.

### Arguments:

1. `data_set` : a list of 1 or more numbers for which you want a moving average.
2. `smoothing_factor` : an atom, the smoothing factor, typically between 0 and 1.

### Returns:

A **sequence**, made of the requested averages, or {} if `data_set` is empty or the supplied period is less than one.

### Comments:

A moving average is used to smooth out a set of data points over a period.

The formula used is:

$$Y_i = Y_{i-1} + F * (X_i - Y_{i-1})$$

Note that only atom elements are included and any sub-sequences elements are ignored.

The smoothing factor controls how data is smoothed. 0 smooths everything to 0, and 1 means no smoothing at all.

Any value for `smoothing_factor` outside the 0.0..1.0 range causes `smoothing_factor` to be set to the periodic factor ( $2/(N+1)$ ).

### Example 1:

```
? emovavg( {7,2,8,5,6}, 0.75 )
-- Ans: {6.65,3.1625,6.790625,5.44765625,5.861914063}
? emovavg( {7,2,8,5,6}, 0.25 )
-- Ans: {5.95,4.9625,5.721875,5.54140625,5.656054687}
? emovavg( {7,2,8,5,6}, -1 )
-- Ans: {6.066666667,4.711111111,5.807407407,5.538271605,5.69218107}
```

### See Also:

[average](#)

## median

```
include std/stats.e
namespace stats
public function median(object data_set, object subseq_opt = ST_ALLNUM)
```

returns the mid point of the data points.

### Arguments:

1. `data_set` : a list of 1 or more numbers for which you want the mean.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

### Returns:

An **object**, either {} if there are no items in the set, or an **atom** (the median) otherwise.

### Comments:

median is the item for which half the items are below it and half are above it.

All elements are included; any sequence elements are assumed to have the value zero.

The equation for average is:

```
median(X) ==> sort(X)[N/2]
```

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

### Example 1:

```
? median( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,4} ) -- Ans: 5
```

### See Also:

[average](#) | [geomean](#) | [harmean](#) | [movavg](#) | [emovavg](#)

## raw\_frequency

```
include std/stats.e
namespace stats
public function raw_frequency(object data_set, object subseq_opt = ST_ALLNUM)
```

returns the frequency of each unique item in the data set.

### Arguments:

1. `data_set` : a list of 1 or more numbers for which you want the frequencies.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

### Returns:

A **sequence**. This will contain zero or more 2-element sub-sequences. The first element is the frequency count and the second element is the data item that was counted. The returned values are in descending order, meaning that the highest frequencies are at the beginning of the returned list.

### Comments:

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

### Example 1:

```
? raw_frequency("the cat is the hatter")
```

This returns

```
{
{5,116},
{4,32},
{3,104},
{3,101},
{2,97},
{1,115},
{1,114},
{1,105},
{1,99}
}
```

## mode

```
include std/stats.e
namespace stats
public function mode(sequence data_set, object subseq_opt = ST_ALLNUM)
```

returns the most frequent point(s) of the data set.

### Arguments:

1. `data_set` : a list of 1 or more numbers for which you want the mode.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

### Returns:

A **sequence**. The list of modal items in the data set.

### Comments:

It is possible for the mode to return more than one item when more than one item in the set has the same highest frequency count.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

### Example 1:

```
mode( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,4} ) -- Ans: {6}
mode( {8,2,8,5,6,6,4,8,6,6,3,3,4,1,8,4} ) -- Ans: {8,6}
```

### See Also:

[average](#) | [geomean](#) | [harmean](#) | [movavg](#) | [emovavg](#)

## central\_moment

```
include std/stats.e
namespace stats
```

```
public function central_moment(sequence data_set, object datum, integer order_mag = 1,
    object subseq_opt = ST_ALLNUM)
```

returns the distance between a supplied value and the mean, to some supplied order of magnitude. This is used to get a measure of the *shape* of a data set.

### Arguments:

1. `data_set` : a list of 1 or more numbers whose mean is used.
2. `datum`: either a single value or a list of values for which you require the central moments.
3. `order_mag`: An integer. This is the order of magnitude required. Usually a number from 1 to 4, but can be anything.
4. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

### Returns:

An **object**. The same data type as `datum`. This is the set of calculated central moments.

### Comments:

For each of the items in `datum`, its central moment is calculated as:

```
CM = power( ITEM - AVG, MAGNITUDE)
```

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

### Example 1:

```
central_moment("the cat is the hatter", "the",1) --> {23.14285714, 11.14285714, 8.142857143}
central_moment("the cat is the hatter", 't',2) --> 535.5918367
central_moment("the cat is the hatter", 't',3) --> 12395.12536
```

### See Also:

[average](#)

## sum\_central\_moments

```
include std/stats.e
namespace stats
public function sum_central_moments(object data_set, integer order_mag = 1,
    object subseq_opt = ST_ALLNUM)
```

returns sum of the central moments of each item in a data set.

### Arguments:

1. `data_set` : a list of 1 or more numbers whose mean is used.
2. `order_mag`: An integer. This is the order of magnitude required. Usually a number from 1 to 4, but can be anything.
3. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

## Returns:

An **atom**. The total of the central moments calculated for each of the items in `data_set`.

## Comments:

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

## Example 1:

```
sum_central_moments("the cat is the hatter", 1) --> -8.526512829e-14
sum_central_moments("the cat is the hatter", 2) --> 19220.57143
sum_central_moments("the cat is the hatter", 3) --> -811341.551
sum_central_moments("the cat is the hatter", 4) --> 56824083.71
```

## See Also:

[central\\_moment](#) | [average](#)

## skewness

```
include std/stats.e
namespace stats
public function skewness(object data_set, object subseq_opt = ST_ALLNUM)
```

returns a measure of the asymmetry of a data set. Usually the `data_set` is a probability distribution but it can be anything. This value is used to assess how suitable the data set is in representing the required analysis. It can help detect if there are too many extreme values in the data set.

## Arguments:

1. `data_set` : a list of 1 or more numbers whose mean is used.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

## Returns:

An **atom**. The skewness measure of the data set.

## Comments:

Generally speaking, a negative return indicates that most of the values are lower than the mean, while positive values indicate that most values are greater than the mean. However this might not be the case when there are a few extreme values on one side of the mean.

The larger the magnitude of the returned value, the more the data is skewed in that direction.

A returned value of zero indicates that the mean and median values are identical and that the data is symmetrical.

If the data can contain sub-sequences, such as strings, you need to let the the function

know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

### Example 1:

```
skewness("the cat is the hatter") --> -1.36166186
skewness("thecatisthehatter")    --> 0.1093730315
```

### See Also:

[kurtosis](#)

## kurtosis

```
include std/stats.e
namespace stats
public function kurtosis(object data_set, object subseq_opt = ST_ALLNUM)
```

returns a measure of the spread of values in a dataset when compared to a *normal* probability curve.

### Arguments:

1. `data_set` : a list of 1 or more numbers whose kurtosis is required.
2. `subseq_opt` : an object. When this is `ST_ALLNUM` (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

### Returns:

An **object**. If this is an atom it is the kurtosis measure of the data set. Otherwise it is a sequence containing an error integer. The return value `{0}` indicates that an empty dataset was passed, `{1}` indicates that the standard deviation is zero (all values are the same).

### Comments:

Generally speaking, a negative return indicates that most of the values are further from the mean, while positive values indicate that most values are nearer to the mean.

The larger the magnitude of the returned value, the more the data is 'peaked' or 'flatter' in that direction.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use `ST_IGNSTR` as the `subseq_opt` parameter value otherwise use `ST_ZEROSTR`. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, `ST_ALLNUM`.

**Note** It is faster if the data only contains numbers.

### Example 1:

```
kurtosis("thecatisthehatter") --> -1.737889192
```



See Also:

skewness

# DATA

- eds.e
- primes.e
- flags.e
- hash.e
- map.e
- stack.e
- scinot.e

# Euphoria Database (EDS)

## Error Status Constants

### enum

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

DB\_OK | DB\_OPEN\_FAIL | DB\_EXISTS\_ALREADY | DB\_LOCK\_FAIL | DB\_BAD\_NAME |  
DB\_FATAL\_FAIL

### DB\_OK

```
include std/eds.e
namespace eds
DB_OK
```

### DB\_OPEN\_FAIL

```
include std/eds.e
namespace eds
DB_OPEN_FAIL
```

### DB\_EXISTS\_ALREADY

```
include std/eds.e
namespace eds
DB_EXISTS_ALREADY
```

### DB\_LOCK\_FAIL

```
include std/eds.e
namespace eds
DB_LOCK_FAIL
```

### DB\_BAD\_NAME

```
include std/eds.e
namespace eds
DB_BAD_NAME
```

### DB\_FATAL\_FAIL

```
include std/eds.e
namespace eds
```

**DB\_FATAL\_FAIL**

## Lock Type Constants

### enum

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

**DB\_LOCK\_NO** | **DB\_LOCK\_SHARED** | **DB\_LOCK\_EXCLUSIVE** | **DB\_LOCK\_READ\_ONLY**

### DB\_LOCK\_NO

```
include std/eds.e
namespace eds
DB_LOCK_NO
```

### DB\_LOCK\_SHARED

```
include std/eds.e
namespace eds
DB_LOCK_SHARED
```

### DB\_LOCK\_EXCLUSIVE

```
include std/eds.e
namespace eds
DB_LOCK_EXCLUSIVE
```

### DB\_LOCK\_READ\_ONLY

```
include std/eds.e
namespace eds
DB_LOCK_READ_ONLY
```

## Error Code Constants

### enum

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

**MISSING\_END** | **NO\_DATABASE** | **BAD\_SEEK** | **NO\_TABLE** | **DUP\_TABLE** | **BAD\_RECNO**  
| **INSERT\_FAILED** | **LAST\_ERROR\_CODE** | **BAD\_FILE**

**MISSING\_END**

```
include std/eds.e
namespace eds
MISSING_END
```

**NO\_DATABASE**

```
include std/eds.e
namespace eds
NO_DATABASE
```

**BAD\_SEEK**

```
include std/eds.e
namespace eds
BAD_SEEK
```

**NO\_TABLE**

```
include std/eds.e
namespace eds
NO_TABLE
```

**DUP\_TABLE**

```
include std/eds.e
namespace eds
DUP_TABLE
```

**BAD\_RECNO**

```
include std/eds.e
namespace eds
BAD_RECNO
```

**INSERT\_FAILED**

```
include std/eds.e
namespace eds
INSERT_FAILED
```

**LAST\_ERROR\_CODE**

```
include std/eds.e
namespace eds
LAST_ERROR_CODE
```

## BAD\_FILE

```
include std/eds.e
namespace eds
BAD_FILE
```

## Indexes for Connection Option Structure.

### enum

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

CONNECT\_LOCK | CONNECT\_TABLES | CONNECT\_FREE

## CONNECT\_LOCK

```
include std/eds.e
namespace eds
CONNECT_LOCK
```

## CONNECT\_TABLES

```
include std/eds.e
namespace eds
CONNECT_TABLES
```

## CONNECT\_FREE

```
include std/eds.e
namespace eds
CONNECT_FREE
```

## Database Connection Options

### DISCONNECT

```
include std/eds.e
namespace eds
public constant DISCONNECT
```

Disconnect a connected database

## LOCK\_METHOD

```
include std/eds.e
namespace eds
public constant LOCK_METHOD
```

Locking method to use

## INIT\_TABLES

```
include std/eds.e
namespace eds
public constant INIT_TABLES
```

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

## INIT\_FREE

```
include std/eds.e
namespace eds
public constant INIT_FREE
```

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

## CONNECTION

```
include std/eds.e
namespace eds
public constant CONNECTION
```

Fetch the details about the alias

## Variables

### db\_fatal\_id

```
include std/eds.e
namespace eds
public integer db_fatal_id
```

This is an *Exception handler*.

Set this to a valid routine\_id value for a procedure that will be called whenever the library detects a serious error. Your procedure will be passed a single text string that describes the error. It may also call `db_get_errors` to get more detail about the cause of the error.

## Routines

### db\_get\_errors

```
include std/eds.e
namespace eds
public function db_get_errors(integer clearing = 1)
```

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

### Arguments:

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

### Returns:

A **sequence**, each element is a set of four fields.

1. Error Code.
2. Error Text.
3. Name of library routine that recorded the error.
4. Parameters passed to that routine.

### Comments:

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

### Example 1:

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

## db\_dump

```
include std/eds.e
namespace eds
public procedure db_dump(object file_id, integer low_level_too = 0)
```

prints the current database in readable form to file fn.

### Arguments:

1. fn : the destination file for printing the current Euphoria database;
2. low\_level\_too : a boolean. If *true*, a byte-by-byte binary dump is presented as well; otherwise this step is skipped. If omitted, *false* is assumed.

### Errors:

If the current database is not defined, an error will occur.

### Comments:

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

### Example 1:

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



## check\_free\_list

```
include std/eds.e
namespace eds
public procedure check_free_list()
```

detects corruption of the free list in a Euphoria database.

### Comments:

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

## Managing Databases

## db\_connect

```
include std/eds.e
namespace eds
public function db_connect(sequence dbalias, sequence path = "", sequence dboptions = {})
```

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

### Arguments:

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

### Returns:

An **integer**, status code, either DB\_OK if creation successful or anything else on an error.

### Comments:

- This does not create or open a database. It only associates a symbolic name with a database path. This name can then be used in the calls to `db_create`, `db_open`, and `db_select` instead of the physical database name.
- If the file in the path does not have an extension, ".edb" will be added automatically.
- The `dboptions` can contain any of the options detailed below. These can be given as a single string of the form "option=value, option=value, ..." or as a sequence containing option-value pairs, { {option,value}, {option,value}, ... } *Note*: The options can be in any order.
- The options are:
  - **LOCK\_METHOD** : an integer specifying which type of access can be granted to the database. This must be one of DB\_LOCK\_NO, DB\_LOCK\_EXCLUSIVE, DB\_LOCK\_SHARDED or DB\_LOCK\_READ\_ONLY.
  - **INIT\_TABLES** : an integer giving the initial number of tables to reserve space for. The default is 5 and the minimum is 1.
  - **INIT\_FREE** : an integer giving the initial amount of free space pointers to reserve space for. The default is 5 and the minimum is 0.
- If a symbolic name has already been defined for a database, you can get it's full path and options by calling this function with `dboptions` set to `CONNECTION`. The returned value is a sequence of two elements. The first is the full path name and the second is a list of the option values. These options are indexed by `[CONNECT_LOCK]`, `[CONNECT_TABLES]`, and `[CONNECT_FREE]`.
- If a symbolic name has already been defined for a database, you remove the symbolic

name by calling this function with `dboptions` set to `DISCONNECT`.

### Example 1:

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

### Example 2:

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

### Example 3:

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

### See Also:

[db\\_create](#) | [db\\_open](#) | [db\\_select](#)

## db\_create

```
include std/eds.e
namespace eds
public function db_create(sequence path, integer lock_method = DB_LOCK_NO,
                        integer init_tables = DEF_INIT_TABLES,
                        integer init_free = DEF_INIT_FREE)
```

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

### Arguments:

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

### Returns:

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

### Comments:

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

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

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

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

### Example 1:

```

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

```

### See Also:

[db\\_open](#) | [db\\_select](#)

## db\_open

```

include std/eds.e
namespace eds
public function db_open(sequence path, integer lock_method = DB_LOCK_NO)

```

opens an existing Euphoria database.

### Arguments:

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

### Returns:

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

The return codes are:

```

public constant
    DB_OK = 0          -- success
    DB_OPEN_FAIL = -1  -- could not open the file
    DB_LOCK_FAIL = -3  -- could not lock the file in the
                      -- manner requested

```

### Comments:

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

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

### Example 1:

```

tries = 0
while 1 do
    err = db_open("mydata", DB_LOCK_SHARED)
    if err = DB_OK then
        exit
    elsif err = DB_LOCK_FAIL then
        tries += 1
        if tries > 10 then
            puts(2, "too many tries, giving up\n")
            abort(1)
        else
            sleep(5)
        end if
    end if
end while

```

```

        end if
    else
        puts(2, "Couldn't open the database!\n")
        abort(1)
    end if
end while

```

### See Also:

[db\\_create](#) | [db\\_select](#)

## db\_select

```

include std/eds.e
namespace eds
public function db_select(sequence path, integer lock_method = - 1)

```

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

### Arguments:

1. path : a sequence, the path to the database to be the new current database.
2. lock\_method : an integer. Optional locking method.

### Returns:

An **integer**, DB\_OK on success or an error code.

### Comments:

- Subsequent database operations will apply to this database. path is the path of the database file as it was originally opened with db\_open or db\_create.
- When you create (db\_create) or open (db\_open) a database, it automatically becomes the current database. Use db\_select when you want to switch back and forth between open databases, perhaps to copy records from one to the other. After selecting a new database, you should select a table within that database using db\_select\_table.
- If the lock\_method is omitted and the database has not already been opened, this function will fail. However, if lock\_method is a valid lock type for db\_open and the database is not open yet, this function will attempt to open it. It may still fail if the database cannot be opened.

### Example 1:

```

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

```

### Example 2:

```

if db_select("customer", DB_LOCK_SHARED) != DB_OK then
    puts(2, "Could not open or select Customer database\n")
end if

```

### See Also:

[db\\_open](#) | [db\\_select](#)

## db\_close

```

include std/eds.e
namespace eds

```

```
public procedure db_close()
```

unlocks and closes the current database.

### Comments:

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

## Managing Tables

### db\_select\_table

```
include std/eds.e
namespace eds
public function db_select_table(sequence name)
```

### Arguments:

1. name : a sequence which defines the name of the new current table.

On success, the table with name given by name becomes the current table.

### Returns:

An **integer**, either DB\_OK on success or DB\_OPEN\_FAIL otherwise.

### Errors:

An error occurs if the current database is not defined.

### Comments:

All record-level database operations apply automatically to the current table.

### Example 1:

```
if db_select_table("salary") != DB_OK then
    puts(2, "Couldn't find salary table!\n")
    abort(1)
end if
```

### See Also:

[db\\_table\\_list](#)

### db\_current\_table

```
include std/eds.e
namespace eds
public function db_current_table()
```

gets the name of currently selected table.

### Arguments:

1. None.

### Returns:

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

### Example 1:

```
s = db_current_table()
```

### See Also:

[db\\_select\\_table](#), [db\\_table\\_list](#)

## db\_create\_table

```
include std/eds.e
namespace eds
public function db_create_table(sequence name, integer init_records = DEF_INIT_RECORDS)
```

creates a new table within the current database.

### Arguments:

1. name : a sequence, the name of the new table.
2. init\_records : The number of records to initially reserve space for. (Default is 50)

### Returns:

An **integer**, either DB\_OK on success or DB\_EXISTS\_ALREADY on failure.

### Errors:

An error occurs if the current database is not defined.

### Comments:

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

### Example 1:

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

### See Also:

[db\\_select\\_table](#) | [db\\_table\\_list](#)

## db\_delete\_table

```
include std/eds.e
namespace eds
public procedure db_delete_table(sequence name)
```

deletes a table in the current database.

### Arguments:

1. name : a sequence, the name of the table to delete.

### Errors:

An error occurs if the current database is not defined.

### Comments:

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

### See Also:

[db\\_table\\_list](#) | [db\\_select\\_table](#) | [db\\_clear\\_table](#)

## db\_clear\_table

```
include std/eds.e
namespace eds
public procedure db_clear_table(sequence name, integer init_records = DEF_INIT_RECORDS)
```

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

### Arguments:

1. name : a sequence, the name of the table to clear.

### Errors:

An error occurs if the current database is not defined.

### Comments:

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

### See Also:

[db\\_table\\_list](#), [db\\_select\\_table](#), [db\\_delete\\_table](#)

## db\_rename\_table

```
include std/eds.e
namespace eds
public procedure db_rename_table(sequence name, sequence new_name)
```

renames a table in the current database.

### Arguments:

1. name : a sequence, the name of the table to rename
2. new\_name : a sequence, the new name for the table

### Errors:

- An error occurs if the current database is not defined.
- If name does not exist on the current database, or if new\_name does exist on the current database, an error will occur.

### Comments:

The table to be renamed can be the current table, or some other table in the current

database.

### See Also:

[db\\_table\\_list](#)

## db\_table\_list

```
include std/eds.e
namespace eds
public function db_table_list()
```

lists all tables in the current database.

### Returns:

A **sequence**, of all the table names in the current database. Each element of this sequence is a sequence, the name of a table.

### Errors:

An error occurs if the current database is undefined.

### Example 1:

```
sequence names = db_table_list()
for i = 1 to length(names) do
    puts(1, names[i] & '\n')
end for
```

### See Also:

[db\\_select\\_table](#), [db\\_create\\_table](#)

## Managing Records

## db\_find\_key

```
include std/eds.e
namespace eds
public function db_find_key(object key, object table_name = current_table_name)
```

finds the record in the current table with supplied key.

### Arguments:

1. key : the identifier of the record to be looked up.
2. table\_name : optional name of table to find key in

### Returns:

An **integer**, either greater or less than zero:

- If above zero, the record identified by key was found on the current table, and the returned integer is its record number.
- If less than zero, the record was not found. The returned integer is the opposite of what the record number would have been, had the record been found.
- If equal to zero, an error occurred.

### Errors:



If the current table is not defined, it returns 0 .

### Comments:

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

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

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

### Example 1:

```
rec_num = db_find_key("Millennium")
if rec_num > 0 then
    ? db_record_key(rec_num)
    ? db_record_data(rec_num)
else
    puts(2, "Not found, but if you insert it,\n")
    printf(2, "it will be %d\n", -rec_num)
end if
```

### See Also:

[db\\_insert](#) | [db\\_replace\\_data](#) | [db\\_delete\\_record](#) | [db\\_get\\_recid](#)

## db\_insert

```
include std/eds.e
namespace eds
public function db_insert(object key, object data, object table_name = current_table_name)
```

inserts a new record into the current table.

### Arguments:

1. **key** : an object, the record key, which uniquely identifies it inside the current table
2. **data** : an object, associated to key.
3. **table\_name** : optional table name to insert record into

### Returns:

An **integer**, either DB\_OK on success or an error code on failure.

### Comments:

Within a table, all keys must be unique. db\_insert will fail with DB\_EXISTS\_ALREADY if a record already exists on current table with the same key value.

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

### Example 1:

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

```
end if
```

### See Also:

[db\\_replace\\_data](#) | [db\\_delete\\_record](#)

## db\_delete\_record

```
include std/eds.e
namespace eds
public procedure db_delete_record(integer key_location, object table_name = current_table_name)
```

deletes record number `key_location` from the current table.

### Arguments:

1. `key_location` : a positive integer, designating the record to delete.
2. `table_name` : optional table name to delete record from.

### Errors:

If the current table is not defined, or `key_location` is not a valid record index, an error will occur. Valid record indexes are between 1 and the number of records in the table.

### Example 1:

```
db_delete_record(55)
```

### See Also:

[db\\_find\\_key](#)

## db\_replace\_data

```
include std/eds.e
namespace eds
public procedure db_replace_data(integer key_location, object data,
                                object table_name = current_table_name)
```

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

### Arguments:

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

### Comments:

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

### Example 1:

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

### See Also:

[db\\_find\\_key](#)

## db\_table\_size

```
include std/eds.e
namespace eds
public function db_table_size(object table_name = current_table_name)
```

gets the size (number of records) of the default table.

### Arguments:

1. table\_name : optional table name to get the size of.

Returns An **integer**, the current number of records in the current table. If a value less than zero is returned, it means that an error occurred.

### Errors:

If the current table is undefined, an error will occur.

### Example 1:

```
-- look at all records in the current table
for i = 1 to db_table_size() do
    if db_record_key(i) = 0 then
        puts(1, "0 key found\n")
        exit
    end if
end for
```

### See Also:

[db\\_replace\\_data](#)

## db\_record\_data

```
include std/eds.e
namespace eds
public function db_record_data(integer key_location, object table_name = current_table_name)
```

returns the data in a record queried by position.

### Arguments:

1. key\_location : the index of the record the data of which is being fetched.
2. table\_name : optional table name to get record data from.

### Returns:

An **object**, the data portion of requested record.

### Note:

This function calls fatal and returns a value of -1 if an error prevented the correct data being returned.

### Comments:

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

### Errors:

If the current table is not defined, or if the record index is invalid, an error will occur.

### Example 1:

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

### See Also:

[db\\_find\\_key](#) | [db\\_replace\\_data](#)

## db\_fetch\_record

```
include std/eds.e
namespace eds
public function db_fetch_record(object key, object table_name = current_table_name)
```

returns the data for the record with supplied key.

### Arguments:

1. key : the identifier of the record to be looked up.
2. table\_name : optional name of table to find key in

### Returns:

An **integer**,

- If less than zero, the record was not found. The returned integer is the opposite of what the record number would have been, had the record been found.
- If equal to zero, an error occurred. A sequence, the data for the record.

### Errors:

If the current table is not defined, it returns 0.

### Comments:

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

### Note:

This function does not support records that data consists of a single non-sequence value. In those cases you will need to use [db\\_find\\_key](#) and [db\\_record\\_data](#).

### Example 1:

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

### See Also:

[db\\_find\\_key](#) | [db\\_record\\_data](#)

## db\_record\_key

```
include std/eds.e
namespace eds
public function db_record_key(integer key_location, object table_name = current_table_name)
```

returns the key of a record given an index.

### Arguments:

1. `key_location` : an integer, the index of the record the key is being requested.
2. `table_name` : optional table name to get record key from.

Returns An **object**, the key of the record being queried by index.

### Note:

This function calls `fatal` and returns a value of -1 if an error prevented the correct data being returned.

### Errors:

If the current table is not defined, or if the record index is invalid, an error will occur.

### Comments:

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

### Example 1:

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

### See Also:

[`db\_record\_data`](#)

## db\_compress

```
include std/eds.e
namespace eds
public function db_compress()
```

compresses the current database.

### Returns:

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

### Comments:

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

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

### Example 1:

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

## db\_current

```
include std/eds.e
namespace eds
public function db_current()
```

gets name of currently selected database.

### Arguments:

1. None.

### Returns:

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

### Comments:

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

### Example 1:

```
s = db_current_database()
```

### See Also:

[db\\_select](#)

## db\_cache\_clear

```
include std/eds.e
namespace eds
public procedure db_cache_clear()
```

forces the database index cache to be cleared.

### Arguments:

1. None

### Comments:

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

### Example 1:

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

## db\_set\_caching

```
include std/eds.e
namespace eds
public function db_set_caching(atom new_setting)
```

sets the key cache behavior.

### Arguments:

1. integer : 0 will turn off caching, 1 will turn it back on.

### Returns:

An **integer**, the previous setting of the option.

### Comments:

Initially, the cache option is turned on. This means that when possible, the keys of a table are kept in RAM rather than read from disk each time `db_select_table` is called. For most databases, this will improve performance when you have more than one table in it.

When caching is turned off, the current cache contents is totally cleared.

### Example 1:

```
x = db_set_caching(0) -- Turn off key caching.
```

## db\_replace\_recid

```
include std/eds.e
namespace eds
public procedure db_replace_recid(integer recid, object data)
```

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

### Arguments:

1. recid : an atom, the recid of the record to be updated.
2. data : an object, the new value of the record.

### Comments:

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

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

### Example 1:

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

### See Also:

`db_replace_data`, `db_find_key`, `db_get_recid`

## db\_record\_recid

```
include std/eds.e
namespace eds
public function db_record_recid(integer recid)
```

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

### Arguments:

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

### Returns:

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

### Comments:

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

### Example 1:

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

### See Also:

[db\\_get\\_recid](#) | [db\\_replace\\_recid](#)

## db\_get\_recid

```
include std/eds.e
namespace eds
public function db_get_recid(object key, object table_name = current_table_name)
```

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

### Arguments:

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

### Returns:

An **atom**, either greater or equal to zero:

- If above zero, it is a `recid`.
- If less than zero, the record wasn't found.
- If equal to zero, an error occurred.

### Errors:

If the table is not defined, an error is raised.

### Comments:

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

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

### Example 1:



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

### See Also:

[db\\_insert](#) | [db\\_replace\\_data](#) | [db\\_delete\\_record](#) | [db\\_find\\_key](#)

# Prime Numbers

## Subroutines

### calc\_primes

```
include std/primes.e
namespace primes
public function calc_primes(integer approx_limit, atom time_limit_p = 10)
```

returns all the prime numbers below a threshold, with a cap on computation time.

#### Arguments:

1. `approx_limit` : an integer, This is not the upper limit but the last prime returned is the *next* prime after or on this value.
2. `time_out_p` : an atom, the maximum number of seconds that this function can run for. The default is 10 (ten) seconds.

#### Returns:

A **sequence**, made of prime numbers in increasing order. The last value is the next prime number that falls on or *after* the value of `approx_limit`.

#### Comments:

- The `approx_limit` argument *does not* represent the largest value to return. The largest value returned will be the next prime number on or after
- The returned sequence contains all the prime numbers less than its last element.
- If the function times out, it may not hold all primes below `approx_limit`, but only the largest ones will be absent. If the last element returned is less than `approx_limit` then the function timed out.
- To disable the timeout, simply give it a negative value.

#### Example 1:

```
? calc_primes(1000, 5)
-- On a very slow computer, you may only get all primes up to say 719.
-- On a faster computer, the last element printed out will be 1009.
-- This call will never take longer than 5 seconds.
```

#### See Also:

[next\\_prime](#) | [prime\\_list](#)

### next\_prime

```
include std/primes.e
namespace primes
public function next_prime(integer n, object fail_signal_p = - 1, atom time_out_p = 1)
```

returns the next prime number on or after the supplied number.

#### Arguments:

1. `n` : an integer, the starting point for the search

2. `fail_signal_p` : an integer, used to signal error. Defaults to -1.

### Returns:

An **integer**, which is prime only if it took less than one second to determine the next prime greater or equal to `n`.

### Comments:

The default value of -1 will alert you about an invalid returned value, since a prime not less than `n` is expected. However, you can pass another value for this parameter.

### Example 1:

```
? next_prime(997)
-- On a very slow computer, you might get -997, but 1009 is expected.
```

### See Also:

[calc\\_primes](#)

## prime\_list

```
include std/primes.e
namespace primes
public function prime_list(integer top_prime_p = 0)
```

returns a list of prime numbers.

### Arguments:

1. `top_prime_p` : The list will end with the prime less than or equal to this value. If `top_prime_p` is zero, the current list of calculated primes is returned.

### Returns:

An **sequence**, a list of prime numbers from 2 to  $\leq$  `top_prime_p`

### Example 1:

```
sequence pList = prime_list(1000)
-- pList will now contain all the primes from 2 up to the largest less than or
-- equal to 1000, which is 997.
```

### See Also:

[calc\\_primes](#) | [next\\_prime](#)

# Flags

## Subroutines

### which\_bit

```
include std/flags.e
namespace flags
public function which_bit(object theValue)
```

tests if the supplied value has only a single bit on in its representation.

#### Arguments:

1. theValue : an object to test.

#### Returns:

An **integer**, either 0 if it contains multiple bits, zero bits or is an invalid value, otherwise the bit number set. The right-most bit is position 1 and the leftmost bit is position 32.

#### Example 1:

```
? which_bit(2) --> 2
? which_bit(0) --> 0
? which_bit(3) --> 0
? which_bit(4)      --> 3
? which_bit(17)     --> 0
? which_bit(1.7)    --> 0
? which_bit(-2)     --> 0
? which_bit("one")  --> 0
? which_bit(0x80000000) --> 32
```

### flags\_to\_string

```
include std/flags.e
namespace flags
public function flags_to_string(object flag_bits, sequence flag_names,
    integer expand_flags = 0)
```

returns a list of strings that represent the human-readable identities of the supplied flag or flags.

#### Arguments:

1. flag\_bits : Either a single 32-bit set of flags (a flag value), or a list of such flag values. The function returns the names for these flag values.
2. flag\_names : A sequence of two-element sub-sequences. Each sub-sequence contains {FlagValue, FlagName}, where *FlagName* is a string and *FlagValue* is the set of bits that set the flag on.
3. expand\_flags: An integer. 0 (the default) means that the flag values in flag\_bits are not broken down to their single-bit values. For example: #0c returns the name of #0c and not the names for #08 and #04. When expand\_flags is non-zero then each bit in the flag\_bits parameter is scanned for a matching name.

#### Returns:

A sequence. This contains the name or names for each supplied flag value or values.

## Comments:

- The number of strings in the returned value depends on `expand_flags` is non-zero and whether `flags_bits` is an atom or sequence.
- When `flag_bits` is an atom, you get returned a sequence of strings, one for each matching name (according to `expand_flags` option).
- When `flag_bits` is a sequence, it is assumed to represent a list of atomic flags. That is, `{#1, #4}` is a set of two flags for which you want their names. In this case, you get returned a sequence that contains one sequence for each element in `flag_bits`, which in turn contain the matching name or names.
- When a flag's name can not be found in `flag_names`, this function returns the *name* of "?".

## Example 1:

```
include std/console.e
sequence s
s = {
  {#00000000, "WS_OVERLAPPED"},
  {#80000000, "WS_POPUP"},
  {#40000000, "WS_CHILD"},
  {#20000000, "WS_MINIMIZE"},
  {#10000000, "WS_VISIBLE"},
  {#08000000, "WS_DISABLED"},
  {#44000000, "WS_CLIPPINGCHILD"},
  {#04000000, "WS_CLIPSIBLINGS"},
  {#02000000, "WS_CLIPCHILDREN"},
  {#01000000, "WS_MAXIMIZE"},
  {#00C00000, "WS_CAPTION"},
  {#00800000, "WS_BORDER"},
  {#00400000, "WS_DLGFAME"},
  {#00100000, "WS_HSCROLL"},
  {#00200000, "WS_VSCROLL"},
  {#00080000, "WS_SYSMENU"},
  {#00040000, "WS_THICKFRAME"},
  {#00020000, "WS_MINIMIZEBOX"},
  {#00010000, "WS_MAXIMIZEBOX"},
  {#00300000, "WS_SCROLLBARS"},
  {#00CF0000, "WS_OVERLAPPEDWINDOW"},
  $
}
display( flags_to_string( {#0C20000,2,9,0}, s,1))
--> {
-->   "WS_BORDER",
-->   "WS_DLGFAME",
-->   "WS_MINIMIZEBOX"
--> },
--> {
-->   "?"
--> },
--> {
-->   "?"
--> },
--> {
-->   "WS_OVERLAPPED"
--> }
--> }
display( flags_to_string( #80000000, s))
--> {
-->   "WS_POPUP"
--> }
display( flags_to_string( #00C00000, s))
--> {
-->   "WS_CAPTION"
--> }
display( flags_to_string( #44000000, s))
--> {
-->   "WS_CLIPPINGCHILD"
--> }
display( flags_to_string( #44000000, s, 1))
```

```
--> {  
-->   "WS_CHILD",  
-->   "WS_CLIPSIBLINGS"  
--> }  
display( flags_to_string( #00000000, s))  
--> {  
-->   "WS_OVERLAPPED"  
--> }  
display( flags_to_string( #00CF0000, s))  
--> {  
-->   "WS_OVERLAPPEDWINDOW"  
--> }  
display( flags_to_string( #00CF0000, s, 1))  
--> {  
-->   "WS_BORDER",  
-->   "WS_DLGFAME",  
-->   "WS_SYSMENU",  
-->   "WS_THICKFRAME",  
-->   "WS_MINIMIZEBOX",  
-->   "WS_MAXIMIZEBOX"  
--> }
```

# Hashing Algorithms

## Type Constants

### enum

```
include std/hash.e
namespace stdhash
public enum
```

HSIEH30 | HSIEH32 | ADLER32 | FLETCHER32 | MD5 | SHA256

## Subroutines

### hash

```
<built-in> function hash(object source, atom algo)
```

calculates a hash value for a *key* using the algorithm *algo*.

### Arguments:

1. *source* : Any Euphoria object
2. *algo* : A code indicating which algorithm to use.
  - HSIEH30 uses Hsieh. Returns a 30-bit (a Euphoria integer). Fast and good dispersion
  - HSIEH32 uses Hsieh. Returns a 32-bit value. Fast and very good dispersion
  - ADLER32 uses Adler. Very fast and reasonable dispersion, especially for small strings
  - FLETCHER32 uses Fletcher. Very fast and good dispersion
  - MD5 uses MD5 (not implemented yet) Slower but very good dispersion. Suitable for signatures.
  - SHA256 uses SHA256 (not implemented yet) Slow but excellent dispersion. Suitable for signatures. More secure than MD5.
  - 0 and above (integers and decimals) and non-integers less than zero use the cyclic variant ( $\text{hash} = \text{hash} * \text{algo} + c$ ). This is a fast and good to excellent dispersion depending on the value of *algo*. Decimals give better dispersion but are slightly slower.

### Returns:

An **object**, could be integer, atom, or sequence depending on the algorithm selected.

# Map (Hash Table)

A **map** is a special array, often called an associative array or dictionary; in a map the data **values** (any Euphoria object) are indexed by **keys** (also any Euphoria object).

When programming think in terms of *key:value* pairs. For example we can code things like this:

```
custrec = new() -- Create a new map
put(custrec, "Name", "Joe Blow")
put(custrec, "Address", "555 High Street")
put(custrec, "Phone", 555675632)
```

This creates three elements in the map, and they are indexed by "Name", "Address" and "Phone", meaning that to get the data associated with those keys we can code:

```
object data = get(custrec, "Phone")
-- data now set to 555675632
```

Note that *only one instance of a given key* can exist in a given map, meaning for example, we could not have two separate "Name" values in the above custrec map.

Maps automatically grow to accommodate all the elements placed into it.

Associative arrays can be implemented in many different ways, depending on what efficiency trade-offs have been made. This implementation allows you to specify how many items you expect the map to hold, or simply start with the default size.

As the number of items in the map grows, the map may increase its size to accommodate larger numbers of items.

## Operation Codes for Put

### enum

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

## Types

### map

```
include std/map.e
namespace map
public type map(object m)
```

defines the datatype 'map'.

### Comments:

Used when declaring a map variable.

### Example 1:

```
map SymbolTable = new() -- Create a new map to hold the symbol table.
```



## Subroutines

### calc\_hash

```
include std/map.e
namespace map
public function calc_hash(object key_p, integer max_hash_p)
```

calculates a Hashing value from the supplied data.

#### Arguments:

1. key\_p : The data for which you want a hash value calculated.
2. max\_hash\_p : The returned value will be no larger than this value.

#### Returns:

An **integer**, the value of which depends only on the supplied data.

#### Comments:

This is used whenever you need a single number to represent the data you supply. It can calculate the number based on all the data you give it, which can be an atom or sequence of any value.

#### Example 1:

```
integer h1
-- calculate a hash value and ensure it will be a value from 1 to 4097.
h1 = calc_hash( symbol_name, 4097 )
```

### threshold

```
include std/map.e
namespace map
public function threshold(integer new_value_p = 0)
```

deprecated.

#### Arguments:

1. new\_value\_p : unused value.

#### Returns:

Zero..

### type\_of

```
include std/map.e
namespace map
public function type_of(map the_map_p)
```

deprecated

#### Arguments:

1. m : A map

## Returns:

Zero.

## rehash

```
include std/map.e
namespace map
public procedure rehash(map the_map_p, integer requested_size_p = 0)
```

changes the width (that is the number of buckets) of a map.

## Arguments:

1. `m` : the map to resize
2. `requested_size_p` : a lower limit for the new size.

## Comments:

If `requested_size_p` is not greater than zero, a new width is automatically derived from the current one.

## See Also:

[statistics](#) | [optimize](#)

## new

```
include std/map.e
namespace map
public function new(integer initial_size_p = DEFAULT_SIZE)
```

creates a new map data structure.

## Arguments:

1. `initial_size_p` : An estimate of how many initial elements will be stored in the map.

## Returns:

An empty **map**.

## Comments:

A new object of type `map` is created. The resources allocated for the map will be automatically cleaned up if the reference count of the returned value drops to zero, or if passed in a call to [delete](#).

## Example 1:

```
map m = new() -- m is now an empty map
x = new()     -- the resources for the map previously stored in x are released automatically
delete( m )   -- the resources for the map are released
```

## new\_extra

```
include std/map.e
namespace map
```

```
public function new_extra(object the_map_p, integer initial_size_p = 8)
```

returns either the supplied map or a new map.

### Arguments:

1. `the_map_p` : An object, that could be an existing map
2. `initial_size_p` : An estimate of how many initial elements will be stored in a new map.

### Returns:

A **map**, If `m` is an existing map then it is returned otherwise this returns a new empty **map**.

### Comments:

This is used to return a new map if the supplied variable isn't already a map.

### Example 1:

```
map m = new_extra( foo() ) -- If foo() returns a map it is used, otherwise
                           -- a new map is created.
```

## compare

```
include std/map.e
namespace map
public function compare(map map_1_p, map map_2_p, integer scope_p = 'd')
```

compares two maps to test equality.

### Arguments:

1. `map_1_p` : A map
2. `map_2_p` : A map
3. `scope_p` : An integer that specifies what to compare.
  - 'k' or 'K' to only compare keys.
  - 'v' or 'V' to only compare values.
  - 'd' or 'D' to compare both keys and values. This is the default.

### Returns:

An **integer**,

- -1 if they are not equal.
- 0 if they are literally the same map.
- 1 if they contain the same keys and/or values.

### Example 1:

```
map map_1_p = foo()
map map_2_p = bar()
if compare(map_1_p, map_2_p, 'k') >= 0 then
    ... -- two maps have the same keys
```

## has

```
include std/map.e
namespace map
public function has(map the_map_p, object key)
```

checks whether map has a given key.

### Arguments:

1. `the_map_p` : the map to inspect
2. `the_key_p` : an object to be looked up

### Returns:

An **integer**, 0 if not present, 1 if present.

### Example 1:

```
map the_map_p
the_map_p = new()
put(the_map_p, "name", "John")
? has(the_map_p, "name") -- 1
? has(the_map_p, "age") -- 0
```

### See Also:

[get](#)

## get

```
include std/map.e
namespace map
public function get(map the_map_p, object key, object default = 0)
```

retrieves the value associated to a key in a map.

### Arguments:

1. `the_map_p` : the map to inspect
2. `the_key_p` : an object, the `the_key_p` being looked tp
3. `default_value_p` : an object, a default value returned if `the_key_p` not found. The default is 0.

### Returns:

An **object**, the value that corresponds to `the_key_p` in `the_map_p`. If `the_key_p` is not in `the_map_p`, `default_value_p` is returned instead.

### Example 1:

```
map ages
ages = new()
put(ages, "Andy", 12)
put(ages, "Budi", 13)

integer age
age = get(ages, "Budi", -1)
if age = -1 then
  puts(1, "Age unknown")
else
  printf(1, "The age is %d", age)
end if
```

### See Also:

[has](#)

## nested\_get

```
include std/map.e
namespace map
public function nested_get(map the_map_p, sequence the_keys_p, object default_value_p = 0)
```

returns the value given a nested key.

### Comments:

Returns the value that corresponds to the object `the_keys_p` in the nested map `the_map_p`. `the_keys_p` is a sequence of keys. If any key is not in the map, the object `default_value_p` is returned instead.

## put

```
include std/map.e
namespace map
public procedure put(map the_map_p, object key, object val, object op = PUT,
    object deprecated = 0)
```

adds or updates an entry on a map.

### Arguments:

1. `the_map_p` : the map where an entry is being added or updated
2. `the_key_p` : an object, the `the_key_p` to look up
3. `the_value_p` : an object, the value to add, or to use for updating.
4. `operation` : an integer, indicating what is to be done with `the_value_p`. Defaults to `PUT`.
5. `trigger_p` : Deprecated. This parameter defaults to zero and is not used.

### Comments:

- The operation parameter can be used to modify the existing value. Valid operations are:
  - `PUT` -- This is the default, and it replaces any value in there already
  - `ADD` -- Equivalent to using the `+=` operator
  - `SUBTRACT` -- Equivalent to using the `-=` operator
  - `MULTIPLY` -- Equivalent to using the `*=` operator
  - `DIVIDE` -- Equivalent to using the `/=` operator
  - `APPEND` -- Appends the value to the existing data
  - `CONCAT` -- Equivalent to using the `&=` operator
  - `LEAVE` -- If it already exists, the current value is left unchanged otherwise the new value is added to the map.

### Example 1:

```
map ages
ages = new()
put(ages, "Andy", 12)
put(ages, "Budi", 13)
put(ages, "Budi", 14)

-- ages now contains 2 entries: "Andy" => 12, "Budi" => 14
```

### See Also:

[remove](#) | [has](#) | [nested\\_put](#)

## nested\_put

```
include std/map.e
namespace map
public procedure nested_put(map the_map_p, sequence the_keys_p, object the_value_p,
    integer operation_p = PUT, object deprecated_trigger_p = 0)
```

adds or updates an entry on a map.

### Arguments:

1. the\_map\_p : the map where an entry is being added or updated
2. the\_keys\_p : a sequence of keys for the nested maps
3. the\_value\_p : an object, the value to add, or to use for updating.
4. operation\_p : an integer, indicating what is to be done with value. Defaults to PUT.
5. deprecated\_trigger\_p : Deprecated. This parameter defaults to zero and is not used.

### Comments:

Valid operations are:

- PUT -- This is the default, and it replaces any value in there already
- ADD -- Equivalent to using the += operator
- SUBTRACT -- Equivalent to using the -= operator
- MULTIPLY -- Equivalent to using the \*= operator
- DIVIDE -- Equivalent to using the /= operator
- APPEND -- Appends the value to the existing data
- CONCAT -- Equivalent to using the &= operator
- If existing entry with the same key is already in the map, the value of the entry is updated.

### Example 1:

```
map city_population
city_population = new()
nested_put(city_population, {"United States", "California", "Los Angeles"},
    3819951 )
nested_put(city_population, {"Canada", "Ontario", "Toronto"},
    2503281 )
```

### See Also:

[put](#)

## remove

```
include std/map.e
namespace map
public procedure remove(map the_map_p, object key)
```

removes an entry with given key from a map.

### Arguments:

1. the\_map\_p : the map to operate on
2. key : an object, the key to remove.

### Comments:

- If key is not on the\_map\_p, the the\_map\_p is returned unchanged.
- If you need to remove all entries, see [clear](#)

### Example 1:

```
map the_map_p
```

```
the_map_p = new()
put(the_map_p, "Amy", 66.9)
remove(the_map_p, "Amy")
-- the_map_p is now an empty map again
```

## See Also:

[clear](#) | [has](#)

## clear

```
include std/map.e
namespace map
public procedure clear(map the_map_p)
```

removes all entries in a map.

## Arguments:

1. `the_map_p` : the map to operate on

## Comments:

- This is much faster than removing each entry individually.
- If you need to remove just one entry, see [remove](#)

## Example 1:

```
map the_map_p
the_map_p = new()
put(the_map_p, "Amy", 66.9)
put(the_map_p, "Betty", 67.8)
put(the_map_p, "Claire", 64.1)
...
clear(the_map_p)
-- the_map_p is now an empty map again
```

## See Also:

[remove](#) | [has](#)

## size

```
include std/map.e
namespace map
public function size(map the_map_p)
```

returns the number of entries in a map.

## Arguments:

`the_map_p` : the map being queried

## Returns:

An **integer**, the number of entries it has.

## Comments:

For an empty map, size will be zero

### Example 1:

```
map the_map_p
put(the_map_p, 1, "a")
put(the_map_p, 2, "b")
? size(the_map_p) -- outputs 2
```

### See Also:

[statistics](#)

## enum

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

## statistics

```
include std/map.e
namespace map
public function statistics(map the_map_p)
```

retrieves characteristics of a map.

### Arguments:

1. `the_map_p`: the map being queried

### Returns:

A **sequence**, of 7 integers:

- `NUM_ENTRIES` -- number of entries
- `NUM_IN_USE` -- number of buckets in use
- `NUM_BUCKETS` -- number of buckets
- `LARGEST_BUCKET` -- size of largest bucket
- `SMALLEST_BUCKET` -- size of smallest bucket
- `AVERAGE_BUCKET` -- average size for a bucket
- `STDEV_BUCKET` -- standard deviation for the bucket length series

### Example 1:

```
sequence s = statistics(mymap)
printf(1, "The average size of the buckets is %d", s[AVERAGE_BUCKET])
```

## keys

```
include std/map.e
namespace map
public function keys(map the_map_p, integer sorted_result = 0)
```

returns all keys in a map.

### Arguments:

1. `the_map_p`: the map being queried
2. `sorted_result`: optional integer. 0 [default] means do not sort the output and 1 means to



sort the output before returning.

### Returns:

A **sequence** made of all the keys in the map.

### Comments:

If `sorted_result` is not used, the order of the keys returned is not predictable.

### Example 1:

```
map the_map_p
the_map_p = new()
put(the_map_p, 10, "ten")
put(the_map_p, 20, "twenty")
put(the_map_p, 30, "thirty")
put(the_map_p, 40, "forty")

sequence keys
keys = keys(the_map_p) -- keys might be {20,40,10,30} or some other order
keys = keys(the_map_p, 1) -- keys will be {10,20,30,40}
```

### See Also:

[has](#) | [values](#) | [pairs](#)

## values

```
include std/map.e
namespace map
public function values(map the_map, object keys = 0, object default_values = 0)
```

returns values, without their keys, from a map.

### Arguments:

1. `the_map` : the map being queried
2. `keys` : optional, key list of values to return.
3. `default_values` : optional default values for keys list

### Returns:

A **sequence**, of all values stored in `the_map`.

### Comments:

- The order of the values returned may not be the same as the putting order.
- Duplicate values are not removed.
- You use the `keys` parameter to return a specific set of values from the map. They are returned in the same order as the `keys` parameter. If no `default_values` is given and one is needed, 0 will be used.
- If `default_values` is an atom, it represents the default value for all values in keys.
- If `default_values` is a sequence, and its length is less than keys, then the last item in `default_values` is used for the rest of the keys.

### Example 1:

```
map the_map_p
the_map_p = new()
put(the_map_p, 10, "ten")
put(the_map_p, 20, "twenty")
put(the_map_p, 30, "thirty")
put(the_map_p, 40, "forty")
```

```
sequence values
values = values(the_map_p)
-- values might be {"twenty","forty","ten","thirty"}
-- or some other order
```

### Example 2:

```
map the_map_p
the_map_p = new()
put(the_map_p, 10, "ten")
put(the_map_p, 20, "twenty")
put(the_map_p, 30, "thirty")
put(the_map_p, 40, "forty")

sequence values
values = values(the_map_p, { 10, 50, 30, 9000 })
-- values WILL be { "ten", 0, "thirty", 0 }
values = values(the_map_p, { 10, 50, 30, 9000 }, {-1,-2,-3,-4})
-- values WILL be { "ten", -2, "thirty", -4 }
```

### See Also:

[get](#) | [keys](#) | [pairs](#)

## pairs

```
include std/map.e
namespace map
public function pairs(map the_map, integer sorted_result = 0)
```

returns all key:value pairs in a map.

### Arguments:

1. the\_map\_p : the map to get the data from
2. sorted\_result : optional integer. 0 [default] means do not sort the output and 1 means to sort the output before returning.

### Returns:

A **sequence**, of all key:value pairs stored in the\_map\_p. Each pair is a sub-sequence in the form {key, value}

### Comments:

If sorted\_result is not used, the order of the values returned is not predicable.

### Example 1:

```
map the_map_p

the_map_p = new()
put(the_map_p, 10, "ten")
put(the_map_p, 20, "twenty")
put(the_map_p, 30, "thirty")
put(the_map_p, 40, "forty")

sequence keyvals
keyvals = pairs(the_map_p)
-- might be {{20,"twenty"},{40,"forty"},{10,"ten"},{30,"thirty"}}

keyvals = pairs(the_map_p, 1)
-- will be {{10,"ten"},{20,"twenty"},{30,"thirty"},{40,"forty"}}
```

## See Also:

[get](#) | [keys](#) | [values](#)

## optimize

```
include std/map.e
namespace map
public procedure optimize(map the_map_p, integer deprecated_max_p = 0,
    atom deprecated_grow_p = 0)
```

rehashes a map to increase performance. This procedure is deprecated in favor of [rehash](#).

### Arguments:

1. the\_map\_p : the map being optimized
2. deprecated\_max\_p : unused
3. deprecated\_grow\_p : unused.

### Comments:

This rehashes the map until either the maximum bucket size is less than the desired maximum or the maximum bucket size is less than the largest size statistically expected (mean + 3 standard deviations).

## See Also:

[statistics](#) | [rehash](#)

## load\_map

```
include std/map.e
namespace map
public function load_map(object input_file_name)
```

loads a map from a file.

### Arguments:

1. file\_name\_p : The file to load from. This file may have been created by the [save\\_map](#) function. This can either be a name of a file or an already opened file handle.

### Returns:

Either a **map**, with all the entries found in file\_name\_p, or **-1** if the file failed to open, or **-2** if the file is incorrectly formatted.

### Comments:

If file\_name\_p is an already opened file handle, this routine will read from that file and not close it. Otherwise, the named file will be opened and closed by this routine.

The input file can be either one created by the [save\\_map](#) function or a manually created or edited text file. See [save\\_map](#) for details about the required layout of the text file.

### Example 1:

```
include std/error.e

object loaded
```

```

map AppOptions
sequence SavedMap = "c:\\myapp\\options.txt"

loaded = load_map(SavedMap)
if equal(loaded, -1) then
    crash("Map '%s' failed to open", SavedMap)
end if

-- By now we know that it was loaded and a new map created,
-- so we can assign it to a 'map' variable.
AppOptions = loaded
if get(AppOptions, "verbose", 1) = 3 then
    ShowInstructions()
end if

```

### See Also:

[new](#) | [save\\_map](#)

## enum

```

include std/map.e
namespace map
public enum

```

## save\_map

```

include std/map.e
namespace map
public function save_map(map the_map_, object file_name_p, integer type_ = SM_TEXT)

```

saves a map to a file.

### Arguments:

1. *m* : a map.
2. *file\_name\_p* : Either a sequence, the name of the file to save to, or an open file handle as returned by [open\(\)](#).
3. *type* : an integer. SM\_TEXT for a human-readable format (default), SM\_RAW for a smaller and faster format, but not human-readable.

### Returns:

An **integer**, the number of keys saved to the file, or -1 if the save failed.

### Comments:

If *file\_name\_p* is an already opened file handle, this routine will write to that file and not close it. Otherwise, the named file will be created and closed by this routine.

The SM\_TEXT type saves the map keys and values in a text format which can be read and edited by standard text editor. Each entry in the map is saved as a KEY/VALUE pair in the form

```
key = value
```

Note that if the 'key' value is a normal string value, it can be enclosed in double quotes. If it is not thus quoted, the first character of the key determines its Euphoria value type. A dash or digit implies an atom, an left-brace implies a sequence, an alphabetic character implies a text string that extends to the next equal '=' symbol, and anything else is ignored.

Note that if a line contains a double-dash, then all text from the double-dash to the end of the line will be ignored. This is so you can optionally add comments to the saved map. Also, any blank lines are ignored too.

All text after the '=' symbol is assumed to be the map item's value data.

Because some map data can be rather long, it is possible to split the text into multiple lines, which will be considered by `load_map` as a single *logical* line. If an line ends with a comma (,) or a dollar sign (\$), then the next actual line is appended to the end of it. After all these physical lines have been joined into one logical line, all combinations of `", \$" and `, \$` are removed.

For example:

```
one = {"first",
"second",
"third",
$
}
second = "A long text ",$
"line that has been", $
" split into three lines"
third = {"first",
"second",
"third"}
```

is equivalent to

```
one = {"first","second","third"}
second = "A long text line that has been split into three lines"
third = {"first","second","third"}
```

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

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

### Example 1:

```
include std/error.e

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

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

### See Also:

`load_map`

### copy

```
include std/map.e
namespace map
public function copy(map source_map, object dest_map = 0, integer put_operation = PUT)
```

duplicates a map.

### Arguments:

1. `source_map` : map to copy from
2. `dest_map` : optional, map to copy to
3. `put_operation` : optional, operation to use when `dest_map` is used. The default is PUT.

### Returns:

If `dest_map` was not provided, an exact duplicate of `source_map` otherwise `dest_map`, which does not have to be empty, is returned with the new values copied from `source_map`, according to the `put_operation` value.

### Example 1:

```
map m1 = new()
put(m1, 1, "one")
put(m1, 2, "two")

map m2 = copy(m1)
printf(1, "%s, %s\n", { get(m2, 1), get(m2, 2) })
-- one, two

put(m1, 1, "one hundred")
printf(1, "%s, %s\n", { get(m1, 1), get(m1, 2) })
-- one hundred, two

printf(1, "%s, %s\n", { get(m2, 1), get(m2, 2) })
-- one, two
```

### Example 2:

```
map m1 = new()
map m2 = new()

put(m1, 1, "one")
put(m1, 2, "two")
put(m2, 3, "three")

copy(m1, m2)

? keys(m2)
-- { 1, 2, 3 }
```

### Example 3:

```
map m1 = new()
map m2 = new()

put(m1, "XY", 1)
put(m1, "AB", 2)
put(m2, "XY", 3)

pairs(m1) --> { {"AB", 2}, {"XY", 1} }
pairs(m2) --> { {"XY", 3} }

-- Add same keys' values.
copy(m1, m2, ADD)

pairs(m2) --> { {"AB", 2}, {"XY", 4} }
```

### See Also:

[put](#)

## new\_from\_kvpairs

```
include std/map.e
namespace map
public function new_from_kvpairs(sequence kv_pairs)
```

converts a set of key:value pairs to a map.

### Arguments:

1. kv\_pairs : A sequence containing any number of subsequences that have the format {KEY, VALUE}. These are loaded into a new map which is then returned by this function.

### Returns:

A **map**, containing the data from kv\_pairs

### Example 1:

```
map m1 = new_from_kvpairs( {
    { "application", "Euphoria" },
    { "version", "4.0" },
    { "genre", "programming language" },
    { "crc", 0x4F71AE10 }
})

v = map:get(m1, "application") --> "Euphoria"
```

## new\_from\_string

```
include std/map.e
namespace map
public function new_from_string(sequence kv_string)
```

converts a set of key:value pairs contained in a string to a map.

### Arguments:

1. kv\_string : A string containing any number of lines that have the format KEY=VALUE. These are loaded into a new map which is then returned by this function.

### Returns:

A **map**, containing the data from kv\_string

### Comments:

This function actually calls **keyvalues** to convert the string to key-value pairs, which are then used to create the map.

### Example 1:

Given that a file called "xyz.config" contains the lines ...

```
application = Euphoria,
version      = 4.0,
genre       = "programming language",
crc         = 4F71AE10
```

```
map m1 = new_from_string( read_file("xyz.config", TEXT_MODE))

printf(1, "%s\n", {map:get(m1, "application")}) --> "Euphoria"
printf(1, "%s\n", {map:get(m1, "genre")})      --> "programming language"
```

```
printf(1, "%s\n", {map:get(m1, "version")})    --> "4.0"
printf(1, "%s\n", {map:get(m1, "crc")})        --> "4F71AE10"
```

## for\_each

```
include std/map.e
namespace map
public function for_each(map source_map, integer user_rid, object user_data = 0,
    integer in_sorted_order = 0, integer signal_boundary = 0)
```

calls a user-defined routine for each of the items in a map.

### Arguments:

1. source\_map : The map containing the data to process
2. user\_rid: The routine\_id of a user defined processing function
3. user\_data: An object. Optional. This is passed, unchanged to each call of the user defined routine. By default, zero (0) is used.
4. in\_sorted\_order: An integer. Optional. If non-zero the items in the map are processed in ascending key sequence otherwise the order is undefined. By default they are not sorted.
5. signal\_boundary: A integer; 0 (the default) means that the user routine is not called if the map is empty and when the last item is passed to the user routine, the Progress Code is not negative.

### Returns:

An integer: 0 means that all the items were processed, and anything else is whatever was returned by the user routine to abort the for\_each process.

### Comments:

- The user defined routine is a function that must accept four parameters.
  1. Object: an Item Key
  2. Object: an Item Value
  3. Object: The user\_data value. This is never used by for\_each itself, merely passed to the user routine.
  4. Integer: Progress code.
    - The abs value of the progress code is the ordinal call number. That is 1 means the first call, 2 means the second call, etc ...
    - If the progress code is negative, it is also the last call to the routine.
    - If the progress code is zero, it means that the map is empty and thus the item key and value cannot be used.
    - **note** that if signal\_boundary is zero, the Progress Code is never less than 1.
- The user routine must return 0 to get the next map item. Anything else will cause for\_each to stop running, and is returned to whatever called for\_each.
- Note that any changes that the user routine makes to the map do not affect the order or number of times the routine is called. for\_each takes a copy of the map keys and data before the first call to the user routine and uses the copied data to call the user routine.

### Example 1:

```
include std/map.e
include std/math.e
include std/io.e

function Process_A(object k, object v, object d, integer pc)
```



```

        writeln("[ ] = [ ]", {k, v})
        return 0
    end function

function Process_B(object k, object v, object d, integer pc)
    if pc = 0 then
        writeln("The map is empty")
    else
        integer c
        c = abs(pc)
        if c = 1 then
            writeln("---[ ]---", {d}) -- Write the report title.
        end if
        writeln("[ ]: [ :15] = [ ]", {c, k, v})
        if pc < 0 then
            writeln(repeat('-', length(d) + 6), { }) -- Write the report end.
        end if
    end if
    return 0
end function

map m1 = new()
map:put(m1, "application", "Euphoria")
map:put(m1, "version", "4.0")
map:put(m1, "genre", "programming language")
map:put(m1, "crc", "4F71AE10")

-- Unsorted
map:for_each(m1, routine_id("Process_A"))
-- Sorted
map:for_each(m1, routine_id("Process_B"), "List of Items", 1)

```

The output from the first call could be...

```

application = Euphoria
version = 4.0
genre = programming language
crc = 4F71AE10

```

The output from the second call should be...

```

---List of Items---
1: application      = Euphoria
2: crc              = 4F71AE10
3: genre            = programming language
4: version          = 4.0
-----

```

# Stack

## Constants

### Stack types

#### FIFO

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

FIFO: like people standing in line: first item in is first item out

#### FILO

```
include std/stack.e
namespace stack
public constant FILO
```

FILO: like for a stack of plates : first item in is last item out

## Types

#### stack

```
include std/stack.e
namespace stack
public type stack(object obj_p)
```

A stack is a sequence of objects with some internal data.

## Subroutines

#### new

```
include std/stack.e
namespace stack
public function new(integer typ = FILO)
```

creates a new stack.

#### Arguments:

1. `stack_type` : an integer, defining the semantics of the stack. The default is FILO.

#### Returns:

An empty **stack**, note that the variable storing the stack must not be an integer. The resources allocated for the stack will be automatically cleaned up if the reference count of the returned value drops to zero, or if passed in a call to `delete`.

## Comments:

There are two sorts of stacks, designated by the types FIFO and FILO:

- A FIFO stack is one where the first item to be pushed is popped first. People standing in queue form a FIFO stack.
- A FILO stack is one where the item pushed last is popped first. A column of coins is of the FILO kind.

## See Also:

[is\\_empty](#)

### is\_empty

```
include std/stack.e
namespace stack
public function is_empty(stack sk)
```

determines whether a stack is empty.

## Arguments:

1. sk : the stack being queried.

## Returns:

An **integer**, 1 if the stack is empty, else 0.

## See Also:

[size](#)

### size

```
include std/stack.e
namespace stack
public function size(stack sk)
```

returns how many elements a stack has.

## Arguments:

1. sk : the stack being queried.

## Returns:

An **integer**, the number of elements in sk.

### at

```
include std/stack.e
namespace stack
public function at(stack sk, integer idx = 1)
```

fetches a value from the stack without removing it from the stack.

## Arguments:

1. `sk` : the stack being queried
2. `idx` : an integer, the place to inspect. The default is 1 (top item).

### Returns:

An **object**, the `idx`-th item of the stack.

### Errors:

If the supplied value of `idx` does not correspond to an existing element, an error occurs.

### Comments:

- For FIFO stacks (queues), the top item is the oldest item in the stack.
- For FILO stacks, the top item is the newest item in the stack.

`idx` can be less than 1, in which case it refers relative to the end item. Thus, 0 stands for the end element.

### Example 1:

```
stack sk = new(FILO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)

at(sk, 0)  --> 5
at(sk, -1) --> "abc"
at(sk, 1)  --> 2.3
at(sk, 2)  --> "abc"
```

### Example 2:

```
stack sk = new(FIFO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)
at(sk, 0)  --> 2.3
at(sk, -1) --> "abc"
at(sk, 1)  --> 5
at(sk, 2)  --> "abc"
```

### See Also:

[size](#) | [top](#) | [peek\\_top](#) | [peek\\_end](#)

## push

```
include std/stack.e
namespace stack
public procedure push(stack sk, object value)
```

adds something to a stack.

### Arguments:

1. `sk` : the stack to augment
2. `value` : an object, the value to push.

### Comments:

`value` appears at the end of FIFO stacks and the top of FILO stacks. The size of the stack

increases by one.

### Example 1:

```
stack sk = new(FIFO)

push(sk,5)
push(sk,"abc")
push(sk, 2.3)
top(sk)  --> 5
last(sk) --> 2.3
```

### Example 2:

```
stack sk = new(FILO)

push(sk,5)
push(sk,"abc")
push(sk, 2.3)
top(sk)  --> 2.3
last(sk) --> 5
```

### See Also:

[pop](#) | [top](#)

## top

```
include std/stack.e
namespace stack
public function top(stack sk)
```

retrieve the top element on a stack.

### Arguments:

1. sk : the stack to inspect.

### Returns:

An **object**, the top element on a stack.

### Comments:

This call is equivalent to `at(sk,1)`.

### Example 1:

```
stack sk = new(FILO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)

top(sk) --> 2.3
```

### Example 2:

```
stack sk = new(FIFO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)
```

```
top(sk) --> 5
```

## See Also:

[at](#) | [pop](#) | [peek\\_top](#) | [last](#)

## last

```
include std/stack.e
namespace stack
public function last(stack sk)
```

retrieves the end element on a stack.

## Arguments:

1. `sk` : the stack to inspect.

## Returns:

An **object**, the end element on a stack.

## Comments:

This call is equivalent to `at(sk,0)`.

## Example 1:

```
stack sk = new(FIFO)

push(sk,5)
push(sk,"abc")
push(sk, 2.3)

last(sk) --> 5
```

## Example 2:

```
stack sk = new(FIFO)

push(sk,5)
push(sk,"abc")
push(sk, 2.3)

last(sk) --> 2.3
```

## See Also:

[at](#) | [pop](#) | [peek\\_end](#) | [top](#)

## pop

```
include std/stack.e
namespace stack
public function pop(stack sk, integer idx = 1)
```

removes an object from a stack.

## Arguments:

1. `sk` : the stack to pop

2. `idx` : integer. The `n`-th item to pick from the stack. The default is 1.

### Returns:

An **item**, from the stack, which is also removed from the stack.

### Errors:

- If the stack is empty, an error occurs.
- If the `idx` is greater than the number of items in the stack, an error occurs.

### Comments:

- For FIFO stacks (queues), the top item is the oldest item in the stack.
- For FILO stacks, the top item is the newest item in the stack.

When `idx` is omitted the 'top' of the stack is removed and returned. When `idx` is supplied, it represents the `n`-th item from the top to be removed and returned. Thus an `idx` of 2 returns the 2nd item from the top, a value of 3 returns the 3rd item from the top, and so on.

### Example 1:

```
stack sk = new(FIFO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? size(sk) -- 3
? pop(sk) -- 1
? size(sk) -- 2
? pop(sk) -- 2
? size(sk) -- 1
? pop(sk) -- 3
? size(sk) -- 0
? pop(sk) -- *error*
```

### Example 2:

```
stack sk = new(FILO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? size(sk) -- 3
? pop(sk) -- 3
? size(sk) -- 2
? pop(sk) -- 2
? size(sk) -- 1
? pop(sk) -- 1
? size(sk) -- 0
? pop(sk) -- *error*
```

### Example 3:

```
stack sk = new(FILO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
push(sk, 4)
-- stack contains {1,2,3,4} (oldest to newest)
? size(sk) -- 4
? pop(sk, 2) -- Pluck out the 2nd newest item .. 3
? size(sk) -- 3
-- stack now contains {1,2,4}
```

### Example 4:

```
stack sk = new(FIFO)
```

```

push(sk, 1)
push(sk, 2)
push(sk, 3)
push(sk, 4)
-- stack contains {1,2,3,4} (oldest to newest)
? size(sk) -- 4
? pop(sk, 2) -- Pluck out the 2nd oldest item .. 2
? size(sk) -- 3
-- stack now contains {1,3,4}

```

## See Also:

[push](#), [top](#), [is\\_empty](#)

## peek\_top

```

include std/stack.e
namespace stack
public function peek_top(stack sk, integer idx = 1)

```

gets an object, relative to the top, from a stack.

## Arguments:

1. sk : the stack to get from.
2. idx : integer. The n-th item to get from the stack. The default is 1.

## Returns:

An **item**, from the stack, which is **not** removed from the stack.

## Errors:

- If the stack is empty, an error occurs.
- If the idx is greater than the number of items in the stack, an error occurs.

## Comments:

This is identical to [pop](#) except that it does not remove the item.

- For FIFO stacks (queues), the top item is the oldest item in the stack.
- For FILO stacks, the top item is the newest item in the stack.

When idx is omitted the 'top' of the stack is returned. When idx is supplied, it represents the n-th item from the top to be returned. Thus an idx of 2 returns the 2nd item from the top, a value of 3 returns the 3rd item from the top, and so on.

## Example 1:

```

stack sk = new(FIFO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? peek_top(sk) -- 1
? peek_top(sk,2) -- 2
? peek_top(sk,3) -- 3
? peek_top(sk,4) -- *error*
? peek_top(sk, size(sk)) -- 3 (end item)

```

## Example 2:

```

stack sk = new(FILO)
push(sk, 1)
push(sk, 2)

```



```

push(sk, 3)
? peek_top(sk) -- 3
? peek_top(sk,2) -- 2
? peek_top(sk,3) -- 1
? peek_top(sk,4) -- *error*
? peek_top(sk, size(sk)) -- 1 (end item)

```

## See Also:

pop, top, is\_empty, size, peek\_end

## peek\_end

```

include std/stack.e
namespace stack
public function peek_end(stack sk, integer idx = 1)

```

gets an object, relative to the end, from a stack.

## Arguments:

1. sk : the stack to get from.
2. idx : integer. The n-th item from the end to get from the stack. The default is 1.

## Returns:

An **item**, from the stack, which is **not** removed from the stack.

## Errors:

- If the stack is empty, an error occurs.
- If the idx is greater than the number of items in the stack, an error occurs.

## Comments:

- For FIFO stacks (queues), the end item is the newest item in the stack.
- For FILO stacks, the end item is the oldest item in the stack.

When idx is omitted the 'end' of the stack is returned. When idx is supplied, it represents the n-th item from the end to be returned. Thus an idx of 2 returns the 2nd item from the end, a value of 3 returns the 3rd item from the end, and so on.

## Example 1:

```

stack sk = new(FIFO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? peek_end(sk) -- 3
? peek_end(sk,2) -- 2
? peek_end(sk,3) -- 1
? peek_end(sk,4) -- *error*
? peek_end(sk, size(sk)) -- 3 (top item)

```

## Example 2:

```

stack sk = new(FILO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? peek_end(sk) -- 1
? peek_end(sk,2) -- 2
? peek_end(sk,3) -- 3
? peek_end(sk,4) -- *error*

```

```
? peek_end(sk, size(sk)) -- 3 (top item)
```

## See Also:

[pop](#) | [top](#) | [is\\_empty](#) | [size](#) | [peek\\_top](#)

## swap

```
include std/stack.e
namespace stack
public procedure swap(stack sk)
```

swaps the top two elements of a stack.

## Arguments:

1. `sk` : the stack to swap.

## Returns:

A **copy**, of the original **stack**, with the top two elements swapped.

## Comments:

- For FIFO stacks (queues), the top item is the oldest item in the stack.
- For FILO stacks, the top item is the newest item in the stack.

## Errors:

If the stack has less than two elements, an error occurs.

## Example 1:

```
stack sk = new(FILO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)
push(sk, "")

? peek_top(sk, 1) --> ""
? peek_top(sk, 2) --> 2.3

swap(sk)

? peek_top(sk, 1) --> 2.3
? peek_top(sk, 2) --> ""
```

## Example 2:

```
stack sk = new(FIFO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)
push(sk, "")

peek_top(sk, 1) --> 5
peek_top(sk, 2) --> "abc"

swap(sk)

peek_top(sk, 1) --> "abc"
peek_top(sk, 2) --> 5
```

## dup

```
include std/stack.e
namespace stack
public procedure dup(stack sk)
```

repeats the top element of a stack.

### Arguments:

1. sk : the stack.

### Comments:

- For FIFO stacks (queues), the top item is the oldest item in the stack.
- For FILO stacks, the top item is the newest item in the stack.

### Side Effect:

The value of top is pushed onto the stack, thus the stack size grows by one.

### Errors:

If the stack has no elements, an error occurs.

### Example 1:

```
stack sk = new(FILO)

push(sk, 5)
push(sk, "abc")
push(sk, "")

dup(sk)

peek_top(sk, 1) --> ""
peek_top(sk, 2) --> "abc"
size(sk)        --> 3

dup(sk)

peek_top(sk, 1) --> ""
peek_top(sk, 2) --> ""
peek_top(sk, 3) --> "abc"
size(sk)        --> 4
```

### Example 1:

```
stack sk = new(FIFO)

push(sk, 5)
push(sk, "abc")
push(sk, "")

dup(sk)

peek_top(sk, 1) --> 5
peek_top(sk, 2) --> "abc"
size(sk)        --> 3

dup(sk)

peek_top(sk, 1) --> 5
peek_top(sk, 2) --> 5
peek_top(sk, 3) --> "abc"
size(sk)        --> 4
```

## set

```
include std/stack.e
namespace stack
public procedure set(stack sk, object val, integer idx = 1)
```

updates a value on the stack.

### Arguments:

1. sk : the stack being queried
2. val : an object, the value to place on the stack
3. idx : an integer, the place to inspect. The default is 1 (the top item)

### Errors:

If the supplied value of idx does not correspond to an existing element, an error occurs.

### Comments:

- For FIFO stacks (queues), the top item is the oldest item in the stack.
- For FILO stacks, the top item is the newest item in the stack.

idx can be less than one, in which case it refers to an element relative to the end of the stack. Thus 0 stands for the end element.

### See Also:

[size](#) | [top](#)

## clear

```
include std/stack.e
namespace stack
public procedure clear(stack sk)
```

wipes out a stack.

### Arguments:

1. sk : the stack to clear.

### Side Effect:

The stack contents is emptied.

### See Also:

[new](#) | [is\\_empty](#)

# Scientific Notation Parsing

## Parsing routines

The parsing functions require a sequence containing a correctly formed scientific notation representation of a number. The general pattern is an optional negative sign (-), a number, usually with a decimal point, followed by an upper case or lower case 'e', then optionally a plus (+) or a minus (-) sign, and an integer. There should be no spaces or other characters. The following are valid numbers:

```
1e0
3.1415e-2
-9.0E+3
```

This library evaluates scientific notation to the highest level of precision possible using Euphoria atoms. An atom in 32-bit euphoria can have up to 16 digits of precision (19 in 64-bit euphoria). A number represented by scientific notation could contain up to 17 (or 20) digits. The 17th (or 20th) supplied digit may have an effect upon the value of the atom due to rounding errors in the calculations.

This does not mean that if the 17th (or 20th) digit is 5 or higher, you should include it. The calculations are much more complicated, because a decimal fraction has to be converted to a binary fraction, and there is not really a one-to-one correspondence between the decimal digits and the bits in the resulting atom. The 18th or higher digit, however, will never have an effect on the resulting atom.

The biggest and smallest (magnitude) atoms possible are:

```
32-bit:
1.7976931348623157e+308
4.9406564584124654e-324
```

## Floating Point Types

### floating\_point

```
include std/scinot.e
public enum type floating_point
```

### NATIVE

```
include std/scinot.e
enum type floating_point NATIVE
```

NATIVE Use whatever is the appropriate format based upon the version of euphoria being used (DOUBLE for 32-bit, EXTENDED for 64-bit)

### DOUBLE

```
include std/scinot.e
enum type floating_point DOUBLE
```

### DOUBLE:

Description IEEE 754 double (64-bit) floating point format. The native 32-bit euphoria floating point representation.

## EXTENDED

```
include std/scinot.e
enum type floating_point EXTENDED
```

The native 64-bit euphoria floating point representation.

## bits\_to\_bytes

```
include std/scinot.e
public function bits_to_bytes(sequence bits)
```

Takes a sequence of bits (all elements either 0 or 1) and converts it into a sequence of bytes.

### Arguments:

1. bits : sequence of ones and zeroes

Returns a sequence of 8-bit integers

## bytes\_to\_bits

```
include std/scinot.e
public function bytes_to_bits(sequence bytes)
```

Converts a sequence of bytes (all elements integers between 0 and 255) and converts it into a sequence of bits.

### Arguments:

1. bytes : sequence of values from 0-255

### Returns:

Sequence of bits (ones and zeroes)

## scientific\_to\_float

```
include std/scinot.e
public function scientific_to_float(sequence s, floating_point fp = NATIVE)
```

Takes a string representation of a number in scientific notation and the requested precision (DOUBLE or EXTENDED) and returns a sequence of bytes in the raw format of an IEEE 754 double or extended precision floating point number. This value can be passed to the euphoria library function, `float64_to_atom` or `float80_to_atom`, respectively.

### Arguments:

1. s : string representation of a number, e.g., "1.23E4"
2. fp : the required precision for the ultimate representation
  1. DOUBLE Use IEEE 754, the euphoria representation used in 32-bit euphoria

2. EXTENDED Use Extended Floating Point, the euphoria representation in 64-bit euphoria

### Returns:

Sequence of bytes that represents the physical form of the converted floating point number.

### Note:

Does not check if the string exceeds IEEE 754 double precision limits.

## scientific\_to\_atom

```
include std/scinot.e
public function scientific_to_atom(sequence s, floating_point fp = NATIVE)
```

Takes a string representation of a number in scientific notation and returns an atom.

### Arguments:

1. s : string representation of a number (such as "1.23E4" ).
2. fp : the required precision for the ultimate representation.
  1. DOUBLE Use IEEE 754, the euphoria representation used in 32-bit Euphoria.
  2. EXTENDED Use Extended Floating Point, the euphoria representation in 64-bit Euphoria.

### Returns:

Euphoria atom floating point number.

# NETWORK

- `socket.e`
- `net/common.e`
- `net/dns.e`
- `net/http.e`
- `net/url.e`



# Core Sockets

## Error Information

### error\_code

```
include std/socket.e
namespace sockets
public function error_code()
```

gets the error code.

#### Returns:

Integer **OK** on no error, otherwise any one of the `ERR_` constants to follow.

### OK

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

No error occurred.

### ERR\_ACCESS

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

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

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

### ERR\_ADDRINUSE

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

Address is already in use.

### ERR\_ADDRNOTAVAIL

```
include std/socket.e
namespace sockets
```

```
public constant ERR_ADDRNOTAVAIL
```

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

## ERR\_AFNOSUPPORT

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

Address family not supported by the protocol family.

## ERR\_AGAIN

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

Kernel resources to complete the request are temporarily unavailable.

## ERR\_ALREADY

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

Operation is already in progress.

## ERR\_CONNABORTED

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

Software has caused a connection to be aborted.

## ERR\_CONNREFUSED

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

Connection was refused.

## ERR\_CONNRESET

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

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

## ERR\_DESTADDRREQ

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

Destination address required.

## ERR\_FAULT

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

Address creation has failed internally.

## ERR\_HOSTUNREACH

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

No route to the host specified could be found.

## ERR\_INPROGRESS

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

A blocking call is inprogress.

## ERR\_INTR

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

A blocking call was cancelled or interrupted.

## ERR\_INVALID

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

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

## ERR\_IO

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

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

## ERR\_ISCONN

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

Socket is already connected.

## ERR\_ISDIR

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

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

## ERR\_LOOP

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

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

## ERR\_MFILE

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

The queue is not empty upon routine call.

## ERR\_MSGSIZE

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

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

## ERR\_NAMETOOLONG

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

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

## ERR\_NETDOWN

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

The network subsystem is down or has failed

## ERR\_NETRESET

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

Network has dropped it's connection on reset.

## ERR\_NETUNREACH

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

Network is unreachable.

## ERR\_NFILE

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

Not a file. (Unix Domain Sockets).

## ERR\_NOBUFS

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

No buffer space is available.

## ERR\_NOENT

```
include std/socket.e
namespace sockets
```

```
public constant ERR_NOENT
```

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

## ERR\_NOTCONN

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

Socket is not connected.

## ERR\_NOTDIR

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

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

## ERR\_NOTINITIALISED

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

Socket system is not initialized (Windows only)

## ERR\_NOTSOCK

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

The descriptor is not a socket.

## ERR\_OPNOTSUPP

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

Operation is not supported on this type of socket.

## ERR\_PROTONOSUPPORT

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

Protocol not supported.

## ERR\_PROTOTYPE

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

Protocol is the wrong type for the socket.

## ERR\_ROFS

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

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

## ERR\_SHUTDOWN

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

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

## ERR\_SOCKETNOSUPPORT

```
include std/socket.e
namespace sockets
public constant ERR_SOCKETNOSUPPORT
```

Socket type is not supported.

## ERR\_TIMEDOUT

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

Connection has timed out.

## ERR\_WOULDBLOCK

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

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

## Socket Backend Constants

These values are used by the Euphoria backend to pass information to this library. The TYPE constants are used to identify to the [info](#) function which family of constants are

being retrieved (AF protocols, socket types, and socket options, respectively).

## ESOCK\_UNDEFINED\_VALUE

```
include std/socket.e
namespace sockets
public constant ESOCK_UNDEFINED_VALUE
```

when a particular constant was not defined by C, the backend returns this value

## ESOCK\_UNKNOWN\_FLAG

```
include std/socket.e
namespace sockets
public constant ESOCK_UNKNOWN_FLAG
```

if the backend doesn't recognize the flag in question

## ESOCK\_TYPE\_AF

```
include std/socket.e
namespace sockets
public constant ESOCK_TYPE_AF
```

## ESOCK\_TYPE\_TYPE

```
include std/socket.e
namespace sockets
public constant ESOCK_TYPE_TYPE
```

## ESOCK\_TYPE\_OPTION

```
include std/socket.e
namespace sockets
public constant ESOCK_TYPE_OPTION
```

## Socket Type Euphoria Constants

These values are used to retrieve the known values for family and sock\_type parameters of the `create` function from the Euphoria backend. (The reason for doing it this way is to retrieve the values defined in C, instead of duplicating them here.) These constants are guaranteed to never change, and to be the same value across platforms.

## EAF\_UNSPEC

```
include std/socket.e
namespace sockets
public constant EAF_UNSPEC
```

Address family is unspecified



## EAFF\_UNIX

```
include std/socket.e
namespace sockets
public constant EAF_UNIX
```

Local communications

## EAFF\_INET

```
include std/socket.e
namespace sockets
public constant EAF_INET
```

IPv4 Internet protocols

## EAFF\_INET6

```
include std/socket.e
namespace sockets
public constant EAF_INET6
```

IPv6 Internet protocols

## EAFF\_APPLETALK

```
include std/socket.e
namespace sockets
public constant EAF_APPLETALK
```

Appletalk

## EAFF\_BTH

```
include std/socket.e
namespace sockets
public constant EAF_BTH
```

Bluetooth (currently Windows-only)

## ESOCK\_STREAM

```
include std/socket.e
namespace sockets
public constant ESOCK_STREAM
```

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

## ESOCK\_DGRAM

```
include std/socket.e
namespace sockets
public constant ESOCK_DGRAM
```

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

## ESOCK\_RAW

```
include std/socket.e
namespace sockets
public constant ESOCK_RAW
```

Provides raw network protocol access.

## ESOCK\_RDM

```
include std/socket.e
namespace sockets
public constant ESOCK_RDM
```

Provides a reliable datagram layer that does not guarantee ordering.

## ESOCK\_SEQPACKET

```
include std/socket.e
namespace sockets
public constant ESOCK_SEQPACKET
```

Obsolete and should not be used in new programs

## Socket Type Constants

These values are passed as the family and sock\_type parameters of the [create](#) function. They are OS-dependent.

## AF\_UNSPEC

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

Address family is unspecified

## AF\_UNIX

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

Local communications

## AF\_INET

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

IPv4 Internet protocols

## AF\_INET6

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

IPv6 Internet protocols

## AF\_APPLETALK

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

Appletalk

## AF\_BTH

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

Bluetooth (currently Windows-only)

## SOCK\_STREAM

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

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

## SOCK\_DGRAM

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

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

## SOCK\_RAW

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

Provides raw network protocol access.

## SOCK\_RDM

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

Provides a reliable datagram layer that does not guarantee ordering.

## SOCK\_SEQPACKET

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

Obsolete and should not be used in new programs

## Select Accessor Constants

Use with the result of [select](#).

### enum

```
include std/socket.e
namespace sockets
public enum
```

## SELECT\_SOCKET

```
include std/socket.e
namespace sockets
SELECT_SOCKET
```

The socket

## SELECT\_IS\_READABLE

```
include std/socket.e
namespace sockets
SELECT_IS_READABLE
```

Boolean (1/0) value indicating the readability.

## SELECT\_IS\_WRITABLE

```
include std/socket.e
namespace sockets
SELECT_IS_WRITABLE
```

Boolean (1/0) value indicating the writeability.

## SELECT\_IS\_ERROR

```
include std/socket.e
namespace sockets
SELECT_IS_ERROR
```

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

## Shutdown Options

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

### SD\_SEND

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

Shutdown the send operations.

### SD\_RECEIVE

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

Shutdown the receive operations.

### SD\_BOTH

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

Shutdown both send and receive operations.

## Socket Options

Pass to the optname parameter of the functions `get_option` and `set_option`.

These options are highly OS specific and are normally not needed for most socket communication. They are provided here for your convenience. If you should need to set socket options, please refer to your OS reference material.

There may be other values that your OS defines and some defined here are not supported on all operating systems.

## Socket Options In Common

## SOL\_SOCKET

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

## SO\_DEBUG

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

## SO\_ACCEPTCONN

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

## SO\_REUSEADDR

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

## SO\_KEEPALIVE

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

## SO\_DONTROUTE

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

## SO\_BROADCAST

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

## SO\_LINGER

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

## SO\_SNDBUF

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

## SO\_RCVBUF

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

## SO\_SNDLOWAT

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

## SO\_RCVLOWAT

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

## SO\_SNDTIMEO

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

## SO\_RCVTIMEO

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

## SO\_ERROR

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

## SO\_TYPE

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

## SO\_OOBINLINE

```
include std/socket.e
namespace sockets
public constant SO_OOBINLINE
```

## Windows Socket Options

## SO\_USELOOPBACK

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

## SO\_DONTLINGER

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

## SO\_REUSEPORT

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

## SO\_CONNDATA

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

## SO\_CONNOPT

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

## SO\_DISCDATA

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

## SO\_DISCOPT



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

## SO\_CONNDATALEN

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

## SO\_CONNOPTLEN

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

## SO\_DISCDATALEN

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

## SO\_DISCOPTLEN

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

## SO\_OPENTYPE

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

## SO\_MAXDG

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

## SO\_MAXPATHDG

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

## SO\_SYNCHRONOUS\_ALERT

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

## SO\_SYNCHRONOUS\_NONALERT

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

## LINUX Socket Options

### SO\_SNDBUFSIZE

```
include std/socket.e
namespace sockets
public constant SO_SNDBUFSIZE
```

### SO\_RCVBUFSIZE

```
include std/socket.e
namespace sockets
public constant SO_RCVBUFSIZE
```

### SO\_NO\_CHECK

```
include std/socket.e
namespace sockets
public constant SO_NO_CHECK
```

### SO\_PRIORITY

```
include std/socket.e
namespace sockets
public constant SO_PRIORITY
```

### SO\_BSDCOMPAT

```
include std/socket.e
namespace sockets
public constant SO_BSDCOMPAT
```

### SO\_PASSCRED

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

## SO\_PEERCREC

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

- Security levels - as per NRL IPv6 - do not actually do anything

## SO\_SECURITY\_AUTHENTICATION

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

## SO\_SECURITY\_ENCRYPTION\_TRANSPORT

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

## SO\_SECURITY\_ENCRYPTION\_NETWORK

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

## SO\_BINDTODEVICE

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

## LINUX Socket Filtering Options

## SO\_ATTACH\_FILTER

```
include std/socket.e
namespace sockets
public constant SO_ATTACH_FILTER
```

## SO\_DETACH\_FILTER

```
include std/socket.e
namespace sockets
public constant SO_DETACH_FILTER
```

## SO\_PEERNAME

```
include std/socket.e
namespace sockets
public constant SO_PEERNAME
```

## SO\_TIMESTAMP

```
include std/socket.e
namespace sockets
public constant SO_TIMESTAMP
```

## SCM\_TIMESTAMP

```
include std/socket.e
namespace sockets
public constant SCM_TIMESTAMP
```

## SO\_PEERSEC

```
include std/socket.e
namespace sockets
public constant SO_PEERSEC
```

## SO\_PASSSEC

```
include std/socket.e
namespace sockets
public constant SO_PASSSEC
```

## SO\_TIMESTAMPNS

```
include std/socket.e
namespace sockets
public constant SO_TIMESTAMPNS
```

## SCM\_TIMESTAMPNS

```
include std/socket.e
namespace sockets
public constant SCM_TIMESTAMPNS
```

## SO\_MARK

```
include std/socket.e
namespace sockets
public constant SO_MARK
```

## SO\_TIMESTAMPING

```
include std/socket.e
namespace sockets
public constant SO_TIMESTAMPING
```

## SCM\_TIMESTAMPING

```
include std/socket.e
namespace sockets
public constant SCM_TIMESTAMPING
```

## SO\_PROTOCOL

```
include std/socket.e
namespace sockets
public constant SO_PROTOCOL
```

## SO\_DOMAIN

```
include std/socket.e
namespace sockets
public constant SO_DOMAIN
```

## SO\_RXQ\_OVFL

```
include std/socket.e
namespace sockets
public constant SO_RXQ_OVFL
```

## Send Flags

Pass to the flags parameter of [send](#) and [receive](#)

## MSG\_OOB

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

Sends out-of-band data on sockets that support this notion (e.g., of type [SOCK\\_STREAM](#)); the underlying protocol must also support out-of-band data.

## MSG\_PEEK

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

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

## MSG\_DONTROUTE

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

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

## MSG\_TRYHARD

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

## MSG\_CTRUNC

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

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

## MSG\_PROXY

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

## MSG\_TRUNC

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

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

## MSG\_DONTWAIT

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

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

## MSG\_EOR

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

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

## MSG\_WAITALL

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

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

## MSG\_FIN

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

## MSG\_SYN

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

## MSG\_CONFIRM

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

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

## MSG\_RST

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

## MSG\_ERRQUEUE

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

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

## MSG\_NOSIGNAL

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

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

## MSG\_MORE

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

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

## Server and Client Sides

### enum

```
include std/socket.e
namespace sockets
export enum
```

## SOCKET\_SOCKET

```
include std/socket.e
namespace sockets
SOCKET_SOCKET
```

Accessor index for socket handle of a socket type

## SOCKET\_SOCKADDR\_IN



```
include std/socket.e
namespace sockets
SOCKET_SOCKADDR_IN
```

Accessor index for the sockaddr\_in pointer of a socket type

## socket

```
include std/socket.e
namespace sockets
public type socket(object o)
```

Socket type

## create

```
include std/socket.e
namespace sockets
public function create(integer family, integer sock_type, integer protocol)
```

creates a new socket.

### Arguments:

1. family: an integer
2. sock\_type: an integer, the type of socket to create
3. protocol: an integer, the communication protocol being used

family options:

- AF\_UNIX
- AF\_INET
- AF\_INET6
- AF\_APPLETALK
- AF\_BTH

sock\_type options:

- SOCK\_STREAM
- SOCK\_DGRAM
- SOCK\_RAW
- SOCK\_RDM
- SOCK\_SEQPACKET

### Returns:

An **object**, an atom, representing an integer code on failure, else a sequence representing a valid socket id.

### Comments:

On *Windows* you must have Windows Sockets version 2.2 or greater installed. This means at least Windows 2000 Professional or Windows 2000 Server.

### Example 1:

```
socket = create(AF_INET, SOCK_STREAM, 0)
```

## close

```
include std/socket.e
namespace sockets
public function close(socket sock)
```

closes a socket.

### Arguments:

1. sock: the socket to close

### Returns:

An **integer**, 0 on success and -1 on error.

### Comments:

It may take several minutes for the OS to declare the socket as closed.

## shutdown

```
include std/socket.e
namespace sockets
public function shutdown(socket sock, atom method = SD_BOTH)
```

partially or fully close a socket.

### Arguments:

1. sock : the socket to shutdown
2. method : the way used to close the socket

### Returns:

An **integer**, 0 on success and -1 on error.

### Comments:

Three constants are defined that can be sent to method:

- **SD\_SEND** -- shutdown the send operations.
- **SD\_RECEIVE** -- shutdown the receive operations.
- **SD\_BOTH** -- shutdown both send and receive operations.

It may take several minutes for the OS to declare the socket as closed.

## select

```
include std/socket.e
namespace sockets
public function select(object sockets_read, object sockets_write, object sockets_err,
    integer timeout = 0, integer timeout_micro = 0)
```

determines the read, write and error status of one or more sockets.

### Arguments:

1. sockets\_read : either one socket or a sequence of sockets to check for reading.
2. sockets\_write : either one socket or a sequence of sockets to check for writing.
3. sockets\_err : either one socket or a sequence of sockets to check for errors.
4. timeout : maximum time to wait to determine a sockets status, seconds part
5. timeout\_micro : maximum time to wait to determine a sockets status, microsecond part

## Returns:

A **sequence**, of the same size of all unique sockets containing { `socket`, `read_status`, `write_status`, `error_status` } for each socket passed 2 to the function. Note that the sockets returned are not guaranteed to be in any particular order.

## Comments:

Using `select`, you can check to see if a socket has data waiting and is read to be read, if a socket can be written to and if a socket has an error status.

`select` allows for fine-grained control over your sockets; it allows you to specify that a given socket only be checked for reading or for only reading and writing, etc.

## send

```
include std/socket.e
namespace sockets
public function send(socket sock, sequence data, atom flags = 0)
```

sends TCP data to a socket connected remotely.

## Arguments:

1. `sock` : the socket to send data to
2. `data` : a sequence of atoms, what to send
3. `flags` : flags (see [Send Flags](#))

## Returns:

An **integer**, the number of characters sent, or -1 for an error.

## receive

```
include std/socket.e
namespace sockets
public function receive(socket sock, atom flags = 0)
```

receives data from a bound socket.

## Arguments:

1. `sock` : the socket to get data from
2. `flags` : flags (see [Send Flags](#))

## Returns:

A **sequence**, either a full string of data on success, or an atom indicating the error code.

## Comments:

This function will not return until data is actually received on the socket, unless the flags parameter contains `MSG_DONTWAIT`.

`MSG_DONTWAIT` only works on Linux kernels 2.4 and above. To be cross-platform you should use `select` to determine if a socket is readable, i.e. has data waiting.

## get\_option

```
include std/socket.e
namespace sockets
public function get_option(socket sock, integer level, integer optname)
```

gets options for a socket.

### Arguments:

1. sock : the socket
2. level : an integer, the option level
3. optname : requested option (See [Socket Options](#))

### Returns:

An **object**, either:

- On error, {"ERROR",error\_code}.
- On success, either an atom or a sequence containing the option value, depending on the option.

### Comments:

Primarily for use in multicast or more advanced socket applications. Level is the option level, and option\_name is the option for which values are being sought. Level is usually [SOL\\_SOCKET](#).

### Returns:

An **atom**, On error, an atom indicating the error code.

A **sequence** or **atom**, On success, either an atom or a sequence containing the option value.

### See Also:

[get\\_option](#)

## set\_option

```
include std/socket.e
namespace sockets
public function set_option(socket sock, integer level, integer optname, object val)
```

sets options for a socket.

### Arguments:

1. sock : an atom, the socket id
2. level : an integer, the option level
3. optname : requested option (See [Socket Options](#))
4. val : an object, the new value for the option

### Returns:

An **integer**, 0 on success, -1 on error.

### Comments:

Primarily for use in multicast or more advanced socket applications. Level is the option level, and option\_name is the option for which values are being set. Level is usually [SOL\\_SOCKET](#).

### See Also:

[get\\_option](#)

## Client Side Only

### connect

```
include std/socket.e
namespace sockets
public function connect(socket sock, sequence address, integer port = - 1)
```

establishes an outgoing connection to a remote computer. Only works with TCP sockets.

#### Arguments:

1. sock : the socket
2. address : ip address to connect, optionally with :PORT at the end
3. port : port number

#### Returns:

An **integer**, 0 for success and non-zero on failure. See the ERR\_\* constants for supported values.

#### Comments:

address can contain a port number. If it does not, it has to be supplied to the port parameter.

#### Example 1:

```
success = connect(sock, "11.1.1.1") -- uses default port 80
success = connect(sock, "11.1.1.1:110") -- uses port 110
success = connect(sock, "11.1.1.1", 345) -- uses port 345
```

## Server Side Only

### bind

```
include std/socket.e
namespace sockets
public function bind(socket sock, sequence address, integer port = - 1)
```

joins a socket to a specific local internet address and port so later calls only need to provide the socket.

#### Arguments:

1. sock : the socket
2. address : the address to bind the socket to
3. port : optional, if not specified you must include :PORT in the address parameter.

#### Returns:

An **integer**, 0 on success and -1 on failure.

#### Example 1:

```
-- Bind to all interfaces on the default port 80.
success = bind(socket, "0.0.0.0")
-- Bind to all interfaces on port 8080.
```

```
success = bind(socket, "0.0.0.0:8080")
-- Bind only to the 243.17.33.19 interface on port 345.
success = bind(socket, "243.17.33.19", 345)
```

## listen

```
include std/socket.e
namespace sockets
public function listen(socket sock, integer backlog)
```

starts monitoring a connection. Only works with TCP sockets.

### Arguments:

1. sock : the socket
2. backlog : the number of connection requests that can be kept waiting before the OS refuses to hear any more.

### Returns:

An **integer**, 0 on success and an error code on failure.

### Comments:

Once the socket is created and bound, this will indicate to the operating system that you are ready to being listening for connections.

The value of backlog is strongly dependent on both the hardware and the amount of time it takes the program to process each connection request.

This function must be executed after **bind**.

## accept

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

produces a new socket for an incoming connection.

### Arguments:

1. sock: the server socket

### Returns:

An **atom**, on error

A **sequence**, {socket client, sequence client\_ip\_address} on success.

### Comments:

Using this function allows communication to occur on a "side channel" while the main server socket remains available for new connections.

accept must be called after bind and listen.

## UDP Only

## send\_to

```
include std/socket.e
namespace sockets
public function send_to(socket sock, sequence data, sequence address, integer port = - 1,
    atom flags = 0)
```

sends a UDP packet to a given socket.

### Arguments:

1. sock: the server socket
2. data: the data to be sent
3. ip: the ip where the data is to be sent to (ip:port) is acceptable
4. port: the port where the data is to be sent on (if not supplied with the ip)
5. flags : flags (see [Send Flags](#))

### Returns:

An integer status code.

### See Also:

[receive\\_from](#)

## receive\_from

```
include std/socket.e
namespace sockets
public function receive_from(socket sock, atom flags = 0)
```

receives a UDP packet from a given socket.

### Arguments:

1. sock: the server socket
2. flags : flags (see [Send Flags](#))

### Returns:

A sequence containing { client\_ip, client\_port, data } or an atom error code.

### See Also:

[send\\_to](#)

## Information

## service\_by\_name

```
include std/socket.e
namespace sockets
public function service_by_name(sequence name, object protocol = 0)
```

gets service information by name.

### Arguments:

1. name : service name.
2. protocol : protocol. Default is not to search by protocol.

### Returns:

A **sequence**, containing { official protocol name, protocol, port number } or an atom indicating the error code.

### Example 1:

```
object result = getservbyname("http")
-- result = { "http", "tcp", 80 }
```

### See Also:

[service\\_by\\_port](#)

## service\_by\_port

```
include std/socket.e
namespace sockets
public function service_by_port(integer port, object protocol = 0)
```

gets service information by port number.

### Arguments:

1. port : port number.
2. protocol : protocol. Default is not to search by protocol.

### Returns:

A **sequence**, containing { official protocol name, protocol, port number } or an atom indicating the error code.

### Example 1:

```
object result = getservbyport(80)
-- result = { "http", "tcp", 80 }
```

### See Also:

[service\\_by\\_name](#)

## info

```
include std/socket.e
namespace sockets
public function info(integer Type)
```

gets constant definitions from the backend.

### Arguments:

1. type : The type of information requested.

### Returns:

A **sequence**, containing the list of definitions from the backend. The resulting list can be indexed into using the Euphoria constants. Or an atom indicating an error.

### Example 1:



```
object result = info(ESOCK_TYPE_AF)
-- result = { AF_UNIX, AF_INET, AF_INET6, AF_APPLETALK, AF_BTH, AF_UNSPEC }
```

## See Also:

[Socket Options](#), [Socket Backend Constants](#), [Socket Type Euphoria Constants](#)

# Common Internet Routines

## IP Address Handling

### is\_inetaddr

```
include std/net/common.e
namespace common
public function is_inetaddr(sequence address)
```

checks if x is an IP address in the form (#.#.#.#[:#])

#### Parameters:

1. address : the address to check

#### Returns:

An **integer**, 1 if x is an inetaddr, 0 if it is not

#### Comments:

Some ip validation algorithms do not allow 0.0.0.0 . We do here because many times you will want to bind to 0.0.0.0 . However, you cannot connect to 0.0.0.0 of course.

With sockets, normally binding to 0.0.0.0 means bind to all interfaces that the computer has.

### parse\_ip\_address

```
include std/net/common.e
namespace common
public function parse_ip_address(sequence address, integer port = - 1)
```

converts a text "address:port" into {"address", port} format.

#### Parameters:

1. address : ip address to connect, optionally with :PORT at the end
2. port : optional, if not specified you may include :PORT in the address parameter otherwise the default port 80 is used.

#### Returns:

A **sequence**, of two elements: "address" and integer port number.

#### Comments:

If port is supplied, it overrides any ":PORT" value in the input address.

#### Example 1:

```
addr = parse_ip_address("11.1.1.1") --> {"11.1.1.1", 80} -- default port
addr = parse_ip_address("11.1.1.1:110") --> {"11.1.1.1", 110}
addr = parse_ip_address("11.1.1.1", 345) --> {"11.1.1.1", 345}
```

## URL Parsing

## URL\_ENTIRE

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

## URL\_PROTOCOL

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

## URL\_HTTP\_DOMAIN

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

## URL\_HTTP\_PATH

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

## URL\_HTTP\_QUERY

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

## URL\_MAIL\_ADDRESS

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

## URL\_MAIL\_USER

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

## URL\_MAIL\_DOMAIN

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

## URL\_MAIL\_QUERY

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

## parse\_url

```
include std/net/common.e
namespace common
public function parse_url(sequence url)
```

Parse a common URL. Currently supported URLs are http(s), ftp(s), gopher(s) and mailto.

### Parameters:

1. url : url to be parsed

### Returns:

A **sequence**, containing the URL details. You should use the `URL_` constants to access the values of the returned sequence. You should first check the protocol (`URL_PROTOCOL`) that was returned as the data contained in the return value can be of different lengths.

On a parse error, -1 will be returned.

### Example 1:

```
object url_data = parse_url("http://john.com/index.html?name=jeff")
-- url_data = {
--   "http://john.com/index.html?name=jeff", -- URL_ENTIRE
--   "http",                                -- URL_PROTOCOL
--   "john.com",                             -- URL_DOMAIN
--   "/index.html",                          -- URL_PATH
--   "?name=jeff"                            -- URL_QUERY
-- }

url_data = parse_url("mailto:john@mail.doe.com?subject=Hello%20John%20Doe")
-- url_data = {
--   "mailto:john@mail.doe.com?subject=Hello%20John%20Doe",
--   "mailto",
--   "john@mail.doe.com",
--   "john",
--   "mail.doe.com",
--   "?subject=Hello%20John%20Doe"
-- }
```

# DNS

## Constants

### enum

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

### enum

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

### DNS\_QUERY\_STANDARD

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

### DNS\_QUERY\_ACCEPT\_TRUNCATED\_RESPONSE

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

### DNS\_QUERY\_USE\_TCP\_ONLY

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

### DNS\_QUERY\_NO\_RECURSION

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

### DNS\_QUERY\_BYPASS\_CACHE

```
include std/net/dns.e
namespace dns
```

```
public constant DNS_QUERY_BYPASS_CACHE
```

## DNS\_QUERY\_NO\_WIRE\_QUERY

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

## DNS\_QUERY\_NO\_LOCAL\_NAME

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

## DNS\_QUERY\_NO\_HOSTS\_FILE

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

## DNS\_QUERY\_NO\_NETBT

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

## DNS\_QUERY\_WIRE\_ONLY

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

## DNS\_QUERY\_RETURN\_MESSAGE

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

## DNS\_QUERY\_TREAT\_AS\_FQDN

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

## DNS\_QUERY\_DONT\_RESET\_TTL\_VALUES

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

## DNS\_QUERY\_RESERVED

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

## NS\_C\_IN

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

## NS\_C\_ANY

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

## NS\_KT\_RSA

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

## NS\_KT\_DH

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

## NS\_KT\_DSA

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

## NS\_KT\_PRIVATE

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

## NS\_T\_A

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

## NS\_T\_NS

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

## NS\_T\_PTR

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

## NS\_T\_MX

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

## NS\_T\_AAAA

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

## NS\_T\_A6

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

## NS\_T\_ANY

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

## General Subroutines

### host\_by\_name



```
include std/net/dns.e
namespace dns
public function host_by_name(sequence name)
```

gets the host information by name.

### Parameters:

1. name : host name

### Returns:

A sequence, containing

```
{
  official name,
  { alias1, alias2, ... },
  { ip1, ip2, ... },
  address_type
}
```

### Example 1:

```
object data = host_by_name("www.google.com")
-- data = {
--   "www.l.google.com",
--   {
--     "www.google.com"
--   },
--   {
--     "74.125.93.104",
--     "74.125.93.147",
--     ...
--   },
--   2
-- }
```

## host\_by\_addr

```
include std/net/dns.e
namespace dns
public function host_by_addr(sequence address)
```

gets the host information by address.

### Parameters:

1. address : host address

### Returns:

A sequence, containing

```
{
  official name,
  { alias1, alias2, ... },
  { ip1, ip2, ... },
  address_type
}
```

### Example 1:

```
object data = host_by_addr("74.125.93.147")
```

```
-- data = {  
--   "www.l.google.com",  
--   {  
--     "www.google.com"  
--   },  
--   {  
--     "74.125.93.104",  
--     "74.125.93.147",  
--     ...  
--   },  
--   2  
-- }
```

# HTTP Client

## Error Codes

### enum

```
include std/net/http.e
namespace http
public enum
Increments by - 1
```

## Constants

### enum

```
include std/net/http.e
namespace http
public enum
```

### enum

```
include std/net/http.e
namespace http
public enum
```

### ENCODE\_NONE

```
include std/net/http.e
namespace http
ENCODE_NONE
```

No encoding is necessary

### ENCODE\_BASE64

```
include std/net/http.e
namespace http
ENCODE_BASE64
```

Use Base64 encoding

## Configuration Subroutines

### set\_proxy\_server

```
include std/net/http.e
```

```
namespace http
public procedure set_proxy_server(sequence ip, integer port)
```

Configure http client to use a proxy server

### Parameters:

- proxy\_ip - IP address of the proxy server
- proxy\_port - Port of the proxy server

## Get/Post Routines

### http\_post

```
include std/net/http.e
namespace http
public function http_post(sequence url, object data, object headers = 0,
    natural follow_redirects = 10, natural timeout = 15)
```

Post data to a HTTP resource.

### Parameters:

- url - URL to send post request to
- data - Form data (described later)
- headers - Additional headers added to request
- follow\_redirects - Maximum redirects to follow
- timeout - Maximum number of seconds to wait for a response

### Returns:

An integer error code or a 2 element sequence. Element 1 is a sequence of key/value pairs representing the result header information. element 2 is the body of the result.

If result is a negative integer, that represents a local error condition.

If result is a positive integer, that represents a HTTP error value from the server.

### Data Sequence:

This sequence should contain key value pairs representing the expected form elements of the called URL. For a simple url-encoded form:

```
{ {"name", "John Doe"}, {"age", "22"}, {"city", "Small Town"}}
```

All Keys and Values should be a sequence.

If the post requires multipart form encoding then the sequence is a little different. The first element of the data sequence must be **MULTIPART\_FORM\_DATA**. All subsequent field values should be key/value pairs as described above **except** for a field representing a file upload. In that case the sequence should be:

```
{ FIELD-NAME, FILE-VALUE, FILE-NAME, MIME-TYPE, ENCODING-TYPE }
```

Encoding type can be

- **ENCODE\_NONE**
- **ENCODE\_BASE64**

An example for a multipart form encoded post request data sequence

```
{
  { "name", "John Doe" },
  { "avatar", file_content, "me.png", "image/png", ENCODE_BASE64 },
}
```

```
{ "city", "Small Town" }  
}
```

## See Also:

[http\\_get](#)

## http\_get

```
include std/net/http.e  
namespace http  
public function http_get(sequence url, object headers = 0, natural follow_redirects = 10,  
    natural timeout = 15)
```

Get a HTTP resource.

## Returns:

An integer error code or a 2 element sequence. Element 1 is a sequence of key/value pairs representing the result header information. Element 2 is the body of the result.

If result is a negative integer, that represents a local error condition.

If result is a positive integer, that represents a HTTP error value from the server.

## Example 1:

```
include std/console.e -- for display()  
include std/net/http.e  
  
object result = http_get("http://example.com")  
if atom(result) then  
    printf(1, "Web error: %d\n", result)  
    abort(1)  
end if  
  
display(result[1]) -- header key/value pairs  
printf(1, "Content: %s\n", { result[2] })
```

## See Also:

[http\\_post](#)

# URL handling

## Parsing

### parse\_querystring

```
include std/net/url.e
namespace url
public function parse_querystring(object query_string)
```

Parse a query string into a map

#### Parameters:

1. query\_string: Query string to parse

#### Returns:

map containing the key/value pairs

#### Example 1:

```
map qs = parse_querystring("name=John&age=18")
printf(1, "%s is %s years old\n", { map:get(qs, "name"), map:get(qs, "age") })
```

## URL Parse Accessor Constants

Use with the result of `parse`.

#### Notes:

If the host name, port, path, username, password or query string are not part of the URL they will be returned as an integer value of zero.

### enum

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

### URL\_PROTOCOL

```
include std/net/url.e
namespace url
URL_PROTOCOL
```

The protocol of the URL

### URL\_HOSTNAME

```
include std/net/url.e
namespace url
```

**URL\_HOSTNAME**

The hostname of the URL

## URL\_PORT

```
include std/net/url.e
namespace url
URL_PORT
```

The TCP port that the URL will connect to

## URL\_PATH

```
include std/net/url.e
namespace url
URL_PATH
```

The protocol-specific pathname of the URL

## URL\_USER

```
include std/net/url.e
namespace url
URL_USER
```

The username of the URL

## URL\_PASSWORD

```
include std/net/url.e
namespace url
URL_PASSWORD
```

The password the URL

## URL\_QUERY\_STRING

```
include std/net/url.e
namespace url
URL_QUERY_STRING
```

The HTTP query string

## parse

```
include std/net/url.e
namespace url
public function parse(sequence url, integer querystring_also = 0)
```

Parse a URL returning its various elements.

### Parameters:

1. url: URL to parse
2. querystring\_also: Parse the query string into a map also?

### Returns:

A multi-element sequence containing:

1. protocol
2. host name
3. port
4. path
5. user name
6. password
7. query string

Or, zero if the URL could not be parsed.

### Notes:

If the host name, port, path, username, password or query string are not part of the URL they will be returned as an integer value of zero.

### Example 1:

```
sequence parsed =
    parse("http://user:pass@www.debian.org:80/index.html?name=John&age=39")
-- parsed is
-- {
--     "http",
--     "www.debian.org",
--     80,
--     "/index.html",
--     "user",
--     "pass",
--     "name=John&age=39"
-- }
```

## URL encoding and decoding

### encode

```
include std/net/url.e
namespace url
public function encode(sequence what, sequence spacecode = "+")
```

Converts all non-alphanumeric characters in a string to their percent-sign hexadecimal representation, or plus sign for spaces.

### Parameters:

1. what : the string to encode
2. spacecode : what to insert in place of a space

### Returns:

A **sequence**, the encoded string.

### Comments:

spacecode defaults to + as it is more correct, however, some sites want %20 as the space encoding.



### Example 1:

```
puts(1, encode("Fred & Ethel"))  
-- Prints "Fred+%26+Ethel"
```

### See Also:

[decode](#)

## decode

```
include std/net/url.e  
namespace url  
public function decode(sequence what)
```

Convert all encoded entities to their decoded counter parts

### Parameters:

1. what: what value to decode

### Returns:

A decoded sequence

### Example 1:

```
puts(1, decode("Fred+%26+Ethel"))  
-- Prints "Fred & Ethel"
```

### See Also:

[encode](#)

# LOW LEVEL

- `dll.e`
- `error.e`
- `eumem.e`
- `machine.e`
- `memconst.e`

# Dynamic Linking to External Code

## C Type Constants

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

### Example 1:

See [define\\_c\\_proc](#)

### See Also:

[define\\_c\\_proc](#) | [define\\_c\\_func](#) | [define\\_c\\_var](#)

### C\_CHAR

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

char 8-bits

### C\_BYTE

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

byte 8-bits

### C\_UCHAR

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

unsigned char 8-bits

### C\_UBYTE

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

ubyte 8-bits

### C\_SHORT

```
include std/dll.e
```

```
namespace dll
public constant C_SHORT
```

short 16-bits

## C\_WORD

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

word 16-bits

## C\_USHORT

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

unsigned short 16-bits

## C\_INT

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

int 32-bits

## C\_BOOL

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

bool 32-bits

## C\_UINT

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

unsigned int 32-bits

## C\_LONG

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

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

## C\_ULONG

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

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

## C\_SIZE\_T

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

size\_t unsigned long 32-bits except on 64-bit *unix*, where it is 64-bits

## C\_POINTER

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

any valid pointer

## C\_LONGLONG

```
include std/dll.e
namespace dll
public constant C_LONGLONG
```

longlong 64-bits

## C\_LONG\_PTR

```
include std/dll.e
namespace dll
public constant C_LONG_PTR
```

signed integer sizeof pointer

## C\_HANDLE

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

handle sizeof pointer

## C\_HWND

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

hwnd sizeof pointer

## C\_DWORD

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

dword 32-bits

## C\_WPARAM

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

wparam sizeof pointer

## C\_LPARAM

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

lparam sizeof pointer

## C\_HRESULT

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

hresult 32-bits

## C\_FLOAT

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

float 32-bits

## C\_DOUBLE

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

double 64-bits

## C\_DWORDLONG

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

dwordlong 64-bits

## External Euphoria Type Constants

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

## E\_INTEGER

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

integer

## E\_ATOM

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

atom

## E\_SEQUENCE

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

sequence

## E\_OBJECT

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

object

## sizeof

```
<built-in> function sizeof( atom data_type )
```

### Arguments:

1. `data_type` : The C data-type constant.

### Returns:

An **integer**, the size, in bytes of the specified data type.

## Constants

### NULL

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

The C NULL pointer

## Subroutines

### open\_dll

```
include std/dll.e
namespace dll
public function open_dll(sequence file_name)
```

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

### Arguments:

1. `file_name` :
  - a string sequence : the name of one shared library to open
  - a sequence of strings : a list filenames to be opened

### Returns:

An **atom**, actually a 32-bit address. Return 0 zero if the .dll can not be found.

### Errors:

The length of `file_name` (or any filename contained therein) should not exceed 1\_024 characters.

### Comments:

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

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

The value returned by `open_dll` can be passed to `define_c_proc`, `define_c_func`, or



`define_c_var`.

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

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

### Example 1:

```

                                include std/dll.e
                                atom user32

-- on windows

user32 = open_dll("user32.dll")
if user32 = 0 then
    puts(1, "Failure: could not open user32.dll!\n")
else
    puts(1, "Success: user32.dll opened" )
end if

? gets(0)

--> Success: user32dll opened.c

```

### Example 2:

```

                                include std/dll.e
                                atom mysql_lib

-- on Linux

mysql_lib = open_dll({"libmysqlclient.so", "libmysqlclient.so.15",
                    "libmysqlclient.so.15.0"})
if mysql_lib = 0 then
    puts(1, "Failure: could not find the mysql client library\n")
else
    puts(1, "Success" )
end if

--> Failure: could not find the mysql client library
-- mysql not install on this system

```

### See Also:

[define\\_c\\_func](#) | [define\\_c\\_proc](#) | [define\\_c\\_var](#) | [c\\_func](#) | [c\\_proc](#)

## define\_c\_var

```

include std/dll.e
namespace dll
public function define_c_var(atom lib, sequence variable_name)

```

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

### Arguments:

1. `lib` : an atom, the address of a *Unix* .so or *Windows* .dll, as returned by `open_dll`.
2. `variable_name` : a sequence, the name of a public C variable defined within the library.

### Returns:

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

### Comments:

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

### Example 1:

see `.../euphoria/demo/linux/mylib.ex`

### See Also:

[c\\_proc](#) | [define\\_c\\_func](#) | [c\\_func](#) | [open\\_dll](#)

## define\_c\_proc

```
include std/dll.e
namespace dll
public function define_c_proc(object lib, object routine_name, sequence arg_types)
```

defines the characteristics of either a C function, or a machine-code routine that you wish to call as a procedure from your Euphoria program.

### Arguments:

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

### Returns:

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

### Errors:

The length of name should not exceed 1\_024 characters.

### Comments:

Use the returned routine id as the first argument to `c_proc` when you wish to call the routine from Euphoria.

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

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

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

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

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

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

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

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

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

### Example 1:

```
atom user32
integer ShowWindow

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

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

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

### See Also:

[c\\_proc](#) | [define\\_c\\_func](#) | [c\\_func](#) | [open\\_dll](#)

## define\_c\_func

```
include std/dll.e
namespace dll
public function define_c_func(object lib, object routine_name, sequence arg_types,
    atom return_type)
```

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

### Arguments:

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

### Returns:

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

### Errors:

The length of name should not exceed 1\_024 characters.

### Comments:

Use the returned routine id as the first argument to `c_proc` when you wish to call the

routine from Euphoria.

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

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

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

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

- E\_INTEGER = #06000004
- E\_ATOM = #07000004
- E\_SEQUENCE = #08000004
- E\_OBJECT = #09000004

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

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

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

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

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

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

### Example 1:

```
atom user32
integer LoadIcon

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

-- It takes a C pointer and a C int as parameters.
-- It returns a C int as a result.
LoadIcon = define_c_func(user32, "LoadIconA",
                        {C_POINTER, C_INT}, C_INT)

-- We use "LoadIconA" here because we know that LoadIconA
-- needs the stdcall convention, as do
-- all standard .dll routines in the WINDOWS API.
-- To specify the cdecl convention, we would have used "+LoadIconA".

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

## See Also:

[demo\callmach.ex](#) | [c\\_func](#) | [define\\_c\\_proc](#) | [c\\_proc](#) | [open\\_dll](#)

## c\_func

```
<built-in> function c_func(integer rid, sequence args={})
```

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

## Arguments:

1. rid : an integer, the routine\_id of the external function being called.
2. args : a sequence, the list of parameters to pass to the function

## Returns:

An **object**, whose type and meaning was defined on calling [define\\_c\\_func](#).

## Errors:

If rid is not a valid routine id, or the arguments do not match the prototype of the routine being called, an error occurs.

## Comments:

rid must have been returned by [define\\_c\\_func](#), **not** by [routine\\_id](#). The type checks are different, and you would get a machine level exception in the best case.

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

If you pass an argument value which contains a fractional part, where the C function expects a C integer type, the argument will be rounded towards zero. For example: 5.9 will be passed as 5 and -5.9 will be passed as -5.

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

## Example 1:

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

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

-- the C function BeginPaint takes a C int argument and
-- a C pointer, and returns a C int as a result:
BeginPaint = define_c_func(user32, "BeginPaint",
                          {C_INT, C_POINTER}, C_INT)

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

## See Also:

[c\\_proc](#) | [define\\_c\\_proc](#) | [open\\_dll](#) | [Platform-Specific Issues](#)

## c\_proc

```
<built-in> procedure c_proc(integer rid, sequence args={})
```

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

### Arguments:

1. rid : an integer, the routine\_id of the external function being called.
2. args : a sequence, the list of parameters to pass to the function

### Errors:

If rid is not a valid routine id, or the arguments do not match the prototype of the routine being called, an error occurs.

### Comments:

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

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

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

### Example 1:

```
atom user32, hwnd, rect
integer GetClientRect

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

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

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

### See Also:

[c\\_func](#) | [define\\_c\\_func](#) | [open\\_dll](#) | [Platform-Specific Issues](#)

## call\_back

```
include std/dll.e
namespace dll
public function call_back(object id)
```

gets a machine address for an Euphoria procedure.

### Arguments:

1. id : an object, either the id returned by `routine_id` (for the function or procedure), or a pair {'+', id}.

### Returns:

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

### Errors:

The length of name should not exceed 1\_024 characters.

### Comments:

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

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 or FreeBSD signal function. For example, you might want to catch the SIGTERM signal, and do a graceful shutdown. Some Web hosts send a SIGTERM to a CGI process that has used too much CPU time.

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

### Example 1:

See: .../euphoria/demo/win32/window.exw

### Example 2:

See: .../euphoria/demo/linux/qsor.sort.ex

### See Also:

[routine\\_id](#)

# Errors and Warnings

## Subroutines

### crash

```
include std/error.e
namespace error
public procedure crash(sequence fmt, object data = {})
```

crashes the running program and displays a formatted error message.

#### Arguments:

1. `fmt` : a sequence representing the message text. It may have format specifiers in it
2. `data` : an object, defaulted to {}.

#### Comments:

Formatting is the same as with `printf`.

The actual message being shown, both on standard error and in `ex.err` (or whatever file last passed to `crash_file`), is `sprintf(fmt, data)`. The program terminates as for any runtime error.

#### Example 1:

```
if PI = 3 then
  crash("The structure of universe just changed -- reload solar_system.ex")
end if
```

#### Example 2:

```
if token = end_of_file then
  crash("Test file #%d is bad, text read so far is %s\n",
        {file_number, read_so_far})
end if
```

#### See Also:

[crash\\_file](#) | [crash\\_message](#) | [printf](#)

### crash\_message

```
include std/error.e
namespace error
public procedure crash_message(sequence msg)
```

specifies a final message to be displayed to your user, in the event that Euphoria has to shut down your program due to an error.

#### Arguments:

1. `msg` : a sequence to display. It must only contain printable characters.

#### Comments:



There can be as many calls to `crash_message` as needed in a program. Whatever was defined last will be used in case of a runtime error.

### Example 1:

```
crash_message("The password you entered must have at least 8 characters.")
pwd_key = input_text[1..8]
-- if ##input_text## is too short,
-- user will get a more meaningful message than
-- "index out of bounds".
```

### See Also:

[crash](#) | [crash\\_file](#)

## crash\_file

```
include std/error.e
namespace error
public procedure crash_file(sequence file_path)
```

specifies a file path name in place of "ex.err" where you want any diagnostic information to be written.

### Arguments:

1. `file_path` : a sequence, the new error and traceback file path.

### Comments:

There can be as many calls to `crash_file` as needed. Whatever was defined last will be used in case of an error at runtime, whether it was triggered by [crash](#) or not.

### See Also:

[crash](#) | [crash\\_message](#)

## abort

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

aborts execution of the program.

### Arguments:

1. `error` : an integer, the exit code to return.

### Comments:

`error` is expected to lie in the 0..255 range. Zero is usually interpreted as the sign of a successful completion.

Other values can indicate various kinds of errors. *Windows* batch (.bat) programs can read this value using the `errorlevel` feature. Non integer values are rounded down. A *Euphoria* program can read this value using [system\\_exec](#).

`abort` is useful when a program is many levels deep in subroutine calls, and execution must end immediately, perhaps due to a severe error that has been detected.

If you do not use `abort` then the interpreter will normally return an exit status code of zero. If your program fails with a *Euphoria*-detected compile-time or run-time error then

a code of one is returned.

### Example 1:

```
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](#)

## warning\_file

```
include std/error.e
namespace error
public procedure warning_file(object file_path)
```

specifies a file path where to output warnings.

### Arguments:

1. `file_path` : an object indicating where to dump any warning that were produced.

### Comments:

By default, warnings are displayed on the standard error, and require pressing the Enter key to keep going. Redirecting to a file enables skipping the latter step and having a console window open, while retaining ability to inspect the warnings in case any was issued.

Any atom `>= 0` causes standard error to be used, thus reverting to default behaviour.

Any atom `< 0` suppresses both warning generation and output. Use this latter in extreme cases only.

On an error, some output to the console is performed anyway, so that whatever warning file was specified is ignored then.

### Example 1:

```
warning_file("warnings.lst")
-- some code
warning_file(0)
-- changed opinion: warnings will go to standard error as usual
```

### See Also:

[without warning](#) | [warning](#)

## warning

```
<built-in> procedure warning(sequence message)
```

causes the specified warning message to be displayed as a regular warning.

### Arguments:

1. message : a double quoted literal string, the text to display.

### Comments:

Writing a library has specific requirements, since the code you write will be mainly used inside code you did not write. It may be desirable then to influence, from inside the library, that code you did not write.

This is what warning, in a limited way, does. It enables to generate custom warnings in code that will include yours. Of course, you can also generate warnings in your own code, for instance as a kind of memo. The `without warning` top level statement disables such warnings.

The warning is issued with the `custom_warning` level. This level is enabled by default, but can be turned off any time.

Using any kind of expression in message will result in a blank warning text.

### Example 1:

```
-- mylib.e
procedure foo(integer n)
  warning("The foo() procedure is obsolete, use bar() instead.")
  ? n
end procedure

-- some_app.exw
include mylib.e
foo(123)
```

will result, when `some_app.exw` is run with warning, in the following text being displayed in the console (terminal) window

```
123
Warning: ( custom_warning ):
The foo() procedure is obsolete, use bar() instead.

Press Enter...
```

### See Also:

[warning\\_file](#) | [without warning](#)

## crash\_routine

```
include std/error.e
namespace error
public procedure crash_routine(integer func)
```

specifies a function to be called when an error takes place at run time.

### Arguments:

1. func : an integer, the `routine_id` of the function to link in.

### Comments:

The supplied function must have only one argument, which should be integer or more general. Defaulted parameters in crash routines are not supported yet.

Euphoria maintains a linked list of routines to execute upon a crash. `crash_routine` adds a new function to the list. The routines defined first are executed last. You cannot unlink a routine once it is linked, nor inspect the crash routine chain.

Currently, the crash routines are pass zero. Future versions may attempt to convey more information to them. If a crash routine returns anything else than zero, the remaining routines in the chain are skipped.

Crash routines are not fully fledged exception handlers, and they cannot resume execution at current or next statement. However, they can read the generated crash file, and might perform any action, including restarting the program.

### Example 1:

```
function report_error(integer dummy)
  mylib:email("maintainer@remote_site.org", "ex.err")
  return 0 and dummy
end function
crash_routine(routine_id("report_error"))
```

### See Also:

[crash\\_file](#) | [routine\\_id](#) | [Debugging and Profiling](#)

# Pseudo Memory

One use is to emulate PBR, such as Euphoria's map and stack types.

## ram\_space

```
include std/eumem.e
namespace eumem
export sequence ram_space
```

The (pseudo) RAM heap space. Use `malloc` to gain ownership to a heap location and `free` to release it back to the system.

## malloc

```
include std/eumem.e
namespace eumem
export function malloc(object mem_struct_p = 1, integer cleanup_p = 1)
```

allocates a block of (pseudo) memory.

### Arguments:

1. `mem_struct_p` : The initial structure (sequence) to occupy the allocated block. If this is an integer, a sequence of zero this long is used. The default is the number one, meaning that the default initial structure is {0}
2. `cleanup_p` : Identifies whether the memory should be released automatically when the reference count for the handle for the allocated block drops to zero, or when passed to delete. If zero, then the block must be freed using the `free` procedure.

### Returns:

A **handle**, to the acquired block. Once you acquire the handle you can use it as needed. Note that if `cleanup_p` is one, then the variable holding the handle must be capable of storing an atom (do not use an integer) as a double floating point value.

### Example 1:

```
my_spot = malloc()
ram_space[my_spot] = my_data
```

## free

```
include std/eumem.e
namespace eumem
export procedure free(atom mem_p)
```

deallocates a block of (pseudo) memory.

### Arguments:

1. `mem_p` : The handle to a previously acquired `ram_space` location.

### Comments:

This allows the location to be used by other parts of your application. You should no longer access this location again because it could be acquired by some other process in your application. This routine should only be called if you passed zero as `cleanup_p` to `malloc`.

### Example 1:

```
my_spot = malloc(1,0)
ram_space[my_spot] = my_data
-- . . . do some processing . . .
free(my_spot)
```

## valid

```
include std/eumem.e
namespace eumem
export function valid(object mem_p, object mem_struct_p = 1)
```

validates a block of (pseudo) memory.

### Arguments:

1. `mem_p` : The handle to a previously acquired `ram_space` location.
2. `mem_struct_p` : If an integer, this is the length of the sequence that should be occupying the `ram_space` location pointed to by `mem_p`.

### Returns:

An **integer**,

0 if either the `mem_p` is invalid or if the sequence at that location is the wrong length.  
1 if the handle and contents are okay.

### Comments:

This can only check the length of the contents at the location. Nothing else is checked at that location.

### Example 1:

```
my_spot = malloc()
ram_space[my_spot] = my_data
. . . do some processing . .
if valid(my_spot, length(my_data)) then
    free(my_spot)
end if
```

# Machine Level Access

## Machine Level Access Summary

Warning: Some of these routines require a knowledge of machine-level programming. You could crash your system!

These routines, along with `peek`, `poke` and `call`, let you access all of the features of your computer. You can read and write to any allowed memory location, and you can create and execute machine code subroutines.

If you are manipulating 32-bit addresses or values, remember to use variables declared as `atom`. The integer type only goes up to 31 bits.

If you choose to call `machine_proc` or `machine_func` directly (to save a bit of overhead) you *must* pass valid arguments or Euphoria could crash.

### Example 1:

- `demo/callmach.ex` -- calling a machine language routine

### peek\_longs

```
include std/machine.e
namespace machine
public function peek_longs(object x)
```

@nodoc

### MAP\_ANONYMOUS

```
include std/machine.e
namespace machine
export constant MAP_ANONYMOUS
```

### MAP\_FAILED

```
include std/machine.e
namespace machine
export constant MAP_FAILED
```

## Safe Mode

### Safe Mode Summary

During the development of your application, you can define the word `SAFE` to cause `machine.e` to use alternative memory functions. These functions are slower but help in the debugging stages. In general, `SAFE` mode should not be enabled during production phases but only for development phases.

To define the word `SAFE` run your application with the `-D SAFE` command line option, or add to the top of your main file:

```
with define safe
```

before the first appearance of `include std/machine.e`

The implementation of the Machine Level Access routines used are controlled with the define word `SAFE`. The use of `SAFE` switches the routines included here to use debugging versions which will allow you to catch all kinds of bugs that might otherwise may not always crash your program where in the line your program is written. There may be bugs that are invisible until you port the program they are in to another platform. There has been no bench marking for how much of a speed penalty there is using `SAFE`.

You can take advantage of `SAFE` debugging by:

- If necessary, call `register_block(address, length, memory_protection)` to add additional "external" blocks of memory to the `safe_address_list`. These are blocks of memory that are safe to use but which you did not acquire through Euphoria's `allocate`, `allocate_data`, `allocate_code` or `allocate_protect`, `allocate_string`, `allocate_wstring`. Call `unregister_block(address)` when you want to prevent further access to an external block. When `SAFE` is not enabled these functions will do nothing and will be converted into nothing by the inline code in the front-end.
- You will be notified if memory that you haven't allocated is accessed, or if memory is freed twice, or if memory is used in the wrong way. Your application will can be ready for D.E.P. enabled systems even if the system you test on doesn't have D.E.P..
- If a bug is caught, you will hear some "beep" sounds. Press Enter to clear the screen and see the error message. There will be a descriptive crash message and a traceback in `ex.err` so you can find the statement that is making the illegal memory access.

## check\_calls

Define block checking policy.

```
include std/machine.e
public integer check_calls
```

### Comments:

If this integer is 1, (the default), check all blocks for edge corruption after each `Executable Memory`, `call`, `c_proc` or `c_func`. To save time, your program can turn off this checking by setting `check_calls` to 0.

## edges\_only

```
include std/machine.e
public integer edges_only
```

Determine whether to flag accesses to remote memory areas.

### Comments:

If this integer is 1 (the default under *Windows*), only check for references to the leader or trailer areas just outside each registered block, and don't complain about addresses that are far out of bounds (it's probably a legitimate block from another source)

For a stronger check, set this to 0 if your program will never read/write an unregistered block of memory.

On *Windows* people often use unregistered blocks. Please do not be one of them.



## check\_all\_blocks

```
include std/machine.e
check_all_blocks()
```

Scans the list of registered blocks for any corruption.

### Comments:

safe.e maintains a list of acquired memory blocks. Those gained through `allocate` are automatically included. Any other block, for debugging purposes, must be registered by `register_block` and unregistered by `unregister_block`.

The list is scanned and, if any block shows signs of corruption, it is displayed on the screen and the program terminates. Otherwise, nothing happens.

Unless `SAFE` is defined, this routine does nothing. It is there to make switching between debugged and normal version of your program easier.

### See Also:

`register_block` | `unregister_block`

## register\_block

```
include std/machine.e
procedure register_block(machine_addr block_addr, positive_int block_len,
    valid_memory_protection_constant memory_protection = PAGE_READ_WRITE )
```

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

### Arguments:

1. `block_addr` : an atom, the start address of the block
2. `block_len` : an integer, the size of the block.
3. `protection` : a constant integer, of the memory protection constants found in machine.e, that describes what access we have to the memory.

### Comments:

In memory.e, this procedure does nothing. It is there to simplify switching between the normal and debug version of the library.

This routine is only meant to be used for debugging purposes. safe.e tracks the blocks of memory that your program is allowed to `peek`, `poke`, `mem_copy` etc. These are normally just the blocks that you have allocated using Euphoria's `allocate` routine, and which you have not yet freed using Euphoria's `free`. In some cases, you may acquire additional, external, blocks of memory, perhaps as a result of calling a C routine.

If you are debugging your program using safe.e, you must register these external blocks of memory or safe.e will prevent you from accessing them. When you are finished using an external block you can unregister it using `unregister_block`.

### Example 1:

```
atom addr

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

[See Also:](#)

[unregister\\_block](#) | [Safe Mode](#)

## unregister\_block

```
include std/machine.e
public procedure unregister_block(machine_addr block_addr)
```

removes a block of memory from the list of safe blocks maintained by safe.e (the debug version of memory.e).

### Arguments:

1. `block_addr` : an atom, the start address of the block

### Comments:

In `memory.e`, this procedure does nothing. It is there to simplify switching between the normal and debug version of the library.

This routine is only meant to be used for debugging purposes. Use it to unregister blocks of memory that you have previously registered using [register\\_block](#). By unregistering a block, you remove it from the list of safe blocks maintained by `safe.e`. This prevents your program from performing any further reads or writes of memory within the block.

See [register\\_block](#) for further comments and an example.

### See Also:

[register\\_block](#) | [Safe Mode](#)

## Data Execute Mode and Data Execute Protection

Data Execute Mode makes data that will be returned from [allocate](#) executable. On some systems you will not be allowed to run code in memory returned from [allocate](#) unless this mode has been enabled. This restriction is called Data Execute Protection or D.E.P.. When writing software you should use [allocate\\_code](#) or [allocate\\_protect](#) to get memory for execution. This is more efficient and more secure than using Data Execute mode. Because many hacker exploits of software use data buffers and then trick software into running this data, Data Execute Protection stops an entire class of exploits.

If you get a Data Execute Protection Exception from running software, it means that D.E.P. could have thwarted an attack! Your application crashes and your computer wasn't infected. However, many people will decide that they want to disable D.E.P. because they know that they call memory returned by [allocate](#) or perhaps they are simply careless.

## Type Sorted Function List

### Executable Memory

Executable Memory is the way to run code on the stack in a completely portable way.

### Use the following Routines:

Use [allocate\\_code](#) to allocate some executable machine-code, [call](#) to call the code, and [free\\_code](#) to free the machine-code.

## Using Data Bytes

In C, bytes are called 'char' or 'BOOL' or 'boolean'. They sometimes are used for very small numbers but mostly, they are used in C-strings. See [Using Strings](#).

Use [allocate\\_data](#) to allocate data and return an address. Use [poke](#) to save atoms or sequences to at an address. Use [peek](#)s or [peek](#) to read from an address. Use [mem\\_set](#) and [mem\\_copy](#) to set and copy sections of memory. Use [free](#) to free or use [delete](#) if you enabled cleanup in [allocate\\_data](#).

## Using Data Words

Words are 16-bit integers and are big enough to hold most integers in common use as far as whole numbers go. So they often are used to hold numbers. In C, they are declared as WORD or short.

Use [allocate\\_data](#) to allocate data and return its address. Use [poke2](#) to write to the data at an address. Use [peek2](#) or [peek2s](#) to read from an address. Use [free](#) to free or use [delete](#) if you enabled cleanup in [allocate\\_data](#).

## Using Data Double Words

Double words are 32-bit integers. In C, they are typically declared as int, or long (on Windows and other 32-bit architectures), or DWORD. They are big enough to hold pointers to other values in memory on 32-bit architectures.

Use [allocate\\_data](#) to allocate data and return its address. Use [poke4](#) to write to the data at an address. Use [peek4](#) or [peek4s](#) to read from an address. Use [free](#) to free or use [delete](#) if you enabled cleanup in [allocate\\_data](#).

## Using Data Quad Words

Quad words are 64-bit integers. In C, they are typically declared as long long int, or long int (on 64-bit architectures other than Windows). They are big enough to hold pointers to other values in memory on 64-bit architectures.

Use [allocate\\_data](#) to allocate data and return its address. Use [poke8](#) to write to the data at an address. Use [peek8u](#) or [peek8s](#) to read from an address. Use [free](#) to free or use [delete](#) if you enabled cleanup in [allocate\\_data](#).

## Using Pointers

A Euphoria atom should be used to store pointer values. On 32-bit architectures, pointers may be larger than a Euphoria integer. On 64-bit architectures, a Euphoria integer is large enough to hold pointer values, since current 64-bit architectures use only a 48-bit memory space

To portably peek and poke pointers, you should use [peek\\_pointer](#) and [poke\\_pointer](#). These routines automatically detect the architecture and use the correct size for a pointer.

## Using Long Integers

When interfacing with C code, some data will be defined as long or long int. This data type

can be tricky to use in a portable manner, due to the way that different architectures and operating systems define it.

On all 32-bit architectures on which Euphoria runs, a long int is defined as 32-bits. On 64-bit Windows, a long int is also 32-bits. However, on other 64-bit operating systems, a long int is defined as 64-bits.

To portably peek and poke long int data, you should use [peek\\_longs](#), [peek\\_longu](#) and [poke\\_long](#). You can also use `sizeof( C_LONG )` to determine the size (in bytes) of a native long int.

## Using Strings

You can create legal ANSI and 16-bit UNICODE Strings with these routines. In C, strings are often declared as some pointer to a character: `char *` or `wchar *`.

Microsoft Windows uses 8-bit ANSI and 16-bit UNICODE in its routines.

Use [allocate\\_string](#) or [allocate\\_wstring](#) to allocate a string pointer. Use [peek\\_string](#), [peek\\_wstring](#), [peek4](#), to read from memory byte strings, word strings and double word strings respectively. Use [poke](#), [poke2](#), or [poke4](#) to write to memory byte strings, word strings and double word strings. Use [free](#) to free or use [delete](#) if you enabled cleanup in [allocate\\_data](#).

## Using Pointer Arrays

Use [allocate\\_string\\_pointer\\_array](#) to allocate a string array from a sequence of strings. Use [allocate\\_pointer\\_array](#) to allocate and then write to an array for pointers . Use [free\\_pointer\\_array](#) to deallocate or use [delete](#) if you enabled cleanup in [allocate\\_data](#).

## Memory Allocation

### allocate

```
include std/machine.e
namespace machine
public function allocate(memory :positive_int n, types :boolean cleanup = 0)
```

This does the same as [allocate\\_data](#) but allows the `DATA_EXECUTE` defined word to cause it to return executable memory.

**See Also:**

[allocate\\_data](#) | [allocate\\_code](#) | [free](#)

### allocate\_data

```
include std/machine.e
namespace machine
public function allocate_data(memory :positive_int n, types :boolean cleanup = 0)
```

allocates a contiguous block of data memory.

**Arguments:**

1. `n` : a positive integer, the size of the requested block.

2. `cleanup` : an integer, if non-zero, then the returned pointer will be automatically freed when its reference count drops to zero, or when passed as a parameter to `delete`.

### Returns:

An **atom**, the address of the allocated memory or 0 if the memory can't be allocated.

**NOTE** you must use either an atom or object to receive the returned value as sometimes the returned memory address is too larger for an integer to hold.

### Comments:

- Since `allocate` acquires memory from the system, it is your responsibility to return that memory when your application is done with it. There are two ways to do that - automatically or manually.
  - *Automatically* - If the `cleanup` parameter is non-zero, then the memory is returned when the variable that receives the address goes out of scope **and** is not referenced by anything else. Alternatively you can force it be released by calling the `delete` function.
  - *Manually* - If the `cleanup` parameter is zero, then you must call the `free` function at some point in your program to release the memory back to the system.
- When your program terminates, the operating system will reclaim all memory that your applicaiton acquired anyway.
- An address returned by this function shouldn't be passed to `call`. For that purpose you should use `allocate_code` instead.
- The address returned will be at least 8-byte aligned.

### Example 1:

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

### See Also:

[Using Data Bytes](#) | [Using Data Words](#) | [Using Data Double Words](#) | [Using Strings](#) | [allocate\\_code](#) | [free](#)

## allocate\_pointer\_array

```
include std/machine.e
namespace machine
public function allocate_pointer_array(sequence pointers, types :boolean cleanup = 0)
```

allocates a NULL terminated pointer array.

### Arguments:

1. `pointers` : a sequence of pointers to add to the pointer array.
2. `cleanup` : an integer, if non-zero, then the returned pointer will be automatically freed when its reference count drops to zero, or when passed as a parameter to `delete`

### Comments:

This function adds the NULL terminator.

### Example 1:

```
atom pa
pa = allocate_pointer_array({ allocate_string("1"), allocate_string("2") })
```

### See Also:

[Using Pointer Arrays](#) | [allocate\\_string\\_pointer\\_array](#) | [free\\_pointer\\_array](#)

## free\_pointer\_array

```
include std/machine.e
namespace machine
public procedure free_pointer_array(atom pointers_array)
```

Free a NULL terminated pointers array.

### Arguments:

1. pointers\_array : memory address of where the NULL terminated array exists at.

### Comments:

This is for NULL terminated lists, such as allocated by [allocate\\_pointer\\_array](#). Do not call `free_pointer_array` for a pointer that was allocated to be cleaned up automatically. Instead, use `delete`.

### See Also:

[allocate\\_pointer\\_array](#) | [allocate\\_string\\_pointer\\_array](#)

## allocate\_string\_pointer\_array

```
include std/machine.e
namespace machine
public function allocate_string_pointer_array(object string_list, types :boolean cleanup = 0)
```

Allocate a C-style null-terminated array of strings in memory

### Arguments:

1. string\_list : sequence of strings to store in RAM.
2. cleanup : an integer, if non-zero, then the returned pointer will be automatically freed when its reference count drops to zero, or when passed as a parameter to `delete`

### Returns:

An **atom**, the address of the memory block where the string pointer array was stored.

### Example 1:

```
atom p = allocate_string_pointer_array({ "One", "Two", "Three" })
-- Same as C: char *p = { "One", "Two", "Three", NULL };
```

### See Also:

[Using Pointer Arrays](#) | [free\\_pointer\\_array](#)

## allocate\_wstring

```
include std/machine.e
namespace machine
public function allocate_wstring(sequence s, types :boolean cleanup = 0)
```

Create a C-style null-terminated `wchar_t` string in memory

### Arguments:

1. `s` : a unicode (utf16) string

### Returns:

An **atom**, the address of the allocated string, or 0 on failure.

### See Also:

[Using Strings](#) | [allocate\\_string](#)

## Reading from Memory

### peek

```
<built-in> function peek(object addr_n_length)
```

fetches a byte, or some bytes, from an address in memory.

### Arguments:

1. `addr_n_length` : an object, either of
  - an atom `addr` -- to fetch one byte at `addr`, or
  - a pair `{addr,len}` -- to fetch `len` bytes at `addr`

### Returns:

An **object**, either an integer if the input was a single address, or a sequence of integers if a sequence was passed. In both cases, integers returned are bytes, in the range 0..255.

### Errors:

**Peeking** in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a EUPHORIA error.

When supplying a `{address, count}` sequence, the count must not be negative.

### Comments:

Since addresses are 32-bit numbers on 32-bit architectures, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several bytes at once using the second form of `peek` than it is to read one byte at a time in a loop. The returned sequence has the length you asked for on input.

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

### Example 1:

```
-- The following are equivalent:
-- first way
s = {peek(100), peek(101), peek(102), peek(103)}

-- second way
s = peek({100, 4})
```

### See Also:

## peeks

```
<built-in> function peeks(object addr_n_length)
```

fetches a byte, or some bytes, from an address in memory.

### Arguments:

1. `addr_n_length` : an object, either of
  - an atom `addr` : to fetch one byte at `addr`, or
  - a pair `{addr,len}` : to fetch `len` bytes at `addr`

### Returns:

An **object**, either an integer if the input was a single address, or a sequence of integers if a sequence was passed. In both cases, integers returned are bytes, in the range - 128..127.

### Errors:

**Peeking** in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a Euphoria error.

When supplying a `{address, count}` sequence, the count must not be negative.

### Comments:

Since addresses are 32-bit numbers on 32-bit architectures, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several bytes at once using the second form of `peek` than it is to read one byte at a time in a loop. The returned sequence has the length you asked for on input.

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

### Example 1:

```
-- The following are equivalent:
-- first way
s = {peeks(100), peek(101), peek(102), peek(103)}

-- second way
s = peeks({100, 4})
```

### See Also:

[Using Data Bytes](#) | [poke](#) | [peek4s](#) | [allocate](#) | [free](#) | [peek2s](#), [peek](#)

## peek2s

```
<built-in> function peek2s(object addr_n_length)
```

Fetches a *signed* word, or some *signed* words , from an address in memory.

### Arguments:



1. `addr_n_length` : an object, either of
  - an atom `addr` -- to fetch one word at `addr`, or
  - a pair { `addr`, `len` }, to fetch `len` words at `addr`

### Returns:

An **object**, either an integer if the input was a single address, or a sequence of integers if a sequence was passed. In both cases, integers returned are double words, in the range -32768..32767.

### Errors:

Peeking in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a EUPHORIA error.

When supplying a {address, count} sequence, the count must not be negative.

### Comments:

Since addresses are 32-bit numbers on 32-bit architectures, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several words at once using the second form of `peek` than it is to read one word at a time in a loop. The returned sequence has the length you asked for on input.

Remember that `peek2s` takes just one argument, which in the second form is actually a 2-element sequence.

The only difference between `peek2s` and `peek2u` is how words with the highest bit set are returned. `peek2s` assumes them to be negative, while `peek2u` just assumes them to be large and positive.

### Example 1:

```
-- The following are equivalent:
-- first way
s = {peek2s(100), peek2s(102), peek2s(104), peek2s(106)}

-- second way
s = peek2s({100, 4})
```

### See Also:

Using Data Words | [poke2](#) | [peeks](#) | [peek4s](#) | [allocate](#) | [free](#) | [peek2u](#)

## peek2u

```
<built-in> function peek2u(object addr_n_length)
```

fetches an *unsigned* word, or some *unsigned* words, from an address in memory.

### Arguments:

1. `addr_n_length` : an object, either of
  - an atom `addr` -- to fetch one double word at `addr`, or
  - a pair {`addr`,`len`} -- to fetch `len` double words at `addr`

### Returns:

An **object**, either an integer if the input was a single address, or a sequence of integers if a sequence was passed. In both cases, integers returned are words, in the range

0..65535.

## Errors:

Peeking in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a Euphoria error.

When supplying a {address, count} sequence, the count must not be negative.

## Comments:

Since addresses are 32-bit numbers on 32-bit architectures, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several words at once using the second form of peek than it is to read one word at a time in a loop. The returned sequence has the length you asked for on input.

Remember that peek2u takes just one argument, which in the second form is actually a 2-element sequence.

The only difference between peek2s and peek2u is how words with the highest bit set are returned. peek2s assumes them to be negative, while peek2u just assumes them to be large and positive.

## Example 1:

```
-- The following are equivalent:
-- first way
Get 4 2-byte numbers starting address 100.
s = {peek2u(100), peek2u(102), peek2u(104), peek2u(106)}

-- second way
Get 4 2-byte numbers starting address 100.
s = peek2u({100, 4})
```

## See Also:

Using Data Words | [poke2](#) | [peek](#) | [peek2s](#) | [allocate](#) | [free](#) | [peek4u](#)

## peek4s

```
<built-in> function peek4s(object addr_n_length)
```

fetches a *signed* double words, or some *signed* double words, from an address in memory.

## Arguments:

1. addr\_n\_length : an object, either of
  - an atom addr -- to fetch one double word at addr, or
  - a pair { addr, len } -- to fetch len double words at addr

## Returns:

An **object**, either an atom if the input was a single address, or a sequence of atoms if a sequence was passed. In both cases, atoms returned are double words, in the range -  $(2^{31})..2^{31}-1$ .

## Errors:

Peeking in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a Euphoria error.

When supplying a {address, count} sequence, the count must not be negative.

### Comments:

Since addresses are 32-bit numbers on 32-bit architectures, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several double words at once using the second form of peek than it is to read one double word at a time in a loop. The returned sequence has the length you asked for on input.

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

The only difference between peek4s and peek4u is how double words with the highest bit set are returned. peek4s assumes them to be negative, while peek4u just assumes them to be large and positive.

### Example 1:

```
-- The following are equivalent:
-- first way
s = {peek4s(100), peek4s(104), peek4s(108), peek4s(112)}

-- second way
s = peek4s({100, 4})
```

### See Also:

Using Data Double Words | poke4 | peeks | peek4u | allocate | free | peek2s

## peek8s

```
<built-in> function peek8s(object addr_n_length)
```

fetches a *signed* quad words, or some *signed* quad words, from an address in memory.

### Arguments:

1. addr\_n\_length : an object, either of
  - an atom addr -- to fetch one double word at addr, or
  - a pair { addr, len } -- to fetch len quad words at addr

### Returns:

An **object**, either an atom if the input was a single address, or a sequence of atoms if a sequence was passed. In both cases, atoms returned are quad words, in the range - power(2,63)..power(2,63)-1.

### Errors:

Peeking in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a Euphoria error.

When supplying a {address, count} sequence, the count must not be negative.

### Comments:

Since addresses are 32-bit numbers on 32-bit architectures, 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 quad words at once using the second form of peek than it is to read one quad word at a time in a loop. The returned sequence has the length you asked for on input.

Remember that peek8s takes just one argument, which in the second form is actually a 2-element sequence.

The only difference between peek8s and peek8u is how quad words with the highest bit set are returned. peek4s assumes them to be negative, while peek4u just assumes them to be large and positive.

### Example 1:

```
-- The following are equivalent:
-- first way
s = {peek8s(100), peek8s(108), peek8s(116), peek8s(124)}

-- second way
s = peek8s({100, 4})
```

### See Also:

[Using Data Double Words](#) | [poke4](#) | [peeks](#) | [peek4u](#) | [allocate](#) | [free](#) | [peek2s](#)

## peek4u

```
<built-in> function peek4u(object addr_n_length)
```

fetches an *unsigned* double word, or some *unsigned* double words, from an address in memory.

### Arguments:

1. addr\_n\_length : an object, either of
  - an atom addr -- to fetch one double word at addr, or
  - a pair {addr,len} -- to fetch len double words at addr

### Returns:

An **object**, either an atom if the input was a single address, or a sequence of atoms if a sequence was passed. In both cases, atoms returned are double words, in the range  $0..2^{32}-1$ .

### Errors:

Peeking in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a Euphoria error.

When supplying a {address, count} sequence, the count must not be negative.

### Comments:

Since addresses are 32-bit numbers on 32-bit architectures, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several double words at once using the second form of peek than it is to read one double word at a time in a loop. The returned sequence has the length you

asked for on input.

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

The only difference between `peek4s` and `peek4u` is how double words with the highest bit set are returned. `peek4s` assumes them to be negative, while `peek4u` just assumes them to be large and positive.

### Example 1:

```
-- The following are equivalent:
-- first way
s = {peek4u(100), peek4u(104), peek4u(108), peek4u(112)}

-- second way
s = peek4u({100, 4})
```

### See Also:

[Using Data Double Words](#) | [poke4](#) | [peek](#) | [peek4s](#) | [allocate](#) | [free](#) | [peek2u](#)

## peek8u

```
<built-in> function peek8u(object addr_n_length)
```

fetches an *unsigned* quad word, or some *unsigned* quad words, from an address in memory.

### Arguments:

1. `addr_n_length` : an object, either of
  - an atom `addr` -- to fetch one double word at `addr`, or
  - a pair `{addr,len}` -- to fetch `len` double words at `addr`

### Returns:

An **object**, either an atom if the input was a single address, or a sequence of atoms if a sequence was passed. In both cases, atoms returned are quad words, in the range `0..power(2,64)-1`.

### Errors:

Peeking in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a Euphoria error.

When supplying a `{address, count}` sequence, the count must not be negative.

### Comments:

Since addresses are 32-bit numbers on 32-bit architectures, 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 quad words at once using the second form of `peek` than it is to read one quad word at a time in a loop. The returned sequence has the length you asked for on input.

Remember that `peek8u` takes just one argument, which in the second form is actually a 2-element sequence.

The only difference between `peek8s` and `peek8u` is how quad words with the highest bit set are returned. `peek8s` assumes them to be negative, while `peek8u` just assumes them

to be large and positive.

### Example 1:

```
-- The following are equivalent:
--first way
s = {peek8u(100), peek8u(108), peek8u(116), peek8u(124)}

-- second way
s = peek8u({100, 4})
```

### See Also:

Using Data Double Words | [poke4](#) | [peek](#) | [peek4s](#) | [allocate](#) | [free](#) | [peek2u](#)

## peek\_longu

```
<built-in> function peek_longu(object addr_n_length)
```

fetches an *unsigned* integer, or some *unsigned* integers, from an address in memory.

### Arguments:

1. `addr_n_length` : an object, either of
  - an atom `addr` -- to fetch one double word at `addr`, or
  - a pair `{addr,len}` -- to fetch `len` double words at `addr`

### Returns:

An **object**, either an atom if the input was a single address, or a sequence of atoms if a sequence was passed. In both cases, atoms returned are based on the native size of a "long int." On *Windows* and all other 32-bit architectures, the number will be in the range  $0..power(2,32)-1$ . On other 64-bit architectures, the number will be in the range of  $0..power(2,64)-1$ .

### Errors:

Peeking in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a *Euphoria* error.

When supplying a `{address, count}` sequence, the count must not be negative.

### Comments:

Since addresses are 32-bit numbers on 32-bit architectures, 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 integers at once using the second form of `peek` than it is to read one integer at a time in a loop. The returned sequence has the length you asked for on input.

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

The only difference between `peek longs` and `peek_longu` is how double words with the highest bit set are returned. `peek4s` assumes them to be negative, while `peek_longu` just assumes them to be large and positive.

### Example 1:

```
-- The following are equivalent (on a 32-bit architecture, or Windows):
```

```
-- first way
s = {peek_longu(100), peek4u(104), peek4u(108), peek4u(112)}

-- second way
s = peek_longu({100, 4})
```

## See Also:

Using Data Double Words | [poke4](#) | [peek](#) | [peek4s](#) | [allocate](#) | [free](#) | [peek2u](#) | [peek2s](#) | [peek8u](#) | [peek8s](#) | [peek longs](#) | [poke\\_long](#)

## peek\_string

```
<built-in> function peek_string(atom addr)
```

reads an ASCII string in RAM, starting from a supplied address.

## Arguments:

1. `addr` : an atom, the address at which to start reading.

## Returns:

A **sequence**, of bytes, the string that could be read.

## Errors:

Further, peeking in memory that does not belong to your process is something the operating system could prevent, and you'd crash with a machine level exception.

## Comments:

An ASCII string is any sequence of bytes and ends with a 0 byte. If you `peek_string` at some place where there is no string, you will get a sequence of garbage.

## See Also:

Using Strings | [peek](#) | [peek\\_wstring](#) | [allocate\\_string](#)

## peek\_pointer

```
<built-in> function peek_pointer(object addr_n_length)
```

## peek\_wstring

```
include std/machine.e
namespace machine
public function peek_wstring(atom addr)
```

returns a unicode (utf16) string that are stored at machine address `a`.

## Arguments:

1. `addr` : an atom, the address of the string in memory

## Returns:

The **string**, at the memory position. The terminator is the null word (two bytes equal to

0).

### See Also:

[Using Strings](#) | [peek\\_string](#)

## Writing to Memory

### poke

```
<built-in> procedure poke(atom addr, object x)
```

stores one or more bytes, starting at a memory location.

### Arguments:

1. `addr` : an atom, the address at which to store
2. `x` : an object, either a byte or a non empty sequence of bytes.

### Errors:

Poking in memory you do not own may be blocked by the OS, and cause a machine exception. The `-D SAFE` option will make `poke` catch this sort of issues.

### Comments:

The lower 8-bits of each byte value (such as `remainder(x, 256)`) is actually stored in memory.

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

Writing to the screen memory with `poke` can be much faster than using `puts` or `printf`, but the programming is more difficult. In most cases the speed is not needed. For example, the Euphoria editor, `ed`, never uses `poke`.

### Example 1:

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

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

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

### Example 2:

`demo/callmach.ex`

### See Also:

[Using Data Bytes](#) | [peek](#) | [peeks](#) | [poke4](#) | [allocate](#) | [free](#) | [poke2](#) | [mem\\_copy](#) | [mem\\_set](#)

### poke2

```
<built-in> procedure poke2(atom addr, object x)
```

stores one or more words, starting at a memory location.



## Arguments:

1. `addr` : an atom, the address at which to store
2. `x` : an object, either a word or a non empty sequence of words.

## Errors:

Poking in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a Euphoria error.

## Comments:

There is no point in having `poke2s` or `poke2u`. For example, both 32768 and -32768 are stored as #F000 when stored as words. It is up to whoever reads the value to figure it out.

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

Writing to the screen memory with `poke2` can be much faster than using `puts` or `printf`, but the programming is more difficult. In most cases the speed is not needed. For example, the Euphoria editor, `ed`, never uses `poke2`.

The 2-byte values to be stored can be negative or positive. You can read them back with either `peek2s` or `peek2u`. Actually, only `remainder(x,65536)` is being stored.

## Example 1:

```
a = allocate(100)  -- allocate 100 bytes in memory

-- poke one 2-byte value at a time:
poke2(a, 12345)
poke2(a+2, #FF00)
poke2(a+4, -12345)

-- poke 3 2-byte values at once:
poke2(a, {12345, #FF00, -12345})
```

## See Also:

[Using Data Words](#) | [peek2s](#) | [peek2u](#) | [poke](#) | [poke4](#) | [allocate](#) | [free](#)

## poke4

```
<built-in> procedure poke4(atom addr, object x)
```

stores one or more double words, starting at a memory location.

## Arguments:

1. `addr` : an atom, the address at which to store
2. `x` : an object, either a double word or a non empty sequence of double words.

## Errors:

Poking in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a Euphoria error.

## Comments:

There is no point in having `poke4s` or `poke4u`. For example, both  $+2^{31}$  and  $-(2^{31})$  are

stored as #F0000000. It is up to whoever reads the value to figure it out.

It is faster to write several double words at once by poking a sequence of values, than it is to write one double words at a time in a loop.

Writing to the screen memory with poke4 can be much faster than using puts or printf, but the programming is more difficult. In most cases the speed is not needed. For example, the Euphoria editor, ed, never uses poke4.

The 4-byte values to be stored can be negative or positive. You can read them back with either peek4s or peek4u. However, the results are unpredictable if you want to store values with a fractional part or a magnitude greater than  $2^{32}$ , even though Euphoria represents them all as atoms.

### Example 1:

```
a = allocate(100)  -- allocate 100 bytes in memory

-- poke one 4-byte value at a time:
poke4(a, 9712345)
poke4(a+4, #FF00FF00)
poke4(a+8, -12345)

-- poke 3 4-byte values at once:
poke4(a, {9712345, #FF00FF00, -12345})
```

### See Also:

[Using Data Double Words](#) | [peek4s](#) | [peek4u](#) | [poke](#) | [poke2](#) | [allocate](#) | [free](#) | [call](#)

## poke8

```
<built-in> procedure poke8(atom addr, object x)
```

stores one or more quad words, starting at a memory location.

### Arguments:

1. addr : an atom, the address at which to store
2. x : an object, either a quad word or a non empty sequence of double words.

### Errors:

Poking in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a Euphoria error.

### Comments:

There is no point in having poke8s or poke8u. For example, both +power(2,63) and -power(2,63) are stored as #F000000000000000. It is up to whoever reads the value to figure it out.

It is faster to write several quad words at once by poking a sequence of values, than it is to write one quad words at a time in a loop.

The 8-byte values to be stored can be negative or positive. You can read them back with either peek8s or peek8u. However, the results are unpredictable if you want to store values with a fractional part or a magnitude greater than power(2,64), even though Euphoria represents them all as atoms.

### Example 1:

```

a = allocate(100)  -- allocate 100 bytes in memory

-- poke one 8-byte value at a time:
poke8(a, 9712345)
poke8(a+8, #FF00FF00)
poke8(a+16, -12345)

-- poke 3 8-byte values at once:
poke8(a, {9712345, #FF00FF00, -12345})

```

## See Also:

[Using Data Double Words](#) | [peek4s](#) | [peek4u](#) | [poke](#) | [poke2](#) | [allocate](#) | [free](#) | [call](#)

## poke\_long

```
<built-in> procedure poke_long(atom addr, object x)
```

stores one or more integers, starting at a memory location.

## Arguments:

1. *addr* : an atom, the address at which to store
2. *x* : an object, either an integer or a non empty sequence of double words.

## Errors:

Poking in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a Euphoria error.

## Comments:

There is no point in having `poke_longs` or `poke_longu`. For example, both `+power(2,31)` and `-power(2,31)` are stored as `#F0000000` on a 32-bit architecture. It is up to whoever reads the value to figure it out.

On all *Windows* and other 32-bit operating systems, the `poke_long` uses 4-byte integers. On 64-bit architectures using operating systems other than *Windows*, `poke_long` uses 8-byte integers.

It is faster to write several integers at once by poking a sequence of values, than it is to write one double words at a time in a loop.

The 4-byte (or 8-byte) values to be stored can be negative or positive. You can read them back with either `peek_longs` or `peek_longu`. However, the results are unpredictable if you want to store values with a fractional part or a magnitude greater than the size of a native long int, even though Euphoria represents them all as atoms.

## Example 1:

```

a = allocate(100)  -- allocate 100 bytes in memory

-- poke one 4-byte value at a time (on Windows or other 32-bit operating system):
poke_long(a, 9712345)
poke_long(a+4, #FF00FF00)
poke_long(a+8, -12345)

-- poke 3 long int values at once:
poke_long(a, {9712345, #FF00FF00, -12345})

```

## See Also:

## poke\_pointer

```
<built-in> procedure poke_pointer(atom addr, object x)
```

stores one or more pointers, starting at a memory location.

### Arguments:

1. `addr` : an atom, the address at which to store
2. `x` : an object, either an integer or a non empty sequence of pointers.

### Errors:

Poking in memory you do not own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a Euphoria error.

### Comments:

There is no point in having `poke_pointers` or `poke_pointersu`. For example, both `+power(2,31)` and `-power(2,31)` are stored as `#F0000000` on a 32-bit architecture. It is up to whoever reads the value to figure it out.

On all 32-bit operating systems, the `poke_pointer` uses 4-byte integers. On 64-bit architectures using operating systems, `poke_pointer` uses 8-byte integers.

It is faster to write several pointers at once by poking a sequence of values, than it is to write one double words at a time in a loop.

The 4-byte (or 8-byte) values to be stored can be negative or positive. You can read them back with either `peek_pointer` or any other `peek` function of the correct size. However, the results are unpredictable if you want to store values with a fractional part or a magnitude greater than the size of a native pointer, even though Euphoria represents them all as atoms.

### Example 1:

```
a = allocate(100)  -- allocate 100 bytes in memory

-- poke one 4-byte value at a time (on a 32-bit operating system):
poke_pointer(a, 9712345)
poke_pointer(a+4, #FF00FF00)
poke_pointer(a+8, -12345)

-- poke 3 long int values at once:
poke_pointer(a, {9712345, #FF00FF00, -12345})
```

### See Also:

[Using Data Double Words](#) | [peek4s](#) | [peek4u](#) | [peek8u](#) | [peek8s](#) | [peek\\_pointer](#) | [poke](#) | [poke2](#) | [allocate](#) | [free](#) | [call](#)

## poke\_string

```
include std/machine.e
namespace machine
public function poke_string(atom buffaddr, integer bufsize, sequence s)
```

Stores a C-style null-terminated ANSI string in memory

## Arguments:

1. `buffaddr`: an atom, the RAM address to to the string at.
2. `buffsize`: an integer, the number of bytes available, starting from `buffaddr`.
3. `s`: a sequence, the string to store at `buffaddr`.

## Comments:

- This does not allocate an RAM. You must supply the preallocated area.
- This can only be used on ANSI strings. It cannot be used for double-byte strings.
- If `s` is not a string, nothing is stored and a zero is returned.

## Returns:

An atom. If this is zero, then nothing was stored, otherwise it is the address of the first byte after the stored string.

## Example 1:

```
atom title

title = allocate(1000)
if poke_string(title, 1000, "The Wizard of Oz") then
  -- successful
else
  -- failed
end if
```

## See Also:

Using Strings | [allocate](#) | [allocate\\_string](#)

## poke\_wstring

```
include std/machine.e
namespace machine
public function poke_wstring(atom buffaddr, integer buffsize, sequence s)
```

stores a C-style null-terminated Double-Byte string in memory.

## Arguments:

1. `buffaddr`: an atom, the RAM address to to the string at.
2. `buffsize`: an integer, the number of bytes available, starting from `buffaddr`.
3. `s`: a sequence, the string to store at `buffaddr`.

## Comments:

- This does not allocate an RAM. You must supply the preallocated area.
- This uses two bytes per string character. **Note** that `buffsize` is the number of *bytes* available in the buffer and not the number of *characters* available.
- If `s` is not a double-byte string, nothing is stored and a zero is returned.

## Returns:

An atom. If this is zero, then nothing was stored, otherwise it is the address of the first byte after the stored string.

## Example 1:

```
atom title

title = allocate(1000)
if poke_wstring(title, 1000, "The Wizard of Oz") then
```

```
-- successful
else
  -- failed
end if
```

### See Also:

[Using Strings](#) | [allocate](#) | [allocate\\_wstring](#)

## Memory Manipulation

### mem\_copy

```
<built-in> procedure mem_copy(atom destination, atom origin, integer len)
```

copies a block of memory from an address to another.

#### Arguments:

1. destination : an atom, the address at which data is to be copied
2. origin : an atom, the address from which data is to be copied
3. len : an integer, how many bytes are to be copied.

#### Comments:

The bytes of memory will be copied correctly even if the block of memory at destination overlaps with the block of memory at origin.

`mem_copy(destination, origin, len)` is equivalent to: `poke(destination, peek({origin, len}))` but is much faster.

#### Example 1:

```
dest = allocate(50)
src = allocate(100)
poke(src, {1,2,3,4,5,6,7,8,9})
mem_copy(dest, src, 9)
```

### See Also:

[Using Data Bytes](#) | [mem\\_set](#) | [peek](#) | [poke](#) | [allocate](#) | [free](#)

### mem\_set

```
<built-in> procedure mem_set(atom destination, integer byte_value, integer how_many))
```

sets a contiguous range of memory locations to a single value.

#### Arguments:

1. destination : an atom, the address starting the range to set.
2. byte\_value : an integer, the value to copy at all addresses in the range.
3. how\_many : an integer, how many bytes are to be set.

#### Comments:

The low order 8 bits of `byte_value` are actually stored in each byte. `mem_set(destination, byte_value, how_many)` is equivalent to: `poke(destination, repeat(byte_value, how_many))` but is much faster.

### Example 1:

```
destination = allocate(1000)
mem_set(destination, ' ', 1000)
-- 1000 consecutive bytes in memory will be set to 32
-- (the ASCII code for ' ')
```

### See Also:

[Using Data Bytes](#) | [peek](#) | [poke](#) | [allocate](#) | [free](#) | [mem\\_copy](#)

## Calling Into Memory

### call

```
<built-in> procedure call(atom addr)
```

calls a machine language routine which was stored in memory prior.

### Arguments:

1. `addr` : an atom, the address at which to transfer execution control.

### Comments:

The machine code routine must execute a RET instruction #C3 to return control to Euphoria. The routine should save and restore any registers that it uses.

You can allocate a block of memory for the routine and then poke in the bytes of machine code using `allocate_code`. You might allocate other blocks of memory for data and parameters that the machine code can operate on using `allocate`. The addresses of these blocks could be part of the machine code.

If your machine code uses the stack, use `c_proc` instead of `call`.

### Example 1:

demo/callmach.ex

### See Also:

[Executable Memory](#) | [allocate\\_code](#) | [free\\_code](#) | [c\\_proc](#) | [define\\_c\\_proc](#)

## Allocating and Writing to memory:

### allocate\_code

```
include std/machine.e
namespace machine
public function allocate_code(object data, memconst :valid_wordsize wordsize = 1)
```

allocates and copies data into executable memory.

### Arguments:

1. `a_sequence_of_machine_code` : is the machine code to be put into memory to be later called with `call`
2. `the word length` : of the said code. You can specify your code as 1-byte, 2-byte or 4-byte chunks if you wish. If your machine code is byte code specify 1. The default is 1.

## Returns:

An **address**, The function returns the address in memory of the code, that can be safely executed whether DEP is enabled or not or 0 if it fails. On the other hand, if you try to execute a code address returned by `allocate` with DEP enabled the program will receive a machine exception.

## Comments:

Use this for the machine code you want to run in memory. The copying is done for you and when the routine returns the memory may not be readable or writeable but it is guaranteed to be executable. If you want to also write to this memory **after the machine code has been copied** you should use `allocate_protect` instead and you should read about having memory executable and writeable at the same time is a bad idea. You mustn't use `free` on memory returned from this function. You may instead use `free_code` but since you will probably need the code throughout the life of your program's process this normally is not necessary. If you want to put only data in the memory to be read and written use `allocate`.

## See Also:

[Executable Memory](#) | [allocate](#) | [free\\_code](#) | [allocate\\_protect](#)

## allocate\_string

```
include std/machine.e
namespace machine
public function allocate_string(sequence s, types :boolean cleanup = 0)
```

Allocate a C-style null-terminated string in memory

## Arguments:

1. `s` : a sequence, the string to store in RAM.
2. `cleanup` : an integer, if non-zero, then the returned pointer will be automatically freed when its reference count drops to zero, or when passed as a parameter to `delete`.

## Returns:

An **atom**, the address of the memory block where the string was stored, or 0 on failure.

## Comments:

Only the 8 lowest bits of each atom in `s` is stored. Use `allocate_wstring` for storing double byte encoded strings.

There is no `allocate_string_low` function. However, you could easily craft one by adapting the code for `allocate_string`.

Since `allocate_string` allocates memory, you are responsible to `free` the block when done with it if `cleanup` is zero. If `cleanup` is non-zero, then the memory can be freed by calling `delete`, or when the pointer's reference count drops to zero.

## Example 1:

```
atom title

title = allocate_string("The Wizard of Oz")
```

## See Also:

[Using Strings](#) | [allocate](#) | [allocate\\_wstring](#)



## allocate\_protect

```
include std/machine.e
namespace machine
public function allocate_protect(object data, memconst :valid_wordsize wordsize = 1,
    valid_memory_protection_constant protection)
```

Allocates and copies data into memory and gives it protection using [Standard Library Memory Protection Constants](#) or [Microsoft Windows Memory Protection Constants](#). The user may only pass in one of these constants. If you only wish to execute a sequence as machine code use `allocate_code`. If you only want to read and write data into memory use `allocate`.

See [MSDN: Microsoft's Memory Protection Constants](#)

### Arguments:

1. `data` : is the machine code to be put into memory.
2. `wordsize` : is the size each element of data will take in memory. Are they 1-byte, 2-bytes, 4-bytes or 8-bytes long? Specify here. The default is 1.
3. `protection` : is the particular *Windows* protection.

### Returns:

An **address**, The function returns the address to the required memory or 0 if it fails. This function is guaranteed to return memory on the 8 byte boundary. It also guarantees that the memory returned with at least the protection given (but you may get more).

If you want to call `allocate_protect( data, PAGE_READWRITE )`, you can use `allocate` instead. It is more efficient and simpler.

If you want to call `allocate_protect( data, PAGE_EXECUTE )`, you can use `allocate_code` instead. It is simpler.

You must not use `free` on memory returned from this function, instead use `free_code`.

### See Also:

[Executable Memory](#)

## Memory Disposal

### free

```
include std/machine.e
namespace machine
public procedure free(object addr)
```

frees up a previously allocated block of memory.

### Arguments:

1. `addr`, either a single atom or a sequence of atoms; these are addresses of a blocks to free.

### Comments:

- Use `free` to return blocks of memory the 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.
- `addr` must have been allocated previously using `allocate`. You cannot use it to relinquish part of a block. Instead, you have to allocate a block of the new size, copy useful contents from old block there and then `free` the old block.
- If the memory was allocated and automatic cleanup was specified, then do not call `free`

- directly. Instead, use `delete`.
- An addr of zero is simply ignored.

### Example 1:

demo/callmach.ex

### See Also:

[Using Data Bytes](#) | [Using Data Words](#) | [Using Data Double Words](#) | [Using Strings](#) | [allocate\\_data](#) | [free\\_code](#)

## free\_code

```
include std/machine.e
public procedure free_code( atom addr, integer size, valid_wordsize wordsize = 1 )
```

frees up allocated code memory.

### Arguments:

1. `addr` : must be an address returned by `allocate_code` or `allocate_protect`. Do **not** pass memory returned from `allocate` here!
2. `size` : is the length of the sequence passed to `allocate_code` or the size you specified when you called `allocate_protect`.
3. `wordsize`: `valid_wordsize` default = 1

### Comments:

Chances are you will not need to call this function because code allocations are typically public scope operations that you want to have available until your process exits.

### See Also:

[Executable Memory](#) | [allocate\\_code](#) | [free](#)

## Automatic Resource Management

Euphoria objects are automatically garbage collected when they are no longer referenced anywhere. Euphoria also provides the ability to manage resources associated with euphoria objects. These resources could be open file handles, allocated memory, or other euphoria objects. There are two built-in routines for managing these external resources.

## delete\_routine

```
<built-in> function delete_routine( object x, integer rid )
```

associates a routine for cleaning up after a Euphoria object.

### Comments:

`delete_routine` associates a euphoria object with a routine id meant to clean up any allocated resources. It always returns an atom (double) or a sequence, depending on what was passed (integers are promoted to atoms).

The routine specified by `delete_routine` should be a procedure that takes a single parameter, being the object to be cleaned up after. Objects are cleaned up under one of two circumstances. The first is if it's called as a parameter to `delete`. After the call, the association with the delete routine is removed.

The second way for the delete routine to be called is when its reference count is reduced to 0. Before its memory is freed, the delete routine is called. A default delete will be used if the cleanup parameter to one of the `allocate` routines is true.

`delete_routine` may be called multiple times for the same object. In this case, the routines are called in reverse order compared to how they were associated.

## delete

```
<built-in> procedure delete( object x )
```

calls the cleanup routines associated with the object, and removes the association with those routines.

### Comments:

The cleanup routines associated with the object are called in reverse order than they were added. If the object is an integer, or if no cleanup routines are associated with the object, then nothing happens.

After the cleanup routines are called, the value of the object is unchanged, though the cleanup routine will no longer be associated with the object.

## Types and Constants

### std\_library\_address

```
include std/machine.e
namespace machine
public type std_library_address(object addr)
```

an address returned from `allocate` or `allocate_protect` or `allocate_code` or the value 0.

### Returns:

An **integer**, The type will return 1 if the parameter, an address, was returned from one of these Machine Level functions (and has not yet been freed)

### Comments:

This type is equivalent to `atom` unless `SAFE` is defined. Only values that satisfy this type may be passed into `free` or `free_code`.

### valid\_memory\_protection\_constant

```
include std/machine.e
public type valid_memory_protection_constant(object a)
```

protection constants type

### machine\_addr

```
include std/machine.e
public type machine_addr(object a)
```

a 32-bit non-null machine address

## safe\_address

```
include std/machine.e
public function safe_address(machine_addr start, natural len,
                             positive_int action )
```

action is some bitwise-or combination of the following constants: A\_READ, A\_WRITE and A\_EXECUTE.

### Returns:

When **Safe Mode** is turned on, this returns true iff it is ok to perform action all addresses from start to start+len-1.

When **Safe Mode** is not turned on, this always returns true.

### Comments:

This is used mostly inside the safe library itself to check whenever you call Machine Level Access Functions or Procedures. It should only be used for debugging purposes.

## ADDRESS\_LENGTH

```
include std/machine.e
namespace machine
public constant ADDRESS_LENGTH
```

The number of bytes required to hold a pointer.

## PAGE\_SIZE

```
include std/machine.e
namespace machine
public constant PAGE_SIZE
```

The operating system's memory page length in bytes.

## Indirect Routine Calling

## Accessing Euphoria coded routines

## routine\_id

```
<built-in> function routine_id(sequence routine_name)
```

returns an integer id number for a user-defined Euphoria procedure or function.

### Arguments:

1. routine\_name : a string, the name of the procedure or function.

### Returns:

An **integer**, known as a routine id, -1 if the named routine can't be found, else zero or more.

## Errors:

routine\_name should not exceed 1,024 characters.

## Comments:

The id number can be passed to `call_proc` or `call_func`, to indirectly call the routine named by routine\_name. This id depends on the internal process of parsing your code, not on routine\_name.

The routine named routine\_name must be visible (that is callable) at the place where routine\_id is used to get the id number. If it is not, -1 is returned.

Indirect calls to the routine can appear earlier in the program than the definition of the routine, but the id number can only be obtained in code that comes after the definition of the routine - see example 2 below.

Once obtained, a valid routine id can be used at any place in the program to call a routine indirectly via `call_proc` or `call_func`, including at places where the routine is no longer in scope.

Some typical uses of routine\_id are:

1. Creating a subroutine that takes another routine as a parameter. (See Example 2 below)
2. Using a sequence of routine id's to make a case (switch) statement. Using the `switch statement` is more efficient.
3. Setting up an Object-Oriented system.
4. Getting a routine id so you can pass it to `call_back`. (See [Platform-Specific Issues](#))
5. Getting a routine id so you can pass it to `task_create`. (See [Multitasking in Euphoria](#))
6. Calling a routine that is defined later in a program. This is no longer needed from v4.0 onward.

Note that C routines, callable by Euphoria, also have ids, but they cannot be used where routine ids are, because of the different type checking and other technical issues.

## See Also:

[define\\_c\\_proc](#) | [define\\_c\\_func](#)

## Example 1:

```
procedure foo()
    puts(1, "Hello World\n")
end procedure

integer foo_num
foo_num = routine_id("foo")

call_proc(foo_num, {}) -- same as calling foo()
```

## Example 2:

```
function apply_to_all(sequence s, integer f)
    -- apply a function to all elements of a sequence
    sequence result
    result = {}
    for i = 1 to length(s) do
        -- we can call add1() here although it comes later in the program
        result = append(result, call_func(f, {s[i]}))
    end for
    return result
end function

function add1(atom x)
```

```

    return x + 1
end function

-- add1() is visible here, so we can ask for its routine id
? apply_to_all({1, 2, 3}, routine_id("add1"))
-- displays {2,3,4}

```

### See Also:

[call\\_proc](#) | [call\\_func](#) | [call\\_back](#) | [define\\_c\\_func](#) | [define\\_c\\_proc](#) | [task\\_create](#) | [Platform-Specific Issues](#) | [Indirect routine calling](#)

## call\_func

```
<built-in> function call_func(integer id, sequence args={})
```

calls the user-defined Euphoria function by routine id.

### Arguments:

1. id : an integer, the routine id of the function to call
2. args : a sequence, the parameters to pass to the function.

### Returns:

The **value**, the called function returns.

### Errors:

If id is negative or otherwise unknown, an error occurs.

If the length of args is not the number of parameters the function takes, an error occurs.

### Comments:

id must be a valid routine id returned by [routine\\_id](#).

args must be a sequence of argument values of length n, where n is the number of arguments required by the called function. Defaulted parameters currently cannot be synthesized while making an indirect call.

If the function with id id does not take any arguments then args should be {}.

### Example 1:

Take a look at the sample program called demo/csort.ex

### See Also:

[call\\_proc](#) | [routine\\_id](#) | [c\\_func](#)

## call\_proc

```
<built-in> procedure call_proc(integer id, sequence args={})
```

calls a user-defined Euphoria procedure by routine id.

### Arguments:

1. id : an integer, the routine id of the procedure to call
2. args : a sequence, the parameters to pass to the function.

### Errors:

If id is negative or otherwise unknown, an error occurs.

If the length of args is not the number of parameters the function takes, an error occurs.

### Comments:

id must be a valid routine id returned by `routine_id`.

args must be a sequence of argument values of length n, where n is the number of arguments required by the called procedure. Defaulted parameters currently cannot be synthesized while making an indirect call.

If the procedure with id id does not take any arguments then args should be {}.

### Example 1:

```

public integer foo_id

procedure x()
    call_proc(foo_id, {1, "Hello World\n"})
end procedure

procedure foo(integer a, sequence s)
    puts(a, s)
end procedure

foo_id = routine_id("foo")

x()

```

### See Also:

`call_func` | `routine_id` | `c_proc`

## Accessing Euphoria Internals

### machine\_func

<built-in> `function machine_func(integer machine_id, object args={})`

performs a machine-specific operation that returns a value.

### Returns:

Depends on the called internal facility.

### Comments:

This function is mainly used by the standard library files to implement machine dependent operations such as graphics and sound effects. This routine should normally be called indirectly via one of the library routines in a Euphoria include file. User programs normally do not need to call `machine_func`.

A direct call might cause a machine exception if done incorrectly.

### See Also:

`machine_proc`

## machine\_proc

```
<built-in> procedure machine_proc(integer machine_id, object args={})
```

perform a machine-specific operation that does not return a value.

### Comments:

This procedure is mainly used by the standard library files to implement machine dependent operations such as graphics and sound effects. This routine should normally be called indirectly via one of the library routines in a Euphoria include file. User programs normally do not need to call `machine_proc`.

A direct call might cause a machine exception if done incorrectly.

### See Also:

`machine_func`



# Memory Constants

## Microsoft Windows Memory Protection Constants

microsoftmemoryprotectionconstants

These constant names are taken right from Microsoft's Memory Protection constants.

### PAGE\_EXECUTE

```
include std/memconst.e
namespace memconst
public constant PAGE_EXECUTE
```

You may run the data in this page

### PAGE\_EXECUTE\_READ

```
include std/memconst.e
namespace memconst
public constant PAGE_EXECUTE_READ
```

You may read or run the data

### PAGE\_EXECUTE\_READWRITE

```
include std/memconst.e
namespace memconst
public constant PAGE_EXECUTE_READWRITE
```

You may run, read or write in this page

### PAGE\_EXECUTE\_WRITECOPY

```
include std/memconst.e
namespace memconst
public constant PAGE_EXECUTE_WRITECOPY
```

You may run or write in this page

### PAGE\_WRITECOPY

```
include std/memconst.e
namespace memconst
public constant PAGE_WRITECOPY
```

You may write to this page.

## PAGE\_READWRITE

```
include std/memconst.e
namespace memconst
public constant PAGE_READWRITE
```

You may read or write in this page.

## PAGE\_READONLY

```
include std/memconst.e
namespace memconst
public constant PAGE_READONLY
```

You may only read data in this page

## PAGE\_NOACCESS

```
include std/memconst.e
namespace memconst
public constant PAGE_NOACCESS
```

You have no access to this page

## Standard Library Memory Protection Constants

Memory Protection Constants are the same constants names and meaning across all platforms yet possibly of different numeric value. They are only necessary for [allocate\\_protect](#)

The constant names are created like this: You have four aspects of protection READ, WRITE, EXECUTE and COPY. You take the word PAGE and you concatenate an underscore and the aspect in the order above. For example: PAGE\_WRITE\_EXECUTE The sole exception to this nomenclature is when you will have no access to the page the constant is called PAGE\_NONE.

## PAGE\_NONE

```
include std/memconst.e
namespace memconst
public constant PAGE_NONE
```

You have no access to this page.

## PAGE\_READ\_EXECUTE

```
include std/memconst.e
namespace memconst
public constant PAGE_READ_EXECUTE
```

You may read or run the data An alias to PAGE\_EXECUTE\_READ

## PAGE\_READ\_WRITE

```
include std/memconst.e
namespace memconst
public constant PAGE_READ_WRITE
```

You may read or write to this page An alias to PAGE\_READWRITE

## PAGE\_READ

```
include std/memconst.e
namespace memconst
public constant PAGE_READ
```

You may only read to this page An alias to PAGE\_READONLY

## PAGE\_READ\_WRITE\_EXECUTE

```
include std/memconst.e
namespace memconst
public constant PAGE_READ_WRITE_EXECUTE
```

You may run, read or write in this page An alias to PAGE\_EXECUTE\_READWRITE

## PAGE\_WRITE\_EXECUTE\_COPY

```
include std/memconst.e
namespace memconst
public constant PAGE_WRITE_EXECUTE_COPY
```

You may run or write to this page. Data will copied for use with other processes when you first write to it.

## PAGE\_WRITE\_COPY

```
include std/memconst.e
namespace memconst
public constant PAGE_WRITE_COPY
```

You may write to this page. Data will copied for use with other processes when you first write to it.

# GRAPHIC

- [graphcst.e](#)
- [graphics.e](#)
- [image.e](#)

# Graphics Constants

## Error Code Constants

### enum

```
include std/graphcst.e
namespace graphcst
public enum
```

BMP\_SUCCESS | BMP\_OPEN\_FAILED | BMP\_UNEXPECTED\_EOF |  
BMP\_UNSUPPORTED\_FORMAT | BMP\_INVALID\_MODE

## video\_config Sequence Accessors

### enum

```
include std/graphcst.e
namespace graphcst
public enum
```

VC\_COLOR | VC\_MODE | VC\_LINES | VC\_COLUMNS | VC\_XPIXELS | VC\_YPIXELS |  
VC\_NCOLORS | VC\_PAGES | VC\_SCRNLINES | VC\_SCRNCOLS

See Also:

[video\\_config](#)

## Colors

### enum

```
include std/graphcst.e
namespace graphcst
public enum
```

BLACK | BLUE | GREEN | CYAN | RED | MAGENTA | BROWN | WHITE | GRAY |  
BRIGHT\_BLUE | BRIGHT\_GREEN | BRIGHT\_CYAN | BRIGHT\_RED | BRIGHT\_MAGENTA  
| YELLOW | BRIGHT\_WHITE

Example 1:

Example 2:

### true\_fgcolor

```
include std/graphcst.e
namespace graphcst
```

```
export sequence true_fgcolor
```

## true\_bgcolor

```
include std/graphcst.e
namespace graphcst
export sequence true_bgcolor
```

## BLINKING

```
include std/graphcst.e
namespace graphcst
public constant BLINKING
```

Add to color number to get blinking text on *windows*.

## BYTES\_PER\_CHAR

```
include std/graphcst.e
namespace graphcst
public constant BYTES_PER_CHAR
```

## color

```
include std/graphcst.e
namespace graphcst
public type color(object x)
```

## Subroutines

### mixture

```
include std/graphcst.e
namespace graphcst
public type mixture(object s)
```

Mixture Type

#### Comments:

A **mixture** is a "{red, green, blue} triple of intensities which enables you to define custom colors."

Intensities must be from 0 (weakest) to 63 (strongest).

Thus, the brightest white is {63, 63, 63}.

## video\_config

```
include std/graphcst.e
```

```
namespace graphcst
public function video_config()
```

returns a description of the current video configuration.

### Returns:

A **sequence**, of 10 non-negative integers, laid out as follows:

1. color monitor? -- 0 if monochrome, 1 otherwise
2. current video mode
3. number of text rows in console buffer
4. number of text columns in console buffer
5. screen width in pixels
6. screen height in pixels
7. number of colors
8. number of display pages
9. number of text rows for current screen size
10. number of text columns for current screen size

### Comments:

A public enum is available for convenient access to the returned configuration data:

- VC\_COLOR
- VC\_MODE
- VC\_LINES
- VC\_COLUMNS
- VC\_XPIXELS
- VC\_YPIXELS
- VC\_NCOLORS
- VC\_PAGES
- VC\_SCRNLINES
- VC\_SCRNCOLS

This routine makes it easy for you to parameterize a program so it will work in many different graphics modes.

### Example 1:

```
include std/graphcst.e
sequence vc

vc = video_config()
? vc

--> vc could be {1, 3, 300, 132, 0, 0, 32, 8, 37, 90}
-- this is a Windows console

--> vc could be {1, 3, 24, 80, 0, 0, 16, 1, 24, 80}
-- this is a Linux terminal
```

### See Also:

[graphics\\_mode](#)

## Color Set Selection

### enum

```
include std/graphcst.e
namespace graphcst
public enum
```

[FGSET](#) | [BGSET](#)

## FGSET

```
include std/graphcst.e  
namespace graphcst  
FGSET
```

Foreground ( text) set of colors

## BGSET

```
include std/graphcst.e  
namespace graphcst  
BGSET
```

Background set of colors



# Graphics - Cross Platform

## Subroutines

### position

```
<built-in> procedure position(integer row, integer column)
```

#### Arguments:

1. row : an integer, the index of the row to position the cursor on.
2. column : an integer, the index of the column to position the cursor on.

sets the cursor to where the next character will be output.

#### Comments:

Set the cursor to line row, column column, where the top left corner of the screen is line 1 one, column 1 one. The next character displayed on the screen will be printed at this location.

position will report an error if the location is off the screen.

The *windows* console does not check for rows, as the physical height of the console may be vastly less than its logical height.

#### Example 1:

```
position(2,1)
-- the cursor moves to the beginning of the second line from the top

puts(1, '*' )
-- the '*' appears at row two, column one
```

#### See Also:

[get\\_position](#)

### get\_position

```
include std/graphics.e
namespace graphics
public function get_position()
```

returns the current line and column position of the cursor.

#### Returns:

A **sequence**, {line, column}, the current position of the text mode cursor.

#### Comments:

The coordinate system for displaying text is different from the one for displaying pixels. Pixels are displayed such that the top-left is (x=0,y=0) and the first coordinate controls the horizontal, left-right location. In pixel-graphics modes you can display both text and pixels. `get_position` returns the current line and column for the text that you are displaying, not the pixels that you may be plotting.

There is no corresponding routine for getting the current pixel position, because there is no such thing.

### Example 1:

```

                                include std/graphics.e
position(2,1)
puts(1, '*' )
--
--*
-- ^
-- the current position

? get_position()
--> {2,2}
-- the position 'was' one to the right of the '*'
-- then `{2,2}` was output

```

### See Also:

[position](#)

## text\_color

```

include std/graphics.e
namespace graphics
public procedure text_color(color c)

```

sets the foreground text color.

### Arguments:

1. *c* : the new text color. AddBLINKING to get blinking text in some modes.

### Comments:

Text that you print after calling `text_color` will have the desired color.

When your program terminates, the last color that you selected and actually printed on the screen will remain in effect. Thus you may have to print something, maybe just '\n', in WHITE to restore white text, especially if you are at the bottom line of the screen, ready to scroll up.

### Example 1:

```
text_color(BRIGHT_BLUE)
```

### Example 2:

```

include std/graphics.e

bk_color( BLACK )

integer i=0
while i<=15 do
  for j=1 to 4 do
    text_color( i )
    printf(1, " %2d XXX", i )
    i += 1
  end for
  puts(1, "\n" )
end while

```

```
--> Linux terminal colors are:
```

--> Windows console colors are:

See Also:

[bk\\_color](#) | [clear\\_screen](#)

## bk\_color

```
include std/graphics.e
namespace graphics
public procedure bk_color(color c)
```

sets the background color to one of the sixteen standard colors.

Arguments:

1. *c* : the new text color. AddBLINKING to get blinking text in some modes.

Comments:

To restore the original background color when your program finishes, ( often 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 1:

```
bk_color(BLACK)
```

See Also:

[text\\_color](#)

## console\_colors

```
include std/graphics.e
namespace graphics
public function console_colors(sequence colorset = {})
```

sets the codes for the colors used in `text_color` and `bk_color`.

Arguments:

1. *colorset* : A sequence in one of two formats.
  1. Containing two sets of exactly sixteen color numbers in which the first set are foreground (text) colors and the other set are background colors.
  2. Containing a set of exactly sixteen color numbers. These are to be applied to both foreground and background.

Returns:

A sequence: This contains two sets of sixteen color values currently in use for foreground and background respectively.

## Comments:

- If the colorset is omitted then this just returns the current values without changing anything.
- A color set contains sixteen values. You can access the color value for a specific color by using [X + 1] where 'X' is one of the Euphoria color constants such as RED or BLUE.
- This can be used to change the meaning of the standard color codes for some consoles that are not using standard values. For example, the *Unix* default color value for RED is 1 and BLUE is 4, but you might need this to be swapped. See code Example 1. Another use might be to suppress highlighted (bold) colors. See code Example 2.

### Example 1:

```
sequence cs
cs = console_colors() -- Get the current FG and BG color values.
cs[FGSET][RED + 1] = 4 -- set RED to 4
cs[FGSET][BLUE + 1] = 1 -- set BLUE to 1
cs[BGSET][RED + 1] = 4 -- set RED to 4
cs[BGSET][BLUE + 1] = 1 -- set BLUE to 1
console_colors(cs)
```

### Example 2:

```
-- Prevent highlighted background colors
sequence cs
cs = console_colors()
for i = GRAY + 1 to BRIGHT_WHITE + 1 do
    cs[BGSET][i] = cs[BGSET][i - 8]
end for
console_colors(cs)
```

## See Also:

[text\\_color](#) | [bk\\_color](#)

## wrap

```
include std/graphics.e
namespace graphics
public procedure wrap(object on = 1)
```

determines whether text will wrap when hitting the rightmost column.

## Arguments:

1. on : an object, 0 to truncate text, anything else to wrap.

## Comments:

By default text will wrap.

Use wrap in text modes or pixel-graphics modes when you are displaying long lines of text.

### Example 1:

```
puts(1, repeat('x', 100) & "\n\n")
-- now have a line of 80 'x' followed a line of 20 more 'x'
wrap(0)
puts(1, repeat('x', 100) & "\n\n")
-- creates just one line of 80 'x'
```

## See Also:

[puts](#) | [position](#)

## scroll

```
include std/graphics.e
namespace graphics
public procedure scroll(integer amount, console :positive_int top_line,
    console :positive_int bottom_line)
```

scrolls a region of text on the screen.

### Arguments:

1. amount : an integer, the number of lines by which to scroll. This is  $\geq 0$  to scroll up and  $< 0$  to scroll down.
2. top\_line : the 1-based number of the topmost line to scroll.
3. bottom\_line : the 1-based number of the bottom-most line to scroll.

### Comments:

- New blank lines will appear at the vacated lines.
- You could perform the scrolling operation using a series of calls to [toputs](#), but scroll is much faster.
- The position of the cursor after scrolling is not defined.

### Example 1:

```
.../euphoria/bin/ed.ex
```

### See Also:

[clear\\_screen](#) | [text\\_rows](#)

## Graphics Modes

## graphics\_mode

```
include std/graphics.e
namespace graphics
public function graphics_mode(object m = - 1)
```

attempts to set up a new graphics mode.

### Arguments:

1. x : an object, but it will be ignored.

### Returns:

An **integer**, always returns zero.

### Platform:

*Windows*

### Comments:

- This has no effect on *Unix* platforms.
- On *Windows* it causes a console to be shown if one has not already been created.

### See Also:

video\_config

# Graphics - Image Routines

A **BMP** is "a Microsoft defined standard for a pixel based image file."

## graphics\_point

```
include std/image.e
namespace image
public type graphics_point(object p)
```

## Bitmap Handling

## read\_bitmap

```
include std/image.e
namespace image
public function read_bitmap(sequence file_name)
```

reads a bitmap (.BMP) file into a 2-d sequence of sequences (image)

### Arguments:

1. `file_name` : a sequence, the path to a.bmp file to read from. The extension is not assumed if missing.

### Returns:

An **object**, on success, a sequence of the form {palette,image}. On failure, an error code is returned.

### Comments:

In the returned value, the first element is a list of three membered sequences, each containing three color intensity values in the range 0 to 255, and the second, a list of pixel rows. Each pixel in a row is represented by its color index in the said first element of the return value.

The file should be in the bitmap format. The most common variations of the format are supported.

Bitmaps of 2, 4, 16 or 256 colors are supported. If the file is not in a good format, an error code (atom) is returned instead

```
public constant
    BMP_OPEN_FAILED = 1,
    BMP_UNEXPECTED_EOF = 2,
    BMP_UNSUPPORTED_FORMAT = 3
```

You can create your own bitmap picture files using Windows Paintbrush and many other graphics programs. You can then incorporate these pictures into your Euphoria programs.

### Example 1:

```
x = read_bitmap("c:\\windows\\arcade.bmp")
```

### See Also:

[save\\_bitmap](#)**save\_bitmap**

```
include std/image.e
namespace image
public function save_bitmap(two_seq palette_n_image, sequence file_name)
```

create a .BMP bitmap file, given a palette and a 2-d sequence of sequences of colors.

**Arguments:**

1. `palette_n_image` : a {palette, image} pair, like [read\\_bitmap](#) returns
2. `file_name` : a sequence, the name of the file to save to.

**Returns:**

An **integer**, 0 on success.

**Comments:**

This routine does the opposite of [read\\_bitmap\(\)](#). The first element of `palette_n_image` is a list of sequences each sequence containing exactly three color intensity values in the range 0 to 255. The second element is a list of sequences of colors. The inner sequences must have the same length. Each element in the each inner sequence represents the color index in `palette_n_image` of a pixel. Each inner sequence is a row in the image.

The result will be one of the following codes:

```
public constant
    BMP_SUCCESS = 0,
    BMP_OPEN_FAILED = 1,
    BMP_INVALID_MODE = 4 -- invalid graphics mode
                        -- or invalid argument
```

`save_bitmap` produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with `read_bitmap`. Windows Paintbrush and some other tools do not support 4-color bitmaps.

**Example 1:**

```
code = save_bitmap({paletteData, imageData},
                  "c:\\example\\a1.bmp")
```

**See Also:**

[read\\_bitmap](#)



# EUPHORIA

- [euphoria/info.e](#)
- [euphoria/keywords.e](#)
- [euphoria/syncolor.e](#)
- [euphoria/tokenize.e](#)
- [std/unittest.e](#)
- [euphoria/debug/debug.e](#)

# Euphoria Information

## Build Type Constants

### is\_developmental

```
include euphoria/info.e
namespace info
public constant is_developmental
```

Is this build a developmental build?

### is\_release

```
include euphoria/info.e
namespace info
public constant is_release
```

Is this build a release build?

## Numeric Version Information

## Compiled Platform Information

### platform\_name

```
include euphoria/info.e
namespace info
public function platform_name()
```

Get the platform name

#### Returns:

A **sequence**, containing the platform name, i.e. Windows, Linux, FreeBSD or OS X.

### arch\_bits

```
include euphoria/info.e
namespace info
public function arch_bits()
```

Get the native architecture word size.

#### Returns:

A **sequence** in the form of "%d-bit", where %d is the word size for the architecture for which this version of euphoria was built.

## version

```
include euphoria/info.e
namespace info
public function version()
```

Get the version, as an integer, of the host Euphoria

### Returns:

An **integer**, representing Major, Minor and Patch versions. Version 4.0.0 will return 40000, 4.0.1 will return 40001, 5.6.2 will return 50602, 5.12.24 will return 512624, etc...

## version\_major

```
include euphoria/info.e
namespace info
public function version_major()
```

Get the major version of the host Euphoria

### Returns:

An **integer**, representing the Major version number. Version 4.0.0 will return 4, version 5.6.2 will return 5, and so on.

## version\_minor

```
include euphoria/info.e
namespace info
public function version_minor()
```

Get the minor version of the hosting Euphoria

### Returns:

An **integer**, representing the Minor version number. Version 4.0.0 will return 0, 4.1.0 will return 1, 5.6.2 will return 6, and so on.

## version\_patch

```
include euphoria/info.e
namespace info
public function version_patch()
```

Get the patch version of the hosting Euphoria

### Returns:

An **integer**, representing the Path version number. Version 4.0.0 will return 0, 4.0.1 will return 1, 5.6.2 will return 2, and so on.

## version\_node

```
include euphoria/info.e
```

```
namespace info
public function version_node(integer full = 0)
```

Get the source code node id of the hosting Euphoria

#### Parameters:

- full - If TRUE, the full node id is returned. If FALSE only the first 12 characters of the node id is returned. Typically the short node id is considered unique.

#### Returns:

A text **sequence**, containing the source code management systems node id that globally identifies the executing Euphoria.

### version\_revision

```
include euphoria/info.e
namespace info
public function version_revision()
```

Get the source code revision of the hosting Euphoria

#### Returns:

A text **sequence**, containing the source code management systems revision number that the executing Euphoria was built from.

### version\_date

```
include euphoria/info.e
namespace info
public function version_date(integer full = 0)
```

Get the compilation date of the hosting Euphoria

#### Parameters:

- full - Standard return value is a string formatted as CCYY-MM-DD. However, if this is a development build or the full parameter is TRUE (1), then the result will be formatted as CCYY-MM-DD HH:MM:SS.

#### Returns:

A text **sequence** containing the commit date of the the associated SCM revision.

The date/time is UTC.

## String Version Information

### version\_type

```
include euphoria/info.e
namespace info
public function version_type()
```

Get the type version of the hosting Euphoria

### Returns:

A **sequence**, representing the Type version string. Version 4.0.0 alpha 1 will return alpha 1. 4.0.0 beta 2 will return beta 2. 4.0.0 final, or release, will return release.

## version\_string

```
include euphoria/info.e
namespace info
public function version_string(integer full = 0)
```

Get a normal version string

### Parameters:

1. full - Return full version information regardless of developmental/production status.

### Returns:

A **#sequence**, representing the entire version information in one string. The amount of detail you get depends on if this version of Euphoria has been compiled as a developmental version (more detailed version information) or if you have indicated TRUE for the full argument.

Example return values

- "4.0.0 alpha 3 (ab8e98ab3ce4,2010-11-18)"
- "4.0.0 release (8d8874dc9e0a, 2010-12-22)"
- "4.1.5 development (12332:e8d8787af7de, 2011-07-18 12:55:03)"

## version\_string\_short

```
include euphoria/info.e
namespace info
public function version_string_short()
```

Get a short version string

### Returns:

A **sequence**, representing the Major, Minor and Patch all in one string.

### Example return values:

- "4.0.0"
- "4.0.2"
- "5.6.2"

## version\_string\_long

```
include euphoria/info.e
namespace info
public function version_string_long(integer full = 0)
```

Get a long version string

### Parameters:

1. full - Return full version information regardless of developmental/production status.

### Returns:

A **#sequence**, representing the entire version information in one string. The amount of detail you get depends on if this version of Euphoria has been compiled as a developmental version (more detailed version information) or if you have indicated TRUE for the full argument.

Example return values

- "4.0.0 alpha 3 (ab8e98ab3ce4,2010-11-18) for Windows 32-bit"
- "4.0.0 release (8d8874dc9e0a, 2010-12-22) for Linux 32-bit"
- "4.1.5 development (12332:e8d8787af7de, 2011-07-18 12:55:03) for OS X 64-bit"

## Copyright Information

### euphoria\_copyright

```
include euphoria/info.e
namespace info
public function euphoria_copyright()
```

Get the copyright statement for Euphoria

### Returns:

A **sequence**, containing 2 sequences: product name and copyright message

### Example 1:

```
sequence info = euphoria_copyright()
-- info = {
--     "Euphoria v4.0.0 alpha 3",
--     "Copyright (c) XYZ, ABC\n" &
--     "Copyright (c) ABC, DEF"
-- }
```

### pcre\_copyright

```
include euphoria/info.e
namespace info
public function pcre_copyright()
```

Get the copyright statement for PCRE.

### Returns:

A **sequence**, containing 2 sequences: product name and copyright message.

### See Also:

[euphoria\\_copyright\(\)](#)

### all\_copyrights

```
include euphoria/info.e
namespace info
public function all_copyrights()
```

Get all copyrights associated with this version of Euphoria.

### Returns:

A **sequence**, of product names and copyright messages.

```
{
  { ProductName, CopyrightMessage },
  { ProductName, CopyrightMessage },
  ...
}
```

## Timing Information

### start\_time

```
include euphoria/info.e
namespace info
public function start_time()
```

Euphoria start time.

This time represents the time Euphoria itself started. This time is recorded before any of the users code is opened, parsed or executed. It can provide accurate timing information as to how long it takes for your application to go from start time to usable time.

### Returns:

An **atom** representing the start time of Euphoria itself

## Configure Information

### include\_paths

```
<built-in> function include_paths(integer convert)
```

Returns the list of include paths, in the order in which they are searched

### Parameters:

1. convert : an integer, nonzero to include converted path entries that were not validated yet.

### Returns:

A **sequence**, of strings, each holding a fully qualified include path.

### Comments:

convert is checked only under *Windows*. If a path has accented characters in it, then it may or may not be valid to convert those to the OEM code page. Setting convert to a nonzero value will force conversion for path entries that have accents and which have not been checked to be valid yet. The extra entries, if any, are returned at the end of the returned sequence.

### The paths are ordered in the order they are searched:

1. current directory
2. configuration file,

3. command line switches,
4. EUINC
5. a default based on EUDIR.

### Example 1:

```
sequence s = include_paths(0)
-- s might contain
{
  "/usr/euphoria/tests",
  "/usr/euphoria/include",
  "./include",
  "../include"
}
```

### See Also:

[eu.cfg](#) | [include](#) | [option\\_switches](#)



# Keyword Data

Keywords and routines built in to Euphoria.

## Constants

### keywords

```
include euphoria/keywords.e
namespace keywords
public constant keywords
```

Sequence of Euphoria keywords

### builtins

```
include euphoria/keywords.e
namespace keywords
public constant builtins
```

Sequence of Euphoria's built-in function names

# Syntax Coloring

Syntax Color Break Euphoria statements into words with multiple colors. The editor and pretty printer (eprint.ex) both use this file.

## Subroutines

### set\_colors

```
include euphoria/syncolor.e
namespace syncolor
public procedure set_colors(sequence pColorList)
```

### init\_class

```
include euphoria/syncolor.e
namespace syncolor
public procedure init_class()
```

### new

```
include euphoria/syncolor.e
namespace syncolor
public function new()
```

Create a new colorizer state

**See Also:**

[reset](#) | [SyntaxColor](#)

### reset

```
include euphoria/syncolor.e
namespace syncolor
public procedure reset(atom state = g_state)
```

### keep\_newlines

```
include euphoria/syncolor.e
namespace syncolor
public procedure keep_newlines(integer val = 1, atom state = g_state)
```

### SyntaxColor

```
include euphoria/syncolor.e
namespace syncolor
```

```
public function SyntaxColor(sequence pline, atom state = g_state, multiline_token multi = 0)
```

Parse Euphoria code into tokens of like colors.

### Parameters:

1. pline the source code to color
2. state (default g\_state) the tokenizer to use
3. multi the multiline token from the previous line

Break up a new-line terminated line into colored text segments identifying the various parts of the Euphoria language. They are broken into separate tokens.

### Returns:

A sequence that looks like:

```
{color1, "text1"}, {color2, "text2"}, ... }
```

### Comments:

In order to properly color multiline syntax (strings and comments), you should pass a value for multi. This value can be attained by calling `last_multiline_token` after coloring the previous line.

# Euphoria Source Tokenizer

## tokenize return sequence key

### enum

```
include euphoria/tokenize.e
namespace tokenize
public enum
```

## Tokens

### enum

```
include euphoria/tokenize.e
namespace tokenize
public enum
```

### T\_CHAR

```
include euphoria/tokenize.e
namespace tokenize
T_CHAR
```

quoted character

### T\_STRING

```
include euphoria/tokenize.e
namespace tokenize
T_STRING
```

string

## T\_NUMBER formats and T\_types

## Token accessors

### enum

```
include euphoria/tokenize.e
namespace tokenize
public enum
```

## ET error codes

## enum

```
include euphoria/tokenize.e
namespace tokenize
public enum
```

## error\_string

```
include euphoria/tokenize.e
namespace tokenize
public function error_string(integer err)
```

Get an error message string for a given error code.

## new

```
include euphoria/tokenize.e
namespace tokenize
public function new()
```

Create a new tokenizer state

**See Also:**

[reset](#) | [tokenize\\_string](#) | [tokenize\\_file](#)

## reset

```
include euphoria/tokenize.e
namespace tokenize
public procedure reset(atom state = g_state)
```

Reset the state to begin parsing a new file

**See Also:**

[new](#) | [tokenize\\_string](#) | [tokenize\\_file](#)

## get/set options

## keep\_builtins

```
include euphoria/tokenize.e
namespace tokenize
public procedure keep_builtins(integer val = 1, atom state = g_state)
```

Specify whether to identify builtins specially or not

default is FALSE

## keep\_keywords

```
include euphoria/tokenize.e
namespace tokenize
public procedure keep_keywords(integer val = 1, atom state = g_state)
```

Specify whether to identify keywords specially or not

default is TRUE

## keep\_whitespace

```
include euphoria/tokenize.e
namespace tokenize
public procedure keep_whitespace(integer val = 1, atom state = g_state)
```

Return white space (other than newlines) as tokens.

default is FALSE

## keep\_newlines

```
include euphoria/tokenize.e
namespace tokenize
public procedure keep_newlines(integer val = 1, atom state = g_state)
```

Return new lines as tokens.

default is FALSE

## keep\_comments

```
include euphoria/tokenize.e
namespace tokenize
public procedure keep_comments(integer val = 1, atom state = g_state)
```

Return comments as tokens

default is FALSE

## return\_literal\_string

```
include euphoria/tokenize.e
namespace tokenize
public procedure return_literal_string(integer val = 1, atom state = g_state)
```

When returning string tokens, we have the option to process them and return their value, or to return the literal text that made up the original string.

Right now, this option only affects the processing of hex strings.

default is FALSE - process the string and return its value

## string\_strip\_quotes

```
include euphoria/tokenize.e
```

```
namespace tokenize
public procedure string_strip_quotes(integer val = 1, atom state = g_state)
```

When returning string tokens, we have the option to strip the quotes.

default is TRUE

## string\_numbers

```
include euphoria/tokenize.e
namespace tokenize
public procedure string_numbers(integer val = 1, atom state = g_state)
```

Return TDATA for all T\_NUMBER tokens in "string" format.

### Defaults:

- T\_NUMBER tokens return atoms
- T\_CHAR tokens return single integer chars
- T\_EOF tokens return undefined data
- Other tokens return strings

## multiline\_token

```
include euphoria/tokenize.e
namespace tokenize
public type multiline_token(object mlt)
```

## last\_multiline\_token

```
include euphoria/tokenize.e
namespace tokenize
public function last_multiline_token()
```

### Returns:

One of 0, TF\_COMMENT\_MULTIPLE, TF\_STRING\_BACKTICK, TF\_STRING\_TRIPLE.

### Comments:

After calling `tokenize_string`, this function will return a value of 0 if the line did not end in the middle of a multiline construct, or the value for the respective token. This is meant to facilitate proper tokenizing of individual lines of code.

## Subroutines

## tokenize\_string

```
include euphoria/tokenize.e
namespace tokenize
public function tokenize_string(sequence code, atom state = g_state,
    integer stop_on_error = TRUE, multiline_token multi = 0)
```

Tokenize euphoria source code

### Parameters:

1. code The code to be tokenized
2. state (default g\_state) the tokenizer returned by `new`
3. stop\_on\_error (default TRUE)
4. multi one of 0, TF\_COMMENT\_MULTIPLE, TF\_STRING\_BACKTICK, TF\_STRING\_TRIPLE

### Returns:

Sequence of tokens

## tokenize\_file

```
include euphoria/tokenize.e
namespace tokenize
public function tokenize_file(sequence fname, atom state = g_state,
    integer mode = io :BINARY_MODE)
```

Tokenize euphoria source code

### Parameters:

1. fname the file to be read and tokenized
2. state (default g\_state) the tokenizer returned by `new`
3. mode the mode in which to open the file. One of: `io:BINARY_MODE` (default) or `io:TEXT_MODE`. Note that for large files with Windows line endings, text mode may be much slower. See `io:read_file` for more information.

### Returns:

Sequence of tokens

## Debugging

## token\_names

```
include euphoria/tokenize.e
namespace tokenize
public constant token_names
```

Sequence containing token names for debugging

## token\_forms

```
include euphoria/tokenize.e
namespace tokenize
public constant token_forms
```

## show\_tokens

```
include euphoria/tokenize.e
namespace tokenize
public procedure show_tokens(integer fh, sequence tokens)
```

Print token names and data for each token in `tokens` to the file handle `fh`

### Parameters:



- fh - file handle to print information to
- tokens - token sequence to print

### Comments:

This does not take direct output from `tokenize_string` or `tokenize_file`. Instead they take the first element of their return value, the token stream only.

### See Also:

`tokenize_string` | `tokenize_file`

## Unit Testing Framework

### Background

Unit testing is the process of assuring that the smallest programming units are actually delivering functionality that complies with their specification. The units in question are usually individual routines rather than whole programs or applications.

The theory is that if the components of a system are working correctly, then there is a high probability that a system using those components can be made to work correctly.

In Euphoria terms, this framework provides the tools to make testing and reporting on functions and procedures easy and standardized. It gives us a simple way to write a test case and to report on the findings.

Example:

```
include std/unittest.e

test_equal( "Power function test #1", 4, power(2, 2))
test_equal( "Power function test #2", 4, power(16, 0.5))

test_report()
```

Name your test file in the special manner, t\_NAME.e and then simply run eutest in that directory.

```
C:\Euphoria> eutest
t_math.e:
failed: Bad math, expected: 100 but got: 8
2 tests run, 1 passed, 1 failed, 50.0% success

Test failure summary:
FAIL: t_math.e

2 file(s) run 1 file(s) failed, 50.0% success--
```

In this example, we use the `test_equal` function to record the result of a test. The first parameter is the name of the test, which can be anything and is displayed if the test fails. The second parameter is the expected result -- what we expect the function being tested to return. The third parameter is the actual result returned by the function being tested. This is usually written as a call to the function itself.

It is typical to provide as many test cases as would be required to give us confidence that the function is being truly exercised. This includes calling it with typical values and edge-case or exceptional values. It is also useful to test the function's error handling by calling it with bad parameters.

When a test fails, the framework displays a message, showing the test's name, the expected result and the actual result. You can configure the framework to display each test run, regardless of whether it fails or not.

After running a series of tests, you can get a summary displayed by calling the `test_report` procedure. To get a better feel for unit testing, have a look at the provided test cases for the standard library in the *tests* directory.

When included in your program, `unittest.e` sets a crash handler to log a crash as a failure.

### Constants

#### enum

```
include std/unittest.e
```

```
namespace unittest
public enum
```

## Setup Routines

### set\_test\_verbosity

```
include std/unittest.e
namespace unittest
public procedure set_test_verbosity(atom verbosity)
```

set the amount of information that is returned about passed and failed tests.

#### Arguments:

1. verbosity : an atom which takes predefined values for verbosity levels.

#### Comments:

The following values are allowable for verbosity:

- TEST\_QUIET -- 0,
- TEST\_SHOW\_FAILED\_ONLY -- 1
- TEST\_SHOW\_ALL -- 2

However, anything less than TEST\_SHOW\_FAILED\_ONLY is treated as TEST\_QUIET, and everything above TEST\_SHOW\_ALL is treated as TEST\_SHOW\_ALL.

- At the lowest verbosity level, only the score is shown, ie the ratio passed tests/total tests.
- At the medium level, in addition, failed tests display their name, the expected outcome and the outcome they got. This is the initial setting.
- At the highest level of verbosity, each test is reported as passed or failed.

If a file crashes when it should not, this event is reported no matter the verbosity level.

The command line switch "-failed" causes verbosity to be set to medium at startup. The command line switch "-all" causes verbosity to be set to high at startup.

#### See Also:

[test\\_report](#)

### set\_wait\_on\_summary

```
include std/unittest.e
namespace unittest
public procedure set_wait_on_summary(integer to_wait)
```

requests the test report to pause before exiting.

#### Arguments:

1. to\_wait : an integer, zero not to wait, nonzero to wait.

#### Comments:

Depending on the environment, the test results may be invisible if set\_wait\_on\_summary(1) was not called prior, as this is not the default. The command line switch "-wait" performs this call.

#### See Also:

## test\_report

### set\_accumulate\_summary

```
include std/unittest.e
namespace unittest
public procedure set_accumulate_summary(integer accumulate)
```

requests the test report to save run stats in "unittest.dat" before exiting.

#### Arguments:

1. `accumulate` : an integer, zero not to accumulate, nonzero to accumulate.

#### Comments:

The file "unittest.dat" is appended to with {t,f}

where

*t* is total number of tests run

*f* is the total number of tests that failed

### set\_test\_abort

```
include std/unittest.e
namespace unittest
public function set_test_abort(integer abort_test)
```

sets the behavior on test failure, and return previous value.

#### Arguments:

1. `abort_test` : an integer, the new value for this setting.

#### Returns:

An **integer**, the previous value for the setting.

#### Comments:

By default, the tests go on even if a file crashed.

### Reporting

### test\_report

```
include std/unittest.e
namespace unittest
public procedure test_report()
```

outputs the test report.

#### Comments:

The report components are described in the comments section for [set\\_test\\_verbosity](#). Everything prints on the standard error device.

#### See Also:

`set_test_verbosity`

## Tests

### test\_equal

```
include std/unittest.e
namespace unittest
public procedure test_equal(sequence name, object expected, object outcome)
```

records whether a test passes by comparing two values.

#### Arguments:

1. name : a string, the name of the test
2. expected : an object, the expected outcome of some action
3. outcome : an object, some actual value that should equal the referenceexpected.

#### Comments:

- For floating point numbers, a fuzz of1e-9 is used to assess equality.

A test is recorded as passed if equality holds between expected and outcome. The latter is typically a function call, or a variable that was set by some prior action.

While expected and outcome are processed symmetrically, they are not recorded symmetrically, so be careful to pass expected before outcome for better test failure reports.

#### See Also:

`test_not_equal`, `test_true`, `test_false`, `test_pass`, `test_fail`

### test\_not\_equal

```
include std/unittest.e
namespace unittest
public procedure test_not_equal(sequence name, object a, object b)
```

records whether a test passes by comparing two values.

#### Arguments:

1. name : a string, the name of the test
2. expected : an object, the expected outcome of some action
3. outcome : an object, some actual value that should equal the referenceexpected.

#### Comments:

- For atoms, a fuzz of1e-9 is used to assess equality.
- For sequences, no such fuzz is implemented.

A test is recorded as passed if equality does not hold between expected and outcome. The latter is typically a function call, or a variable that was set by some prior action.

#### See Also:

`test_equal`, `test_true`, `test_false`, `test_pass`, `test_fail`

## test\_true

```
include std/unittest.e
namespace unittest
public procedure test_true(sequence name, object outcome)
```

records whether a test passes.

### Arguments:

1. name : a string, the name of the test
2. outcome : an object, some actual value that should not be zero.

### Comments:

This assumes an expected value different from 0. No fuzz is applied when checking whether an atom is zero or not. Use `test_equal` instead in this case.

### See Also:

`test_equal`, `test_not_equal`, `test_false`, `test_pass`, `test_fail`

## assert

```
include std/unittest.e
namespace unittest
public procedure assert(object name, object outcome)
```

records whether a test passes. If it fails, the program also fails.

### Arguments:

1. name : a string, the name of the test
2. outcome : an object, some actual value that should not be zero.

### Comments:

This is identical to `test_true` except that if the test fails, the program will also be forced to fail at this point.

### See Also:

`test_equal`, `test_not_equal`, `test_false`, `test_pass`, `test_fail`

## test\_false

```
include std/unittest.e
namespace unittest
public procedure test_false(sequence name, object outcome)
```

records whether a test passes by comparing two values.

### Arguments:

1. name : a string, the name of the test
2. outcome : an object, some actual value that should be zero

### Comments:

This assumes an expected value of 0. No fuzz is applied when checking whether an atom is zero or not. Use `test_equal` instead in this case.

### See Also:

[test\\_equal](#), [test\\_not\\_equal](#), [test\\_true](#), [test\\_pass](#), [test\\_fail](#)

## test\_fail

```
include std/unittest.e
namespace unittest
public procedure test_fail(sequence name)
```

records that a test failed.

### Arguments:

1. name : a string, the name of the test

### See Also:

[test\\_equal](#), [test\\_not\\_equal](#), [test\\_true](#), [test\\_false](#), [test\\_pass](#)

## test\_pass

```
include std/unittest.e
namespace unittest
public procedure test_pass(sequence name)
```

records that a test passed.

### Arguments:

1. name : a string, the name of the test

### See Also:

[test\\_equal](#), [test\\_not\\_equal](#), [test\\_true](#), [test\\_false](#), [test\\_fail](#)

## Debugging tools

### Call Stack Constants

#### enum

```
include euphoria/debug/debug.e
namespace debug
public enum
```

#### CS\_ROUTINE\_NAME

```
include euphoria/debug/debug.e
namespace debug
CS_ROUTINE_NAME
```

#### CS\_FILE\_NAME

```
include euphoria/debug/debug.e
namespace debug
CS_FILE_NAME
```

#### CS\_LINE\_NO

```
include euphoria/debug/debug.e
namespace debug
CS_LINE_NO
```

#### CS\_ROUTINE\_SYM

```
include euphoria/debug/debug.e
namespace debug
CS_ROUTINE_SYM
```

#### CS\_PC

```
include euphoria/debug/debug.e
namespace debug
CS_PC
```

#### CS\_GLINE

```
include euphoria/debug/debug.e
namespace debug
CS_GLINE
```



## DEBUG\_ROUTINE Enum Type

These constants are used to register euphoria routines that handle various debugger tasks, displaying information or waiting for user input.

### DEBUG\_ROUTINE

```
include euphoria/debug/debug.e
namespace debug
public enum type DEBUG_ROUTINE
```

**SHOW\_DEBUG** A procedure that takes an integer parameter that represents the current line in the global line table

**DISPLAY\_VAR** A procedure that takes a pointer to the variable in the symbol table, and a flag to indicate whether the user requested this variable or not. Euphoria generally calls this when a variable is assigned to.

**UPDATE\_GLOBALS** A procedure called when the debug screen should update the display of any non-private variables

### DEBUG\_SCREEN

```
include euphoria/debug/debug.e
namespace debug
enum type DEBUG_ROUTINE DEBUG_SCREEN
```

### ERASE\_PRIVATES

```
include euphoria/debug/debug.e
namespace debug
enum type DEBUG_ROUTINE ERASE_PRIVATES
```

### ERASE\_SYMBOL

```
include euphoria/debug/debug.e
namespace debug
enum type DEBUG_ROUTINE ERASE_SYMBOL
```

## Debugging Routines

### call\_stack

```
include euphoria/debug/debug.e
namespace debug
public function call_stack()
```

Returns information about the call stack of the code currently running.

#### Returns:

A sequence where each element represents one level in the call stack. See the [Call Stack](#)

**Constants** for constants that can be used to access the call stack information.

1. routine name
2. file name
3. line number

## M\_INIT\_DEBUGGER

```
include euphoria/debug/debug.e
namespace debug
public constant M_INIT_DEBUGGER
```

## initialize\_debugger

```
include euphoria/debug/debug.e
namespace debug
public procedure initialize_debugger(atom init_ptr)
```

Initializes an external debugger. It can also be called from a debugger compiled into a DLL / SO.

### Parameters:

1. init\_ptr : The result of `machine_func( M_INIT_DEBUGGER, {} )`.

## set\_debug\_rid

```
include euphoria/debug/debug.e
namespace debug
public procedure set_debug_rid(DEBUG_ROUTINE rtn, integer rid)
```

## read\_object

```
include euphoria/debug/debug.e
namespace debug
public function read_object(atom sym)
```

## trace\_off

```
include euphoria/debug/debug.e
namespace debug
public procedure trace_off()
```

## disable\_trace

```
include euphoria/debug/debug.e
namespace debug
public procedure disable_trace()
```

## step\_over

```
include euphoria/debug/debug.e
namespace debug
public procedure step_over()
```

## abort\_program

```
include euphoria/debug/debug.e
namespace debug
public procedure abort_program()
```

## get\_current\_line

```
include euphoria/debug/debug.e
namespace debug
public function get_current_line()
```

## symbol\_lookup

```
include euphoria/debug/debug.e
namespace debug
public function symbol_lookup(sequence name, integer line = get_current_line(),
    atom pc = get_pc())
```

## get\_pc

```
include euphoria/debug/debug.e
namespace debug
public function get_pc()
```

## is\_novalue

```
include euphoria/debug/debug.e
namespace debug
public function is_novalue(atom sym_ptr)
```

## debugger\_call\_stack

```
include euphoria/debug/debug.e
namespace debug
public function debugger_call_stack()
```

## break\_routine

```
include euphoria/debug/debug.e
namespace debug
```

```
public function break_routine(atom routine_sym, integer enable)
```

## get\_name

```
include euphoria/debug/debug.e
namespace debug
public function get_name(atom sym)
```

## get\_source

```
include euphoria/debug/debug.e
namespace debug
public function get_source(integer line)
```

## get\_file\_no

```
include euphoria/debug/debug.e
namespace debug
public function get_file_no(integer line)
```

## get\_file\_name

```
include euphoria/debug/debug.e
namespace debug
public function get_file_name(integer file_no)
```

## get\_file\_line

```
include euphoria/debug/debug.e
namespace debug
public function get_file_line(integer line)
```

## get\_next

```
include euphoria/debug/debug.e
namespace debug
public function get_next(atom sym)
```

## is\_variable

```
include euphoria/debug/debug.e
namespace debug
public function is_variable(atom sym_ptr)
```

## get\_parameter\_syms

```
include euphoria/debug/debug.e
namespace debug
public function get_parameter_syms(atom rtn_sym)
```

## get\_symbol\_table

```
include euphoria/debug/debug.e
namespace debug
public function get_symbol_table()
```

# WINDOWS

- win32/msgbox.e
- win32/sounds.e

# Windows Message Box

## Style Constants

Possible style values for `message_box()` style sequence

### MB\_ABORTRETRYIGNORE

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_ABORTRETRYIGNORE
```

Abort, Retry, Ignore

### MB\_APPLMODAL

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_APPLMODAL
```

User must respond before doing something else

### MB\_DEFAULT\_DESKTOP\_ONLY

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_DEFAULT_DESKTOP_ONLY
```

### MB\_DEFBUTTON1

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_DEFBUTTON1
```

First button is default button

### MB\_DEFBUTTON2

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_DEFBUTTON2
```

Second button is default button

### MB\_DEFBUTTON3

```
include std/win32/msgbox.e
```

```
namespace msgbox
public constant MB_DEFBUTTON3
```

Third button is default button

## MB\_DEFBUTTON4

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_DEFBUTTON4
```

Fourth button is default button

## MB\_HELP

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_HELP
```

Windows 95: Help button generates help event

## MB\_ICONASTERISK

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_ICONASTERISK
```

## MB\_ICONERROR

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_ICONERROR
```

## MB\_ICONEXCLAMATION

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_ICONEXCLAMATION
```

Exclamation-point appears in the box

## MB\_ICONHAND

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_ICONHAND
```

A hand appears



## MB\_ICONINFORMATION

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_ICONINFORMATION
```

Lowercase letter i in a circle appears

## MB\_ICONQUESTION

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_ICONQUESTION
```

A question-mark icon appears

## MB\_ICONSTOP

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_ICONSTOP
```

## MB\_ICONWARNING

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_ICONWARNING
```

## MB\_OK

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_OK
```

Message box contains one push button: OK

## MB\_OKCANCEL

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_OKCANCEL
```

Message box contains OK and Cancel

## MB\_RETRYCANCEL

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_RETRYCANCEL
```

Message box contains Retry and Cancel

## MB\_RIGHT

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_RIGHT
```

Windows 95: The text is right-justified

## MB\_RTLREADING

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_RTLREADING
```

Windows 95: For Hebrew and Arabic systems

## MB\_SERVICE\_NOTIFICATION

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_SERVICE_NOTIFICATION
```

Windows NT: The caller is a service

## MB\_SETFOREGROUND

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_SETFOREGROUND
```

Message box becomes the foreground window

## MB\_SYSTEMMODAL

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_SYSTEMMODAL
```

All applications suspended until user responds

## MB\_TASKMODAL

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_TASKMODAL
```

Similar to MB\_APPLMODAL

## MB\_YESNO

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_YESNO
```

Message box contains Yes and No

## MB\_YESNOCANCEL

```
include std/win32/msgbox.e
namespace msgbox
public constant MB_YESNOCANCEL
```

Message box contains Yes, No, and Cancel

## Return Value Constants

possible values returned by MessageBox(). 0 means failure

## IDABORT

```
include std/win32/msgbox.e
namespace msgbox
public constant IDABORT
```

Abort button was selected.

## IDCANCEL

```
include std/win32/msgbox.e
namespace msgbox
public constant IDCANCEL
```

Cancel button was selected.

## IDIGNORE

```
include std/win32/msgbox.e
namespace msgbox
public constant IDIGNORE
```

Ignore button was selected.

## IDNO

```
include std/win32/msgbox.e
namespace msgbox
public constant IDNO
```

No button was selected.

## IDOK

```
include std/win32/msgbox.e
namespace msgbox
public constant IDOK
```

OK button was selected.

## IDRETRY

```
include std/win32/msgbox.e
namespace msgbox
public constant IDRETRY
```

Retry button was selected.

## IDYES

```
include std/win32/msgbox.e
namespace msgbox
public constant IDYES
```

Yes button was selected.

## Subroutines

### message\_box

```
include std/win32/msgbox.e
namespace msgbox
public function message_box(sequence text, sequence title, object style)
```

Displays a window with a title, message, buttons and an icon, usually known as a message box.

#### Parameters:

1. text: a sequence, the message to be displayed
2. title: a sequence, the title the box should have
3. style: an object which defines which icon should be displayed, if any, and which buttons will be presented.

#### Returns:

An **integer**, the button which was clicked to close the message box, or 0 on failure.

#### Comments:

See [Style Constants](#) above for a complete list of possible values for style and [Return Value Constants](#) for the returned value. If style is a sequence, its elements will be or'ed together.

# Windows Sound

## SND\_DEFAULT

```
include std/win32/sounds.e
namespace sound
public constant SND_DEFAULT
```

## SND\_STOP

```
include std/win32/sounds.e
namespace sound
public constant SND_STOP
```

## SND\_QUESTION

```
include std/win32/sounds.e
namespace sound
public constant SND_QUESTION
```

## SND\_EXCLAMATION

```
include std/win32/sounds.e
namespace sound
public constant SND_EXCLAMATION
```

## SND\_ASTERISK

```
include std/win32/sounds.e
namespace sound
public constant SND_ASTERISK
```

## sound

```
include std/win32/sounds.e
namespace sound
public procedure sound(atom sound_type = SND_DEFAULT)
```

Makes a sound.

### Parameters:

1. `sound_type`: An atom. The type of sound to make. The default is `SND_DEFAULT`.

### Comments:

The `sound_type` value can be one of ...

- SND\_ASTERISK
- SND\_EXCLAMATION
- SND\_STOP
- SND\_QUESTION
- SND\_DEFAULT

These are sounds associated with the same Windows events via the Control Panel.

### Example:

```
sound( SND_EXCLAMATION )
```

## UNSUPPORTED

These are features that have been implemented either partly or fully, but are not officially part of the Euphoria Language. They may one day be officially sanctioned and thus fully supported, but that is not certain. And even if an unsupported feature does make it way into the language, it may not be exactly what is documented in this section.

So if you use **any** of these unsupported features then be aware that your code might break in future releases.

## DOS

The DOS operating system is supported only up to Euphoria 3.1

Legacy include files are provided in Euphoria 4.x for programmers still using legacy code. They are located in `../euphoria/include`.



## UTF String Literals

- using word strings hexadecimal (for utf-16) and double word hexadecimal (for utf-32) e.g.

```
u"65 66 67 AE" -- ==> {#65,#66,#67,#AE}
U"65 66 67 AE" -- ==> {#65,#66,#67,#AE}
```

The value of the strings above are equivalent. Spaces separate values to other elements. When you put too many hex characters together for the kind of string they are split up appropriately for you:

```
x"6566 67AE" -- 8-bit ==> {#65,#66,#67,#AE}
u"6566 67AE" -- 16-bit ==> {#6566,#67AE}
U"6566 67AE" -- 32-bit ==> {#6566,#67AE}
U"6566_67AE" -- 32-bit ==> {#656667AE}
                -- Uses '_' to aid readability for long values.
U"656667AE"  -- 32-bit ==> {#656667AE}
```

String literals encoded as ASCII, UTF-8, UTF-16, UTF-32 or really any encoding that uses elements that are 32-bits long or shorter can be built with U"" syntax. Literals of encodings that have 16-bit long or shorter or 8-bit long or shorter elements can be built using u"" syntax or x"" syntax respectively. Use delimiters, such as spaces and underscores, to break the ambiguity and improve readability.

The following is code with a valid UTF8 encoded string:

```
sequence utf8_val = x"3e 65" -- This is ">e"
```

**However**, it is up to the coder to know the correct code-point values for these to make any sense in the encoding the coder is using. That is to say, it is possible for the coder to use the x"", u"", and U"" syntax to create literals that are **not valid** UTF strings.

Hexadecimal strings can be used to encode UTF-8 strings, even though the resulting string does not have to be a valid UTF-8 string.

## Rules for Unicode Strings

1. they begin with the pair u" for UTF-16 and U" for UTF-32 strings, and end with a double-quote (") character
2. they can only contain hexadecimal digits (0-9 A-F a-f), and space, underscore, tab, newline, carriage-return. Anything else is invalid.
3. an underscore is simply ignored, as if it was never there. It is used to aid readability.
4. For UTF-16 strings, each set of four contiguous hex digits represent a single sequence element with a value from 0x0000 to 0xFFFF
5. For UTF-32 strings, each set of eight contiguous hex digits represent a single sequence element with a value from 0x0000 to 0xFFFFFFFF
6. they can span multiple lines
7. The non-hex digits are treated as punctuation and used to delimit individual values.
8. The resulting string does not have to be a valid UTF-16/UTF-32 string.

```
u"1 2 34 5678AbC" == {0x0001, 0x0002, 0x0034, 0x5678, 0x0ABC}
U"1 2 34 5678AbC" == {0x0000_0001, 0x0000_0002, 0x0000_0034, 0x05678ABC}
U"1 2 34 5_678_AbC" == {0x0000_0001, 0x0000_0002, 0x0000_0034, 0x0567_8ABC}
```

## UTF String Literals

- using word strings hexadecimal (for utf-16) and double word hexadecimal (for utf-32) e.g.

```
u"65 66 67 AE" -- ==> {#65,#66,#67,#AE}
U"65 66 67 AE" -- ==> {#65,#66,#67,#AE}
```

The value of the strings above are equivalent. Spaces separate values to other elements. When you put too many hex characters together for the kind of string they are split up appropriately for you:

```

x"6566 67AE"  -- 8-bit  ==> {#65,#66,#67,#AE}
u"6566 67AE"  -- 16-bit ==> {#6566,#67AE}
U"6566 67AE"  -- 32-bit ==> {#6566,#67AE}
U"6566_67AE"  -- 32-bit ==> {#656667AE}
               --          Uses '_' to aid readability for long values.
U"656667AE"   -- 32-bit ==> {#656667AE}

```

String literals encoded as ASCII, UTF-8, UTF-16, UTF-32 or really any encoding that uses elements that are 32-bits long or shorter can be built with U`""` syntax. Literals of encodings that have 16-bit long or shorter or 8-bit long or shorter elements can be built using u`""` syntax or x`""` syntax respectively. Use delimiters, such as spaces and underscores, to break the ambiguity and improve readability.

The following is code with a valid UTF8 encoded string:

```
sequence utf8_val = x"3e 65" -- This is ">e"
```

**However**, it is up to the coder to know the correct code-point values for these to make any sense in the encoding the coder is using. That is to say, it is possible for the coder to use the x`""`, u`""`, and U`""` syntax to create literals that are **not valid** UTF strings.

Hexadecimal strings can be used to encode UTF-8 strings, even though the resulting string does not have to be a valid UTF-8 string.

## Rules for Unicode Strings

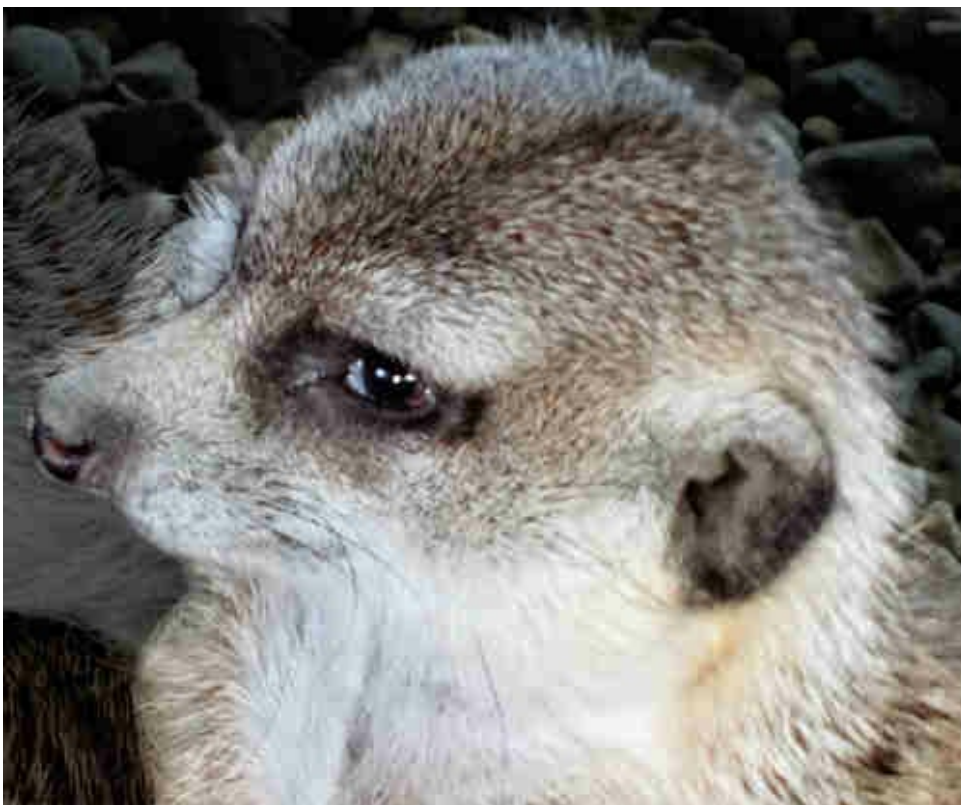
1. they begin with the pair u`"` for UTF-16 and U`"` for UTF-32 strings, and end with a double-quote (`"`) character
2. they can only contain hexadecimal digits (0-9 A-F a-f), and space, underscore, tab, newline, carriage-return. Anything else is invalid.
3. an underscore is simply ignored, as if it was never there. It is used to aid readability.
4. For UTF-16 strings, each set of four contiguous hex digits represent a single sequence element with a value from 0x0000 to 0xFFFF
5. For UTF-32 strings, each set of eight contiguous hex digits represent a single sequence element with a value from 0x0000 to 0xFFFFFFFF
6. they can span multiple lines
7. The non-hex digits are treated as punctuation and used to delimit individual values.
8. The resulting string does not have to be a valid UTF-16/UTF-32 string.

```

u"1 2 34 5678AbC" == {0x0001, 0x0002, 0x0034, 0x5678, 0x0ABC}
U"1 2 34 5678AbC" == {0x0000_0001, 0x0000_0002, 0x0000_0034, 0x05678ABC}
U"1 2 34 5_678_AbC" == {0x0000_0001, 0x0000_0002, 0x0000_0034, 0x0567_8ABC}

```

# NEWS



release 4.1 news

## Version 4.1.0 Date TBD

### Bug Fixes

- [Interwiki link failed for ticket:665](#) Fixed to load socket routines from its DLL only when needed.
- [Interwiki link failed for ticket:744](#) Detect duplicate case values in a `switch statement` and throw an error at compile or parse time
- OS X bug fixes:
  - Callbacks function again, including on 64bit platforms
  - Memory maps function
- Fix std/net/http.e that caused malformed HTTP GET requests
- Updated demo/news.ex with up-to-date URLs for some news web sites.
- Fix std/net/http.e so it can handle cases where the Content-Length header is not present
- Fix std/sequence.e so store() will correctly handle the one-element index case - it was duplicating the entire sequence before.
- [Interwiki link failed for ticket:710](#) Updated tokenizer and syntax coloring to be able to preserve state between lines. The euphoria trace screen and ed.ex now properly colorize multiline strings and comments.
- `tokenize_string` had an infinite loop if the string ended with a single or double quote and a backslash
- euphoria/tokenizer.e does not add a leading zero to floating point numbers without one when `string_numbers` is set
- fixed detection of hex string tokens in `tokenize_string`
- tokenizing better respects the value of `stop_on_error` parameter for `tokenize_string`
- [Interwiki link failed for ticket:884](#) 32-bit translator crash when translating a 64-bit target
- [Interwiki link failed for ticket:886](#) system\_exec with quotes on Windows failed due to quotes being removed
- [Interwiki link failed for ticket:887](#) EDS: crash if create\_table and clear\_table init\_records parameter was less than MAX\_INDEX

### Enhancements

- Euphoria can be built natively as a 64-bit programming language.
- Added 8-byte memory access: `poke8`, `peek8s`, `peek8u`
- eucoverage also outputs a file "big\_routines.html" that shows covered routines from all files sorted by descending routine size
- Added `poke_pointer` and `peek_pointer`
- New `sizeof` built-in for determining size of certain data types.
- [Interwiki link failed for ticket:631](#) Scientific parsing code moved from the euphoria source directory and into the standard library. Routines in std/get.e now return the proper precision data based on the native platform (32 or 64 bits).
- Users can write their own debuggers and use them instead of the built in trace debugger.
- gcc builds now include -fPIC (position independent code) runtime libraries for translating euphoria code into shared objects.
- -lib-pic switch for translator to specify the PIC runtime library to be used
- [Interwiki link failed for ticket:166](#) get\_integer{16,32} will return -1 on EOF.
- Added `deprecate` keyword
- Architecture ifdefs (X86, X86\_64, ARM, BITS32, BITS64, LONG32, LONG64)
- -arch option for translator for cross translating
- -cc-prefix option for translator
- Can `assign to multiple variables` with one statement using sequence semantics.
- Use ? to stand in for default parameters.

- eudis now tabulates counts of forward references
- Added `poke_long`, `peek_longu` and `peek longs`
- [Interwiki link failed for ticket:735](#) The number of lines to be used in `ctrace.out` by `trace(3)` can be configured using `-trace-lines n` command line switch. See [Command line switches](#) for more information.
- [Interwiki link failed for ticket:782](#) When downloading http content, `std/net/http.e` will yield to other tasks
- `t_integer32` type for checking to see if an object is an integer based on 32-bit Euphoria's definition
- Improved identification of `routine_id()` targets by the translator
- Smaller translated DLLs are produced by improved identification of routines that need to be exported
- Eutest now has an `eubin` option for specifying all binaries in a single option.
- Eutest has a `retest` option for retesting all tests that had previously failed.
- Front end optimizations to reduce parsing time
- Added dynamic library uninitialization to reduce memory leaks if a euphoria translated `.dll` or `.so` is unloaded
- [Interwiki link failed for ticket:838](#) Eutest now reports the date of the Interpreter, and when the test was completed.
- Much faster and simpler implementation of maps in `std/map.e` inspired by the implementation of python's dictionary object. Some functions and parameters have been deprecated (such as any distinction between small and large maps), as they no longer make sense for the new implementation.
- [Interwiki link failed for ticket:532](#) `extra-cflags` and `extra-lflags` for translator (thanks to Ira Hill)
- `lock_file` on Windows now supports `LOCK_SHARED` and `LOCK_EXCLUSIVE`
- `tokenize_file` uses `io:BINRARY_MODE` by default instead of `io:TEXT_MODE`, which improves performance on large files with Windows style newlines
- [Interwiki link failed for ticket:883](#) Improved error messages on subscript / slice errors
- `ed.ex` renamed `edx.ex`

## Version 4.0.6 Date TBD

### Bug Fixes

- [Interwiki link failed for ticket:872](#) fix documentation error involving `or_all`
- [Interwiki link failed for ticket:880](#) fix documentaiton error involving `poke2`
- [Interwiki link failed for ticket:708](#) specified that color triples are specified between 0..255 in `read_bitmap` and `save_bitmap`.
- [Interwiki link failed for ticket:801](#) fix translator memory leak for `insert`
- [Interwiki link failed for ticket:799](#) fix memory leak in `gets` when reading EOF
- [Interwiki link failed for ticket:819](#) use operating system sleep functions for fractions of seconds to avoid needless CPU utilization
- [Interwiki link failed for ticket:824](#) fix OpenWatcom installer PCRE directory
- [Interwiki link failed for ticket:823](#) emit error in translator when user specifies a file for the build directory
- [Interwiki link failed for ticket:781](#) `http_post` and `http_get` now follow redirects
- [Interwiki link failed for ticket:835](#) translator properly handles sequences passed to `floor`
- [Interwiki link failed for ticket:830](#) fixed memory leak in `replace`
- [Interwiki link failed for ticket:847](#) fixed memory leak in `remove`
- [Interwiki link failed for ticket:837](#) fixed documentation error involving `load_map`
- Fix `std/sequence.e` so `store()` will correctly handle the one-element index case - it was duplicating the entire sequence before.
- [Interwiki link failed for ticket:638](#) `value` and `get` handle multi-line strings
- [Interwiki link failed for ticket:836](#) `canonical_path` works when path is not on the current drive on Windows
- [Interwiki link failed for ticket:630](#) shrouder ignores binder options that are not applicable
- [Interwiki link failed for ticket:776](#) Updated `walk_dir` parameter documentation
- functions imported from `msvcrt.dll` should use `cdecl` (affects `now_gmt`, `locale:get`, `locale:set` and `locale:datetime`)
- [Interwiki link failed for ticket:882](#) Translated `remove` works correctly with empty sequence
- [Interwiki link failed for ticket:885](#) Translated `splice` properly counts references when splicing an atom

### Enhancements

- command line help is now sorted by option

## Version 4.0.5 October 19, 2012

### Bug Fixes

- [Interwiki link failed for ticket:777](#) When invalid input is sent to 'match' or 'find' the error includes 'match' or 'find' in the error message respectively.
- [Interwiki link failed for ticket:749](#) Fix init checks for while-entry and goto
- [Interwiki link failed for ticket:563](#) Default values for arguments are always parsed and resolved as though they were being evaluated from the point of the routine declaration, not the point where the routine is called
- [Interwiki link failed for ticket:763](#) In some cases, the translator did not keep potential `routine_id` targets when dynamic routine names were used
- [Interwiki link failed for ticket:665](#) documented minimal requirements for various features in EUPHORIA on various platforms.
- [Interwiki link failed for ticket:665](#) set minimal version for Windows in its installer to avoid installing on computers that it wont work on.
- [Interwiki link failed for ticket:767](#) translated `insert()` could segfault when inserting an atom stored as an integer
- [Interwiki link failed for ticket:744](#) Duplicate case values in a switch block no longer result in a failed compile after being translated to C.
- [Interwiki link failed for ticket:775](#) Fixed potential memory leak when a temp is passed to one of the native type check functions: `integer()`, `atom()`, `object()` or `sequence()`
- [Interwiki link failed for ticket:778](#) Translator keeps forward referenced `routine_id` routines in include files
- [Interwiki link failed for ticket:789](#) Make parser read Windows eols the same as unix eols on Linux.
- [Interwiki link failed for ticket:795](#) Corrected `std/serialize.e` to call `define_c_proc` correctly
- [Interwiki link failed for ticket:795](#) Corrected `std/net/http.e` to call `do_a_case` insensitive search for 'content-length'
- [Interwiki link failed for ticket:796](#) when binding and translating use different EXE names
- Fixed memory leak in translator when calls `tohead()` result in an empty sequence

### Enhancements

- [Interwiki link failed for ticket:768](#) Backported support for deserializing 8-byte integers and 10-byte floating point.
- Optimization of `std/map.e remove()` to prevent unnecessary copy on write
- [Interwiki link failed for ticket:787](#) Document cases where you pass an empty sequence into search routines



## Version 4.0.4 April 4, 2012

### Bug Fixes

- [Interwiki link failed for ticket:664](#) Symbol resolution errors now report whether you use a symbol is not declared or is declared more than once, or from not declared in the file you specify (via a namespace), or not a builtin. When declared more than once, you are now told where the symbols were declared.
- [Interwiki link failed for ticket:602](#) socket create documentation corrected to state that it returns an error code on failure.
- [Interwiki link failed for ticket:672](#) fixed dll creation under Windows.
- [Interwiki link failed for ticket:687](#) fixed source file distribution.
- [Interwiki link failed for ticket:681](#) fixed error reporting when the error is the last symbol on a line, but that might be part of an expression that carries over to the next line
- [Interwiki link failed for ticket:694](#) do not short circuit inside of forward function calls
- [Interwiki link failed for ticket:699](#) Include public and export symbols in ex.err output
- [Interwiki link failed for ticket:717](#) Fix docs to correctly describe bitwise functions
- [Interwiki link failed for ticket:725](#) Smarter reading of command line options. Euphoria could consume switches meant for the the end user program
- When there is a user supplied library, the translator does not abort when the library doesn't exist and one of -nobuild, -makefile or -makefile-partial is used
- [Interwiki link failed for ticket:728](#) Fix sequence slice error when invalid command line arguments are passed to euphoria.
- [Interwiki link failed for ticket:730](#) Fixed initialization of private variables. The translator incorrectly assumed that all variables started as integers to prevent them from being dereferenced.
- [Interwiki link failed for ticket:722](#) Use backslashes for the filesystem separator when passing to Watcom even if the supplied data uses forward slashes.
- [Interwiki link failed for ticket:611](#) an no-longer existing install.doc was being referenced by a an install script. This has been updated.
- [Interwiki link failed for ticket:683](#) [Interwiki link failed for ticket:685](#) fixes for building the interpreter itself for MinGW
- [Interwiki link failed for ticket:732](#) fixes in building console less programs using MinGW
- [Interwiki link failed for ticket:721](#) fixes drive letter case discrepancy between various functions defined in sys/filesys.e

### Enhancements

- [Interwiki link failed for ticket:611](#) A more complete explanation of how to install has been added to the documentation.
- [Interwiki link failed for ticket:727](#) The interpreter and translator no longer show you all of their options when you make a mistake at the command line.
- [Interwiki link failed for ticket:727](#) cmd\_parse() can take a new option NO\_HELP\_ON\_ERROR, which means it will not display all of the options on error.
- [Interwiki link failed for ticket:741](#) minor format/refactor win32 demos to use C\_TYPES more win64 compatible & eu4.1 ready.



## Version 4.0.3 June 23, 2011

### Bug Fixes

- [Interwiki link failed for ticket:655](#) Integer values stored as doubles weren't being correctly coerced back to euphoria integers in translated code.
- [Interwiki link failed for ticket:656](#) Translated `not_bits` made incorrect type assumptions
- [Interwiki link failed for ticket:662](#) Switches with all integer cases, but with a range of greater than 1024 between the biggest and smallest were interpreted incorrectly.
- [Interwiki link failed for ticket:661](#) fixed translator linking to use comctl32 library on windows
- [Interwiki link failed for ticket:663](#) Translator -plat switch now uses WINDOWS instead of WIN.
- [Interwiki link failed for ticket:666](#) fixed to allow integers stored as doubles in `be_sockets.c`.
- [Interwiki link failed for ticket:654](#) removed internal use-only standard library routines and constants from the user documentation.
- [Interwiki link failed for ticket:667](#) Fixed optimization of translated IF when the conditions were known to be false.
- [Interwiki link failed for ticket:654](#) Removed from documentation the internal workings of Machine Level Access and reorganized Documentation.
- [Interwiki link failed for ticket:676](#) Changed search order for `locate_file`
- [Interwiki link failed for ticket:675](#) Fixed machine crash in `splice` when splicing an atom before beginning of sequence or after end
- [Interwiki link failed for ticket:665](#) Windows 95 and above is supported. For using sockets you must have Windows Sockets 2.2
- [Interwiki link failed for ticket:680](#) Fixed `socket` type checking.
- [Interwiki link failed for ticket:720](#) Fix propagation of public include among reincluded files

### Enhancements

- Minor changes to eutest output to read its console output
- The interpreter and programs created with the translator (for WATCOM only) will now run on older versions of Windows that don't support sockets unless this program *uses* sockets.
- New math functions `larger_of` and `smaller_of`

## Version 4.0.2 April 5, 2011

### Bug Fixes

- Fixed `canonical_path` performance issues introduced in 4.0.1.
- [Interwiki link failed for ticket:646](#) `dir` can now handle multiple wildcards on non-Windows platforms
- [Interwiki link failed for ticket:647](#) The version detection system has been improved so that all binaries use the same C header file, which should prevent the potential of mismatched versions.
- [Interwiki link failed for ticket:644](#) `canonical_path` leaves alone path components (and anything after them) with wildcards.
- Fixed compiler directives about functions that don't return. Removed some that were obsolete, and corrected for MinGW to use the GCC directives.
- [Interwiki link failed for ticket:648](#) Fix small memory leak from while loops

### New Functionality

- The `std::rand_e` function, **`sample()`**, now implements both *with replacement* and *without replacement* sampling methods.

## Version 4.0.1 March 29, 2011

### Bug Fixes

- Renamed implicit Top Level SubProgram to an illegal name. Previously used "\_toplevel\_", which became a legal name for euphoria 4.0
- [Interwiki link failed for ticket:577](#) object() works same on translator as the interpreter.
- euc now uses quotes around filenames when processing resource files
- [Interwiki link failed for ticket:575](#) OW installer file setenv-ow.bat functionality restored from 4.0.0RC2.
- case issues were removed from pathinfo(), canonical\_path(), and abbreviate\_path() these functions now return raw OS output; it is up to the user to change case when necessary
- [Interwiki link failed for ticket:593](#) Atoms represented as doubles, but that hold the double representation of a euphoria integer, now hash as though they were actually represented as an integer. This ensures that two objects that evaluate as equal() will have the same hash value.
- [Interwiki link failed for ticket:597](#) Invalid negative routine ids were not detected properly by the interpreter, leading to a machine crash.
- Now EUPHORIA can be installed under the Windows' 'Program Files' (with spaces) and the translated code will be compiled.
- Fixed Demos to not rely on EUDIR being set and to not issue warnings
- Improved confirmation in the algorithm that determines where EUPHORIA is.
- [Interwiki link failed for ticket:601](#) Missing htmldoc added to Makefile
- [Interwiki link failed for ticket:604](#) Uninstaller now completely cleans up after the installer. Note %EUBIN%\bin\eu.cfg is left in place if modified.
- Fixed link to PDF documentation
- Added HTML documentation
- [Interwiki link failed for ticket:610](#) Euphoria Installer that includes Watcom will now prevent the user from installing Euphoria under a directory with spaces. Watcom itself has a lot of problems when spaces are in its path
- [Interwiki link failed for ticket:614](#) maybe\_any\_key() was not pausing when a Console Program was run from Windows Explorer.
- [Interwiki link failed for ticket:591](#) updated copyright and version and added documentation reminding us all of the places we need to change that information.
- [Interwiki link failed for ticket:607](#) Fixed translation of integers with decimals (e.g., 2.0) when assigned to constants
- [Interwiki link failed for ticket:598](#) Link windows binaries to comdlg32.dll to make sure GUI calls work with the new manifest.
- [Interwiki link failed for ticket:590](#) Fixed outdated or incorrect documentation on loop statements
- [Interwiki link failed for ticket:594](#) Fixed problem with not being able to link to resource file in a location with spaces.
- [Interwiki link failed for ticket:615](#) Fixed abbreviate\_path for Windows
- [Interwiki link failed for ticket:595](#) When it is necessary, tell user to change directory before using the make program.
- [Interwiki link failed for ticket:592](#) eu.cfg files in the program's directory and the euphoria executable directories are searched before platform specific directories
- [Interwiki link failed for ticket:609](#) Scientific notation not handling a decimal of all zeroes correctly.
- [Interwiki link failed for ticket:621](#) Add -eudir <dir> handler to binder and shrouder
- [Interwiki link failed for ticket:617](#) Fix top level case values when referencing an unqualified constant in another file
- [Interwiki link failed for ticket:620](#) Added comdlg32.dll to mingw linking flags
- [Interwiki link failed for ticket:625](#) Negative subscripts result in runtime

- errors.
- Fixed eu.cfg handling precedence and parameter merge / de-dupe algorithm to keep correct order of switches.
- Load eu.cfg arguments when running programs with no arguments, e.g., "eui app.ex"
- [Interwiki link failed for ticket:619](#) GNU makefile "all" target builds all binaries now
- [Interwiki link failed for ticket:632](#) fix trace screen prompts to prompt to continue
- [Interwiki link failed for ticket:633](#) On Windows, `dir` was incorrectly case sensitive if wildcards were used.
- [Interwiki link failed for ticket:624](#) Fixed regex function `is_match` to use the from parameter
- [Interwiki link failed for ticket:596](#) Worked around GNU C problem of a lack of alias attribute support on some Mac OS X machines.
- [Interwiki link failed for ticket:636](#) Source files checked out from Mercurial (and thus distributed packages) will use the conventions of the OS for line breaks.
- [Interwiki link failed for ticket:639](#) In place RHS slice (on sequence with reference count 1), followed by in place `splice` (on sequence still with reference count 1) works correctly
- [Interwiki link failed for ticket:640](#) Fix `dir` when a file cannot be stat()ed
- [Interwiki link failed for ticket:641](#) Use `dir` instead of just calling raw `machine_func` in `canonical_path` and `abbreviate_path`

## Enhancements

- Added parsing of two digit years to `std/datetime.e` `parse`.
- [Interwiki link failed for ticket:516](#) added `join_path` and `split_path` routines.
- `current_dir()` now always returns an upper case letter for the drive id.
- `canonical_path()` can now leave the case alone, lower the case, correct the case, and even get short file names for programs that still cannot handle quoted arguments at the command line.

## Version 4.0.0 December 22, 2010

4.0.0 was released on December 22, 2010.

For a concise list of what has changed from 3.1.1 to 4.0.0 final, please see [What's new in 4.0?](#) section of this manual.

### Deprecation

- with/without warning lists have changed from ( name1, name2 ) to { name1, name2 } as to be more like Euphoria sequences. In the future the old( name1, name2 ) syntax will be removed.

### Possible Breaking Changes

- `std/sequence.e/series` has changed the functionality of the last parameter. Previously `series(1,1,5)` would produce {1,2,3,4,5,6}. i.e. 5 was the number of items to add onto the starting 1. The last parameter has been changed to be the number of items in the resulting list. Thus, `series(1,1,5)` will now produce {1,2,3,4,5}, i.e. a sequence of 5 items. `series(1,1,0)` before would produce {1}. Now it produces {}, i.e. an empty series.
- [Interwiki link failed for ticket:551](#): `WIN32_GUI`, `WIN32_CONSOLE`, `EUB_CONSOLE`, `EUC_CONSOLE` have been changed to simply refer to `GUI` or `CONSOLE`. On non-Windows platforms, `CONSOLE` will be defined.

### Removed

- `creolehtml` is no longer shipped with Euphoria. It has been enhanced to support multiple output formats and thus its name has been changed to simply `creole`. HTML remains the default output. Usage remains the same thus simply renaming build systems to use `creole` instead of `creolehtml` will work.

### Bug Fixes

- [Interwiki link failed for ticket:438](#), removed path test in `demos/santiy.ex` as it does not function correctly with `bound`, `translated` or even a non-standard `eui` location and actually cannot, thus it was removed.
- [Interwiki link failed for ticket:514](#), Fixed bug with `internaldir` implementation that would prevent displaying the content of a directory if given without a trailing slash on Windows.
- [Interwiki link failed for ticket:517](#), Added a bounds check that could cause the translator or binder to crash.
- [Interwiki link failed for ticket:518](#), Prevented `write_coverage` from being called twice on `CTRL+C/error` condition.
- [Interwiki link failed for ticket:519](#), `preproc` and `net` demos are now in the debian package.
- [Interwiki link failed for ticket:530](#), `t_command_line_quote` test fixed on Windows.
- [Interwiki link failed for ticket:533](#), Debian package copyright was updated in accordance to Debian policy.
- [Interwiki link failed for ticket:540](#), `get_key` was described in `bothio.e` and `console.e`, removed from `io.e`
- [Interwiki link failed for ticket:545](#), `canonical_path` did not properly insert the drive letter on Windows when the path began with a forward slash `/`.
- [Interwiki link failed for ticket:548](#), Fixed error in emitted C in some translated for loops.

- [Interwiki link failed for ticket:550](#), Examples for regex `matches` and `all_matches` now properly either supply or use the default from parameter.
- [Interwiki link failed for ticket:555](#), Fixed parsing of constants when first statement is a constant assigned by a built-in function.
- [Interwiki link failed for ticket:556](#), Fixed type inference for return value from `rand` in translator.
- [Interwiki link failed for ticket:557](#), `euphoria.h` had gotten out of sync when some OPs were removed.
- [Interwiki link failed for ticket:558](#), Fixed crash caused by undeclared variable assignment by properly subscripting `[i]` when looking up forward references in the toplevel subroutine
- [Interwiki link failed for ticket:560](#), Functions that started with an unqualified variable from another file being assigned by the return value of an unqualified function from another file could result in a crash.
- [Interwiki link failed for ticket:564](#), Documentation fix on parameter name for `calc_hash`.
- Fix backend and interpreter to avoid "press any key" prompts when running as a console from a shared console window.
- Ensure forward type checks aren't resolved until after the variable being type checked has been resolved.

## Enhancements/Changes

- Made previously private method `iscon` in `std/console.e` a public method named `has_console` which will return TRUE/FALSE if the current application has a console window attached.
- `cmd_parse` now splits onto two lines an option whose command is longer than the maximum pad size and its description.
- PDF documentation is now much better, generated from LaTeX sources.
- Bundled creole program supports multiple output formats now, the addition of LaTeX for great printed or PDF documentation from your creole sources.
- Bundled utility `bench.ex` now outputs timing information to `STDERR` by default. `--stdout` can be supplied if output to `STDOUT` is desired. It now also displays the min and max iteration times in addition to the already average and total.
- `demo/net/pastey.ex` demo has been updated to function with OpenEuphoria's pastey service. It can also now accept file input via `stdin`.
- `-version` on main products now reports build date in addition to previous information.
- `euphoria/info.e` version methods `version_string` and `version_string_long` now have the ability to report the enhanced version information.
- Optimized for loops to check for integer initial value and limits.

## Version 4.0.0 Release Candidate 2 December 8, 2010

### Deprecation

- `find_from` and `match_from` have been deprecated. `find` and `match` accept an optional argument (start) allowing these functions to be a 100% drop in replacement.
- `OPT_EXTRAS` in `std/cmdline.e` has been replaced by a more favored name `EXTRAS`.
- `iff` from `std/utls.e` has been replaced by a more favored name `iif`.

### Removed

- [Interwiki link failed for ticket:371](#), `replace_all` has been removed as it was a duplicate of the more powerful `match_replace` routine.
- [Interwiki link failed for ticket:376](#), `mouse.e` and `std/mouse.e`
- [Interwiki link failed for ticket:484](#), `wildcard_file` is very DOS centric, doesn't act right at all on modern consoles. It has been removed.
- [Interwiki link failed for ticket:486](#), `can_add` docs have been removed, they pointed to the name change of `can_add` to `binop_ok`, changed during beta stage.
- [Interwiki link failed for ticket:487](#), `wildcard:new()`, method really didn't make sense as a planning stage for regex usage as too much would have to change, a simply call to `new` did not save much and possibly just caused bad programming methods to be used.
- Support for alternate style `eu.cfg` sections, i.e. `bind:unix` and `unix:bind` were previously supported, now only the documented method: `bind:unix` is accepted.

### Bug Fixes

- [Interwiki link failed for ticket:118](#), `object()` tests now function properly when translated.
- [Interwiki link failed for ticket:169](#), `find_nested` no longer defaults the `rtn_id` parameter to -1 as that is the "invalid" return value of `routine_id` in which case a typo in your routine id would be silently ignored
- [Interwiki link failed for ticket:335](#), `eui` now only accepts `-v`, `--version` as parameters to display the version number instead of `-v`, `--v`, `-version` and `--version`.
- [Interwiki link failed for ticket:338](#), Fixed *Data Execution Prevention* for FreeBSD systems.
- [Interwiki link failed for ticket:339](#), Fixed locale for FreeBSD systems.
- [Interwiki link failed for ticket:341](#), Removed unused variables in the standard library.
- [Interwiki link failed for ticket:343](#), Resolution of unqualified symbols from other files is deferred until all files that could cause symbol resolution conflicts have been read.
- [Interwiki link failed for ticket:345](#), Forward patches now update the stack space for a routine when they create temps.
- [Interwiki link failed for ticket:349](#), Fixed resolution of qualified public symbols when the namespace points to the wrong file, but the namespace file directly includes the file with the actual routine
- [Interwiki link failed for ticket:352](#), A function with a defaulted parameter that is both forward referenced and inlined no longer crashes.
- [Interwiki link failed for ticket:358](#), The programs `eutest`, `creolehtml`, and `eudoc` now all support a command line option to display their version number.
- [Interwiki link failed for ticket:362](#), The handing of regular expressions



which match the text but didn't have any matching sub-groups was not correct nor documented.

- [Interwiki link failed for ticket:366](#), Created a new module, base64, to implement the standard Base-64 encoding algorithms.
- [Interwiki link failed for ticket:367](#), [http\\_post](#) properly handles multi-part form data.
- [Interwiki link failed for ticket:372](#), When an application ends, it closes all the opened files. However if it was ending due to an syntax error, it was closing those files before trying to access the message text database that had been opened, thus causing a seek() to fail and crash the application.
- [Interwiki link failed for ticket:378](#), On Linux and FreeBSD, the socket tests failed to detect the correct error code.
- [Interwiki link failed for ticket:392](#), [seek](#) was not returning the correct failure code on some errors.
- [Interwiki link failed for ticket:396](#), Continue operations are now properly back patched.
- [Interwiki link failed for ticket:391](#), Watcom build system was lacking the ability to build the manual.
- [Interwiki link failed for ticket:402](#), [maybe\\_any\\_key](#) now works when run from the command line version of EUPHORIA even when run without a command-line shell.
- [Interwiki link failed for ticket:403](#), Many documentation examples used ? func() and showed the output in string format which ? does not do. It was misleading to the new person to Euphoria. Found instances have been updated.
- [Interwiki link failed for ticket:405](#), dis.ex no longer creates a build directory for no reason.
- [Interwiki link failed for ticket:409](#), Calls to Head() that should have altered the sequence in place did not, resulting in slower code.
- [Interwiki link failed for ticket:417](#), Accidental inclusion of TOC was removed
- [Interwiki link failed for ticket:418](#), -debug eu.cfg switch text was corrected
- [Interwiki link failed for ticket:418](#), Clarified what (all) and (translator) means
- [Interwiki link failed for ticket:425](#), Fixed crash when branches were inlined into the top level
- [Interwiki link failed for ticket:426](#), Eutest uses binary binder
- [Interwiki link failed for ticket:429](#), tokenize.e no longer drops the first character of a backtick string
- [Interwiki link failed for ticket:431](#), tokenize.e properly parses \xxx escapes
- [Interwiki link failed for ticket:434](#), tokenize.e no longer strips leading zeros on numbers when using the [string\\_numbers](#) option.
- [Interwiki link failed for ticket:435](#), tokenize.e handles 0?NN numbers properly now. Returns [T\\_NUMBER](#) as the token type and either [TF\\_INT](#) or [TF\\_HEX](#) as the form. If [string\\_numbers](#) is enabled, the prefix is returned as part of the string, i.e. integer a = 0b0101 will return "0b0101".
- [Interwiki link failed for ticket:439](#), tokenize.e fixed breakage with slice operator due to new string number parsing.
- [Interwiki link failed for ticket:448](#), Fixed splice() translation.
- [Interwiki link failed for ticket:453](#), Reworked the way open files are cleaned up so that coverage works properly
- [Interwiki link failed for ticket:457](#), cmd\_parse() now correctly honors the NO\_HELP option and allows the coder to override the default help switches.
- [Interwiki link failed for ticket:461](#), Fixed error checking for invalid C routines for c\_func / c\_proc.
- [Interwiki link failed for ticket:463](#), Fixed large file support for MinGW
- [Interwiki link failed for ticket:464](#), Fixed translated for loops that could result in incorrectly emitted brackets
- [Interwiki link failed for ticket:465](#), Fixed stack space calculations for forward proc to func conversion and type checks.
- [Interwiki link failed for ticket:466](#), Fixed line reporting on compile time type check error.



- [Interwiki link failed for ticket:467](#), Fixed interpreter, translator and binder for handling multiple parameters when one comes from a eu.cfg file and the other from the command line, but the given option was designed to only be used ONCE, such as -batch
- [Interwiki link failed for ticket:469](#), Fixed translated block comments
- [Interwiki link failed for ticket:471](#), When using the -lib parameter to euc, it's canonical path is used and it's existence is checked before translation has begun to prevent wasting time only to find the linker fails.
- [Interwiki link failed for ticket:472](#), eui --help display is now in a logical display order.
- [Interwiki link failed for ticket:473](#), euc --help display is now in a logical display order.
- [Interwiki link failed for ticket:475](#), Fixed memory leak with interpreted `rand`
- [Interwiki link failed for ticket:476](#), euc can now translated single character base filenames, i.e. h.ex
- [Interwiki link failed for ticket:477](#), canonical\_path expansion of ~ now works in MSYS and CMD.exe with the MinGW build.
- [Interwiki link failed for ticket:479](#), Installer now writes a eu.cfg, appends at the confirmation of the user.
- [Interwiki link failed for ticket:481](#), RD\_INPLACE, RD\_PRESORTED, RD\_SORT are now documented individually.
- [Interwiki link failed for ticket:485](#), Fixed scanner initialization to prevent invalid accesses.
- [Interwiki link failed for ticket:490](#), Fixed large file support for Watcom
- [Interwiki link failed for ticket:491](#), cmd\_parse now appends everything after the first extra to the OPT\_EXTRAS entry when NO\_VALIDATION\_AFTER\_FIRST\_EXTRA is supplied as a parsing option.
- [Interwiki link failed for ticket:501](#), rand\_range(hi,lo) now works with lo > 30-bits.
- [Interwiki link failed for ticket:505](#), Fixed front-end command line processing
- [Interwiki link failed for ticket:509](#), fix pointer handling in regex back end code
- [Interwiki link failed for ticket:503](#), When translating, temps that were thought to be either sequences or objects, but were ultimately atoms were not having their possible min / max values reset, leading to incorrect C code being emitted.
- Fixed `seek` return value for large files on Linux.
- `connect` return value was documented incorrectly.
- std/cmdline.e, cmd\_parse sets the NO\_CASE option when option does not have HAS\_CASE.
- std/cmdline.e, cmd\_parse sets the NO\_PARAMETER option when option does not have HAS\_PARAMETER.
- std/cmdline.e, cmd\_parse sets the ONCE option when option does not have MULTIPLE.
- The euphoria coded backend (eu.ex) in some cases did not handle recursive calls correctly.
- Too numerous to list: Many documentation typo, spelling mistakes and formatting errors have been corrected.
- Removed many unnecessary `maybe_any_key` uses from the general demos
- net/http.e now properly handles { key,value}, ..., { encoding\_type, {key,val}, ...} and already encoded string data, "name=John%20Doe".
- Fixed eutext.ex and std/unittest.e. Under some circumstances, they would report 100% success even though there were some failures.
- Fixed std/filesys.e and std/locale.e for use on OpenBSD and NetBSD.
- Use POSIX random() to initialize random seed1 on non-Windows platforms.
- Updated t\_io.e as OpenBSD and NetBSD allows seeking on STDIN, STDOUT and STDERR.
- Now ensure internal C strings in be\_pcre are properly null terminated.
- Fixed version display information for NetBSD

## Enhancements/Changes

- [Interwiki link failed for ticket:334](#), \*nix generic distribution build scripts are now combined for easier maintenance.
- [Interwiki link failed for ticket:341](#), [Interwiki link failed for ticket:344](#), Many unused variables have been cleaned up in the standard library.
- [Interwiki link failed for ticket:363](#), euc now has an optional parameter-rc-file that will compile and bind the resource file to windows executables.
- [Interwiki link failed for ticket:411](#), documented the \$ character as applied to a sequence terminator.
- [Interwiki link failed for ticket:413](#), Qualified the standard library. With so many forward lookups this allows for a pretty large speedup when using multiple standard library includes.
- [Interwiki link failed for ticket:499](#), Add support for using '1', '2', and 'j' in place of F1, F2, and the DOWN\_ARROW key in the Trace screen. This allows Unix users to use trace(1) even if we don't recognize escape sequences for their particular \$TERM
- [Interwiki link failed for ticket:513](#), Moved `get_text` from `std/text.e` to `std/locale.e`
- manual-upload target was added to GNU and Watcom makefiles
- Formatted buzz.ex example
- euc will always remove it's temporary build directory unless the-keep option is supplied. If one wants to keep the build directory for some reason, they could probably use -build-dir as well, for example: `euc -build-dir my-build hello.ex` which will automatically keep the build directory since it was user specified.
- Converted bin/lines.ex to use new language constructs and the standard library as an example of how an application take advantage of 4.0. Code base went from 591 lines of code to 195 lines of code. Bugs were fixed, comment percentage calculations and header/footer lines were added in the new, smaller version as well.

This program also now has the ability to sort results by numerous options in normal or reverse order.

- Moved network demos from `demos/` to `demos/net` for better organization.
- `euphoria/tokenize.e` BLANK concept has changed. `tokenize` use to consume all newlines and only report double blanks. It now simply tokenizes the data and returns a newline if requested. `keep_blanks` has been renamed to `keep_newlines` and `T_BLANK` has been renamed to `T_NEWLINE`. Thus the tokenizer doesn't perform any 'parser' functions, it simply tokenizes the source.
- Added `show_tokens`, `token_names` and `token_forms` to `euphoria/tokenize.e` to help in debugging both `tokenize` internal routines and applications that make use of `euphoria/tokenize.e`
- Installer now creates a `eu.cfg` directory in `EUDIR/bin`
- Speed improvements to `map:put()`
- Demoted several "bin" programs to demos including:
  - `ascii.ex`
  - `eprint.ex`
  - `eused.ex`
  - `guru.ex`
  - `key.ex`
  - `search.ex`
  - `where.ex`
- All demos and bundled bin programs are unit tested to ensure at least eui - test passes
- Removed `analyze.ex` as it never was a finished, deployable product
- `eu.cfg` "win32" sections (`[win32]`, `[bind:win32]`, `[translate:win32]`, `[bind:win32]`) have now all been changed to "windows", not "win32"
- Removed `demo/demo.doc` and instead included in the header of each demo program what they do and included them into a section in the manual about demos.
- -test parameter now displays warnings as well as errors
- Translator speed optimizations.

- Improved logging and error checking for sock\_server.ex demo
- Renamed bin/lines.ex to bin/euloc.ex since it's more Euphoria centric now.
- Removed left-over translator command line parameter-fastfp which was for DOS only.
- Reuse memory buffer in HSIEH32 hash implementation
- abbreviate\_path is used to cleanup the display from euc regarding the Build Directory.
- printf's third argument is now optional. printf(1, "Hello\n", {}) is no longer needed, it can be shortened to printf(1, "Hello\n")
- Added another hashing algorithm. HSIEH30 is identical to HSIEH32 but will only ever return a 30-bit integer (a Euphoria integer).
- Removed hash elements from map.e and placed them in a new standard library module, hash.e
- Performance tweaks to maps.
- Removed support for emake.bat build scripts, please use direct build or makefiles, both of which euc supports directly.

## Version 4.0.0 Release Candidate 1 November 8, 2010

The release of Euphoria 4.0 is like no other. It's updates are massive. The change log here is not designed to detail every minor change that has taken place during the 4.0 development cycle. Included in this release note are the language changes only.

The entire standard library is brand new. The manual should be consulted to learn about the new standard library, it's changes are not documented here as it would just be a duplicate of the manual API sections. We will, however, mention a few major additions to the API library that has required binary changes in the backend:

### Major Library Additions

- Dictionary Type
- Regular Expressions
- Sockets

### Contributors

Another thing you will notice that is slightly different about this release note is that we are not attributing "Change ABC" to person "DEF." Many of the changes made have been an iterative process involving many people. Euphoria 4.0 has had a large number of contributors. We will, however, list all those that have contributed, the list is in last name alphabetical order:

- Jiri Babor
- Chris Bensler
- Jim C. Brown
- CoJaBo
- Jeremy Cowgar
- Robert Craig
- Chris Cuvier
- Jason Glade
- Ryan W. Johnson
- C.K. Lester
- Matthew Lewis
- Junko Miura
- Marco Antonio Achury Palma
- Derek Parnell
- Shawn Pringle
- Michael Sabal
- Kathy Smith
- Yuku (Aku)

If we have forgotten your name, please forgive us and bring it to our attention, the addition will be made promptly.

### Bug Fixes

- 1855414. `open()` max path length is now determined by the underlying operating system and not a generic default. `open()` also now returns -1 when the filename is too long instead of causing a fatal error.
- 1608870. `dir()` now handles \*.abc correctly, not showing a file ending with .abcd. `dir()` also now supports wildcard characters (\* and ?) on all platforms.

## Changes

- DOS support has been withdrawn. OpenEuphoria from version 4 onwards will not be specifically supporting DOS editions of the language.
- Comments may now be embedded in data passed to **value()** in **get.e**.
- Documentation moved to a new format.

## New Programs

- eutest - Unit testing system for Euphoria

## New Features

- New standard include files are in `include/std` to resolve many conflicts.
- Include file names with accent characters now supported.
- Enhanced symbol resolution to take into account information regarding which files were included by which files.
- Namespaces for a source file now can be used for identifiers in the specified file and for global identifiers in all files included by the specified file.
- Command line arguments for the translator allow for creating binaries with debugging symbols, and to specify a different runtime library.
- In trace mode, '?' will show the last defined variable of the requested name.
- Include directories can now be specified based on command line arguments and config files in addition to environment variables.
- Improved accuracy in scanning numbers in scientific notation. Scanned numbers are accurate to the full precision of the IEEE 754 floating point standard.
- New **loop do ... until** *condition* end loop construct, which differs from a while loop in that it performs its test at the end of the block, rather than at the start.
- New keywords to give greater control over the instruction flow:
  - **continue**: start next iteration of a loop;
  - **retry**: restarts the current iteration of a loop
  - **entry**: marks the entry point into a loop, skipping initial test
  - **break**: exit an if block or switch block
  - **goto**: jump to a label that is in the same scope
- The **exit**, **break**, **continue** and **retry** keywords now can take an optional parameter, which enables to exit several blocks at a time, or (re)starting an iteration of a loop which is not the innermost one.
- Block headers now may mention a label. This label can be used as the optional parameter of flow control keywords.
- Variables can now be initialized right on the spot at which they are declared, just like constants.
- Any routine parameter can be defaulted, i.e. given a default value that is plugged in if omitted on a call. Any expression can be used, and parameters of the same call can even be used.
- New **switch ... end switch** construct, which more efficiently implements a series of **elsif**, using the compact **case** statement.
- **Unit testing added to Euphoria.**
- Condition compiling keywords (**ifdef**, **elsifdef**, **end ifdef**) and **with define=xyz** or command line **-D XYZ** to insert/omit code in interpreter IL code and in translated C code.
- New enum keyword that allows for *parse time* sequential constant creation.
- The namespace **eu** is predefined, and can be used to fully qualify built-in routines.
- `with warning` has been enhanced in order to individually turn warnings on or off.
- New scope: **export**. Identifiers with the export scope can only be seen from

files that:

1. directly include the file where the identifiers are defined
- New scope: **public**. Identifiers with the public scope can only be seen from files that:
    1. directly include the file where the identifiers are defined
    2. directly include a file that uses the "public include file.e" construct to pass public identifiers
  - Routine resolution changes
    1. Routines the same name as an internal no longer override the internal by default. You must use the keyword **override**.
    2. An unqualified call to routine that exists as an internal calls the internal unless overridden with the override keyword. global, public and export functions are not called. A namespace must be used.
  - -STRICT option added that will display **all** warnings regardless of the file's with/without warning setting.
  - -BATCH option designed to run in an automated environment. Causes any "Press Enter" type prompt due to error to be suppressed. Exit code will be 1 on success, 0 on failure as normal.
  - -TEST option allows for editing/IDE environments to perform a syntax check on the euphoria code in question. Causes euphoria interpreter to do all parsing, syntax checking, etc... but does not execute the code. Exit code will be 1 on success, 0 on failure as normal. Editors/IDE's may need both -test and -batch.
  - dis.ex (in the source directory) will parse a euphoria program and output the symbol table and the IL code in a readable format.
  - Variables may be in any part of a routine, or **infor**, **while**, **if**, **loop** and **switch** blocks, in which case the scope of the variable ends when its block ends.

## What's new in 4.0?

Euphoria v4.0 is a very large jump in functionality from the previous stable release, 3.1.1.

Euphoria has a brand new standard library consisting of over 800 public members. Too numerous to list here, please see the reference section of this manual.

## General Changes

- New manual and documentation system
- New logo
- Switched to using our own ticket system
- Switched to using our own self hosted Mercurial SCM system

## Executable name changes

Old	New	Description
ex and exwc	eui	Euphoria Interpreter
ec and ecw	euc	Euphoria to C Translator
bind.bat and bind	eubind	Euphoria Binder
shroud.bat and shroud	eushroud	Euphoria Shrouder

## Language Enhancements

- Conditional compilation using the `ifdef` statement.
- **Raw strings**, which can include multilined text.
- **Multiline comments** using the C-styled comments `/* .. */`, which can be nested.
- **Binary, Octal and alternative Decimal and Hexadecimal number format**-0b10 (2), 0t10 (8), 0d10 (10), 0x10 (16)
- **Hexadecimal string** formats. Use `\x` to embed any byte value into a standard string, or create an entire hexadecimal byte string using `x" ... "`
- Function results can now be ignored.
- Optional list terminator. The final item in a list can be the dollar symbol `$`). This is just a place holder for the *end-of-list*, making it easier to add and delete items from the source code without having to adjust the commas.
- **Enumerated values/types** (enum, enum type)
- Built-in eu: **namespace**
- Declare variable anywhere, not just at the top of a routine.
- Scoped variables (declared inside an if for example)
- Assign on declaration. You can now declare a variable and assign it an initial value on the same statement.
- The `object()` built-in function can now be used to safely test if a variable has been initialized or not.
- Forward referencing. You no longer need to lexically declare a routine before using it.
- Additional loop constructs ...
  - **loop/until**
  - You can **label** a loop
  - while X with **entry**
  - **exit, continue, retry**. All with an optional "label"
  - **goto**
- Additional conditional constructs

- switch statement with or without **fallthru**
  - You can **label** an if or switch
  - break keyword allows exiting from **if / switch** blocks
- Default/optional parameters for routines
- Additional scope modifiers
  - **export**
  - **public** (public include)
  - **override**
- Built in **sockets**
- Built in **Regular Expressions**
- Resource clean up that can be triggered manually, or when an object's reference count goes to zero
- Automatic inlining of small routines, **with / without inline**
- Built in, optimized sequence operations (**remove**, **insert**, **splice**, **replace**, **head**, **tail**)
- Built in peek and poke 2 byte values, 1 byte signed values, peek null terminated strings, **peek**, **peek2**, **peek\_string**, **poke** and **poke2**
- Fine grained control over which, if any, warnings will be generated by Euphoria, **with / without warning**.

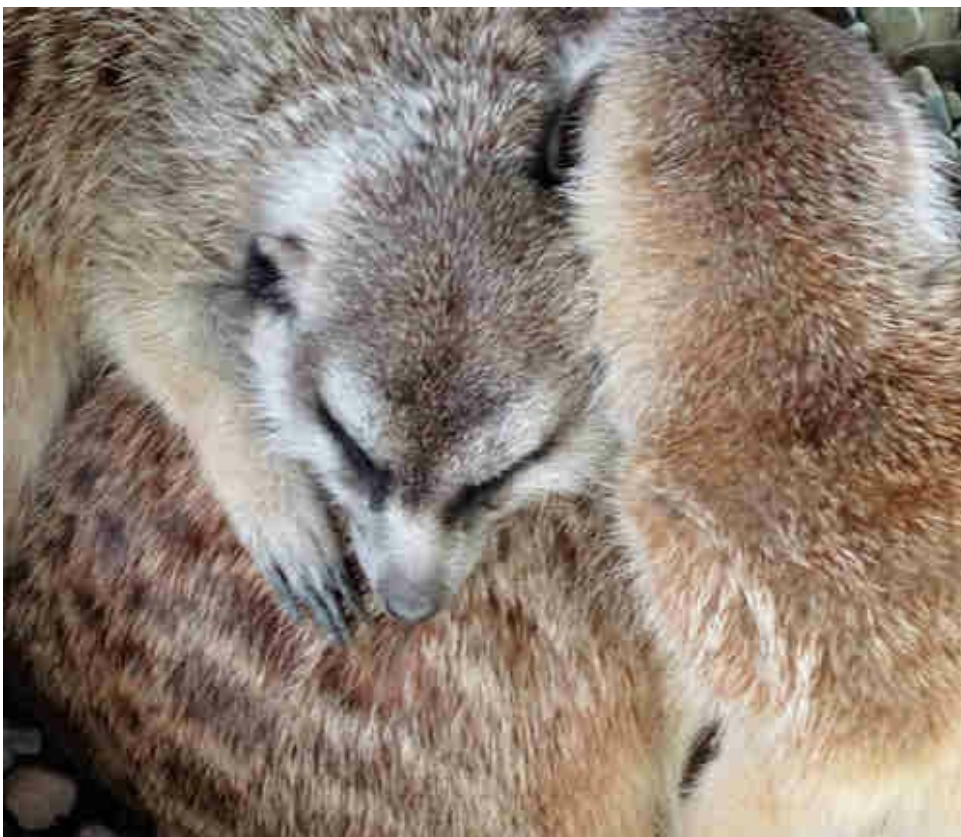
## Tool Additions / Enhancements

- General
  - **User Defined Preprocessor**
  - **Configuration system** (eu.cfg)
  - Version display for all tools
- Interpreter
  - New test mode, **Command line switches**
  - Batch mode for unattended execution such as a CGI application, **Command line switches**
- **Translator**
  - Compiles directly
  - Can compile in debug mode using the -debug argument
  - Can write a makefile
  - Can compile/bind a resource file on Windows
  - Now includes eudbg.lib, eu.a and eudbg.a files in addition to the eu.lib file enabling one to link against debug libraries and also use the MinGW compiler directly without having to recompile sources.
- New independent shrouder
- Coverage Analysis
- Disassembler
- **EuDist - Distributing Programs**
- **EuDOC - Source Documentation Tool**
- **EuTEST - Unit Testing**



# Changelog

# INDEX



- [glossary](#)
- [index](#)

## Glossary

Index of controlled vocabulary used in this documentation.

# Index