

# Euphoria Programming Language

version 3.0

## Reference Manual

(c) 2005 Rapid Deployment Software

Permission is freely granted to anyone  
to copy this manual.

## TABLE OF CONTENTS

### Part I - Core Language

1. Introduction
  - 1.1 Example Program
  - 1.2 Installation
  - 1.3 Running a Program
    - 1.3.1 Running under Windows
    - 1.3.2 Use of a Swap File
  - 1.4 Editing a Program
  - 1.5 Distributing a Program
    - 1.5.1 Licensing
2. Language Definition
  - 2.1 Objects
    - 2.1.1 Atoms and Sequences
    - 2.1.2 Character Strings and Individual Characters
    - 2.1.3 Comments
  - 2.2 Expressions
    - 2.2.1 Relational Operators
    - 2.2.2 Logical Operators
    - 2.2.3 Arithmetic Operators
    - 2.2.4 Operations on Sequences
    - 2.2.5 Subscripting of Sequences
    - 2.2.6 Slicing of Sequences
    - 2.2.7 Concatenation of Sequences and Atoms - The & Operator
    - 2.2.8 Sequence-Formation
    - 2.2.9 Other Operations on Sequences
      - o length
      - o repeat

- [append / prepend](#)
- 2.2.10 [Precedence Chart](#)
- 2.3 [Euphoria versus Conventional Languages](#)
- 2.4 [Declarations](#)
- 2.4.1 [Identifiers](#)
  - [procedures](#)
  - [functions](#)
  - [types](#)
  - [variables](#)
  - [constants](#)
- 2.4.2 [Scope](#)
- 2.4.3 [Specifying the Type of a Variable](#)
- 2.5 [Statements](#)
- 2.5.1 [assignment statement](#)
  - [Assignment with Operator](#)
- 2.5.2 [procedure call](#)
- 2.5.3 [if statement](#)
- 2.5.4 [while statement](#)
  - [Short-Circuit Evaluation](#)
- 2.5.5 [for statement](#)
- 2.5.6 [return statement](#)
- 2.5.7 [exit statement](#)
- 2.6 [Special Top-Level Statements](#)
- 2.6.1 [include](#)
- 2.6.2 [with / without](#)
- 3. [Debugging and Profiling](#)
  - 3.1 [Debugging](#)
    - 3.1.1 [The Trace Screen](#)
    - 3.1.2 [The Trace File](#)
  - 3.2 [Profiling](#)
    - 3.2.1 [Some Further Notes on Time Profiling](#)

## Part II - Library Routines

- 1. [Introduction](#)
- 2. [Routines by Application Area](#)
  - 2.1 [Predefined Types](#)
  - 2.2 [Sequence Manipulation](#)
  - 2.3 [Searching and Sorting](#)
  - 2.4 [Pattern Matching](#)
  - 2.5 [Math](#)

- 2.6 [Bitwise Logical Operations](#)
  - 2.7 [File and Device I/O](#)
  - 2.8 [Mouse Support \(DOS32\)](#)
  - 2.9 [Operating System](#)
  - 2.10 [Special Machine-Dependent Routines](#)
  - 2.11 [Debugging](#)
  - 2.12 [Graphics & Sound](#)
  - 2.13 [Machine Level Interface](#)
  - 2.14 [Dynamic Calls](#)
  - 2.15 [Calling C Functions](#)
  - 2.16 [Multitasking](#)
3. Alphabetical Listing of all Routines
- [From A to B](#)
  - [From C to D](#)
  - [From E to G](#)
  - [From H to O](#)
  - [From P to R](#)
  - [From S to T](#)
  - [From U to Z](#)

... continue [Part I - Core Language](#)

# Part I - Core Language

## 1. Introduction

**Euphoria** is a programming language with the following advantages over conventional languages:

- a remarkably simple, flexible, powerful language definition that is easy to learn and use.
- dynamic storage allocation. Variables grow or shrink without the programmer having to worry about allocating and freeing chunks of memory. Objects of any size can be assigned to an element of a Euphoria sequence (array).
- a high-performance, state-of-the-art interpreter that's **30 times** faster than conventional interpreters such as Perl and Python.
- an optimizing Euphoria To C Translator, that can boost your speed even further, often by a factor of 2x to 5x versus the already-fast interpreter.
- extensive run-time checking for: out-of-bounds subscripts, uninitialized variables, bad parameter values for library routines, illegal value assigned to a variable and many more. There are no mysterious machine exceptions -- you will always get a full English description of any problem that occurs with your program at run-time, along with a call-stack trace-back and a dump of all of your variable values. Programs can be debugged quickly, easily and more thoroughly.
- features of the underlying hardware are completely hidden. Programs are not aware of word-lengths, underlying bit-level representation of values, byte-order etc.
- a full-screen source debugger and an execution profiler are included, along with a full-screen, multi-file editor. On a color monitor, the editor displays Euphoria programs in multiple colors, to highlight comments, reserved words, built-in functions, strings, and level of nesting of brackets. It optionally performs auto-completion of statements, saving you typing effort and reducing syntax errors. This editor is written in Euphoria, and the source code is provided to you without restrictions. You are free to modify it, add features, and redistribute it as you wish.
- Euphoria programs run under Linux, FreeBSD, 32-bit Windows, and any DOS environment, and are not subject to any 640K memory limitations. You can create programs that use the full multi-megabyte memory of your computer, and a swap file is automatically used when a program needs more memory than exists on your machine.
- You can make a single, stand-alone .exe file from your program.
- Euphoria routines are naturally generic. The example program below shows a single routine that will sort any type of data -- integers, floating-point numbers, strings etc. Euphoria is not an "object-oriented" language, yet it achieves many of the benefits of these languages in a much simpler way.
- Euphoria is completely free and open source.

## 1.1 Example Program

The following is an example of a complete Euphoria program.

```
~~~~~

sequence list, sorted_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 list
    list = {9, 10, 3, 1, 4, 5, 8, 7, 6, 2}
    sorted_list = merge_sort(list)
    ? sorted_list
end procedure

print_sorted_list()      -- this command starts the program

~~~~~
```

The above example contains 4 separate commands that are processed in order. The first declares two variables: list and sorted\_list to be **sequences** (flexible arrays). The second defines a **function** merge\_sort(). The third defines a **procedure** print\_sorted\_list(). The final command calls procedure print\_sorted\_list().

The output from the program will be:

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.

**merge\_sort()** will just as easily sort {1.5, -9, 1e6, 100} or {"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 "ex allsorts" to see timings on several different sorting algorithms for increasing numbers of objects. For a quick tutorial example of Euphoria programming see **euphoria\demo\bench\filesort.ex**.

---

## 1.2 Installation

To install Euphoria on your machine, first read the file **install.doc**. Installation simply involves copying the **euphoria** files to your hard disk under a directory named "euphoria", and then modifying your **autoexec.bat** file so that **euphoria\bin** is on your search path, and the environment variable **EUDIR** is set to the euphoria directory.

When installed, the **euphoria** directory will look something like this:

### **\euphoria**

readme.doc

readme.htm

License.txt

### **\bin**

Interpreters ex.exe and exw.exe. Translators ec.exe and ecw.exe. Or on Linux/FreeBSD, Interpreter exu and Translator ecu. There are also utility programs such as ed.bat, guru.bat etc.

### **\include**

standard include files, e.g. graphics.e

### **\source**

the complete source code (interpreter, translator, binder)

### **\doc**

refman.doc, library.doc, and several other plain-text documentation files

### **\html**

HTML files corresponding to each of the .doc files in the doc directory

### **\tutorial**

small tutorial programs to help you learn Euphoria

### **\demo**

generic demo programs that run on all platforms

### **\dos32**

DOS32-specific demo programs (optional)

### **\win32**

WIN32-specific demo programs (optional)

### **\linux**

Linux/FreeBSD-specific demo programs (optional)

### **\langwar**

language war game (pixel-graphics version for DOS, or text version for Linux/FreeBSD)

### **\bench**

benchmark programs

The Linux subdirectory is not included in the DOS/Windows distribution, and the dos32 and win32 subdirectories are not included in the Linux/FreeBSD distribution. In this manual, directory names are shown

using backslash (\). Linux/FreeBSD users should substitute forward slash (/).

---

### 1.3 Running a Program

Euphoria programs are executed by typing **ex**, **exw** or **exu** followed by the name of the main Euphoria file. You can type additional words (known as **arguments**) on this line, known as the **command-line**. Your program can call the built-in function [command\\_line\(\)](#) to read the command-line. The DOS32 version of the Euphoria interpreter is called **ex.exe**. The WIN32 version is called **exw.exe**. The Linux/FreeBSD version is called **exu**. By convention, main Euphoria files have an extension of **.ex**, **.exw** or **.exu**. Other Euphoria files, that are meant to be included in a larger program, end in **.e** or sometimes **.ew** or **.eu**. To save typing, you can leave off the ".ex", and the **ex** command will supply it for you automatically. **exw.exe** will supply ".exw", and **exu** will supply ".exu". If the file can't be found in the current directory, your PATH will be searched. You can redirect standard input and standard output when you run a Euphoria program, for example:

```
ex filesort.ex < raw.txt > sorted.txt
```

or simply,

```
ex filesort < raw.txt > sorted.txt
```

Unlike many other compilers and interpreters, there are no special command-line options for **ex**, **exw** or **exu**. Only the name of your Euphoria file is expected, and if you don't supply it, you will be prompted for it.

For frequently-used programs under DOS/Windows you might want to make a small **.bat** (batch) file, perhaps called **myprog.bat**, containing two statements like:

```
@echo off
ex myprog.ex %1 %2 %3
```

The first statement turns off echoing of commands to the screen. The second runs **ex myprog.ex** with up to 3 command-line arguments. See [command\\_line\(\)](#) for an example of how to read these arguments. If your program takes more arguments, you should add %4 %5 etc. Having a .bat file will save you the minor inconvenience of typing **ex** (or **exw**) all the time, i.e. you can just type:

```
myprog
```

instead of:

```
ex myprog
```

Unfortunately DOS will not allow redirection of standard input and output when you use a **.bat** file

Under Linux/FreeBSD, you can type the path to the Euphoria interpreter on the first line of your main file, e.g. if your program is called foo.exu:

```
#!/home/rob/euphoria/bin/exu

procedure foo()
    ? 2+2
end procedure

foo()
```

Then if you make your file executable:

```
chmod +x foo.exu
```

You can just type:

```
foo.exu
```

to run your program. You could even shorten the name to simply "foo". Euphoria ignores the first line when it starts with `#!`. Be careful though that your first line ends with the Linux/FreeBSD-style `\n`, and not the DOS/Windows-style `\r\n`, or the Linux/FreeBSD shell might get confused. If your file is shrouded, you must give the path to backendu, not exu.

You can also run **bind.bat** (DOS32), or **bindw.bat** (WIN32) or **bindu** (Linux/FreeBSD) to combine your Euphoria program with **ex.exe**, **exw.exe** or **exu**, to make a stand-alone executable file (**.exe** file on DOS/Windows). With a stand-alone **.exe** file you *can* redirect standard input and output. Binding is discussed further in [1.5 Distributing a Program](#).

Using the [Euphoria to C Translator](#), you can also make a stand-alone **.exe** file, and it will normally run much faster than a bound program.

**exu** or **ex.exe** and **exw.exe** will be in the **euphoria\bin** directory which must be on your search path. The environment variable EUDIR should be set to the main Euphoria directory, e.g. **c:\euphoria**.

### 1.3.1 Running under Windows

You can run Euphoria programs directly from the Windows environment, or from a DOS shell that you have opened from Windows. By "associating" **.ex** files with **ex.exe**, and **.exw** files with **exw.exe** you can simply double-click on a **.ex** or **.exw** file to run it. Under Windows you would define a new file type for **.ex**, by clicking on My Computer / view / options / file types. It is possible to have several Euphoria programs active in different windows. If you turn your program into a **.exe** file, you can simply double-click on it to run it.

### 1.3.2 Use of a Swap File

If you run a Euphoria program under Linux/FreeBSD or Windows (or in a DOS shell under Windows), and the program runs out of physical memory, it will start using "virtual memory". The operating system provides this virtual memory automatically by swapping out the least-recently-used code and data to a system swap file. To change the size of the Windows swap file, click on Control Panel / 386 Enhanced / "virtual memory...". Under OS/2 you can adjust the "DPMI\_MEMORY\_LIMIT" by clicking the Virtual DOS machine icon / "DOS Settings" to allocate more extended memory for your program.

Under pure DOS, outside of Windows, there is no system swap file so the DOS-extender built in to **ex.exe** (DOS32) will create one for possible use by your program. See [platform.doc](#).

## 1.4 Editing a Program

You can use any text editor to edit a Euphoria program. However, Euphoria comes with its own special editor that is written entirely in Euphoria. Type: **ed** followed by the complete name of the file you wish to edit (the **.ex/.exw/.exu** extension is not assumed). You can use this editor to edit any kind of text file. When you edit a Euphoria file, some extra features such as color syntax highlighting and auto-completion of certain



statements, are available to make your job easier.

Whenever you run a Euphoria program and get an error message, during compilation or execution, you can simply type **ed** with no file name and you will be automatically positioned in the file containing the error, at the correct line and column, and with the error message displayed at the top of the screen.

Under Windows you can associate **ed.bat** with various kinds of text files that you want to edit. Color syntax highlighting is provided for **.ex**, **.exw**, **.exu**, **.e**, **.ew**, **.eu**, and **.pro** ([profile](#)) files.

Most keys that you type are inserted into the file at the cursor position. Hit the **Esc** key once to get a menu bar of special commands. The arrow keys, and the Insert/Delete/Home/End/PageUp/PageDown keys are also active. Under Linux/FreeBSD some keys may not be available, and alternate keys are provided. See the file **euphoria\doc\ed.doc** ([euphoria/html/ed.htm](#)) for a complete description of the editing commands. **Esc h** (help) will let you view **ed.doc** from your editing session.

If you need to understand or modify any detail of the editor's operation, you can edit the file **ed.ex** in **euphoria\bin** (be sure to make a backup copy so you don't lose your ability to edit). If the name **ed** conflicts with some other command on your system, simply rename the file **euphoria\bin\ed.bat** to something else. Because this editor is written in Euphoria, it is remarkably concise and easy to understand. The same functionality implemented in a language like C, would take far more lines of code.

**ed** is a simple text-mode editor that runs on DOS, Linux, FreeBSD and the Windows console. See also **David Cuny's** excellent **ee.ex** editor for DOS and Linux/FreeBSD. You can download **ee.ex** from the [Euphoria Web site](#). There are also some Windows editors oriented to Euphoria. These are also on [the Web site](#).

---

## 1.5 Distributing a Program

Euphoria provides you with 4 distinct ways of distributing a program.

In the first method you simply ship your users **ex.exe** or **exw.exe** or **exu** file, along with your main Euphoria **.ex**, **.exw**, or **.exu** file and any **.e** include files that are needed (including any of the standard ones from **euphoria\include**). If the Euphoria source files are placed together in one directory and **ex.exe**, **exw.exe** or **exu** is placed in the same directory or somewhere on the search path, then your user can run your program by typing **ex** (**exw**) or (**exu**) followed by the path of your main **.ex**, **.exw**, or **.exu** file. You might also provide a small **.bat** file so people won't actually have to type **ex** (**exw**). This method assumes that you are willing to share your Euphoria source code with your users.

The Binder gives you two more methods of distribution. You can **shroud** your program, or you can **bind** your program. **Shrouding** combines all of the **.e** files that your program needs, along with your main file to create a single **.il** file. **Binding** combines your shrouded program with **backend.exe**, **backendw.exe** or **backendu** to create a **single, stand-alone executable (.exe)** file. For example, if your program is called "myprog.ex" you can create "myprog.exe" which will run identically. For more information about shrouding and binding, see [bind.doc](#).

Finally, with the [Euphoria To C Translator](#), you can **translate** your Euphoria program into C and then compile it with a C compiler to get an executable (**.exe**) file.

### 1.5.1 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 **ex.exe**, **exw.exe** and **exu** files so anyone can run your program. You can also distribute the interpreter backend: **backend.exe**, **backendw.exe** and **backendu**. You can **shroud** or **bind** 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 euphoria\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.RapidEuphoria.com> of our Web page, but we do not require any such acknowledgment.

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 [Euphoria To C Translator](#).

... continue [2. Language Definition](#)

# Part I - Core Language

## 1. Introduction

**Euphoria** is a programming language with the following advantages over conventional languages:

- a remarkably simple, flexible, powerful language definition that is easy to learn and use.
- dynamic storage allocation. Variables grow or shrink without the programmer having to worry about allocating and freeing chunks of memory. Objects of any size can be assigned to an element of a Euphoria sequence (array).
- a high-performance, state-of-the-art interpreter that's **30 times** faster than conventional interpreters such as Perl and Python.
- an optimizing Euphoria To C Translator, that can boost your speed even further, often by a factor of 2x to 5x versus the already-fast interpreter.
- extensive run-time checking for: out-of-bounds subscripts, uninitialized variables, bad parameter values for library routines, illegal value assigned to a variable and many more. There are no mysterious machine exceptions -- you will always get a full English description of any problem that occurs with your program at run-time, along with a call-stack trace-back and a dump of all of your variable values. Programs can be debugged quickly, easily and more thoroughly.
- features of the underlying hardware are completely hidden. Programs are not aware of word-lengths, underlying bit-level representation of values, byte-order etc.
- a full-screen source debugger and an execution profiler are included, along with a full-screen, multi-file editor. On a color monitor, the editor displays Euphoria programs in multiple colors, to highlight comments, reserved words, built-in functions, strings, and level of nesting of brackets. It optionally performs auto-completion of statements, saving you typing effort and reducing syntax errors. This editor is written in Euphoria, and the source code is provided to you without restrictions. You are free to modify it, add features, and redistribute it as you wish.
- Euphoria programs run under Linux, FreeBSD, 32-bit Windows, and any DOS environment, and are not subject to any 640K memory limitations. You can create programs that use the full multi-megabyte memory of your computer, and a swap file is automatically used when a program needs more memory than exists on your machine.
- You can make a single, stand-alone .exe file from your program.
- Euphoria routines are naturally generic. The example program below shows a single routine that will sort any type of data -- integers, floating-point numbers, strings etc. Euphoria is not an "object-oriented" language, yet it achieves many of the benefits of these languages in a much simpler way.
- Euphoria is completely free and open source.

## 1.1 Example Program

The following is an example of a complete Euphoria program.

```
~~~~~

sequence list, sorted_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 list
    list = {9, 10, 3, 1, 4, 5, 8, 7, 6, 2}
    sorted_list = merge_sort(list)
    ? sorted_list
end procedure

print_sorted_list()    -- this command starts the program

~~~~~
```

The above example contains 4 separate commands that are processed in order. The first declares two variables: list and sorted\_list to be **sequences** (flexible arrays). The second defines a **function** merge\_sort(). The third defines a **procedure** print\_sorted\_list(). The final command calls procedure print\_sorted\_list().

The output from the program will be:

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.

**merge\_sort()** will just as easily sort {1.5, -9, 1e6, 100} or {"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 "ex allsorts" to see timings on several different sorting algorithms for increasing numbers of objects. For a quick tutorial example of Euphoria programming see [euphoria\demo\bench\filesort.ex](#).

---

## 1.2 Installation

To install Euphoria on your machine, first read the file [install.doc](#). Installation simply involves copying the **euphoria** files to your hard disk under a directory named "euphoria", and then modifying your **autoexec.bat** file so that [euphoria\bin](#) is on your search path, and the environment variable **EUDIR** is set to the euphoria directory.

When installed, the **euphoria** directory will look something like this:

### [\euphoria](#)

readme.doc

readme.htm

License.txt

### [\bin](#)

Interpreters ex.exe and exw.exe. Translators ec.exe and ecw.exe. Or on Linux/FreeBSD, Interpreter exu and Translator ecu. There are also utility programs such as ed.bat, guru.bat etc.

### [\include](#)

standard include files, e.g. graphics.e

### [\source](#)

the complete source code (interpreter, translator, binder)

### [\doc](#)

refman.doc, library.doc, and several other plain-text documentation files

### [\html](#)

HTML files corresponding to each of the .doc files in the doc directory

### [\tutorial](#)

small tutorial programs to help you learn Euphoria

### [\demo](#)

generic demo programs that run on all platforms

### [\dos32](#)

DOS32-specific demo programs (optional)

### [\win32](#)

WIN32-specific demo programs (optional)

### [\linux](#)

Linux/FreeBSD-specific demo programs (optional)

### [\langwar](#)

language war game (pixel-graphics version for DOS, or text version for Linux/FreeBSD)

### [\bench](#)

benchmark programs

The Linux subdirectory is not included in the DOS/Windows distribution, and the dos32 and win32 subdirectories are not included in the Linux/FreeBSD distribution. In this manual, directory names are shown

using backslash (\). Linux/FreeBSD users should substitute forward slash (/).

---

### 1.3 Running a Program

Euphoria programs are executed by typing **ex**, **exw** or **exu** followed by the name of the main Euphoria file. You can type additional words (known as **arguments**) on this line, known as the **command-line**. Your program can call the built-in function [command\\_line\(\)](#) to read the command-line. The DOS32 version of the Euphoria interpreter is called **ex.exe**. The WIN32 version is called **exw.exe**. The Linux/FreeBSD version is called **exu**. By convention, main Euphoria files have an extension of **.ex**, **.exw** or **.exu**. Other Euphoria files, that are meant to be included in a larger program, end in **.e** or sometimes **.ew** or **.eu**. To save typing, you can leave off the ".ex", and the **ex** command will supply it for you automatically. **exw.exe** will supply ".exw", and **exu** will supply ".exu". If the file can't be found in the current directory, your PATH will be searched. You can redirect standard input and standard output when you run a Euphoria program, for example:

```
ex filesort.ex < raw.txt > sorted.txt
```

or simply,

```
ex filesort < raw.txt > sorted.txt
```

Unlike many other compilers and interpreters, there are no special command-line options for **ex**, **exw** or **exu**. Only the name of your Euphoria file is expected, and if you don't supply it, you will be prompted for it.

For frequently-used programs under DOS/Windows you might want to make a small **.bat** (batch) file, perhaps called **myprog.bat**, containing two statements like:

```
@echo off
ex myprog.ex %1 %2 %3
```

The first statement turns off echoing of commands to the screen. The second runs **ex myprog.ex** with up to 3 command-line arguments. See [command\\_line\(\)](#) for an example of how to read these arguments. If your program takes more arguments, you should add %4 %5 etc. Having a .bat file will save you the minor inconvenience of typing **ex** (or **exw**) all the time, i.e. you can just type:

```
myprog
```

instead of:

```
ex myprog
```

Unfortunately DOS will not allow redirection of standard input and output when you use a **.bat** file

Under Linux/FreeBSD, you can type the path to the Euphoria interpreter on the first line of your main file, e.g. if your program is called foo.exu:

```
#!/home/rob/euphoria/bin/exu

procedure foo()
    ? 2+2
end procedure

foo()
```

Then if you make your file executable:

```
chmod +x foo.exu
```

You can just type:

```
foo.exu
```

to run your program. You could even shorten the name to simply "foo". Euphoria ignores the first line when it starts with **#!**. Be careful though that your first line ends with the Linux/FreeBSD-style `\n`, and not the DOS/Windows-style `\r\n`, or the Linux/FreeBSD shell might get confused. If your file is shrouded, you must give the path to backendu, not exu.

You can also run **bind.bat** (DOS32), or **bindw.bat** (WIN32) or **bindu** (Linux/FreeBSD) to combine your Euphoria program with **ex.exe**, **exw.exe** or **exu**, to make a stand-alone executable file (**.exe** file on DOS/Windows). With a stand-alone **.exe** file you *can* redirect standard input and output. Binding is discussed further in [1.5 Distributing a Program](#).

Using the [Euphoria to C Translator](#), you can also make a stand-alone **.exe** file, and it will normally run much faster than a bound program.

**exu** or **ex.exe** and **exw.exe** will be in the **euphoria\bin** directory which must be on your search path. The environment variable EUDIR should be set to the main Euphoria directory, e.g. **c:\euphoria**.

### 1.3.1 Running under Windows

You can run Euphoria programs directly from the Windows environment, or from a DOS shell that you have opened from Windows. By "associating" **.ex** files with **ex.exe**, and **.exw** files with **exw.exe** you can simply double-click on a **.ex** or **.exw** file to run it. Under Windows you would define a new file type for **.ex**, by clicking on My Computer / view / options / file types. It is possible to have several Euphoria programs active in different windows. If you turn your program into a **.exe** file, you can simply double-click on it to run it.

### 1.3.2 Use of a Swap File

If you run a Euphoria program under Linux/FreeBSD or Windows (or in a DOS shell under Windows), and the program runs out of physical memory, it will start using "virtual memory". The operating system provides this virtual memory automatically by swapping out the least-recently-used code and data to a system swap file. To change the size of the Windows swap file, click on Control Panel / 386 Enhanced / "virtual memory...". Under OS/2 you can adjust the "DPMI\_MEMORY\_LIMIT" by clicking the Virtual DOS machine icon / "DOS Settings" to allocate more extended memory for your program.

Under pure DOS, outside of Windows, there is no system swap file so the DOS-extender built in to **ex.exe** (DOS32) will create one for possible use by your program. See [platform.doc](#).

## 1.4 Editing a Program

You can use any text editor to edit a Euphoria program. However, Euphoria comes with its own special editor that is written entirely in Euphoria. Type: **ed** followed by the complete name of the file you wish to edit (the **.ex/.exw/.exu** extension is not assumed). You can use this editor to edit any kind of text file. When you edit a Euphoria file, some extra features such as color syntax highlighting and auto-completion of certain

statements, are available to make your job easier.

Whenever you run a Euphoria program and get an error message, during compilation or execution, you can simply type **ed** with no file name and you will be automatically positioned in the file containing the error, at the correct line and column, and with the error message displayed at the top of the screen.

Under Windows you can associate **ed.bat** with various kinds of text files that you want to edit. Color syntax highlighting is provided for **.ex**, **.exw**, **.exu**, **.e**, **.ew**, **.eu**, and **.pro** ([profile](#)) files.

Most keys that you type are inserted into the file at the cursor position. Hit the **Esc** key once to get a menu bar of special commands. The arrow keys, and the Insert/Delete/Home/End/PageUp/PageDown keys are also active. Under Linux/FreeBSD some keys may not be available, and alternate keys are provided. See the file **euphoria\doc\ed.doc** ([euphoria/html/ed.htm](#)) for a complete description of the editing commands. **Esc h** (help) will let you view **ed.doc** from your editing session.

If you need to understand or modify any detail of the editor's operation, you can edit the file **ed.ex** in **euphoria\bin** (be sure to make a backup copy so you don't lose your ability to edit). If the name **ed** conflicts with some other command on your system, simply rename the file **euphoria\bin\ed.bat** to something else. Because this editor is written in Euphoria, it is remarkably concise and easy to understand. The same functionality implemented in a language like C, would take far more lines of code.

**ed** is a simple text-mode editor that runs on DOS, Linux, FreeBSD and the Windows console. See also **David Cuny's** excellent **ee.ex** editor for DOS and Linux/FreeBSD. You can download **ee.ex** from the [Euphoria Web site](#). There are also some Windows editors oriented to Euphoria. These are also on [the Web site](#).

---

## 1.5 Distributing a Program

Euphoria provides you with 4 distinct ways of distributing a program.

In the first method you simply ship your users **ex.exe** or **exw.exe** or **exu** file, along with your main Euphoria **.ex**, **.exw**, or **.exu** file and any **.e** include files that are needed (including any of the standard ones from **euphoria\include**). If the Euphoria source files are placed together in one directory and **ex.exe**, **exw.exe** or **exu** is placed in the same directory or somewhere on the search path, then your user can run your program by typing **ex** (**exw**) or (**exu**) followed by the path of your main **.ex**, **.exw**, or **.exu** file. You might also provide a small **.bat** file so people won't actually have to type **ex** (**exw**). This method assumes that you are willing to share your Euphoria source code with your users.

The Binder gives you two more methods of distribution. You can **shroud** your program, or you can **bind** your program. **Shrouding** combines all of the **.e** files that your program needs, along with your main file to create a single **.il** file. **Binding** combines your shrouded program with **backend.exe**, **backendw.exe** or **backendu** to create a **single, stand-alone executable (.exe)** file. For example, if your program is called "myprog.ex" you can create "myprog.exe" which will run identically. For more information about shrouding and binding, see [bind.doc](#).

Finally, with the [Euphoria To C Translator](#), you can **translate** your Euphoria program into C and then compile it with a C compiler to get an executable (**.exe**) file.



### 1.5.1 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 **ex.exe**, **exw.exe** and **exu** files so anyone can run your program. You can also distribute the interpreter backend: **backend.exe**, **backendw.exe** and **backendu**. You can **shroud** or **bind** 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 euphoria\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.RapidEuphoria.com> of our Web page, but we do not require any such acknowledgment.

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 [Euphoria To C Translator](#).

... continue [2. Language Definition](#)

## 1.1 Example Program

The following is an example of a complete Euphoria program.

```
~~~~~

sequence list, sorted_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 list
    list = {9, 10, 3, 1, 4, 5, 8, 7, 6, 2}
    sorted_list = merge_sort(list)
    ? sorted_list
end procedure

print_sorted_list()      -- this command starts the program

~~~~~
```

The above example contains 4 separate commands that are processed in order. The first declares two variables: list and sorted\_list to be **sequences** (flexible arrays). The second defines a **function** merge\_sort(). The third defines a **procedure** print\_sorted\_list(). The final command calls procedure print\_sorted\_list().

The output from the program will be:

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.

**merge\_sort()** will just as easily sort {1.5, -9, 1e6, 100} or {"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 "ex allsorts" to see timings on several different sorting algorithms for increasing numbers of objects. For a quick tutorial example of Euphoria programming see [euphoria\demo\bench\filesort.ex](#).

---

## 1.2 Installation

To install Euphoria on your machine, first read the file [install.doc](#). Installation simply involves copying the **euphoria** files to your hard disk under a directory named "euphoria", and then modifying your **autoexec.bat** file so that **euphoria\bin** is on your search path, and the environment variable **EUDIR** is set to the euphoria directory.

When installed, the **euphoria** directory will look something like this:

### **\euphoria**

readme.doc  
readme.htm  
License.txt

### **\bin**

Interpreters ex.exe and exw.exe. Translators ec.exe and ecw.exe. Or on Linux/FreeBSD, Interpreter exu and Translator ecu. There are also utility programs such as ed.bat, guru.bat etc.

### **\include**

standard include files, e.g. graphics.e

### **\source**

the complete source code (interpreter, translator, binder)

### **\doc**

refman.doc, library.doc, and several other plain-text documentation files

### **\html**

HTML files corresponding to each of the .doc files in the doc directory

### **\tutorial**

small tutorial programs to help you learn Euphoria

### **\demo**

generic demo programs that run on all platforms

### **\dos32**

DOS32-specific demo programs (optional)

### **\win32**

WIN32-specific demo programs (optional)

### **\linux**

Linux/FreeBSD-specific demo programs (optional)

### **\langwar**

language war game (pixel-graphics version for DOS, or text version for Linux/FreeBSD)

### **\bench**

benchmark programs

The Linux subdirectory is not included in the DOS/Windows distribution, and the dos32 and win32 subdirectories are not included in the Linux/FreeBSD distribution. In this manual, directory names are shown using backslash (\). Linux/FreeBSD users should substitute forward slash (/).

---

## 1.3 Running a Program

Euphoria programs are executed by typing **ex**, **exw** or **exu** followed by the name of the main Euphoria file. You can type additional words (known as **arguments**) on this line, known as the **command-line**. Your program can call the built-in function [command\\_line\(\)](#) to read the command-line. The DOS32 version of the Euphoria interpreter is called **ex.exe**. The WIN32 version is called **exw.exe**. The Linux/FreeBSD version is called **exu**. By convention, main Euphoria files have an extension of **.ex**, **.exw** or **.exu**. Other Euphoria files, that are meant to be included in a larger program, end in **.e** or sometimes **.ew** or **.eu**. To save typing, you can leave off the ".ex", and the **ex** command will supply it for you automatically. **exw.exe** will supply ".exw", and **exu** will supply ".exu". If the file can't be found in the current directory, your PATH will be searched. You can redirect standard input and standard output when you run a Euphoria program, for example:

```
ex filesort.ex < raw.txt > sorted.txt
```

or simply,

```
ex filesort < raw.txt > sorted.txt
```

Unlike many other compilers and interpreters, there are no special command-line options for **ex**, **exw** or **exu**. Only the name of your Euphoria file is expected, and if you don't supply it, you will be prompted for it.

For frequently-used programs under DOS/Windows you might want to make a small **.bat** (batch) file, perhaps called **myprog.bat**, containing two statements like:

```
@echo off
ex myprog.ex %1 %2 %3
```

The first statement turns off echoing of commands to the screen. The second runs **ex myprog.ex** with up to 3 command-line arguments. See [command\\_line\(\)](#) for an example of how to read these arguments. If your program takes more arguments, you should add %4 %5 etc. Having a .bat file will save you the minor inconvenience of typing **ex** (or **exw**) all the time, i.e. you can just type:

```
myprog
```

instead of:

```
ex myprog
```

Unfortunately DOS will not allow redirection of standard input and output when you use a **.bat** file

Under Linux/FreeBSD, you can type the path to the Euphoria interpreter on the first line of your main file, e.g. if your program is called foo.exu:

```
#!/home/rob/euphoria/bin/exu

procedure foo()
    ? 2+2
end procedure

foo()
```

Then if you make your file executable:

```
chmod +x foo.exu
```

You can just type:

foo.exu

to run your program. You could even shorten the name to simply "foo". Euphoria ignores the first line when it starts with **#!**. Be careful though that your first line ends with the Linux/FreeBSD-style `\n`, and not the DOS/Windows-style `\r\n`, or the Linux/FreeBSD shell might get confused. If your file is shrouded, you must give the path to backendu, not exu.

You can also run **bind.bat** (DOS32), or **bindw.bat** (WIN32) or **bindu** (Linux/FreeBSD) to combine your Euphoria program with **ex.exe**, **exw.exe** or **exu**, to make a stand-alone executable file (**.exe** file on DOS/Windows). With a stand-alone **.exe** file you *can* redirect standard input and output. Binding is discussed further in [1.5 Distributing a Program](#).

Using the [Euphoria to C Translator](#), you can also make a stand-alone .exe file, and it will normally run much faster than a bound program.

**exu** or **ex.exe** and **exw.exe** will be in the **euphoria\bin** directory which must be on your search path. The environment variable EUDIR should be set to the main Euphoria directory, e.g. **c:\euphoria**.

### 1.3.1 Running under Windows

You can run Euphoria programs directly from the Windows environment, or from a DOS shell that you have opened from Windows. By "associating" **.ex** files with **ex.exe**, and **.exw** files with **exw.exe** you can simply double-click on a **.ex** or **.exw** file to run it. Under Windows you would define a new file type for **.ex**, by clicking on My Computer / view / options / file types. It is possible to have several Euphoria programs active in different windows. If you turn your program into a **.exe** file, you can simply double-click on it to run it.

### 1.3.2 Use of a Swap File

If you run a Euphoria program under Linux/FreeBSD or Windows (or in a DOS shell under Windows), and the program runs out of physical memory, it will start using "virtual memory". The operating system provides this virtual memory automatically by swapping out the least-recently-used code and data to a system swap file. To change the size of the Windows swap file, click on Control Panel / 386 Enhanced / "virtual memory...". Under OS/2 you can adjust the "DPML\_MEMORY\_LIMIT" by clicking the Virtual DOS machine icon / "DOS Settings" to allocate more extended memory for your program.

Under pure DOS, outside of Windows, there is no system swap file so the DOS-extender built in to **ex.exe** (DOS32) will create one for possible use by your program. See [platform.doc](#).

---



## 1.4 Editing a Program

You can use any text editor to edit a Euphoria program. However, Euphoria comes with its own special editor that is written entirely in Euphoria. Type: **ed** followed by the complete name of the file you wish to edit (the .ex/.exw/.exu extension is not assumed). You can use this editor to edit any kind of text file. When you edit a Euphoria file, some extra features such as color syntax highlighting and auto-completion of certain statements, are available to make your job easier.

Whenever you run a Euphoria program and get an error message, during compilation or execution, you can simply type **ed** with no file name and you will be automatically positioned in the file containing the error, at the correct line and column, and with the error message displayed at the top of the screen.

Under Windows you can associate **ed.bat** with various kinds of text files that you want to edit. Color syntax highlighting is provided for **.ex**, **.exw**, **.exu**, **.e**, **.ew**, **.eu**, and **.pro** ([profile](#)) files.

Most keys that you type are inserted into the file at the cursor position. Hit the **Esc** key once to get a menu bar of special commands. The arrow keys, and the Insert/Delete/Home/End/PageUp/PageDown keys are also active. Under Linux/FreeBSD some keys may not be available, and alternate keys are provided. See the file [euphoria\doc\ed.doc](#) ([euphoria\html\ed.htm](#)) for a complete description of the editing commands. **Esc h** (help) will let you view **ed.doc** from your editing session.

If you need to understand or modify any detail of the editor's operation, you can edit the file **ed.ex** in [euphoria\bin](#) (be sure to make a backup copy so you don't lose your ability to edit). If the name **ed** conflicts with some other command on your system, simply rename the file [euphoria\bin\ed.bat](#) to something else. Because this editor is written in Euphoria, it is remarkably concise and easy to understand. The same functionality implemented in a language like C, would take far more lines of code.

**ed** is a simple text-mode editor that runs on DOS, Linux, FreeBSD and the Windows console. See also **David Cuny's** excellent **ee.ex** editor for DOS and Linux/FreeBSD. You can download **ee.ex** from the [Euphoria Web site](#). There are also some Windows editors oriented to Euphoria. These are also on [the Web site](#).

---

## 1.5 Distributing a Program

Euphoria provides you with 4 distinct ways of distributing a program.

In the first method you simply ship your users **ex.exe** or **exw.exe** or **exu** file, along with your main Euphoria .ex, .exw, or .exu file and any .e include files that are needed (including any of the standard ones from [euphoria\include](#)). If the Euphoria source files are placed together in one directory and ex.exe, exw.exe or exu is placed in the same directory or somewhere on the search path, then your user can run your program by typing **ex** (**exw**) or (**exu**) followed by the path of your main .ex, .exw, or .exu file. You might also provide a small **.bat** file so people won't actually have to type **ex** (**exw**). This method assumes that you are willing to share your Euphoria source code with your users.

The Binder gives you two more methods of distribution. You can **shroud** your program, or you can **bind** your program. **Shrouding** combines all of the .e files that your program needs, along with your main file to create a single .il file. **Binding** combines your shrouded program with backend.exe, backendw.exe or backendu to create a **single, stand-alone executable (.exe)** file. For example, if your program is called "myprog.ex" you can create "myprog.exe" which will run identically. For more information about shrouding and binding, see [bind.doc](#).

Finally, with the [Euphoria To C Translator](#), you can **translate** your Euphoria program into C and then compile it with a C compiler to get an executable (.exe) file.

### 1.5.1 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 **ex.exe**, **exw.exe** and **exu** files so anyone can run your program. You can also distribute the interpreter backend: **backend.exe**, **backendw.exe** and **backendu**. You can **shroud** or **bind** 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 euphoria\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.RapidEuphoria.com> of our Web page, but we do not require any such acknowledgment.

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 [Euphoria To C Translator](#).

... continue [2. Language Definition](#)

## 2. Language Definition

### 2.1 Objects

#### 2.1.1 Atoms and Sequences

All data **objects** in Euphoria are either **atoms** or **sequences**. An **atom** is a single numeric value. A **sequence** is a collection of numeric values.

The **objects** contained in a sequence can be an arbitrary mix of atoms or sequences. A sequence is represented by a list of objects in brace brackets, separated by commas. 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

-- 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
```

Numbers can also be entered in hexadecimal. For example:

```
#FE          -- 254
#A000        -- 40960
#FFFF000008  -- 68718428168
-#10         -- -16
```

Only the capital letters A, B, C, D, E, F are allowed in hex numbers. Hex numbers are always positive, unless you add a minus sign in front of the # character. So for instance #FFFFFFFF is a huge positive number (4294967295), \*not\* -1, as some machine-language programmers might expect.

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)}
```

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. Of course in the world of physics, atoms were split into smaller parts many years ago, but in Euphoria you can't split them. They 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).

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

#### Performance Note:

Does this mean that all atoms are stored in memory as 8-byte floating-point numbers? No. The Euphoria interpreter usually stores integer-valued atoms as machine integers (4 bytes) to save space and improve execution speed. When fractional results occur or numbers get too big, conversion to floating-point happens automatically.

### 2.1.2 Character Strings and Individual Characters

A **character string** is just a **sequence** of characters. It may be entered using quotes e.g.

```
"ABCDEFGG"
```

Character strings may be manipulated and operated upon just like any other sequences. For example the above string 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 "ABCDEFGG" 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 length-0, also known as the **empty sequence**. As a matter of programming style, it is natural to use "" to suggest a length-0 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 aren't really any "characters" in Euphoria, just numbers (atoms).

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.

Special characters may be entered using a back-slash:

```
\n      newline
\r      carriage return
\t      tab
\\      backslash
\"      double quote
\'      single quote
```

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

### 2.1.3 Comments

Comments are started by two dashes and extend to the end of the current line. e.g.

```
-- this is a comment
```

Comments are ignored by the compiler and have no effect on execution speed. The editor displays comments in red.

On the first line (only) of your program, you can use a special comment beginning with #!, e.g.

```
#!/home/rob/euphoria/bin/exu
```

This informs the Linux shell that your file should be executed by the Euphoria interpreter, and gives the full path to the interpreter. If you make your file executable, you can run it, just by typing its name, and without the need to type "exu". On DOS and Windows this line is just treated as a comment (though Apache Web server on Windows does recognize it.). If your file is a shrouded .il file, use backendu instead of exu.

---

## 2.2 Expressions

Like other programming languages, Euphoria lets you calculate results by forming expressions. However, in Euphoria you can perform calculations on entire sequences of data with one expression, where 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.

### 2.2.1 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.

### 2.2.2 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**.

### 2.2.3 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

```

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.

Division by zero, as well as bad arguments to math library routines, e.g. square root of a negative number, log of a non-positive number etc. cause an immediate error message and your program is aborted.

The only reason that you might use unary plus is to emphasize to the reader of your program that a number is positive. The interpreter does not actually calculate anything for this.

### 2.2.4 Operations on Sequences

All of the relational, logical and arithmetic operators described above, as well as the math routines described in [Part II - Library Routines](#), 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}
```

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 1's and 0's. 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)
```

## 2.2.5 Subscripting of Sequences

A single element of a sequence may be selected by giving the element number in square brackets. Element



numbers start at 1. Non-integer subscripts are rounded down to an integer.

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 there is no simple expression 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 a set of constants that name the various fields. This will make your program easier to read. For the example above you might have:

```
constant NAME = 1
constant FIRST_NAME = 1, LAST_NAME = 2

constant SALARY = 2
constant AGE = 3
constant WEIGHT = 4
```

You could then access the person's name with `employee[NAME]`, or if you wanted the last name you could say `employee[NAME][LAST_NAME]`.

Array of structures:

```
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 the length of 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 `$` symbol equals the length of the sequence. `$` may only appear between square braces. 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 other 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. 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.

Not only can a Euphoria program easily represent all conventional data structures but you can create very useful, flexible structures that would be extremely hard to declare in a conventional language. See [2.3 Euphoria versus Conventional 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.

## 2.2.6 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 length-0 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
```

## 2.2.7 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 (where atoms are considered here to have length 1). 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)]`.

## 2.2.8 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. e.g.

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

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

## 2.2.9 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 **ex.exe/exw.exe/exu**, so they'll always be there, and so they'll be fast. They are described in detail in [Part II - Library Routines](#), 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(s)

**length()** tells you 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. You'll get an error if you ask for the length of an atom. e.g.

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

### repeat(item, count)

**repeat()** makes a sequence that consists of an item repeated count times. e.g.

```
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(s, item) / prepend(s, item)

**append()** creates a new sequence by adding an item to the end of a sequence s. **prepend()** creates 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 1 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}

```

### 2.2.10 Precedence Chart

The precedence of operators in expressions is as follows:

<b>highest precedence:</b>	function/type calls
	unary- unary+ not
	* /
	+ -
	&
	< > <= >= = !=
	and or xor
<b>lowest precedence:</b>	{ , , , }

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.

The equals symbol '=' used in an [assignment statement](#) is not an operator, it's just part of the syntax of the language.

## 2.3 Euphoria versus Conventional Languages

By basing Euphoria on this one, simple, general, recursive data structure, a tremendous amount of the complexity normally found in programming languages has been avoided. The arrays, structures, unions, arrays of records, multidimensional arrays, etc. of other languages can all be easily represented in Euphoria with sequences. So can higher-level structures such as lists, stacks, queues, trees etc.

Furthermore, in Euphoria you can have sequences of mixed type; you can assign any object to an element of a sequence; and sequences easily grow or shrink in length without your having to worry about storage allocation issues. The exact layout of a data structure does not have to be declared in advance, and can change dynamically as required. It is easy to write generic code, where, for instance, you push or pop a mix of various kinds of data objects using a single stack. Making a flexible list that contains a variety of different kinds of data objects is trivial in Euphoria, but requires dozens of lines of ugly code in other languages.

Data structure manipulations are very efficient since the Euphoria interpreter will point to large data objects rather than copy them.

Programming in Euphoria is based entirely on creating and manipulating flexible, dynamic sequences of data. Sequences are *it* - there are no other data structures to learn. You operate in a simple, safe, elastic world of *values*, that is far removed from the rigid, tedious, dangerous world of bits, bytes, pointers and machine crashes.

Unlike other languages such as LISP and Smalltalk, Euphoria's "garbage collection" of unused storage is a continuous process that never causes random delays in execution of a program, and does not pre-allocate huge regions of memory.

The language definitions of conventional languages such as C, C++, Ada, etc. are very complex. Most programmers become fluent in only a subset of the language. The ANSI standards for these languages read like complex legal documents.

You are forced to write different code for different data types simply to copy the data, ask for its current length, concatenate it, compare it etc. The manuals for those languages are packed with routines such as "strcpy", "strncpy", "memcpy", "strcat", "strlen", "strcmp", "memcmp", etc. that each only work on one of the many types of data.

Much of the complexity surrounds issues of data type. How do you define new types? Which types of data can be mixed? How do you convert one type into another in a way that will keep the compiler happy? When you need to do something requiring flexibility at run-time, you frequently find yourself trying to fake out the compiler.

In these languages the numeric value 4 (for example) can have a different meaning depending on whether it is an int, a char, a short, a double, an int \* etc. In Euphoria, 4 is the atom 4, period. Euphoria has something called types as we shall see later, but it is a much simpler concept.

Issues of dynamic storage allocation and deallocation consume a great deal of programmer coding time and debugging time in these other languages, and make the resulting programs much harder to understand. Programs that must run continuously often exhibit storage "leaks", since it takes a great deal of discipline to safely and properly free all blocks of storage once they are no longer needed.

Pointer variables are extensively used. The pointer has been called the "go to" of data structures. It forces programmers to think of data as being bound to a fixed memory location where it can be manipulated in all sorts of low-level, non-portable, tricky ways. A picture of the actual hardware that your program will run on is never far from your mind. Euphoria does not have pointers and does not need them.

---

## 2.4 Declarations

### 2.4.1 Identifiers

**Identifiers**, which consist of variable names and other user-defined symbols, may be of any length. Upper and lower case are distinct. Identifiers must start with a letter and then be followed by letters, digits or underscores. The following **reserved words** have special meaning in Euphoria and may not be used as identifiers:

and	end	include	to
by	exit	not	type
constant	for	or	while

<code>do</code>	<code>function</code>	<code>procedure</code>	<code>with</code>
<code>else</code>	<code>global</code>	<code>return</code>	<code>without</code>
<code>elsif</code>	<code>if</code>	<code>then</code>	<code>xor</code>

The Euphoria editor displays these words in blue.

Identifiers can be used in naming the following:

- procedures
- functions
- types
- variables
- constants

## procedures

These perform some computation and may have 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.

### 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.

## 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

```

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}

```

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

## 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 [2.4.3 Specifying the Type of a 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 specify the values that may legally be assigned to the variable during execution of your program.

## 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.



## 2.4.2 Scope

A symbol's *scope* is the portion of the program where that symbol's declaration is in effect, i.e. where that symbol is *visible*.

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

Every user-defined symbol must be declared before it is used. You can read a Euphoria program from beginning to end without encountering any user-defined variables or routines that haven't been defined yet. It is possible to call a routine that comes later in the source, but you must use the special functions,

**routine\_id()**, and either **call\_func()** or **call\_proc()** to do it. See [Part II - Library Routines - Dynamic Calls](#).

Procedures, functions and types can call themselves *recursively*. Mutual recursion, where routine A calls routine B which directly or indirectly calls routine A, requires the **routine\_id()** mechanism.

A symbol is defined from the point where it is declared to the end of its **scope**. The scope of a variable declared inside a procedure or function (a **private** variable) ends at the end of the procedure or function. The scope of all other variables, constants, procedures, functions and types ends at the end of the source file in which they are declared and they are referred to as **local**, unless the keyword **global** precedes their declaration, in which case their scope extends indefinitely.

When you **include** a Euphoria file in a main file (see [2.6 Special Top-Level Statements](#)), only the variables and routines declared using the **global** keyword are accessible or even visible to the main file. The other, non-global, declarations in the included file are forgotten at the end of the included file, and you will get an error message, "not declared", if you try to use them in the main file.

Symbols marked as **global** can be used externally. All other symbols can only be used internally 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 global symbol as one of the other authors. If you have the source, you can simply edit one of the include files to correct the problem, but 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.

Euphoria encourages you to restrict the scope of symbols. If all symbols 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 isn't possible, make them **local** to a file, rather than **global** to the whole program.

When Euphoria looks up the declaration of a symbol, it first checks the current routine, then the current file, then globals in other files. Symbols that are more local will *override* symbols that are more global. At the end of the scope of the local symbol, the more global symbol will be visible again.

**Constant** declarations must be outside of any subroutine. Constants can be global or local, but not **private**.

Variable declarations inside a subroutine must all appear at the beginning, before the executable statements of the subroutine.

Declarations at the top level, outside of any subroutine, must not be nested inside a loop or [if-statement](#).

The controlling variable used in a [for-loop](#) is special. It is automatically declared at the beginning of the loop, and its scope ends at the end of the for-loop. If the loop is inside a function or procedure, the loop variable is a **private** variable and may not have the same name as any other **private** variable. When the loop is at the top level, outside of any function or procedure, the loop variable is a **local** variable and may not have the same name as any other **local** variable in that file. You can use the same name in many different for-loops as long as the loops aren't nested. You do not declare loop variables as you would other variables. The range of values specified in the for statement defines the legal values of the loop variable - specifying a type would be redundant and is not allowed.

### 2.4.3 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. Those 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 aren't representable as integers. If your machine doesn't have floating-point hardware, Euphoria will call software floating-point arithmetic routines contained in **ex.exe** (or in Windows). You can force ex.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.

To augment the [predefined types](#), you can create **user-defined types**. All you have to do is define a single-parameter function, but declare it with **type ... end type** instead of **function ... end function**. For example,

```
type hour(integer x)
    return x >= 0 and x <= 23
end type

hour h1, h2

h1 = 10      -- ok
h2 = 25      -- error! program aborts with a message
```

Variables h1 and h2 can only be assigned integer values in the range 0 to 23 inclusive. After each assignment to h1 or h2 the interpreter will call hour(), passing the new value. The value will first be checked to see if it is an integer (because of "integer x"). If it is, the return statement will be executed to test the value of x (i.e. the new value of h1 or h2). If hour() returns true, execution continues normally. If hour() returns false then the program is aborted with a suitable diagnostic message.

"hour" can be used to declare subroutine parameters as well:

```
procedure set_time(hour h)
```

set\_time() can only be called with a reasonable value for parameter h, otherwise the program will abort with a message.

A variable's type will be checked after each assignment to the variable (except where the compiler can predetermine that a check will not be necessary), and the program will terminate immediately if the type function returns false. Subroutine parameter types are checked each time that the subroutine is called. This checking guarantees that a variable can never have a value that does not belong to the type of that variable.

Unlike other languages, the type of a variable does not affect any calculations on the variable. Only the value of the variable matters in an expression. The type just serves as an error check to prevent any "corruption" of the variable.

User-defined types can catch unexpected logical errors in your program. They are not designed to catch or correct user input errors.

Type checking can be turned off or on between subroutines using the **with type\_check** or **without type\_check** [special statements](#). It is initially on by default.

### Note to Benchmarkers:

When comparing the speed of Euphoria programs against programs written in other languages, you should specify **without type\_check** at the top of the file. This gives Euphoria permission to skip run-time type checks, thereby saving some execution time. All other checks are still performed, e.g. subscript checking, uninitialized variable checking etc. Even when you turn off type checking, Euphoria reserves the right to make checks at strategic places, since this can actually allow it to run your program *faster* in many cases. So you may still get a type check failure even when you have turned off type checking. Whether type checking is on or off, you will never get a *machine-level* exception. **You will always get a meaningful message from Euphoria when something goes wrong.** (*This might not be the case when you poke directly into memory, or call routines written in C or machine code.*)

Euphoria's method of defining types is simpler than what you will find in other languages, yet Euphoria provides the programmer with *greater* flexibility in defining the legal values for a type of data. Any algorithm can be used to include or exclude values. You can even declare a variable to be of type **object** which will allow it to take on *any* value. Routines can be written to work with very specific types, or very general types.

For many programs, there is little advantage in defining new types, and you may wish to stick with the four predefined types. Unlike other languages, Euphoria's type mechanism is optional. You don't need it to create a program.

However, for larger programs, strict type definitions can aid the process of debugging. Logic errors are caught close to their source and are not allowed to propagate in subtle ways through the rest of the program. Furthermore, it is easier to reason about the misbehavior of a section of code when you are guaranteed that the variables involved always had a legal value, if not the desired value.

Types also provide meaningful, machine-checkable documentation about your program, making it easier for you or others to understand your code at a later date. Combined with the subscript checking, uninitialized variable checking, and other checking that is always present, strict run-time type checking makes debugging much easier in Euphoria than in most other languages. It also increases the reliability of the final program since many latent bugs that would have survived the testing phase in other languages will have been caught by Euphoria.

#### **Anecdote 1:**

In porting a large C program to Euphoria, a number of latent bugs were discovered. Although this C program was believed to be totally "correct", we found: a situation where an uninitialized variable was being read; a place where element number "-1" of an array was routinely written and read; and a situation where something was written just off the screen. These problems resulted in errors that weren't easily visible to a casual observer, so they had survived testing of the C code.

#### **Anecdote 2:**

The Quick Sort algorithm presented on page 117 of *Writing Efficient Programs* by Jon Bentley has a subscript error! The algorithm will sometimes read the element just *before* the beginning of the array to be sorted, and will sometimes read the element just *after* the end of the array. Whatever garbage is read, the algorithm will still work - this is probably why the bug was never caught. But what if there isn't any (virtual) memory just before or just after the array? Bentley later modifies the algorithm such that this bug goes away -- but he presented this version as being correct. *Even the experts need subscript checking!*

#### **Performance Note:**

When typical user-defined types are used extensively, type checking adds only 20 to 40 percent to

execution time. Leave it on unless you really need the extra speed. You might also consider turning it off for just a few heavily-executed routines. [Profiling](#) can help with this decision.

---

## 2.5 Statements

The following kinds of executable statements are available:

- [assignment statement](#)
- [procedure call](#)
- [if statement](#)
- [while statement](#)
- [for statement](#)
- [return statement](#)
- [exit statement](#)

Semicolons are not used in Euphoria, but you are free to put as many statements as you like on one line, or to split a single statement across many lines. You may not split a statement in the middle of an identifier, string, number or keyword.

### 2.5.1 assignment statement

An **assignment statement** assigns the value of an expression to a simple variable, or to a subscript or slice of a variable. e.g.

```
x = a + b
```

```
y[i] = y[i] + 1
```

```
y[i..j] = {1, 2, 3}
```

The previous value of the variable, or element(s) of the subscripted or sliced variable are discarded. For example, suppose x was a 1000-element sequence that we had initialized with:

```
object x
```

```
x = repeat(0, 1000)  -- a sequence of 1000 zeros
```

and then later we assigned an atom to x with:

```
x = 7
```

This is perfectly legal since x is declared as an **object**. The previous value of x, namely the 1000-element sequence, would simply disappear. Actually, the space consumed by the 1000-element sequence will be automatically recycled due to Euphoria's dynamic storage allocation.

Note that the equals symbol '=' is used for both assignment and for [equality testing](#). There is never any confusion because an assignment in Euphoria is a statement only, it can't be used as an expression (as in C).

## Assignment with Operator

Euphoria also provides some additional forms of the assignment statement.

To save typing, and to make your code a bit neater, you can combine assignment with one of the operators:

**+ - / \* &**

For example, instead of saying:

```
mylongvarname = mylongvarname + 1
```

You can say:

```
mylongvarname += 1
```

Instead of saying:

```
galaxy[q_row][q_col][q_size] = galaxy[q_row][q_col][q_size] * 10
```

You can say:

```
galaxy[q_row][q_col][q_size] *= 10
```

and instead of saying:

```
accounts[start..finish] = accounts[start..finish] / 10
```

You can say:

```
accounts[start..finish] /= 10
```

In general, whenever you have an assignment of the form:

***left-hand-side = left-hand-side op expression***

You can say:

***left-hand-side op= expression***

where ***op*** is one of: **+ - \* / &**

When the left-hand-side contains multiple subscripts/slices, the ***op=*** form will usually execute faster than the longer form. When you get used to it, you may find the ***op=*** form to be slightly more readable than the long form, since you don't have to visually compare the left-hand-side against the copy of itself on the right side.

### 2.5.2 procedure call

A **procedure call** starts execution of a procedure, passing it an optional list of argument values. e.g.

```
plot(x, 23)
```

### 2.5.3 if statement

An **if statement** tests a condition to see if it is 0 (false) or non-zero (true) and then executes the appropriate series of statements. There may be optional **elsif** and **else** clauses. e.g.

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

```

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

The **if** and **elseif** conditions are tested using [short-circuit evaluation](#).

## 2.5.4 while statement

A **while statement** tests a condition to see if it is non-zero (true), and while it is true a loop is executed. e.g.

```

while x > 0 do
    a = a * 2
    x = x - 1
end while

```

## Short-Circuit Evaluation

When the condition tested by **if**, **elseif**, or **while** contains **and** or **or** operators, *short-circuit* evaluation will be used. For example,

```
if a < 0 and b > 0 then ...
```

If  $a < 0$  is false, then Euphoria will not bother to test if  $b$  is greater than 0. It will assume that the overall result is false. Similarly,

```
if a < 0 or b > 0 then ...
```

if  $a < 0$  is true, then Euphoria will immediately decide that the result true, without testing the value of  $b$ .

In general, whenever we have a condition of the form:

```
A and B
```

where  $A$  and  $B$  can be any two expressions, Euphoria will take a short-cut when  $A$  is false and immediately make the overall result false, without even looking at expression  $B$ .

Similarly, with:

```
A or B
```

when  $A$  is true, Euphoria will skip the evaluation of expression  $B$ , and declare the result to be true.

If the expression  $B$  contains a call to a function, and that function has possible **side-effects**, i.e. it might do more than just return a value, you will get a compile-time warning. Older versions (pre-2.1) of Euphoria did not use **short-circuit** evaluation, and it's possible that some old code will no longer work correctly, although a search of the Euphoria archives did not turn up any programs that depend on side-effects in this way.

The expression, B, could contain something that would normally cause a run-time error. If Euphoria skips the evaluation of B, the error will not be discovered. For instance:

```
if x != 0 and 1/x > 10 then -- divide by zero error avoided

while 1 or {1,2,3,4,5} do -- illegal sequence result avoided
```

B could even contain uninitialized variables, out-of-bounds subscripts etc.

This may look like sloppy coding, but in fact it often allows you to write something in a simpler and more readable way. For instance:

```
if atom(x) or length(x)=1 then
```

Without short-circuiting, you would have a problem when x was an atom, since length is not defined for atoms. With short-circuiting, length(x) will only be checked when x is a sequence. Similarly:

```
-- find 'a' or 'A' in s
i = 1
while i <= length(s) and s[i] != 'a' and s[i] != 'A' do
    i += 1
end while
```

In this loop the variable i might eventually become greater than length(s). Without short-circuit evaluation, a subscript out-of-bounds error will occur when s[i] is evaluated on the final iteration. With short-circuiting, the loop will terminate immediately when i <= length(s) becomes false. Euphoria will not evaluate s[i] != 'a' and will not evaluate s[i] != 'A'. No subscript error will occur.

**Short-circuit** evaluation of **and** and **or** takes place for **if**, **elsif** and **while** conditions only. It is not used in other contexts. For example, the assignment statement:

```
x = 1 or {1,2,3,4,5} -- x should be set to {1,1,1,1,1}
```

If short-circuiting were used here, we would set x to 1, and not even look at {1,2,3,4,5}. This would be wrong. Short-circuiting can be used in if/elsif/while conditions because we only care if the result is true or false, and conditions are required to produce an atom as a result.

## 2.5.5 for statement

A **for statement** sets up a special loop with a controlling **loop variable** that runs from an initial value up or down to some final value. e.g.

```
for i = 1 to 10 do
    ? i -- ? is a short form for print()
end for

-- fractional numbers allowed too
for i = 10.0 to 20.5 by 0.3 do
    for j = 20 to 10 by -2 do -- counting down
        ? {i, j}
    end for
end for
```

The **loop variable** is declared automatically and exists until the end of the loop. Outside of the loop the variable has no value and is not even declared. If you need its final value, copy it into another variable before leaving the loop. The compiler will not allow any assignments to a loop variable. The initial value, loop limit and increment must all be atoms. If no increment is specified then +1 is assumed. The limit and increment values are established when the loop is entered, and are not affected by anything that happens during the execution of the loop. See also the [scope of the loop variable in 2.4.2 Scope](#).



## 2.5.6 return statement

A **return statement** returns immediately from a subroutine. If the subroutine is a function or type then a value must also be returned. e.g.

```
return

return {50, "FRED", {}}
```

## 2.5.7 exit statement

An **exit statement** may appear inside a [while-loop](#) or a [for-loop](#). It causes immediate termination of the loop, with control passing to the first statement after the loop. e.g.

```
for i = 1 to 100 do
  if a[i] = x then
    location = i
    exit
  end if
end for
```

It is also quite common to see something like this:

```
constant TRUE = 1

while TRUE do
  ...
  if some_condition then
    exit
  end if
  ...
end while
```

i.e. an "infinite" while-loop that actually terminates via an **exit statement** at some arbitrary point in the body of the loop.

### Performance Note:

Euphoria optimizes this type of loop. At run-time, no test is performed at the top of the loop. There's just a simple unconditional jump from **end while** back to the first statement inside the loop.

With **ex.exe**, if you happen to create a real infinite loop, with no input/output taking place, there is no easy way to stop it. You will have to type Control-Alt-Delete to either reboot, or (under Windows) terminate your DOS prompt session. If the program had files open for writing, it would be advisable to run **scandisk** to check your file system integrity. Only when your program is waiting for keyboard input, will control-c abort the program (unless [allow\\_break\(0\)](#) was used).

With **exw.exe** or **exu**, control-c will always stop your program immediately.

---

## 2.6 Special Top-Level Statements

Before any of your statements are executed, the Euphoria front-end quickly reads your entire program. All statements are syntax checked and converted to a low-level intermediate language (IL). The interpreter immediately executes the IL. The translator converts the IL to C. The binder/shrouder saves the IL on disk for later execution. These three tools all share the same front-end (written in Euphoria).

If your program contains only routine and variable declarations, but no top-level executable statements, then nothing will happen when you run it (other than syntax checking). You need a top-level statement to call your main routine (see [1.1 Example Program](#)). It's quite possible to have a program with nothing but top-level executable statements and no routines. For example you might want to use Euphoria as a simple calculator, typing just a few [print](#) (or [?](#)) statements into a file, and then executing it.

As we have seen, you can use any Euphoria [statement](#), including [for-loops](#), [while-loops](#), [if](#) statements etc. (but not [return](#)), at the top level i.e. *outside* of any [function](#) or [procedure](#). In addition, the following special statements may *only* appear at the top level:

- include
- with / without

### 2.6.1 include

When you write a large program it is often helpful to break it up into logically separate files, by using **include statements**. Sometimes you will want to reuse some code that you have previously written, or that someone else has written. Rather than copy this code into your main program, you can use an **include statement** to refer to the file containing the code. The first form of the include statement is:

#### **include filename**

This reads in (compiles) a Euphoria source file.

Some Examples:

```
include graphics.e
include \mylib\myroutines.e
```

Any top-level code in the included file will be executed.

Any [global symbols](#) that have already been defined in the main file will be visible in the included file.

**N.B.** Only those symbols defined as [global](#) in the included file will be visible (accessible) in the remainder of the program.

If an absolute *filename* is given, Euphoria will use it. When a relative *filename* is given, Euphoria will first look for it in the same directory as the main file given on the **ex** (or **exw** or **exu**) [command-line](#). If it's not there, and you've defined an environment variable, **EUINC**, it will search each directory listed in **EUINC** (from left to right). Finally, if it still hasn't found the file, it will search [euphoria\include](#). This directory contains the standard Euphoria include files. The environment variable **EUDIR** tells **ex.exe/exw.exe/exu** where to find your **euphoria** directory. **EUINC** should be a list of directories, separated by semicolons (colons on Linux), similar in form to your PATH variable. It can be added to your AUTOEXEC.BAT file, e.g.

```
SET EUINC=C:\EU\MYFILES;C:\EU\WIN32LIB
```

This lets you organize your include files according to application areas, and avoid adding numerous unrelated files to euphoria\include.

An included file can include other files. In fact, you can "nest" included files up to 30 levels deep.

Include file names typically end in **.e**, or sometimes **.ew** or **.eu** (when they are intended for use with

Windows or Linux). This is just a convention. It is not required.

If your filename (or path) contains blanks, you must enclose it in double-quotes, otherwise quotes are optional. Also, be sure to double-up your backslashes. For example:

```
include "c:\\program files\\myfile.e"
```

Other than possibly defining a new namespace identifier (see below), an include statement will be quietly ignored if the same file has already been included.

An include statement must be written on a line by itself. Only a comment can appear after it on the same line.

The second form of the include statement is:

**include** *filename* **as** *namespace\_identifier*

This is just like the simple include, but it also defines a *namespace identifier* that can be attached to global symbols in the included file that you want to refer to in the main file. This might be necessary to disambiguate references to those symbols, or you might feel that it makes your code more readable. See [Scope Rules](#) for more.

## 2.6.2 with / without

These special statements affect the way that Euphoria translates your program into internal form. They are not meant to change the logic of your program, but they may affect the diagnostic information that you get from running your program. See [3. Debugging and Profiling](#) for more information.

### with

This turns **on** one of the options: **profile**, **profile\_time**, **trace**, **warning** or **type\_check**. Options **warning** and **type\_check** are initially on, while **profile**, **profile\_time** and **trace** are initially off.

Any warnings that are issued will appear on your screen after your program has finished execution. Warnings indicate very minor problems. A warning will never stop your program from executing.

### without

This turns **off** one of the above options.

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.

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.

... continue [3. Debugging and Profiling](#)



## 2.1 Objects

### 2.1.1 Atoms and Sequences

All data **objects** in Euphoria are either **atoms** or **sequences**. An **atom** is a single numeric value. A **sequence** is a collection of numeric values.

The **objects** contained in a sequence can be an arbitrary mix of atoms or sequences. A sequence is represented by a list of objects in brace brackets, separated by commas. 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

-- 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
```

Numbers can also be entered in hexadecimal. For example:

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

Only the capital letters A, B, C, D, E, F are allowed in hex numbers. Hex numbers are always positive, unless you add a minus sign in front of the # character. So for instance #FFFFFFFF is a huge positive number (4294967295), \*not\* -1, as some machine-language programmers might expect.

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)}
```

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. Of course in the world of physics, atoms were split into smaller parts many years ago, but in Euphoria you can't split them. They 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).

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

**Performance Note:**

Does this mean that all atoms are stored in memory as 8-byte floating-point numbers? No. The Euphoria interpreter usually stores integer-valued atoms as machine integers (4 bytes) to save space and improve execution speed. When fractional results occur or numbers get too big, conversion to floating-point happens automatically.

## 2.1 Objects

### 2.1.1 Atoms and Sequences

All data **objects** in Euphoria are either **atoms** or **sequences**. An **atom** is a single numeric value. A **sequence** is a collection of numeric values.

The **objects** contained in a sequence can be an arbitrary mix of atoms or sequences. A sequence is represented by a list of objects in brace brackets, separated by commas. 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

-- 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
```

Numbers can also be entered in hexadecimal. For example:

```
#FE                -- 254
#A000              -- 40960
#FFFF000008        -- 68718428168
-#10                -- -16
```

Only the capital letters A, B, C, D, E, F are allowed in hex numbers. Hex numbers are always positive, unless you add a minus sign in front of the # character. So for instance #FFFFFFFF is a huge positive number (4294967295), \*not\* -1, as some machine-language programmers might expect.

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)}
```

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. Of course in the world of physics, atoms were split into smaller parts many years ago, but in Euphoria you can't split them. They 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).

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

**Performance Note:**

Does this mean that all atoms are stored in memory as 8-byte floating-point numbers? No. The Euphoria interpreter usually stores integer-valued atoms as machine integers (4 bytes) to save space and improve execution speed. When fractional results occur or numbers get too big, conversion to floating-point happens automatically.



## 2.1.2 Character Strings and Individual Characters

A **character string** is just a **sequence** of characters. It may be entered using quotes e.g.

`"ABCDEFGH"`

Character strings may be manipulated and operated upon just like any other sequences. For example the above string 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.

### 2.1.3 Comments

Comments are started by two dashes and extend to the end of the current line. e.g.

```
-- this is a comment
```

Comments are ignored by the compiler and have no effect on execution speed. The editor displays comments in red.

On the first line (only) of your program, you can use a special comment beginning with #!, e.g.

```
#!/home/rob/euphoria/bin/exu
```

This informs the Linux shell that your file should be executed by the Euphoria interpreter, and gives the full path to the interpreter. If you make your file executable, you can run it, just by typing its name, and without the need to type "exu". On DOS and Windows this line is just treated as a comment (though Apache Web server on Windows does recognize it.). If your file is a shrouded .il file, use backendu instead of exu.

---

## 2.2 Expressions

Like other programming languages, Euphoria lets you calculate results by forming expressions. However, in Euphoria you can perform calculations on entire sequences of data with one expression, where 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.

## 2.2.1 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.

## 2.2.2 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**.

### 2.2.3 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
```

## 2.2.4 Operations on Sequences

All of the relational, logical and arithmetic operators described above, as well as the math routines described in [Part II - Library Routines](#), 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}
```

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 1's and 0's. 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)
```



## 2.2.5 Subscripting of Sequences

A single element of a sequence may be selected by giving the element number in square brackets. Element numbers start at 1. Non-integer subscripts are rounded down to an integer.

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 there is no simple expression 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 a set of constants that name the various fields. This will make your program easier to read. For the example above you might have:

```
constant NAME = 1
constant FIRST_NAME = 1, LAST_NAME = 2
```

```
constant SALARY = 2
constant AGE = 3
constant WEIGHT = 4
```

You could then access the person's name with `employee[NAME]`, or if you wanted the last name you could say `employee[NAME][LAST_NAME]`.

Array of structures:

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

    -- .... etc.
}
```

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

## 2.2.6 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 length-0 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
```

## 2.2.7 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 (where atoms are considered here to have length 1). 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*)].

## 2.2.8 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. e.g.

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

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

### 2.2.9 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 **ex.exe/exw.exe/exu** , so they'll always be there, and so they'll be fast. They are described in detail in [Part II - Library Routines](#), 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(s)

**length()** tells you 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. You'll get an error if you ask for the length of an atom. e.g.

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

## repeat(item, count)

**repeat()** makes a sequence that consists of an item repeated count times. e.g.

```
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(s, item) / prepend(s, item)

**append()** creates a new sequence by adding an item to the end of a sequence s. **prepend()** creates 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 1 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}
```

## 2.2.10 Precedence Chart

The precedence of operators in expressions is as follows:

<b>highest precedence:</b>	function/type calls
	unary- unary+ not
	* /
	+ -
	&
	< > <= >= = !=
	and or xor
<b>lowest precedence:</b>	{ , , , }

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.

The equals symbol '=' used in an [assignment statement](#) is not an operator, it's just part of the syntax of the language.

---

## 2.3 Euphoria versus Conventional Languages

By basing Euphoria on this one, simple, general, recursive data structure, a tremendous amount of the complexity normally found in programming languages has been avoided. The arrays, structures, unions, arrays of records, multidimensional arrays, etc. of other languages can all be easily represented in Euphoria with sequences. So can higher-level structures such as lists, stacks, queues, trees etc.

Furthermore, in Euphoria you can have sequences of mixed type; you can assign any object to an element of a sequence; and sequences easily grow or shrink in length without your having to worry about storage allocation issues. The exact layout of a data structure does not have to be declared in advance, and can change dynamically as required. It is easy to write generic code, where, for instance, you push or pop a mix of various kinds of data objects using a single stack. Making a flexible list that contains a variety of different kinds of data objects is trivial in Euphoria, but requires dozens of lines of ugly code in other languages.

Data structure manipulations are very efficient since the Euphoria interpreter will point to large data objects rather than copy them.

Programming in Euphoria is based entirely on creating and manipulating flexible, dynamic sequences of data. Sequences are *it* - there are no other data structures to learn. You operate in a simple, safe, elastic world of *values*, that is far removed from the rigid, tedious, dangerous world of bits, bytes, pointers and machine crashes.

Unlike other languages such as LISP and Smalltalk, Euphoria's "garbage collection" of unused storage is a continuous process that never causes random delays in execution of a program, and does not pre-allocate huge regions of memory.

The language definitions of conventional languages such as C, C++, Ada, etc. are very complex. Most programmers become fluent in only a subset of the language. The ANSI standards for these languages read like complex legal documents.

You are forced to write different code for different data types simply to copy the data, ask for its current length, concatenate it, compare it etc. The manuals for those languages are packed with routines such as "strcpy", "strncpy", "memcpy", "strcat", "strlen", "strcmp", "memcmp", etc. that each only work on one of the many types of data.

Much of the complexity surrounds issues of data type. How do you define new types? Which types of data can be mixed? How do you convert one type into another in a way that will keep the compiler happy? When you need to do something requiring flexibility at run-time, you frequently find yourself trying to fake out the compiler.

In these languages the numeric value 4 (for example) can have a different meaning depending on whether it is an int, a char, a short, a double, an int \* etc. In Euphoria, 4 is the atom 4, period. Euphoria has something called types as we shall see later, but it is a much simpler concept.

Issues of dynamic storage allocation and deallocation consume a great deal of programmer coding time and debugging time in these other languages, and make the resulting programs much harder to understand. Programs that must run continuously often exhibit storage "leaks", since it takes a great deal of discipline to safely and properly free all blocks of storage once they are no longer needed.

Pointer variables are extensively used. The pointer has been called the "go to" of data structures. It forces programmers to think of data as being bound to a fixed memory location where it can be manipulated in all sorts of low-level, non-portable, tricky ways. A picture of the actual hardware that your program will run on is never far from your mind. Euphoria does not have pointers and does not need them.

---

## 2.4 Declarations

### 2.4.1 Identifiers

**Identifiers**, which consist of variable names and other user-defined symbols, may be of any length. Upper and lower case are distinct. Identifiers must start with a letter and then be followed by letters, digits or underscores. The following **reserved words** have special meaning in Euphoria and may not be used as identifiers:

and	end	include	to
by	exit	not	type
constant	for	or	while
do	function	procedure	with
else	global	return	without
elsif	if	then	xor

The Euphoria editor displays these words in blue.

Identifiers can be used in naming the following:

- procedures
- functions
- types
- variables
- constants

## procedures

These perform some computation and may have 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.

### 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.

## 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
```

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}
```

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



## 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 [2.4.3 Specifying the Type of a 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 specify the values that may legally be assigned to the variable during execution of your program.

## 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.

### 2.4.2 Scope

A symbol's *scope* is the portion of the program where that symbol's declaration is in effect, i.e. where that symbol is *visible*.

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

Every user-defined symbol must be declared before it is used. You can read a Euphoria program from beginning to end without encountering any user-defined variables or routines that haven't been defined yet. It is possible to call a routine that comes later in the source, but you must use the special functions, **routine\_id()**, and either **call\_func()** or **call\_proc()** to do it. See [Part II - Library Routines - Dynamic Calls](#).

Procedures, functions and types can call themselves *recursively*. Mutual recursion, where routine A calls routine B which directly or indirectly calls routine A, requires the **routine\_id()** mechanism.

### 2.4.3 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.

## 2.5 Statements

The following kinds of executable statements are available:

- [assignment statement](#)
- [procedure call](#)
- [if statement](#)
- [while statement](#)
- [for statement](#)
- [return statement](#)
- [exit statement](#)

Semicolons are not used in Euphoria, but you are free to put as many statements as you like on one line, or to split a single statement across many lines. You may not split a statement in the middle of an identifier, string, number or keyword.

### 2.5.1 assignment statement

An **assignment statement** assigns the value of an expression to a simple variable, or to a subscript or slice of a variable. e.g.

```
x = a + b
```

```
y[i] = y[i] + 1
```

```
y[i..j] = {1, 2, 3}
```

The previous value of the variable, or element(s) of the subscripted or sliced variable are discarded. For example, suppose x was a 1000-element sequence that we had initialized with:

```
object x
```

```
x = repeat(0, 1000)  -- a sequence of 1000 zeros
```

and then later we assigned an atom to x with:

```
x = 7
```

This is perfectly legal since x is declared as an **object**. The previous value of x, namely the 1000-element sequence, would simply disappear. Actually, the space consumed by the 1000-element sequence will be automatically recycled due to Euphoria's dynamic storage allocation.

Note that the equals symbol '=' is used for both assignment and for [equality testing](#). There is never any confusion because an assignment in Euphoria is a statement only, it can't be used as an expression (as in C).

## Assignment with Operator

Euphoria also provides some additional forms of the assignment statement.

To save typing, and to make your code a bit neater, you can combine assignment with one of the operators:

**+ - / \* &**

For example, instead of saying:

```
mylongvarname = mylongvarname + 1
```

You can say:

```
mylongvarname += 1
```

Instead of saying:

```
galaxy[q_row][q_col][q_size] = galaxy[q_row][q_col][q_size] * 10
```

You can say:

```
galaxy[q_row][q_col][q_size] *= 10
```

and instead of saying:

```
accounts[start..finish] = accounts[start..finish] / 10
```

You can say:

```
accounts[start..finish] /= 10
```

In general, whenever you have an assignment of the form:

***left-hand-side = left-hand-side op expression***

You can say:

***left-hand-side op= expression***

where **op** is one of: **+ - \* / &**

When the left-hand-side contains multiple subscripts/slices, the **op=** form will usually execute faster than the longer form. When you get used to it, you may find the **op=** form to be slightly more readable than the long form, since you don't have to visually compare the left-hand-side against the copy of itself on the right side.



## 2.5.2 procedure call

A **procedure call** starts execution of a procedure, passing it an optional list of argument values. e.g.

```
plot(x, 23)
```

### 2.5.3 if statement

An **if statement** tests a condition to see if it is 0 (false) or non-zero (true) and then executes the appropriate series of statements. There may be optional **elsif** and **else** clauses. e.g.

```
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 doesn't 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**'s contained in it.

The **if** and **elsif** conditions are tested using [short-circuit evaluation](#).

## 2.5.4 while statement

A **while statement** tests a condition to see if it is non-zero (true), and while it is true a loop is executed. e.g.

```
while x > 0 do
    a = a * 2
    x = x - 1
end while
```

## Short-Circuit Evaluation

When the condition tested by **if**, **elsif**, or **while** contains **and** or **or** operators, *short-circuit* evaluation will be used. For example,

```
if a < 0 and b > 0 then ...
```

If  $a < 0$  is false, then Euphoria will not bother to test if  $b$  is greater than 0. It will assume that the overall result is false. Similarly,

```
if a < 0 or b > 0 then ...
```

if  $a < 0$  is true, then Euphoria will immediately decide that the result true, without testing the value of  $b$ .

In general, whenever we have a condition of the form:

```
A and B
```

where  $A$  and  $B$  can be any two expressions, Euphoria will take a short-cut when  $A$  is false and immediately make the overall result false, without even looking at expression  $B$ .

Similarly, with:

```
A or B
```

when  $A$  is true, Euphoria will skip the evaluation of expression  $B$ , and declare the result to be true.

If the expression  $B$  contains a call to a function, and that function has possible **side-effects**, i.e. it might do more than just return a value, you will get a compile-time warning. Older versions (pre-2.1) of Euphoria did not use **short-circuit** evaluation, and it's possible that some old code will no longer work correctly, although a search of the Euphoria archives did not turn up any programs that depend on side-effects in this way.

The expression,  $B$ , could contain something that would normally cause a run-time error. If Euphoria skips the evaluation of  $B$ , the error will not be discovered. For instance:

```
if x != 0 and 1/x > 10 then -- divide by zero error avoided
```

```
while 1 or {1,2,3,4,5} do -- illegal sequence result avoided
```

$B$  could even contain uninitialized variables, out-of-bounds subscripts etc.

This may look like sloppy coding, but in fact it often allows you to write something in a simpler and more readable way. For instance:

```
if atom(x) or length(x)=1 then
```

Without short-circuiting, you would have a problem when  $x$  was an atom, since `length` is not defined for atoms. With short-circuiting, `length(x)` will only be checked when  $x$  is a sequence. Similarly:

```
-- find 'a' or 'A' in s
i = 1
while i <= length(s) and s[i] != 'a' and s[i] != 'A' do
    i += 1
end while
```

In this loop the variable  $i$  might eventually become greater than `length(s)`. Without short-circuit evaluation, a subscript out-of-bounds error will occur when `s[i]` is evaluated on the final iteration. With short-circuiting, the loop will terminate immediately when `i <= length(s)` becomes false. Euphoria will not evaluate `s[i] != 'a'` and will not evaluate `s[i] != 'A'`. No subscript error will occur.

**Short-circuit** evaluation of **and** and **or** takes place for **if**, **elsif** and **while** conditions only. It is not used in other contexts. For example, the assignment statement:

```
x = 1 or {1,2,3,4,5} -- x should be set to {1,1,1,1,1}
```

If short-circuiting were used here, we would set  $x$  to 1, and not even look at `{1,2,3,4,5}`. This would be wrong. Short-circuiting can be used in if/elsif/while conditions because we only care if the result is true or false, and conditions are required to produce an atom as a result.

## 2.5.5 for statement

A **for statement** sets up a special loop with a controlling **loop variable** that runs from an initial value up or down to some final value. e.g.

```
for i = 1 to 10 do
  ? i    -- ? is a short form for print()
end for

-- fractional numbers allowed too
for i = 10.0 to 20.5 by 0.3 do
  for j = 20 to 10 by -2 do    -- counting down
    ? {i, j}
  end for
end for
```

The **loop variable** is declared automatically and exists until the end of the loop. Outside of the loop the variable has no value and is not even declared. If you need its final value, copy it into another variable before leaving the loop. The compiler will not allow any assignments to a loop variable. The initial value, loop limit and increment must all be atoms. If no increment is specified then +1 is assumed. The limit and increment values are established when the loop is entered, and are not affected by anything that happens during the execution of the loop. See also the [scope of the loop variable in 2.4.2 Scope](#).

### 2.5.6 return statement

A **return statement** returns immediately from a subroutine. If the subroutine is a function or type then a value must also be returned. e.g.

```
return
```

```
return {50, "FRED", {}}
```

## 2.5.7 exit statement

An **exit statement** may appear inside a [while-loop](#) or a [for-loop](#). It causes immediate termination of the loop, with control passing to the first statement after the loop. e.g.

```
for i = 1 to 100 do
  if a[i] = x then
    location = i
    exit
  end if
end for
```

It is also quite common to see something like this:

```
constant TRUE = 1

while TRUE do
  ...
  if some_condition then
    exit
  end if
  ...
end while
```

i.e. an "infinite" while-loop that actually terminates via an **exit statement** at some arbitrary point in the body of the loop.

### Performance Note:

Euphoria optimizes this type of loop. At run-time, no test is performed at the top of the loop. There's just a simple unconditional jump from **end while** back to the first statement inside the loop.

With **ex.exe**, if you happen to create a real infinite loop, with no input/output taking place, there is no easy way to stop it. You will have to type Control-Alt-Delete to either reboot, or (under Windows) terminate your DOS prompt session. If the program had files open for writing, it would be advisable to run **scandisk** to check your file system integrity. Only when your program is waiting for keyboard input, will control-c abort the program (unless [allow\\_break\(0\)](#) was used).

With **exw.exe** or **exu**, control-c will always stop your program immediately.

---

## 2.6 Special Top-Level Statements

Before any of your statements are executed, the Euphoria front-end quickly reads your entire program. All statements are syntax checked and converted to a low-level intermediate language (IL). The interpreter immediately executes the IL. The translator converts the IL to C. The binder/shrouder saves the IL on disk for later execution. These three tools all share the same front-end (written in Euphoria).

If your program contains only routine and variable declarations, but no top-level executable statements, then nothing will happen when you run it (other than syntax checking). You need a top-level statement to call your main routine (see [1.1 Example Program](#)). It's quite possible to have a program with nothing but top-level executable statements and no routines. For example you might want to use Euphoria as a simple calculator, typing just a few [print](#) (or [?](#)) statements into a file, and then executing it.

As we have seen, you can use any Euphoria [statement](#), including [for-loops](#), [while-loops](#), [if](#) statements etc. (but not [return](#)), at the top level i.e. *outside* of any [function](#) or [procedure](#). In addition, the following special statements may *only* appear at the top level:

- include
- with / without



## 2.6.1 include

When you write a large program it is often helpful to break it up into logically separate files, by using **include statements**. Sometimes you will want to reuse some code that you have previously written, or that someone else has written. Rather than copy this code into your main program, you can use an **include statement** to refer to the file containing the code. The first form of the include statement is:

### **include filename**

This reads in (compiles) a Euphoria source file.

Some Examples:

```
include graphics.e
include \mylib\myroutines.e
```

Any top-level code in the included file will be executed.

Any [global symbols](#) that have already been defined in the main file will be visible in the included file.

**N.B.** Only those symbols defined as [global](#) in the included file will be visible (accessible) in the remainder of the program.

If an absolute *filename* is given, Euphoria will use it. When a relative *filename* is given, Euphoria will first look for it in the same directory as the main file given on the **ex** (or **exw** or **exu**) [command-line](#). If it's not there, and you've defined an environment variable, **EUINC**, it will search each directory listed in **EUINC** (from left to right). Finally, if it still hasn't found the file, it will search [euphoria\include](#). This directory contains the standard Euphoria include files. The environment variable **EUDIR** tells **ex.exe/exw.exe/exu** where to find your **euphoria** directory. **EUINC** should be a list of directories, separated by semicolons (colons on Linux), similar in form to your PATH variable. It can be added to your AUTOEXEC.BAT file, e.g.

```
SET EUINC=C:\EU\MYFILES;C:\EU\WIN32LIB
```

This lets you organize your include files according to application areas, and avoid adding numerous unrelated files to euphoria\include.

An included file can include other files. In fact, you can "nest" included files up to 30 levels deep.

Include file names typically end in **.e**, or sometimes **.ew** or **.eu** (when they are intended for use with Windows or Linux). This is just a convention. It is not required.

If your filename (or path) contains blanks, you must enclose it in double-quotes, otherwise quotes are optional. Also, be sure to double-up your backslashes. For example:

```
include "c:\\program files\\myfile.e"
```

Other than possibly defining a new namespace identifier (see below), an include statement will be quietly ignored if the same file has already been included.

An include statement must be written on a line by itself. Only a comment can appear after it on the same line.

The second form of the include statement is:

**include filename as namespace\_identifier**

This is just like the simple include, but it also defines a *namespace identifier* that can be attached to global symbols in the included file that you want to refer to in the main file. This might be necessary to disambiguate references to those symbols, or you might feel that it makes your code more readable. See [Scope Rules](#) for more.

## 2.6.2 with / without

These special statements affect the way that Euphoria translates your program into internal form. They are not meant to change the logic of your program, but they may affect the diagnostic information that you get from running your program. See [3. Debugging and Profiling](#) for more information.

### with

This turns **on** one of the options: **profile**, **profile\_time**, **trace**, **warning** or **type\_check**. Options **warning** and **type\_check** are initially on, while **profile**, **profile\_time** and **trace** are initially off.

Any warnings that are issued will appear on your screen after your program has finished execution. Warnings indicate very minor problems. A warning will never stop your program from executing.

### without

This turns **off** one of the above options.

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.

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.

... continue [3. Debugging and Profiling](#)

## 3. Debugging and Profiling

### 3.1 Debugging

Debugging in Euphoria is much easier than in most other programming languages. 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 **ex.err** is not convenient, 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.

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**.

**with trace** / **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.

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)
```

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. e.g.

```
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. e.g.

```
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 0 for the strange value (99) instead of 1, you can force a diagnostic dump into **ex.err**.

### 3.1.1 The 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. 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:

**F1** - display main output screen - take a look at your program's output so far

**F2** - redisplay trace screen. Press this key while viewing the main output screen to return to the trace display.

**Enter** - execute the currently-highlighted statement only

**down-arrow** - 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 [for-loop](#) or [while-loop](#) without having to witness all iterations of the loop.

**?** - 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 tracing and resume normal execution. Tracing will start again when the next `trace(1)` is executed.

**Q** - quit tracing and let the program run freely to its normal completion. `trace()` statements will be ignored.

**!** - 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 `gets()` (read one line) input. When `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()`.

### 3.1.2 The 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

## 3.2 Profiling

If you specify **with profile** (DOS32, WIN32, Linux, FreeBSD), or **with profile\_time** (DOS32 only) 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't 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 (DOS32 only) 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. View this file with the Euphoria editor to see a color display.

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 - don't 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 didn't 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.

### 3.2.1 Some Further Notes on Time Profiling

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.

These interrupts normally occur 18.2 times per second, but if you call `tick_rate()` you can choose a much higher rate and thus get a more accurate time profile, since it will be based on more samples. By default, if you haven't called `tick_rate()`, then `tick_rate(100)` will be called automatically when you start profiling. You can set it even higher (up to say 1000) but you may start to affect your program's performance.

Each sample requires 4 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's 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. e.g.

```
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.

... continue [Part II - Library Routines](#)



## 3.1 Debugging

Debugging in Euphoria is much easier than in most other programming languages. 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 **ex.err** is not convenient, 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.

### 3.1.1 The 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. 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:

- F1** - display main output screen - take a look at your program's output so far
- F2** - redisplay trace screen. Press this key while viewing the main output screen to return to the trace display.
- Enter** - execute the currently-highlighted statement only
- down-arrow** - 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 [for-loop](#) or [while-loop](#) without having to witness all iterations of the loop.
- ?** - 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 tracing and resume normal execution. Tracing will start again when the next [trace\(1\)](#) is executed.
- Q** - quit tracing and let the program run freely to its normal completion. [trace\(\)](#) statements will be ignored.
- !** - 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 [gets\(\)](#) (read one line) input. When [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\(\)](#).



### 3.1.2 The 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**.

---

## 3.2 Profiling

If you specify **with profile** (DOS32, WIN32, Linux, FreeBSD), or **with profile\_time** (DOS32 only) 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't 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 (DOS32 only) 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. View this file with the Euphoria editor to see a color display.

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 - don't 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 didn't 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.

### 3.2.1 Some Further Notes on Time Profiling

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.

These interrupts normally occur 18.2 times per second, but if you call `tick_rate()` you can choose a much higher rate and thus get a more accurate time profile, since it will be based on more samples. By default, if you haven't called `tick_rate()`, then `tick_rate(100)` will be called automatically when you start profiling. You can set it even higher (up to say 1000) but you may start to affect your program's performance.

Each sample requires 4 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's 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. e.g.

```
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.

... continue [Part II - Library Routines](#)

## Part II - Library Routines

### 1. Introduction

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

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

- x** - a general [object](#) (atom or sequence)
- s** - a [sequence](#)
- a** - an [atom](#)
- i** - an [integer](#)
- fn** - an [integer](#) used as a file number
- st** - a [string sequence](#), or [single-character atom](#)

Some routines are only available on one or two of the four platforms. This is noted with "Platform: **DOS32**" or "Platform: **WIN32**" or "Platform: **Linux**" in the description of the routine, and with (**DOS32**) or (**WIN32**) or (**Linux**) in some other places. Things that work on Linux can generally be assumed to work on FreeBSD. The only exception is the mouse routines.

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

---

### 2. Routines by Application Area

#### 2.1 Predefined Types

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

- [integer](#) - test if an object is an integer
  - [atom](#) - test if an object is an atom
  - [sequence](#) - test if an object is a sequence
  - [object](#) - test if an object is an object (always true)
- 

#### 2.2 Sequence Manipulation

- [length](#) - return the length of a sequence
- [repeat](#) - repeat an object n times to form a sequence of length n
- [reverse](#) - reverse a sequence

- [append](#) - add a new element to the end of a sequence
  - [prepend](#) - add a new element to the beginning of a sequence
- 

## 2.3 Searching and Sorting

- [compare](#) - compare two objects
  - [equal](#) - test if two objects are identical
  - [find](#) - find an object in a sequence
  - [match](#) - find a sequence as a slice of another sequence
  - [sort](#) - sort the elements of a sequence into ascending order
  - [custom\\_sort](#) - sort the elements of a sequence based on a compare function that you supply
- 

## 2.4 Pattern Matching

- [lower](#) - convert an atom or sequence to lower case
  - [upper](#) - convert an atom or sequence to upper case
  - [wildcard\\_match](#) - match a pattern containing ? and \* wildcards
  - [wildcard\\_file](#) - match a file name against a wildcard specification
- 

## 2.5 Math

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

- [sqrt](#) - calculate the square root of an object
  - [rand](#) - generate random numbers
  - [sin](#) - calculate the sine of an angle
  - [arcsin](#) - calculate the angle with a given sine
  - [cos](#) - calculate the cosine of an angle
  - [arccos](#) - calculate the angle with a given cosine
  - [tan](#) - calculate the tangent of an angle
  - [arctan](#) - calculate the arc tangent of a number
  - [log](#) - calculate the natural logarithm
  - [floor](#) - round down to the nearest integer
  - [remainder](#) - calculate the remainder when a number is divided by another
  - [power](#) - calculate a number raised to a power
  - [PI](#) - the mathematical value PI (3.14159...)
- 

## 2.6 Bitwise Logical Operations

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

- [and\\_bits](#) - perform logical AND on corresponding bits
  - [or\\_bits](#) - perform logical OR on corresponding bits
  - [xor\\_bits](#) - perform logical XOR on corresponding bits
  - [not\\_bits](#) - perform logical NOT on all bits
- 

## 2.7 File and Device I/O



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

- 0 - standard input
- 1 - standard output
- 2 - standard error

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

**Note:**

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

<a href="#">open</a>	- open a file or device
<a href="#">close</a>	- close a file or device
<a href="#">flush</a>	- flush out buffered data to a file or device
<a href="#">lock_file</a>	- lock a file or device
<a href="#">unlock_file</a>	- unlock a file or device
<a href="#">print</a> structure	- print a Euphoria object on one line, with braces and commas {,,} to show the structure
<a href="#">pretty_print</a> indentation	- print a Euphoria object in a nice readable form, using multiple lines and appropriate indentation
<a href="#">? x</a>	- shorthand for print(1, x)
<a href="#">sprint</a>	- return a printed Euphoria object as a string sequence
<a href="#">printf</a>	- formatted print to a file or device
<a href="#">sprintf</a>	- formatted print returned as a string sequence
<a href="#">puts</a>	- output a string sequence to a file or device
<a href="#">getc</a>	- read the next character from a file or device
<a href="#">gets</a>	- read the next line from a file or device
<a href="#">get_bytes</a>	- read the next n bytes from a file or device
<a href="#">prompt_string</a>	- prompt the user to enter a string
<a href="#">get_key</a>	- check for key pressed by the user, don't wait
<a href="#">wait_key</a>	- wait for user to press a key
<a href="#">get</a>	- read the representation of any Euphoria object from a file
<a href="#">prompt_number</a>	- prompt the user to enter a number
<a href="#">value</a>	- read the representation of any Euphoria object from a string
<a href="#">seek</a>	- move to any byte position within an open file
<a href="#">where</a>	- report the current byte position in an open file
<a href="#">current_dir</a>	- return the name of the current directory
<a href="#">chdir</a>	- change to a new current directory
<a href="#">dir</a>	- return complete info on all files in a directory
<a href="#">walk_dir</a>	- recursively walk through all files in a directory
<a href="#">allow_break</a>	- allow control-c/control-Break to terminate your program or not

[check\\_break](#) - check if user has pressed control-c or control-Break

---

## 2.8 Mouse Support (DOS32 and Linux)

**Note:** On Windows XP, if you want the DOS mouse to work in a (non-full-screen) window, you must disable QuickEdit mode in the Properties for the DOS Window.

[get\\_mouse](#) - return mouse "events" (clicks, movements)  
[mouse\\_events](#) - select mouse events to watch for  
[mouse\\_pointer](#) - display or hide the mouse pointer

---

## 2.9 Operating System

[time](#) - number of seconds since a fixed point in the past  
[tick\\_rate](#) - set the number of clock ticks per second (DOS32)  
[date](#) - current year, month, day, hour, minute, second etc.  
[command\\_line](#) - command-line used to run this program  
[getenv](#) - get value of an environment variable  
[system](#) - execute an operating system command line  
[system\\_exec](#) - execute a program and get its exit code  
[abort](#) - terminate execution  
[sleep](#) - suspend execution for a period of time  
[platform](#) - find out which operating system are we running on

---

## 2.10 Special Machine-Dependent Routines

[machine\\_func](#) - specialized internal operations with a return value  
[machine\\_proc](#) - specialized internal operations with no return value

---

## 2.11 Debugging

[trace](#) - dynamically turns tracing on or off  
[profile](#) - dynamically turns profiling on or off

---

## 2.12 Graphics & Sound

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

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

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

[clear\\_screen](#) - clear the screen

<a href="#"><u>position</u></a>	- set cursor line and column
<a href="#"><u>get_position</u></a>	- return cursor line and column
<a href="#"><u>graphics_mode</u></a>	- select a new pixel-graphics or text mode (DOS32)
<a href="#"><u>video_config</u></a>	- return parameters of current mode
<a href="#"><u>scroll</u></a>	- scroll text up or down
<a href="#"><u>wrap</u></a>	- control line wrap at right edge of screen
<a href="#"><u>text_color</u></a>	- set foreground text color
<a href="#"><u>bk_color</u></a>	- set background color
<a href="#"><u>palette</u></a>	- change color for one color number (DOS32)
<a href="#"><u>all_palette</u></a>	- change color for all color numbers (DOS32)
<a href="#"><u>get_all_palette</u></a>	- get the palette values for all colors (DOS32)
<a href="#"><u>read_bitmap</u></a>	- read a bitmap (.bmp) file and return a palette and a 2-d sequence of pixels
<a href="#"><u>save_bitmap</u></a>	- create a bitmap (.bmp) file, given a palette and a 2-d sequence of pixels
<a href="#"><u>get_active_page</u></a>	- return the page currently being written to (DOS32)
<a href="#"><u>set_active_page</u></a>	- change the page currently being written to (DOS32)
<a href="#"><u>get_display_page</u></a>	- return the page currently being displayed (DOS32)
<a href="#"><u>set_display_page</u></a>	- change the page currently being displayed (DOS32)
<a href="#"><u>sound</u></a>	- make a sound on the PC speaker (DOS32)

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

<a href="#"><u>cursor</u></a>	- select cursor shape
<a href="#"><u>text_rows</u></a>	- set number of lines on text screen
<a href="#"><u>get_screen_char</u></a>	- get one character from the screen
<a href="#"><u>put_screen_char</u></a>	- put one or more characters on the screen
<a href="#"><u>save_text_image</u></a>	- save a rectangular region from a text screen
<a href="#"><u>display_text_image</u></a>	- display an image on the text screen

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

<a href="#"><u>pixel</u></a>	- set color of a pixel or set of pixels
<a href="#"><u>get_pixel</u></a>	- read color of a pixel or set of pixels
<a href="#"><u>draw_line</u></a>	- connect a series of graphics points with a line
<a href="#"><u>polygon</u></a>	- draw an n-sided figure
<a href="#"><u>ellipse</u></a>	- draw an ellipse or circle
<a href="#"><u>save_screen</u></a>	- save the screen to a bitmap (.bmp) file
<a href="#"><u>save_image</u></a>	- save a rectangular region from a pixel-graphics screen
<a href="#"><u>display_image</u></a>	- display an image on the pixel-graphics screen

## 2.13 Machine Level Interface

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

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

Under Linux you'll typically get a segmentation violation.

**Note:**

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

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

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

<a href="#">peek</a>	- read one or more bytes from memory
<a href="#">peek4s</a>	- read 4-byte signed values from memory
<a href="#">peek4u</a>	- read 4-byte unsigned values from memory
<a href="#">poke</a>	- write one or more bytes to memory
<a href="#">poke4</a>	- write 4-byte values into memory
<a href="#">mem_copy</a>	- copy a block of memory
<a href="#">mem_set</a>	- set a block of memory to a value
<a href="#">call</a>	- call a machine language routine
<a href="#">dos_interrupt</a>	- call a DOS software interrupt routine (DOS32)
<a href="#">allocate</a>	- allocate a block of memory
<a href="#">free</a>	- deallocate a block of memory
<a href="#">allocate_low</a>	- allocate a block of low memory (address less than 1Mb) (DOS32)
<a href="#">free_low</a>	- free a block allocated with <a href="#">allocate_low</a> (DOS32)
<a href="#">allocate_string</a>	- allocate a string of characters with 0 terminator
<a href="#">register_block</a>	- register an externally-allocated block of memory
<a href="#">unregister_block</a>	- unregister an externally-allocated block of memory
<a href="#">get_vector</a>	- return address of interrupt handler (DOS32)
<a href="#">set_vector</a>	- set address of interrupt handler (DOS32)
<a href="#">lock_memory</a>	- ensure that a region of memory will never be swapped out (DOS32)
<a href="#">int_to_bytes</a>	- convert an integer to 4 bytes
<a href="#">bytes_to_int</a>	- convert 4 bytes to an integer
<a href="#">int_to_bits</a>	- convert an integer to a sequence of bits
<a href="#">bits_to_int</a>	- convert a sequence of bits to an integer
<a href="#">atom_to_float64</a>	- convert an atom, to a sequence of 8 bytes in IEEE 64-bit floating-point format
<a href="#">atom_to_float32</a>	- convert an atom, to a sequence of 4 bytes in IEEE 32-bit floating-point format
<a href="#">float64_to_atom</a>	- convert a sequence of 8 bytes in IEEE 64-bit floating-point format, to an atom
<a href="#">float32_to_atom</a>	- convert a sequence of 4 bytes in IEEE 32-bit floating-point format, to an atom
<a href="#">set_rand</a>	- set the random number generator so it will generate a repeatable series of random numbers
<a href="#">use_vesa</a>	- force the use of the VESA graphics standard (DOS32)
<a href="#">crash_file</a>	- specify the file for writing error diagnostics if Euphoria detects an error in your

program.

- [crash\\_message](#) - specify a message to be printed if Euphoria detects an error in your program
  - [crash\\_routine](#) - specify a routine to be called if Euphoria detects an error in your program
- 

## 2.14 Dynamic Calls

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

- [routine\\_id](#) - get a unique identifying number for a Euphoria routine
  - [call\\_proc](#) - call a Euphoria procedure using a routine id
  - [call\\_func](#) - call a Euphoria function using a routine id
- 

## 2.15 Calling C Functions (WIN32 and Linux)

See [platform.doc](#) for a description of WIN32 and Linux programming in Euphoria.

- [open\\_dll](#) - open a Windows dynamic link library (.dll file) or Linux shared library (.so file)
  - [define\\_c\\_proc](#) - define a C function that is VOID (no value returned), or whose value your program will ignore
  - [define\\_c\\_func](#) - define a C function that returns a value that your program will use
  - [define\\_c\\_var](#) - get the memory address of a C variable.
  - [c\\_proc](#) - call a C function, ignoring any return value
  - [c\\_func](#) - call a C function and get the return value
  - [call\\_back](#) - get a 32-bit machine address for a Euphoria routine for use as a call-back address
  - [message\\_box](#) - pop up a small window to get a Yes/No/Cancel response from the user
  - [free\\_console](#) - delete the console text window
  - [instance](#) - get the instance handle for the current program
- 

## 2.16 Multitasking

This collection of routines lets you create multiple, independent tasks. Each task has its own current statement being executed, its own subroutine call stack, and its own set of private variables. The local and global variables of a program are shared amongst all tasks. When a task calls `task_yield()`, it is suspended, and the Euphoria scheduler decides which task to execute next.

The Language War demo program, `lw.ex`, makes heavy use of tasks to create a simulated battle involving numerous independently moving ships, torpedos, phasors etc. See also the `taskwire.exw` Windows demo program, and the `news.exu` demo for Linux and FreeBSD.

- [task\\_clock\\_start](#) - restart the scheduler's clock
- [task\\_clock\\_stop](#) - stop the scheduler's clock
- [task\\_create](#) - create a new task
- [task\\_list](#) - get a list of all tasks
- [task\\_schedule](#) - schedule a task for execution
- [task\\_self](#) - return the task id of the current task
- [task\\_status](#) - the current status (active, suspended, killed) of a task
- [task\\_suspend](#) - Suspend a task.
- [task\\_yield](#) - Yield control, so the scheduler can pick a new task to run.

... continue [3. Alphabetical Listing of All Routines, From A to B](#)

## Part II - Library Routines

### 1. Introduction

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

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

- x** - a general [object](#) (atom or sequence)
- s** - a [sequence](#)
- a** - an [atom](#)
- i** - an [integer](#)
- fn** - an [integer](#) used as a file number
- st** - a [string sequence](#), or [single-character atom](#)

Some routines are only available on one or two of the four platforms. This is noted with "Platform: **DOS32**" or "Platform: **WIN32**" or "Platform: **Linux**" in the description of the routine, and with (**DOS32**) or (**WIN32**) or (**Linux**) in some other places. Things that work on Linux can generally be assumed to work on FreeBSD. The only exception is the mouse routines.

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

---

### 2. Routines by Application Area

#### 2.1 Predefined Types

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

- [integer](#) - test if an object is an integer
  - [atom](#) - test if an object is an atom
  - [sequence](#) - test if an object is a sequence
  - [object](#) - test if an object is an object (always true)
- 

#### 2.2 Sequence Manipulation

- [length](#) - return the length of a sequence
- [repeat](#) - repeat an object n times to form a sequence of length n
- [reverse](#) - reverse a sequence

- [append](#) - add a new element to the end of a sequence
  - [prepend](#) - add a new element to the beginning of a sequence
- 

## 2.3 Searching and Sorting

- [compare](#) - compare two objects
  - [equal](#) - test if two objects are identical
  - [find](#) - find an object in a sequence
  - [match](#) - find a sequence as a slice of another sequence
  - [sort](#) - sort the elements of a sequence into ascending order
  - [custom\\_sort](#) - sort the elements of a sequence based on a compare function that you supply
- 

## 2.4 Pattern Matching

- [lower](#) - convert an atom or sequence to lower case
  - [upper](#) - convert an atom or sequence to upper case
  - [wildcard\\_match](#) - match a pattern containing ? and \* wildcards
  - [wildcard\\_file](#) - match a file name against a wildcard specification
- 

## 2.5 Math

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

- [sqrt](#) - calculate the square root of an object
  - [rand](#) - generate random numbers
  - [sin](#) - calculate the sine of an angle
  - [arcsin](#) - calculate the angle with a given sine
  - [cos](#) - calculate the cosine of an angle
  - [arccos](#) - calculate the angle with a given cosine
  - [tan](#) - calculate the tangent of an angle
  - [arctan](#) - calculate the arc tangent of a number
  - [log](#) - calculate the natural logarithm
  - [floor](#) - round down to the nearest integer
  - [remainder](#) - calculate the remainder when a number is divided by another
  - [power](#) - calculate a number raised to a power
  - [PI](#) - the mathematical value PI (3.14159...)
- 

## 2.6 Bitwise Logical Operations

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

- [and\\_bits](#) - perform logical AND on corresponding bits
  - [or\\_bits](#) - perform logical OR on corresponding bits
  - [xor\\_bits](#) - perform logical XOR on corresponding bits
  - [not\\_bits](#) - perform logical NOT on all bits
- 

## 2.7 File and Device I/O



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

- 0 - standard input
- 1 - standard output
- 2 - standard error

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

**Note:**

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

<a href="#">open</a>	- open a file or device
<a href="#">close</a>	- close a file or device
<a href="#">flush</a>	- flush out buffered data to a file or device
<a href="#">lock_file</a>	- lock a file or device
<a href="#">unlock_file</a>	- unlock a file or device
<a href="#">print</a> structure	- print a Euphoria object on one line, with braces and commas {,,} to show the structure
<a href="#">pretty_print</a> indentation	- print a Euphoria object in a nice readable form, using multiple lines and appropriate indentation
<a href="#">? x</a>	- shorthand for print(1, x)
<a href="#">sprint</a>	- return a printed Euphoria object as a string sequence
<a href="#">printf</a>	- formatted print to a file or device
<a href="#">sprintf</a>	- formatted print returned as a string sequence
<a href="#">puts</a>	- output a string sequence to a file or device
<a href="#">getc</a>	- read the next character from a file or device
<a href="#">gets</a>	- read the next line from a file or device
<a href="#">get_bytes</a>	- read the next n bytes from a file or device
<a href="#">prompt_string</a>	- prompt the user to enter a string
<a href="#">get_key</a>	- check for key pressed by the user, don't wait
<a href="#">wait_key</a>	- wait for user to press a key
<a href="#">get</a>	- read the representation of any Euphoria object from a file
<a href="#">prompt_number</a>	- prompt the user to enter a number
<a href="#">value</a>	- read the representation of any Euphoria object from a string
<a href="#">seek</a>	- move to any byte position within an open file
<a href="#">where</a>	- report the current byte position in an open file
<a href="#">current_dir</a>	- return the name of the current directory
<a href="#">chdir</a>	- change to a new current directory
<a href="#">dir</a>	- return complete info on all files in a directory
<a href="#">walk_dir</a>	- recursively walk through all files in a directory
<a href="#">allow_break</a>	- allow control-c/control-Break to terminate your program or not

[check\\_break](#) - check if user has pressed control-c or control-Break

---

## 2.8 Mouse Support (DOS32 and Linux)

**Note:** On Windows XP, if you want the DOS mouse to work in a (non-full-screen) window, you must disable QuickEdit mode in the Properties for the DOS Window.

[get\\_mouse](#) - return mouse "events" (clicks, movements)  
[mouse\\_events](#) - select mouse events to watch for  
[mouse\\_pointer](#) - display or hide the mouse pointer

---

## 2.9 Operating System

[time](#) - number of seconds since a fixed point in the past  
[tick\\_rate](#) - set the number of clock ticks per second (DOS32)  
[date](#) - current year, month, day, hour, minute, second etc.  
[command\\_line](#) - command-line used to run this program  
[getenv](#) - get value of an environment variable  
[system](#) - execute an operating system command line  
[system\\_exec](#) - execute a program and get its exit code  
[abort](#) - terminate execution  
[sleep](#) - suspend execution for a period of time  
[platform](#) - find out which operating system are we running on

---

## 2.10 Special Machine-Dependent Routines

[machine\\_func](#) - specialized internal operations with a return value  
[machine\\_proc](#) - specialized internal operations with no return value

---

## 2.11 Debugging

[trace](#) - dynamically turns tracing on or off  
[profile](#) - dynamically turns profiling on or off

---

## 2.12 Graphics & Sound

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

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

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

[clear\\_screen](#) - clear the screen

<a href="#"><u>position</u></a>	- set cursor line and column
<a href="#"><u>get_position</u></a>	- return cursor line and column
<a href="#"><u>graphics_mode</u></a>	- select a new pixel-graphics or text mode (DOS32)
<a href="#"><u>video_config</u></a>	- return parameters of current mode
<a href="#"><u>scroll</u></a>	- scroll text up or down
<a href="#"><u>wrap</u></a>	- control line wrap at right edge of screen
<a href="#"><u>text_color</u></a>	- set foreground text color
<a href="#"><u>bk_color</u></a>	- set background color
<a href="#"><u>palette</u></a>	- change color for one color number (DOS32)
<a href="#"><u>all_palette</u></a>	- change color for all color numbers (DOS32)
<a href="#"><u>get_all_palette</u></a>	- get the palette values for all colors (DOS32)
<a href="#"><u>read_bitmap</u></a>	- read a bitmap (.bmp) file and return a palette and a 2-d sequence of pixels
<a href="#"><u>save_bitmap</u></a>	- create a bitmap (.bmp) file, given a palette and a 2-d sequence of pixels
<a href="#"><u>get_active_page</u></a>	- return the page currently being written to (DOS32)
<a href="#"><u>set_active_page</u></a>	- change the page currently being written to (DOS32)
<a href="#"><u>get_display_page</u></a>	- return the page currently being displayed (DOS32)
<a href="#"><u>set_display_page</u></a>	- change the page currently being displayed (DOS32)
<a href="#"><u>sound</u></a>	- make a sound on the PC speaker (DOS32)

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

<a href="#"><u>cursor</u></a>	- select cursor shape
<a href="#"><u>text_rows</u></a>	- set number of lines on text screen
<a href="#"><u>get_screen_char</u></a>	- get one character from the screen
<a href="#"><u>put_screen_char</u></a>	- put one or more characters on the screen
<a href="#"><u>save_text_image</u></a>	- save a rectangular region from a text screen
<a href="#"><u>display_text_image</u></a>	- display an image on the text screen

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

<a href="#"><u>pixel</u></a>	- set color of a pixel or set of pixels
<a href="#"><u>get_pixel</u></a>	- read color of a pixel or set of pixels
<a href="#"><u>draw_line</u></a>	- connect a series of graphics points with a line
<a href="#"><u>polygon</u></a>	- draw an n-sided figure
<a href="#"><u>ellipse</u></a>	- draw an ellipse or circle
<a href="#"><u>save_screen</u></a>	- save the screen to a bitmap (.bmp) file
<a href="#"><u>save_image</u></a>	- save a rectangular region from a pixel-graphics screen
<a href="#"><u>display_image</u></a>	- display an image on the pixel-graphics screen

## 2.13 Machine Level Interface

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

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

Under Linux you'll typically get a segmentation violation.

**Note:**

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

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

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

<a href="#">peek</a>	- read one or more bytes from memory
<a href="#">peek4s</a>	- read 4-byte signed values from memory
<a href="#">peek4u</a>	- read 4-byte unsigned values from memory
<a href="#">poke</a>	- write one or more bytes to memory
<a href="#">poke4</a>	- write 4-byte values into memory
<a href="#">mem_copy</a>	- copy a block of memory
<a href="#">mem_set</a>	- set a block of memory to a value
<a href="#">call</a>	- call a machine language routine
<a href="#">dos_interrupt</a>	- call a DOS software interrupt routine (DOS32)
<a href="#">allocate</a>	- allocate a block of memory
<a href="#">free</a>	- deallocate a block of memory
<a href="#">allocate_low</a>	- allocate a block of low memory (address less than 1Mb) (DOS32)
<a href="#">free_low</a>	- free a block allocated with <a href="#">allocate_low</a> (DOS32)
<a href="#">allocate_string</a>	- allocate a string of characters with 0 terminator
<a href="#">register_block</a>	- register an externally-allocated block of memory
<a href="#">unregister_block</a>	- unregister an externally-allocated block of memory
<a href="#">get_vector</a>	- return address of interrupt handler (DOS32)
<a href="#">set_vector</a>	- set address of interrupt handler (DOS32)
<a href="#">lock_memory</a>	- ensure that a region of memory will never be swapped out (DOS32)
<a href="#">int_to_bytes</a>	- convert an integer to 4 bytes
<a href="#">bytes_to_int</a>	- convert 4 bytes to an integer
<a href="#">int_to_bits</a>	- convert an integer to a sequence of bits
<a href="#">bits_to_int</a>	- convert a sequence of bits to an integer
<a href="#">atom_to_float64</a>	- convert an atom, to a sequence of 8 bytes in IEEE 64-bit floating-point format
<a href="#">atom_to_float32</a>	- convert an atom, to a sequence of 4 bytes in IEEE 32-bit floating-point format
<a href="#">float64_to_atom</a>	- convert a sequence of 8 bytes in IEEE 64-bit floating-point format, to an atom
<a href="#">float32_to_atom</a>	- convert a sequence of 4 bytes in IEEE 32-bit floating-point format, to an atom
<a href="#">set_rand</a>	- set the random number generator so it will generate a repeatable series of random numbers
<a href="#">use_vesa</a>	- force the use of the VESA graphics standard (DOS32)
<a href="#">crash_file</a>	- specify the file for writing error diagnostics if Euphoria detects an error in your

program.

- [crash\\_message](#) - specify a message to be printed if Euphoria detects an error in your program
  - [crash\\_routine](#) - specify a routine to be called if Euphoria detects an error in your program
- 

## 2.14 Dynamic Calls

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

- [routine\\_id](#) - get a unique identifying number for a Euphoria routine
  - [call\\_proc](#) - call a Euphoria procedure using a routine id
  - [call\\_func](#) - call a Euphoria function using a routine id
- 

## 2.15 Calling C Functions (WIN32 and Linux)

See [platform.doc](#) for a description of WIN32 and Linux programming in Euphoria.

- [open\\_dll](#) - open a Windows dynamic link library (.dll file) or Linux shared library (.so file)
  - [define\\_c\\_proc](#) - define a C function that is VOID (no value returned), or whose value your program will ignore
  - [define\\_c\\_func](#) - define a C function that returns a value that your program will use
  - [define\\_c\\_var](#) - get the memory address of a C variable.
  - [c\\_proc](#) - call a C function, ignoring any return value
  - [c\\_func](#) - call a C function and get the return value
  - [call\\_back](#) - get a 32-bit machine address for a Euphoria routine for use as a call-back address
  - [message\\_box](#) - pop up a small window to get a Yes/No/Cancel response from the user
  - [free\\_console](#) - delete the console text window
  - [instance](#) - get the instance handle for the current program
- 

## 2.16 Multitasking

This collection of routines lets you create multiple, independent tasks. Each task has its own current statement being executed, its own subroutine call stack, and its own set of private variables. The local and global variables of a program are shared amongst all tasks. When a task calls `task_yield()`, it is suspended, and the Euphoria scheduler decides which task to execute next.

The Language War demo program, `lw.ex`, makes heavy use of tasks to create a simulated battle involving numerous independently moving ships, torpedos, phasors etc. See also the `taskwire.exw` Windows demo program, and the `news.exu` demo for Linux and FreeBSD.

- [task\\_clock\\_start](#) - restart the scheduler's clock
- [task\\_clock\\_stop](#) - stop the scheduler's clock
- [task\\_create](#) - create a new task
- [task\\_list](#) - get a list of all tasks
- [task\\_schedule](#) - schedule a task for execution
- [task\\_self](#) - return the task id of the current task
- [task\\_status](#) - the current status (active, suspended, killed) of a task
- [task\\_suspend](#) - Suspend a task.
- [task\\_yield](#) - Yield control, so the scheduler can pick a new task to run.

... continue [3. Alphabetical Listing of All Routines, From A to B](#)

## 2. Routines by Application Area

## 2.1 Predefined Types

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

- [integer](#) - test if an object is an integer
  - [atom](#) - test if an object is an atom
  - [sequence](#) - test if an object is a sequence
  - [object](#) - test if an object is an object (always true)
-



## 2.2 Sequence Manipulation

- [length](#) - return the length of a sequence
  - [repeat](#) - repeat an object n times to form a sequence of length n
  - [reverse](#) - reverse a sequence
  - [append](#) - add a new element to the end of a sequence
  - [prepend](#) - add a new element to the beginning of a sequence
-

## 2.3 Searching and Sorting

<a href="#"><u>compare</u></a>	- compare two objects
<a href="#"><u>equal</u></a>	- test if two objects are identical
<a href="#"><u>find</u></a>	- find an object in a sequence
<a href="#"><u>match</u></a>	- find a sequence as a slice of another sequence
<a href="#"><u>sort</u></a>	- sort the elements of a sequence into ascending order
<a href="#"><u>custom_sort</u></a>	- sort the elements of a sequence based on a compare function that you supply

---

## 2.4 Pattern Matching

[lower](#)

- convert an atom or sequence to lower case

[upper](#)

- convert an atom or sequence to upper case

[wildcard\\_match](#)

- match a pattern containing ? and \* wildcards

[wildcard\\_file](#)

- match a file name against a wildcard specification

---

## 2.5 Math

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

<a href="#"><u>sqrt</u></a>	- calculate the square root of an object
<a href="#"><u>rand</u></a>	- generate random numbers
<a href="#"><u>sin</u></a>	- calculate the sine of an angle
<a href="#"><u>arcsin</u></a>	- calculate the angle with a given sine
<a href="#"><u>cos</u></a>	- calculate the cosine of an angle
<a href="#"><u>arccos</u></a>	- calculate the angle with a given cosine
<a href="#"><u>tan</u></a>	- calculate the tangent of an angle
<a href="#"><u>arctan</u></a>	- calculate the arc tangent of a number
<a href="#"><u>log</u></a>	- calculate the natural logarithm
<a href="#"><u>floor</u></a>	- round down to the nearest integer
<a href="#"><u>remainder</u></a>	- calculate the remainder when a number is divided by another
<a href="#"><u>power</u></a>	- calculate a number raised to a power
<a href="#"><u>PI</u></a>	- the mathematical value PI (3.14159...)

---

## 2.6 Bitwise Logical Operations

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

- [and\\_bits](#) - perform logical AND on corresponding bits
  - [or\\_bits](#) - perform logical OR on corresponding bits
  - [xor\\_bits](#) - perform logical XOR on corresponding bits
  - [not\\_bits](#) - perform logical NOT on all bits
-

## 2.7 File and Device I/O

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

- 0 - standard input
- 1 - standard output
- 2 - standard error

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

**Note:**

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

<a href="#">open</a>	- open a file or device
<a href="#">close</a>	- close a file or device
<a href="#">flush</a>	- flush out buffered data to a file or device
<a href="#">lock_file</a>	- lock a file or device
<a href="#">unlock_file</a>	- unlock a file or device
<a href="#">print</a>	-

## 2.8 Mouse Support (DOS32 and Linux)

**Note:** On Windows XP, if you want the DOS mouse to work in a (non-full-screen) window, you must disable QuickEdit mode in the Properties for the DOS Window.

- [get\\_mouse](#) - return mouse "events" (clicks, movements)
  - [mouse\\_events](#) - select mouse events to watch for
  - [mouse\\_pointer](#) - display or hide the mouse pointer
-

## 2.9 Operating System

<u>time</u>	- number of seconds since a fixed point in the past
<u>tick_rate</u>	- set the number of clock ticks per second (DOS32)
<u>date</u>	- current year, month, day, hour, minute, second etc.
<u>command_line</u>	- command-line used to run this program
<u>getenv</u>	- get value of an environment variable
<u>system</u>	- execute an operating system command line
<u>system_exec</u>	- execute a program and get its exit code
<u>abort</u>	- terminate execution
<u>sleep</u>	- suspend execution for a period of time
<u>platform</u>	- find out which operating system are we running on

---



## 2.10 Special Machine-Dependent Routines

- [machine\\_func](#) - specialized internal operations with a return value
  - [machine\\_proc](#) - specialized internal operations with no return value
-

## 2.11 Debugging

- [trace](#) - dynamically turns tracing on or off
  - [profile](#) - dynamically turns profiling on or off
-

## 2.12 Graphics & Sound

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

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

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

<a href="#">clear_screen</a>	- clear the screen
<a href="#">position</a>	- set cursor line and column
<a href="#">get_position</a>	- return cursor line and column
<a href="#">graphics_mode</a>	- select a new pixel-graphics or text mode (DOS32)
<a href="#">video_config</a>	- return parameters of current mode
<a href="#">scroll</a>	- scroll text up or down
<a href="#">wrap</a>	- control line wrap at right edge of screen
<a href="#">text_color</a>	- set foreground text color
<a href="#">bk_color</a>	- set background color
<a href="#">palette</a>	- change color for one color number (DOS32)
<a href="#">all_palette</a>	- change color for all color numbers (DOS32)
<a href="#">get_all_palette</a>	- get the palette values for all colors (DOS32)
<a href="#">read_bitmap</a>	- read a bitmap (.bmp) file and return a palette and a 2-d sequence of pixels
<a href="#">save_bitmap</a>	- create a bitmap (.bmp) file, given a palette and a 2-d sequence of pixels
<a href="#">get_active_page</a>	- return the page currently being written to (DOS32)
<a href="#">set_active_page</a>	- change the page currently being written to (DOS32)
<a href="#">get_display_page</a>	- return the page currently being displayed (DOS32)
<a href="#">set_display_page</a>	- change the page currently being displayed (DOS32)
<a href="#">sound</a>	- make a sound on the PC speaker (DOS32)

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

<a href="#">cursor</a>	- select cursor shape
<a href="#">text_rows</a>	- set number of lines on text screen
<a href="#">get_screen_char</a>	- get one character from the screen
<a href="#">put_screen_char</a>	- put one or more characters on the screen
<a href="#">save_text_image</a>	- save a rectangular region from a text screen
<a href="#">display_text_image</a>	- display an image on the text screen

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

<a href="#">pixel</a>	- set color of a pixel or set of pixels
<a href="#">get_pixel</a>	- read color of a pixel or set of pixels
<a href="#">draw_line</a>	- connect a series of graphics points with a line
<a href="#">polygon</a>	- draw an n-sided figure
<a href="#">ellipse</a>	- draw an ellipse or circle
<a href="#">save_screen</a>	- save the screen to a bitmap (.bmp) file
<a href="#">save_image</a>	- save a rectangular region from a pixel-graphics screen
<a href="#">display_image</a>	- display an image on the pixel-graphics screen



## 2.13 Machine Level Interface

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

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

## 2.14 Dynamic Calls

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

- [routine\\_id](#) - get a unique identifying number for a Euphoria routine
  - [call\\_proc](#) - call a Euphoria procedure using a routine id
  - [call\\_func](#) - call a Euphoria function using a routine id
-

## 2.15 Calling C Functions (WIN32 and Linux)

See [platform.doc](#) for a description of WIN32 and Linux programming in Euphoria.

<a href="#">open_dll</a>	- open a Windows dynamic link library (.dll file) or Linux shared library (.so file)
<a href="#">define_c_proc</a>	- define a C function that is VOID (no value returned), or whose value your program will ignore
<a href="#">define_c_func</a>	- define a C function that returns a value that your program will use
<a href="#">define_c_var</a>	- get the memory address of a C variable.
<a href="#">c_proc</a>	- call a C function, ignoring any return value
<a href="#">c_func</a>	- call a C function and get the return value
<a href="#">call_back</a>	- get a 32-bit machine address for a Euphoria routine for use as a call-back address
<a href="#">message_box</a>	- pop up a small window to get a Yes/No/Cancel response from the user
<a href="#">free_console</a>	- delete the console text window
<a href="#">instance</a>	- get the instance handle for the current program

---

## 2.16 Multitasking

This collection of routines lets you create multiple, independent tasks. Each task has its own current statement being executed, its own subroutine call stack, and its own set of private variables. The local and global variables of a program are shared amongst all tasks. When a task calls `task_yield()`, it is suspended, and the Euphoria scheduler decides which task to execute next.

The Language War demo program, `lw.ex`, makes heavy use of tasks to create a simulated battle involving numerous independently moving ships, torpedos, phasors etc. See also the `taskwire.exw` Windows demo program, and the `news.exu` demo for Linux and FreeBSD.

<a href="#"><u>task_clock_start</u></a>	- restart the scheduler's clock
<a href="#"><u>task_clock_stop</u></a>	- stop the scheduler's clock
<a href="#"><u>task_create</u></a>	- create a new task
<a href="#"><u>task_list</u></a>	- get a list of all tasks
<a href="#"><u>task_schedule</u></a>	- schedule a task for execution
<a href="#"><u>task_self</u></a>	- return the task id of the current task
<a href="#"><u>task_status</u></a>	- the current status (active, suspended, killed) of a task
<a href="#"><u>task_suspend</u></a>	- Suspend a task.
<a href="#"><u>task_yield</u></a>	- Yield control, so the scheduler can pick a new task to run.

... continue [3. Alphabetical Listing of All Routines, From A to B](#)



### 3. Alphabetical Listing of all Routines

#### ?

**Syntax:** ? x

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

**Example:**

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

**See Also:** [pretty\\_print](#), [print](#)

#### abort

**Syntax:** abort(i)

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

**Comments:** abort() is useful when a program is many levels deep in subroutine calls, and execution must end immediately, perhaps due to a severe error that has been detected.

If you don't use abort(), **ex.exe/exw.exe/exu** will normally return an exit status code of 0. If your program fails with a Euphoria-detected compile-time or run-time error then a code of 1 is returned.

**Example:**

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

**See Also:** [crash\\_message](#), [system\\_exec](#)

#### all\_palette

**Platform:** DOS32

**Syntax:** include graphics.e  
all\_palette(s)

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

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

Each element specifies a new color intensity {red, green, blue} for the corresponding color number, starting

with color number 0. The values for red, green and blue must be in the range 0 to 63.

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

**Example Program:** [demo\dos32\bitmap.ex](#)

**See Also:** [get\\_all\\_palette](#), [palette](#), [read\\_bitmap](#), [video\\_config](#), [graphics\\_mode](#)

## allocate

**Syntax:** `include machine.e`

`a = allocate(i)`

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

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

**Example:**

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

**See Also:** [free](#), [allocate\\_low](#), [peek](#), [poke](#), [mem\\_set](#), [call](#)

## allocate\_low

**Platform:** **DOS32**

**Syntax:** `include machine.e`

`i2 = allocate_low(i1)`

**Description:** Allocate `i1` contiguous bytes of low memory, i.e. conventional memory (address below 1 megabyte). Return the address of the block of memory, or return 0 if the memory can't be allocated.

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

**Example Program:** [demo\dos32\dosint.ex](#)

**See Also:** [dos\\_interrupt](#), [free\\_low](#), [allocate](#), [peek](#), [poke](#)

## allocate\_string

**Syntax:** `include machine.e`

`a = allocate_string(s)`

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

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

**Example:**

```
atom title
```

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

**Example Program:** [demo\win32\window.exw](#)

**See Also:** [allocate](#), [free](#)

## allow\_break

**Syntax:** include file.e

`allow_break(i)`

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

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

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

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

**Example:**

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

**See Also:** [check\\_break](#)

## and\_bits

**Syntax:** `x3 = and_bits(x1, x2)`

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

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

To understand the binary representation of a number you should display it in hexadecimal notation. Use the %x format of [printf\(\)](#).

**Example 1:**

```
a = and_bits(#0F0F0000, #12345678)
-- a is #02040000
```

**Example 2:**

```

a = and_bits(#FF, {#123456, #876543, #2211})
-- a is {#56, #43, #11}

```

### Example 3:

```

a = and_bits(#FFFFFFFF, #FFFFFFFF)
-- a is -1
-- Note that #FFFFFFFF is a positive number,
-- but the result of a bitwise logical operation is interpreted
-- as a signed 32-bit number, so it's negative.

```

**See Also:**    [or\\_bits](#), [xor\\_bits](#), [not\\_bits](#), [int\\_to\\_bits](#)

## append

**Syntax:**        `s2 = append(s1, x)`

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

**Comments:**    If `x` is an atom this is equivalent to `s2 = s1 & x`. If `x` is a sequence it is not equivalent.

The extra storage is allocated automatically and very efficiently with Euphoria's dynamic storage allocation. The case where `s1` and `s2` are actually the same variable (as in Example 1 below) is highly optimized.

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

sequence x

```

x = {}
for i = 1 to 10 do
    x = append(x, i)
end for
-- x is now {1,2,3,4,5,6,7,8,9,10}

```

**Example 2:**    Any kind of Euphoria object can be appended to a sequence, e.g.

sequence x, y, z

```

x = {"fred", "barney"}
y = append(x, "wilma")
-- y is now {"fred", "barney", "wilma"}

z = append(append(y, "betty"), {"bam", "bam"})
-- z is now {"fred", "barney", "wilma", "betty", {"bam", "bam"}}

```

**See Also:**    [prepend](#), [concatenation operator &](#), [sequence-formation operator](#)

## arccos

**Syntax:**        `include misc.e`

`x2 = arccos(x1)`

**Description:** Return an angle with cosine equal to `x1`.

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

A value between 0 and [PI](#) radians will be returned.

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

`arccos()` is not as fast as `arctan()`.

**Example:**

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

**See Also:** [cos](#), [arcsin](#), [arctan](#)

## arcsin

**Syntax:** `include misc.e`

`x2 = arcsin(x1)`

**Description:** Return an angle with sine equal to `x1`.

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

A value between  $-\pi/2$  and  $+\pi/2$  (radians) will be returned.

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

`arcsin()` is not as fast as `arctan()`.

**Example:**

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

**See Also:** [sin](#), [arccos](#), [arctan](#)

## arctan

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

**Description:** Return an angle with tangent equal to `x1`.

**Comments:** A value between  $-\pi/2$  and  $\pi/2$  (radians) will be returned.

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

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

**Example:**

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

**See Also:** [tan](#), [arcsin](#), [arccos](#)

## atom

**Syntax:** `i = atom(x)`

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

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

if an object is an atom.

**Example 1:**

```
atom a
a = 5.99
```

**Example 2:**

```
object line

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

**See Also:** [sequence](#), [object](#), [integer](#), [atoms and sequences](#)

## atom\_to\_float32

**Syntax:** include machine.e

s = atom\_to\_float32(a1)

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

**Comments:** Euphoria atoms can have values which are 64-bit IEEE floating-point numbers, so you may lose precision when you convert to 32-bits (16 significant digits versus 7). The range of exponents is much larger in 64-bit format (10 to the 308, versus 10 to the 38), so some atoms may be too large or too small to represent in 32-bit format. In this case you will get one of the special 32-bit values: [inf or -inf \(infinity or -infinity\)](#). To avoid this, you can use atom\_to\_float64().

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

**Example:**

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

**See Also:** [atom\\_to\\_float64](#), [float32\\_to\\_atom](#)

## atom\_to\_float64

**Syntax:** include machine.e

s = atom\_to\_float64(a1)

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

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

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

**Example:**

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

**See Also:** [atom\\_to\\_float32](#), [float64\\_to\\_atom](#)

## bits\_to\_int

**Syntax:**       include machine.e

a = bits\_to\_int(s)

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

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

**Example:**

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

**See Also:**   [int\\_to\\_bits](#), [operations on sequences](#)

## bk\_color

**Syntax:**       include graphics.e

bk\_color(i)

**Description:** Set the background color to one of the 16 standard colors. In **pixel-graphics modes** the whole screen is affected immediately. In **text modes** any new characters that you print will have the new background color. In some text modes there might only be 8 distinct background colors available.

**Comments:** The 16 standard colors are defined as constants in [graphics.e](#)

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

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

In **text modes**, to restore the original background color when your program finishes, e.g. 0 - BLACK, you must call bk\_color(0). If the cursor is at the bottom line of the screen, you may have to actually print something before terminating your program. Printing '\n' may be enough.

**Example:**

```
bk_color(BLACK)
```

**See Also:**   [text\\_color](#), [palette](#)

## bytes\_to\_int

**Syntax:**       include machine.e

a = bytes\_to\_int(s)

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

**Comments:** The result could be greater than the integer type allows, so you should assign it to an **atom**. s would normally contain positive values that have been read using peek() from 4 consecutive memory

locations.

**Example:**

```
atom int32
```

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

**See Also:**    [int\\_to\\_bytes](#), [bits\\_to\\_int](#), [peek](#), [peek4s](#), [peek4u](#), [poke](#)

... continue

from A to B | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)



### 3. Alphabetical Listing of all Routines

#### ?

**Syntax:** ? x

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

**Example:**

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

**See Also:** [pretty\\_print](#), [print](#)

#### abort

**Syntax:** abort(i)

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

**Comments:** abort() is useful when a program is many levels deep in subroutine calls, and execution must end immediately, perhaps due to a severe error that has been detected.

If you don't use abort(), **ex.exe/exw.exe/exu** will normally return an exit status code of 0. If your program fails with a Euphoria-detected compile-time or run-time error then a code of 1 is returned.

**Example:**

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

**See Also:** [crash\\_message](#), [system\\_exec](#)

#### all\_palette

**Platform:** **DOS32**

**Syntax:** include graphics.e  
all\_palette(s)

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

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

Each element specifies a new color intensity {red, green, blue} for the corresponding color number, starting

with color number 0. The values for red, green and blue must be in the range 0 to 63.

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

**Example Program:** [demo\dos32\bitmap.ex](#)

**See Also:** [get\\_all\\_palette](#), [palette](#), [read\\_bitmap](#), [video\\_config](#), [graphics\\_mode](#)

## allocate

**Syntax:** `include machine.e`

`a = allocate(i)`

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

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

**Example:**

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

**See Also:** [free](#), [allocate\\_low](#), [peek](#), [poke](#), [mem\\_set](#), [call](#)

## allocate\_low

**Platform:** **DOS32**

**Syntax:** `include machine.e`

`i2 = allocate_low(i1)`

**Description:** Allocate `i1` contiguous bytes of low memory, i.e. conventional memory (address below 1 megabyte). Return the address of the block of memory, or return 0 if the memory can't be allocated.

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

**Example Program:** [demo\dos32\dosint.ex](#)

**See Also:** [dos\\_interrupt](#), [free\\_low](#), [allocate](#), [peek](#), [poke](#)

## allocate\_string

**Syntax:** `include machine.e`

`a = allocate_string(s)`

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

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

**Example:**

```
atom title
```

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

**Example Program:** [demo\win32\window.exw](#)

**See Also:** [allocate](#), [free](#)

## allow\_break

**Syntax:** include file.e

`allow_break(i)`

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

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

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

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

**Example:**

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

**See Also:** [check\\_break](#)

## and\_bits

**Syntax:** `x3 = and_bits(x1, x2)`

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

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

To understand the binary representation of a number you should display it in hexadecimal notation. Use the %x format of [printf\(\)](#).

**Example 1:**

```
a = and_bits(#0F0F0000, #12345678)
-- a is #02040000
```

**Example 2:**

```
a = and_bits(#FF, {#123456, #876543, #2211})
-- a is {#56, #43, #11}
```

### Example 3:

```
a = and_bits(#FFFFFFFF, #FFFFFFFF)
-- a is -1
-- Note that #FFFFFFFF is a positive number,
-- but the result of a bitwise logical operation is interpreted
-- as a signed 32-bit number, so it's negative.
```

**See Also:**    [or\\_bits](#), [xor\\_bits](#), [not\\_bits](#), [int\\_to\\_bits](#)

## append

**Syntax:**        `s2 = append(s1, x)`

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

**Comments:**    If `x` is an atom this is equivalent to `s2 = s1 & x`. If `x` is a sequence it is not equivalent.

The extra storage is allocated automatically and very efficiently with Euphoria's dynamic storage allocation. The case where `s1` and `s2` are actually the same variable (as in Example 1 below) is highly optimized.

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

sequence x

```
x = {}
for i = 1 to 10 do
  x = append(x, i)
end for
-- x is now {1,2,3,4,5,6,7,8,9,10}
```

**Example 2:**    Any kind of Euphoria object can be appended to a sequence, e.g.

sequence x, y, z

```
x = {"fred", "barney"}
y = append(x, "wilma")
-- y is now {"fred", "barney", "wilma"}

z = append(append(y, "betty"), {"bam", "bam"})
-- z is now {"fred", "barney", "wilma", "betty", {"bam", "bam"}}
```

**See Also:**    [prepend](#), [concatenation operator &](#), [sequence-formation operator](#)

## arccos

**Syntax:**        `include misc.e`

`x2 = arccos(x1)`

**Description:** Return an angle with cosine equal to `x1`.

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

A value between 0 and [PI](#) radians will be returned.

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

`arccos()` is not as fast as `arctan()`.

**Example:**

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

**See Also:** [cos](#), [arcsin](#), [arctan](#)

## arcsin

**Syntax:** `include misc.e`

`x2 = arcsin(x1)`

**Description:** Return an angle with sine equal to `x1`.

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

A value between  $-\pi/2$  and  $+\pi/2$  (radians) will be returned.

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

`arcsin()` is not as fast as `arctan()`.

**Example:**

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

**See Also:** [sin](#), [arccos](#), [arctan](#)

## arctan

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

**Description:** Return an angle with tangent equal to `x1`.

**Comments:** A value between  $-\pi/2$  and  $\pi/2$  (radians) will be returned.

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

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

**Example:**

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

**See Also:** [tan](#), [arcsin](#), [arccos](#)

## atom

**Syntax:** `i = atom(x)`

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

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

if an object is an atom.

**Example 1:**

```
atom a
a = 5.99
```

**Example 2:**

```
object line

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

**See Also:** [sequence](#), [object](#), [integer](#), [atoms and sequences](#)

## atom\_to\_float32

**Syntax:** include machine.e

s = atom\_to\_float32(a1)

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

**Comments:** Euphoria atoms can have values which are 64-bit IEEE floating-point numbers, so you may lose precision when you convert to 32-bits (16 significant digits versus 7). The range of exponents is much larger in 64-bit format (10 to the 308, versus 10 to the 38), so some atoms may be too large or too small to represent in 32-bit format. In this case you will get one of the special 32-bit values: [inf or -inf \(infinity or -infinity\)](#). To avoid this, you can use atom\_to\_float64().

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

**Example:**

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

**See Also:** [atom\\_to\\_float64](#), [float32\\_to\\_atom](#)

## atom\_to\_float64

**Syntax:** include machine.e

s = atom\_to\_float64(a1)

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

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

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

**Example:**

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

**See Also:** [atom\\_to\\_float32](#), [float64\\_to\\_atom](#)

## bits\_to\_int

**Syntax:**       include machine.e

a = bits\_to\_int(s)

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

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

**Example:**

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

**See Also:**   [int\\_to\\_bits](#), [operations on sequences](#)

## bk\_color

**Syntax:**       include graphics.e

bk\_color(i)

**Description:** Set the background color to one of the 16 standard colors. In **pixel-graphics modes** the whole screen is affected immediately. In **text modes** any new characters that you print will have the new background color. In some text modes there might only be 8 distinct background colors available.

**Comments:** The 16 standard colors are defined as constants in [graphics.e](#)

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

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

In **text modes**, to restore the original background color when your program finishes, e.g. 0 - BLACK, you must call bk\_color(0). If the cursor is at the bottom line of the screen, you may have to actually print something before terminating your program. Printing '\n' may be enough.

**Example:**

```
bk_color(BLACK)
```

**See Also:**   [text\\_color](#), [palette](#)

## bytes\_to\_int

**Syntax:**       include machine.e

a = bytes\_to\_int(s)

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

**Comments:** The result could be greater than the integer type allows, so you should assign it to an **atom**. s would normally contain positive values that have been read using peek() from 4 consecutive memory

locations.

**Example:**

```
atom int32
```

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

**See Also:**    [int\\_to\\_bytes](#), [bits\\_to\\_int](#), [peek](#), [peek4s](#), [peek4u](#), [poke](#)

... continue

from A to B | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)



## ?

**Syntax:** ? x

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

**Example:**

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

**See Also:** [pretty\\_print](#), [print](#)

## abort

**Syntax:**        abort(i)

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

**Comments:**     abort() is useful when a program is many levels deep in subroutine calls, and execution must end immediately, perhaps due to a severe error that has been detected.

If you don't use abort(), **ex.exe/exw.exe/exu** will normally return an exit status code of 0. If your program fails with a Euphoria-detected compile-time or run-time error then a code of 1 is returned.

**Example:**

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

**See Also:**     [crash\\_message](#), [system\\_exec](#)

## all\_palette

**Platform:** DOS32

**Syntax:** include graphics.e

all\_palette(s)

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

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

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

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

**Example Program:** [demo\dos32\bitmap.ex](#)

**See Also:** [get\\_all\\_palette](#), [palette](#), [read\\_bitmap](#), [video\\_config](#), [graphics\\_mode](#)

## allocate

**Syntax:**           include machine.e

a = allocate(i)

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

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

**Example:**

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

**See Also:**       [free](#), [allocate\\_low](#), [peek](#), [poke](#), [mem\\_set](#), [call](#)

## allocate\_low

**Platform:** **DOS32**

**Syntax:** include machine.e

i2 = allocate\_low(i1)

**Description:** Allocate i1 contiguous bytes of low memory, i.e. conventional memory (address below 1 megabyte). Return the address of the block of memory, or return 0 if the memory can't be allocated.

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

**Example Program:** [demo\dos32\dosint.ex](#)

**See Also:** [dos\\_interrupt](#), [free\\_low](#), [allocate](#), [peek](#), [poke](#)

## allocate\_string

**Syntax:**           include machine.e

a = allocate\_string(s)

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

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

**Example:**

```
atom title
```

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

**Example Program:**   demo\win32\window.exw

**See Also:**       [allocate](#), [free](#)

## allow\_break

**Syntax:** include file.e

allow\_break(i)

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

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

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

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

**Example:**

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

**See Also:** [check\\_break](#)

## and\_bits

**Syntax:** x3 = and\_bits(x1, x2)

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

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

To understand the binary representation of a number you should display it in hexadecimal notation. Use the %x format of [printf\(\)](#).

### Example 1:

```
a = and_bits(#0F0F0000, #12345678)
-- a is #02040000
```

### Example 2:

```
a = and_bits(#FF, {#123456, #876543, #2211})
-- a is {#56, #43, #11}
```

### Example 3:

```
a = and_bits(#FFFFFFFF, #FFFFFFFF)
-- a is -1
-- Note that #FFFFFFFF is a positive number,
-- but the result of a bitwise logical operation is interpreted
-- as a signed 32-bit number, so it's negative.
```

**See Also:** [or\\_bits](#), [xor\\_bits](#), [not\\_bits](#), [int\\_to\\_bits](#)



## append

**Syntax:** `s2 = append(s1, x)`

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

**Comments:** If `x` is an atom this is equivalent to `s2 = s1 & x`. If `x` is a sequence it is not equivalent.

The extra storage is allocated automatically and very efficiently with Euphoria's dynamic storage allocation. The case where `s1` and `s2` are actually the same variable (as in Example 1 below) is highly optimized.

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

`sequence x`

```
x = {}
for i = 1 to 10 do
    x = append(x, i)
end for
-- x is now {1,2,3,4,5,6,7,8,9,10}
```

**Example 2:** Any kind of Euphoria object can be appended to a sequence, e.g.

`sequence x, y, z`

```
x = {"fred", "barney"}
y = append(x, "wilma")
-- y is now {"fred", "barney", "wilma"}

z = append(append(y, "betty"), {"bam", "bam"})
-- z is now {"fred", "barney", "wilma", "betty", {"bam", "bam"}}
```

**See Also:** [prepend](#), [concatenation operator &](#), [sequence-formation operator](#)

## arccos

**Syntax:** include misc.e

```
x2 = arccos(x1)
```

**Description:** Return an angle with cosine equal to x1.

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

A value between 0 and [PI](#) radians will be returned.

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

arccos() is not as fast as arctan().

**Example:**

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

**See Also:** [cos](#), [arcsin](#), [arctan](#)

## arcsin

**Syntax:** include misc.e

`x2 = arcsin(x1)`

**Description:** Return an angle with sine equal to x1.

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

A value between  $-\pi/2$  and  $+\pi/2$  (radians) will be returned.

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

`arcsin()` is not as fast as `arctan()`.

**Example:**

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

**See Also:** [sin](#), [arccos](#), [arctan](#)

## arctan

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

**Description:** Return an angle with tangent equal to x1.

**Comments:** A value between  $-\pi/2$  and  $\pi/2$  (radians) will be returned.

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

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

**Example:**

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

**See Also:** [tan](#), [arcsin](#), [arccos](#)

## atom

**Syntax:** `i = atom(x)`

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

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

**Example 1:**

```
    atom a
a = 5.99
```

**Example 2:**

```
    object line

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

**See Also:** [sequence](#), [object](#), [integer](#), [atoms and sequences](#)

## atom\_to\_float32

**Syntax:**           include machine.e

s = atom\_to\_float32(a1)

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

**Comments:**      Euphoria atoms can have values which are 64-bit IEEE floating-point numbers, so you may lose precision when you convert to 32-bits (16 significant digits versus 7). The range of exponents is much larger in 64-bit format (10 to the 308, versus 10 to the 38), so some atoms may be too large or too small to represent in 32-bit format. In this case you will get one of the special 32-bit values: [inf or -inf \(infinity or -infinity\)](#). To avoid this, you can use atom\_to\_float64().

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

**Example:**

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

**See Also:**       [atom\\_to\\_float64](#), [float32\\_to\\_atom](#)

## atom\_to\_float64

**Syntax:**           include machine.e

s = atom\_to\_float64(a1)

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

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

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

**Example:**

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

**See Also:**     [atom\\_to\\_float32](#), [float64\\_to\\_atom](#)

## bits\_to\_int

**Syntax:**           include machine.e

a = bits\_to\_int(s)

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

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

**Example:**

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

**See Also:**     [int\\_to\\_bits](#), [operations on sequences](#)



## bk\_color

**Syntax:** include graphics.e

bk\_color(i)

**Description:** Set the background color to one of the 16 standard colors. In **pixel-graphics modes** the whole screen is affected immediately. In **text modes** any new characters that you print will have the new background color. In some text modes there might only be 8 distinct background colors available.

**Comments:** The 16 standard colors are defined as constants in [graphics.e](#)

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

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

In **text modes**, to restore the original background color when your program finishes, e.g. 0 - BLACK, you must call bk\_color(0). If the cursor is at the bottom line of the screen, you may have to actually print something before terminating your program. Printing '\n' may be enough.

**Example:**

```
bk_color(BLACK)
```

**See Also:** [text\\_color](#), [palette](#)

## bytes\_to\_int

**Syntax:** `include machine.e`

`a = bytes_to_int(s)`

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

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

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

**Example:**

```
atom int32
```

```
int32 = bytes_to_int({37,1,0,0})
```

```
-- int32 is 37 + 256*1 = 293
```

**See Also:** [int\\_to\\_bytes](#), [bits\\_to\\_int](#), [peek](#), [peek4s](#), [peek4u](#), [poke](#)

... continue

from A to B | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)

## call

**Syntax:**        call(a)

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

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

**Example Program:**   [demo\callmach.ex](#)

**See Also:**        [allocate](#), [free](#), [peek](#), [poke](#), [poke4](#), [c\\_proc](#), [define\\_c\\_proc](#)

## call\_back

**Platform:**        WIN32, Linux, FreeBSD

**Syntax:**        include dll.e

a = call\_back(i)

or

a = call\_back({i1, i})

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

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

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

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

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

**Example Program:**   [demo\win32\window.exw](#), [demo\linux\qsort.exu](#)

**See Also:**        [routine\\_id](#), [platform.doc](#)

## c\_func

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** a = c\_func(i, s)

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

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

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

The C function could be part of a .dll created by the Euphoria To C Translator.

**Example:**

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

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

-- the C function BeginPaint takes a C int argument and
-- a C pointer, and returns a C int as a result:
BeginPaint = define_c_func(user32, "BeginPaint",
                           {C_INT, C_POINTER}, C_INT)

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

**See Also:** [c\\_proc](#), [define\\_c\\_func](#), [open\\_dll](#), [platform.doc](#)

## c\_proc

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** c\_proc(i, s)

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

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

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

The C function could be part of a .dll created by the Euphoria To C Translator.

**Example:**

```
atom user32, hwnd, rect
integer GetClientRect

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

-- GetClientRect is a VOID C function that takes a C int
-- and a C pointer as its arguments:
```

```
GetClientRect = define_c_proc(user32, "GetClientRect",
                               {C_INT, C_POINTER})
```

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

**See Also:** [c\\_func](#), [call](#), [define\\_c\\_proc](#), [open\\_dll](#), [platform.doc](#)

## call\_func

**Syntax:** `x = call_func(i, s)`

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

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

**Example Program:** [demo\csort.ex](#)

**See Also:** [call\\_proc](#), [routine\\_id](#)

## call\_proc

**Syntax:** `call_proc(i, s)`

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

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

**Example:**

```
global integer foo_id

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

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

foo_id = routine_id("foo")

x()
```

**See Also:** [call\\_func](#), [routine\\_id](#)

## chdir

**Syntax:** `include file.e`

`i = chdir(s)`

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

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

name.

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

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

**Example:**

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

**See Also:** [current\\_dir](#)

## check\_break

**Syntax:** `include file.e`

`i = check_break()`

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

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

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

**Example:**

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

**See Also:** [allow\\_break](#), [get\\_key](#)

## clear\_screen

**Syntax:** `clear_screen()`

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

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

**See Also:** [bk\\_color](#), [graphics\\_mode](#)

## close

**Syntax:** `close(fn)`

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

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

**See Also:** [open](#), [flush](#)

## command\_line

**Syntax:** s = command\_line()

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

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

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

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

### Example 1:

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

```
cmd = command_line()
```

```
-- cmd will be:
```

```
{ "C:\EUPHORIA\BIN\EX.EXE",  
  "myprog",  
  "myfile.dat",  
  "12345",  
  "the end" }
```

### Example 2:

```
-- Your program is bound with the name "myprog.exe"
```

```
-- and is stored in the directory c:\myfiles
```

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

```
cmd = command_line()
```

```
-- cmd will be:
```

```
{ "C:\MYFILES\MYPROG.EXE",  
  "C:\MYFILES\MYPROG.EXE", -- place holder  
  "myfile.dat",  
  "12345",  
  "the end"  
}
```

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

See Also: [getenv](#)

## compare

**Syntax:** `i = compare(x1, x2)`

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

**Example 1:**

```
x = compare({1,2,{3,{4}}},5), {2-1,1+1,{3,{4}}},6-1})
-- identical, x is 0
```

**Example 2:**

```
if compare("ABC", "ABCD") < 0 then -- -1
-- will be true: ABC is "less" because it is shorter
end if
```

**Example 3:**

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

**Example 4:**

```
x = compare('a', "a")
-- x will be -1 because 'a' is an atom
-- while "a" is a sequence
```

See Also: [equal](#), [relational operators](#), [operations on sequences](#)

## cos

**Syntax:** `x2 = cos(x1)`

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

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

**Example:**

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

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

See Also: [sin](#), [tan](#), [log](#), [sqrt](#)

## crash\_file

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

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



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

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

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

**Example:**

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

**See Also:** [abort](#), [crash\\_message](#), [crash\\_routine](#), [debugging and profiling](#)

## crash\_message

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

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

**Comments:** Normally Euphoria prints a diagnostic message such as "subscript out of bounds", or "divide by zero" on the screen, as well as dumping debugging information into **ex.err**. Euphoria's error messages will not be meaningful for your users unless they happen to be Euphoria programmers. By calling `crash_message()` you can control the message that will appear on the screen. Debugging information will still be stored in **ex.err**. You won't lose any information by doing this.

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

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

**Example:**

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

**See Also:** [abort](#), [crash\\_file](#), [crash\\_routine](#), [debugging and profiling](#)

## crash\_routine

**Syntax:** `include machine.e`  
`crash_routine(i)`

**Description:** Pass the routine id of a function that you want Euphoria to call in the event that a run-time error is detected and your program must be shut down. Your function should take one argument of type object. The object that is passed to your function is currently always 0. In future releases of Euphoria, a more meaningful value may be passed. You can call `crash_routine` many times with many different routine id's. When a crash occurs, Euphoria will call your crash routines, the most recently specified first, working

back to the first one specified. Normally each routine should return 0. If any routine returns a non-zero value, the chain of calls will terminate immediately.

**Comments:** By specifying a crash routine, you give your program a chance to handle fatal run-time errors, such as subscript out of bounds, in a more graceful way. You might save some critical data to disk. You might inform the user about what has happened, and what he can do about it. You might also save some key debugging information. In fact, when your crash routine is called, `ex.err` will have already been written. Your crash routine could save `ex.err` somewhere, or even open it and extract information from it, such as the error message.

`crash_routine` can be used with the Interpreter or the Translator. Translated code does not check for as many run-time errors, and does not provide a full `ex.err` dump, but machine-level exceptions are caught, and a crash routine will give you an excellent opportunity to save some variable values to disk for debugging.

The developer of a library might want to specify a crash routine for his library. It could tidy things up by unlocking and closing files, releasing resources etc. The developer of the main program could have his own crash routine. Both routines would be called by Euphoria, unless the first one called (the last one specified) returned non-zero.

A crash routine can't resume execution at the point of the crash, but there is no limitation on what else it can do. It doesn't have to return. It could even reinitialize global variables and effectively restart the program.

If another error occurs while a crash routine is running, a new error dump will occur, but the file name this time will be `ex_crash.err`, rather than `ex.err`. At this point no more calls to crash routines will be allowed. You will have to look at both `ex.err` and `ex_crash.err` to fully understand what took place.

**Example:**

```
function crash(object x)
-- in case of fire ...

-- (on Linux) send an e-mail containing ex.err
system("mail -s \"crash!\" myname@xxx.com < ex.err > /dev/null", 2)

return 0
end function

crash_routine(routine_id("crash"))
```

**See Also:** [abort](#), [crash\\_file](#), [crash\\_message](#), [debugging and profiling](#)

## current\_dir

**Syntax:** include file.e

`s = current_dir()`

**Description:** Return the name of the current working directory.

**Comments:** There will be no slash or backslash on the end of the current directory, except under DOS/Windows, at the top-level of a drive, e.g. C:\

**Example:**

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

See Also: [dir](#), [chdir](#), [getenv](#)

## cursor

**Platform:** WIN32, DOS32

**Syntax:** include graphics.e

cursor(i)

**Description:** Select a style of cursor. [graphics.e](#) contains:

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

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

**Comments:** In [pixel-graphics modes](#) no cursor is displayed.

**Example:**

```
cursor(BLOCK_CURSOR)
```

See Also: [graphics\\_mode](#), [text\\_rows](#)

## custom\_sort

**Syntax:** include sort.e

s2 = custom\_sort(i, s1)

**Description:** Sort the elements of sequence s1, using a compare function with [routine id](#) i.

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

**Example Program:** [demo\csort.ex](#)

See Also: [sort](#), [compare](#), [routine\\_id](#)

## date

**Syntax:** s = date()

**Description:** Return a sequence with the following information:

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

**Example:**

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

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

**See Also:** [time](#)

## define\_c\_func

**Syntax:** include dll.e

i1 = define\_c\_func(x1, x2, s1, i2)

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

When defining a C function, x1 is the address of the library containing the C function, while x2 is the name of the C function. x1 is a value returned by open\_dll(). If the C function can't be found, -1 will be returned as the routine id. On Windows, you can add a '+' character as a prefix to the function name. This indicates to Euphoria that the function uses the **cdecl** calling convention. By default, Euphoria assumes that C routines accept the **stdcall** convention.

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

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

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

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

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

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

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

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.

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

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

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

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

#### Example:

```
atom user32
integer LoadIcon

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

-- It takes a C pointer and a C int as parameters.
-- It returns a C int as a result.
LoadIcon = define_c_func(user32, "LoadIconA",
                        {C_POINTER, C_INT}, C_INT)
-- We use "LoadIconA" here because we know that LoadIconA
-- needs the stdcall convention, as do
-- all standard .dll routines in the WIN32 API.
-- To specify the cdecl convention, we would have used "+LoadIconA".

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

**See Also:** `euphoria\demo\callmach.ex`, [c\\_func](#), [define\\_c\\_proc](#), [c\\_proc](#), [open\\_dll](#), [platform.doc](#)

## define\_c\_proc

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

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

When defining a C function, x1 is the address of the library containing the C function, while x2 is the name of the C function. x1 is a value returned by `open_dll()`. If the C function can't be found, -1 will be returned as the routine id. On Windows, you can add a '+' character as a prefix to the function name. This tells Euphoria that the function uses the **cdecl** calling convention. By default, Euphoria assumes that C routines accept the **stdcall** convention.

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

s1 is a list of the parameter types for the function. A list of C types is contained in [dll.e](#), and [shown above](#). These can be used to define machine code parameters as well.

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

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

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

Currently, there is no way to pass a C structure by value. You can only pass a pointer to a structure.

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

#### Example:

```
atom user32
integer ShowWindow

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

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

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

**See Also:** [c\\_proc](#), [define\\_c\\_func](#), [c\\_func](#), [open\\_dll](#), [platform.doc](#)

## define\_c\_var

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

a1 = define\_c\_var(a2, s)

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

**Comments:** Once you have the address of a C variable, and you know its type, you can use peek() and poke() to read or write the value of the variable.

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

**See Also:** [c\\_proc](#), [define\\_c\\_func](#), [c\\_func](#), [open\\_dll](#), [platform.doc](#)

## dir

**Syntax:** include file.e

x = dir(st)

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

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

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

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

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

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

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

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

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

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

**Comments:** The top level directory, e.g. c:\ does not have "." or ".." entries.

This function is often used just to test if a file or directory exists.

Under **WIN32**, st can have a long file or directory name anywhere in the path.

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

**DOS32:** The file name returned in D\_NAME will be a standard DOS 8.3 name. (See [Archive Web page](#) for a better solution).

**WIN32:** The file name returned in D\_NAME will be a long file name.

**Example:**

```
d = dir(current_dir())
```

-- d might have:

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

d[3][D\_NAME] would be "fred"

**Example Program:** [bin\search.ex](#)

**See Also:** [wildcard\\_file](#), [current\\_dir](#), [open](#)

## display\_image

**Platform:** **DOS32**

**Syntax:** include image.e

display\_image(s1, s2)

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

**Comments:** s2 might be the result of a previous call to save\_image(), or read\_bitmap(), or it could be something you have created.

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

**Example:**

```
display_image({20,30}, {{7,5,9,4,8},
                        {2,4,1,2},
                        {1,0,1,0,4,6,1},
                        {5,5,5,5,5,5}})
```

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

**Example Program:** [demo\dos32\bitmap.ex](#)

**See Also:** [save\\_image](#), [read\\_bitmap](#), [display\\_text\\_image](#)



## display\_text\_image

**Syntax:** include image.e

display\_text\_image(s1, s2)

**Description:** Display the 2-d sequence of characters and attributes contained in s2 at line s1[1], column s1[2]. s2 is a sequence of sequences, where each sequence is a string of characters and attributes to be displayed. The top-left character is displayed at s1. Other characters appear to the right or below this position. The attributes indicate the foreground and background color of the preceding character. On DOS32, the attribute should consist of the foreground color plus 16 times the background color.

**Comments:** s2 would normally be the result of a previous call to save\_text\_image(), although you could construct it yourself.

This routine only works in **text modes**.

You might use save\_text\_image()/display\_text\_image() in a text-mode graphical user interface, to allow "pop-up" dialog boxes, and drop-down menus to appear and disappear without losing what was previously on the screen.

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

**Example:**

```
clear_screen()
display_text_image({1,1}, {{ 'A', WHITE, 'B', GREEN},
                           { 'C', RED+16*WHITE},
                           { 'D', BLUE}})

-- displays:
  AB
  C
  D

-- at the top left corner of the screen.
-- 'A' will be white with black (0) background color,
-- 'B' will be green on black,
-- 'C' will be red on white, and
-- 'D' will be blue on black.
```

**See Also:** [save\\_text\\_image](#), [display\\_image](#), [put\\_screen\\_char](#)

## dos\_interrupt

**Platform:** DOS32

**Syntax:** include machine.e

s2 = dos\_interrupt(i, s1)

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

```
global constant REG_DI = 1,
               REG_SI = 2,
               REG_BP = 3,
               REG_BX = 4,
               REG_DX = 5,
               REG_CX = 6,
```

```

REG_AX = 7,
REG_FLAGS = 8,
REG_ES = 9,
REG_DS = 10

```

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

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

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

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

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

#### Example:

```

sequence registers

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

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

```

**Example Program:** [demo\dos32\dosint.ex](#)

**See Also:** [allocate\\_low](#), [free\\_low](#)

## draw\_line

**Platform:** **DOS32**

**Syntax:** `include graphics.e`

`draw_line(i, s)`

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

#### Example:

```

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

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

```

**See Also:** [polygon](#), [ellipse](#), [pixel](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)

## call

**Syntax:**           call(a)

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

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

**Example Program:**   [demo\callmach.ex](#)

**See Also:**         [allocate](#), [free](#), [peek](#), [poke](#), [poke4](#), [c\\_proc](#), [define\\_c\\_proc](#)

## call\_back

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

a = call\_back(i)

or

a = call\_back({i1, i})

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

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

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

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

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

**Example Program:** [demo\win32\window.exw](#), [demo\linux\qsort.exu](#)

**See Also:** [routine\\_id](#), [platform.doc](#)

## c\_func

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** a = c\_func(i, s)

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

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

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

The C function could be part of a .dll created by the Euphoria To C Translator.

**Example:**

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

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

-- the C function BeginPaint takes a C int argument and
-- a C pointer, and returns a C int as a result:
BeginPaint = define_c_func(user32, "BeginPaint",
                           {C_INT, C_POINTER}, C_INT)

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

**See Also:** [c\\_proc](#), [define\\_c\\_func](#), [open\\_dll](#), [platform.doc](#)

## c\_proc

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** c\_proc(i, s)

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

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

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

The C function could be part of a .dll created by the Euphoria To C Translator.

**Example:**

```
atom user32, hwnd, rect
integer GetClientRect

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

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

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

**See Also:** [c\\_func](#), [call](#), [define\\_c\\_proc](#), [open\\_dll](#), [platform.doc](#)

## call\_func

**Syntax:**        `x = call_func(i, s)`

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

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

**Example Program:**    [demo\csort.ex](#)

**See Also:**        [call\\_proc](#), [routine\\_id](#)



## call\_proc

**Syntax:** `call_proc(i, s)`

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

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

**Example:**

```
global integer foo_id

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

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

foo_id = routine_id("foo")

x()
```

**See Also:** [call\\_func](#), [routine\\_id](#)

## chdir

**Syntax:** include file.e

i = chdir(s)

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

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

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

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

**Example:**

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

**See Also:** [current\\_dir](#)

## check\_break

**Syntax:**           include file.e

i = check\_break()

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

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

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

**Example:**

```

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

**See Also:**       [allow\\_break](#), [get\\_key](#)

## clear\_screen

**Syntax:** `clear_screen()`  
**Description:** Clear the screen using the current background color (may be set by `bk_color()`).  
**Comments:** This works in all **text and pixel-graphics modes**.  
**See Also:** [bk\\_color](#), [graphics\\_mode](#)

## close

**Syntax:** `close(fn)`  
**Description:** Close a file or device and flush out any still-buffered characters.  
**Comments:** Any still-open files will be closed automatically when your program terminates.  
**See Also:** [open](#), [flush](#)

## command\_line

**Syntax:** s = command\_line()

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

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

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

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

### Example 1:

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

```
cmd = command_line()
```

```
-- cmd will be:
```

```
{ "C:\EUPHORIA\BIN\EX.EXE",  
  "myprog",  
  "myfile.dat",  
  "12345",  
  "the end" }
```

### Example 2:

```
-- Your program is bound with the name "myprog.exe"
```

```
-- and is stored in the directory c:\myfiles
```

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

```
cmd = command_line()
```

```
-- cmd will be:
```

```
{ "C:\MYFILES\MYPROG.EXE",  
  "C:\MYFILES\MYPROG.EXE", -- place holder  
  "myfile.dat",  
  "12345",  
  "the end"  
}
```

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

**See Also:** [getenv](#)

## compare

**Syntax:** `i = compare(x1, x2)`

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

**Example 1:**

```
x = compare({1,2,{3,{4}}},5), {2-1,1+1,{3,{4}}},6-1})
-- identical, x is 0
```

**Example 2:**

```
if compare("ABC", "ABCD") < 0 then -- -1
  -- will be true: ABC is "less" because it is shorter
end if
```

**Example 3:**

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

**Example 4:**

```
x = compare('a', "a")
-- x will be -1 because 'a' is an atom
-- while "a" is a sequence
```

**See Also:** [equal](#), [relational operators](#), [operations on sequences](#)

## COS

**Syntax:** `x2 = cos(x1)`

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

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

**Example:**

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

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

**See Also:** [sin](#), [tan](#), [log](#), [sqrt](#)



## crash\_file

**Syntax:** include machine.e

crash\_file(s)

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

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

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

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

**Example:**

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

**See Also:** [abort](#), [crash\\_message](#), [crash\\_routine](#), [debugging and profiling](#)

## crash\_message

**Syntax:**           include machine.e  
crash\_message(s)

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

**Comments:**      Normally Euphoria prints a diagnostic message such as "subscript out of bounds", or "divide by zero" on the screen, as well as dumping debugging information into **ex.err**. Euphoria's error messages will not be meaningful for your users unless they happen to be Euphoria programmers. By calling crash\_message() you can control the message that will appear on the screen. Debugging information will still be stored in **ex.err**. You won't lose any information by doing this.

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

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

**Example:**

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

**See Also:**       [abort](#), [crash\\_file](#), [crash\\_routine](#), [debugging and profiling](#)

## crash\_routine

**Syntax:**       include machine.e

crash\_routine(i)

**Description:** Pass the routine id of a function that you want Euphoria to call in the event that a run-time error is detected and your program must be shut down. Your function should take one argument of type object. The object that is passed to your function is currently always 0. In future releases of Euphoria, a more meaningful value may be passed. You can call crash\_routine many times with many different routine id's. When a crash occurs, Euphoria will call your crash routines, the most recently specified first, working back to the first one specified. Normally each routine should return 0. If any routine returns a non-zero value, the chain of calls will terminate immediately.

**Comments:** By specifying a crash routine, you give your program a chance to handle fatal run-time errors, such as subscript out of bounds, in a more graceful way. You might save some critical data to disk. You might inform the user about what has happened, and what he can do about it. You might also save some key debugging information. In fact, when your crash routine is called, ex.err will have already been written. Your crash routine could save ex.err somewhere, or even open it and extract information from it, such as the error message.

crash\_routine can be used with the Interpreter or the Translator. Translated code does not check for as many run-time errors, and does not provide a full ex.err dump, but machine-level exceptions are caught, and a crash routine will give you an excellent opportunity to save some variable values to disk for debugging.

The developer of a library might want to specify a crash routine for his library. It could tidy things up by unlocking and closing files, releasing resources etc. The developer of the main program could have his own crash routine. Both routines would be called by Euphoria, unless the first one called (the last one specified) returned non-zero.

A crash routine can't resume execution at the point of the crash, but there is no limitation on what else it can do. It doesn't have to return. It could even reinitialize global variables and effectively restart the program.

If another error occurs while a crash routine is running, a new error dump will occur, but the file name this time will be ex\_crash.err, rather than ex.err. At this point no more calls to crash routines will be allowed. You will have to look at both ex.err and ex\_crash.err to fully understand what took place.

**Example:**

```
function crash(object x)
-- in case of fire ...

-- (on Linux) send an e-mail containing ex.err
system("mail -s \"crash!\" myname@xxx.com < ex.err > /dev/null", 2)

return 0
end function

crash_routine(routine_id("crash"))
```

**See Also:**   [abort](#), [crash\\_file](#), [crash\\_message](#), [debugging and profiling](#)

## current\_dir

**Syntax:**           include file.e

s = current\_dir()

**Description:**   Return the name of the current working directory.

**Comments:**     There will be no slash or backslash on the end of the current directory, except under DOS/Windows, at the top-level of a drive, e.g. C:\

**Example:**

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

**See Also:**     [dir](#), [chdir](#), [getenv](#)

## cursor

**Platform:** WIN32, DOS32

**Syntax:** include graphics.e

cursor(i)

**Description:** Select a style of cursor. [graphics.e](#) contains:

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

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

**Comments:** In [pixel-graphics modes](#) no cursor is displayed.

**Example:**

```
cursor(BLOCK_CURSOR)
```

**See Also:** [graphics\\_mode](#), [text\\_rows](#)

## custom\_sort

**Syntax:**           include sort.e

s2 = custom\_sort(i, s1)

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

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

**Example Program:**   [demo\csort.ex](#)

**See Also:**       [sort](#), [compare](#), [routine\\_id](#)

## date

**Syntax:** s = date()

**Description:** Return a sequence with the following information:

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

**Example:**

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

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

**See Also:** [time](#)

## define\_c\_func

**Syntax:** include dll.e  
i1 = define\_c\_func(x1, x2, s1, i2)

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

When defining a C function, x1 is the address of the library containing the C function, while x2 is the name of the C function. x1 is a value returned by open\_dll(). If the C function can't be found, -1 will be returned as the routine id. On Windows, you can add a '+' character as a prefix to the function name. This indicates to Euphoria that the function uses the **cdecl** calling convention. By default, Euphoria assumes that C routines accept the **stdcall** convention.

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

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

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

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

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

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

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

If you are not interested in using the value returned by the C function, you should instead define it with



define\_c\_proc() and call it with c\_proc().

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

Passing floating-point values to a machine code routine will be faster if you use c\_func() rather than call() to call the routine, since you won't have to use atom\_to\_float64() and poke() to get the floating-point values into memory.

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

**Example:**

```
atom user32
integer LoadIcon

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

-- It takes a C pointer and a C int as parameters.
-- It returns a C int as a result.
LoadIcon = define_c_func(user32, "LoadIconA",
                        {C_POINTER, C_INT}, C_INT)

-- We use "LoadIconA" here because we know that LoadIconA
-- needs the stdcall convention, as do
-- all standard .dll routines in the WIN32 API.
-- To specify the cdecl convention, we would have used "+LoadIconA".

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

**See Also:**     euphoria\demo\callmach.ex, [c\\_func](#), [define\\_c\\_proc](#), [c\\_proc](#), [open\\_dll](#), [platform.doc](#)

## define\_c\_proc

**Syntax:**       include dll.e  
i1 = define\_c\_proc(x1, x2, s1)

**Description:**   Define the characteristics of either a C function, or a machine-code routine that you wish to call as a procedure from your Euphoria program. A small integer, known as a **routine id**, will be returned. Use this routine id as the first argument to c\_proc() when you wish to call the routine from Euphoria.

When defining a C function, x1 is the address of the library containing the C function, while x2 is the name of the C function. x1 is a value returned by open\_dll(). If the C function can't be found, -1 will be returned as the routine id. On Windows, you can add a '+' character as a prefix to the function name. This tells Euphoria that the function uses the **cdecl** calling convention. By default, Euphoria assumes that C routines accept the **stdcall** convention.

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

s1 is a list of the parameter types for the function. A list of C types is contained in [dll.e](#), and [shown above](#). These can be used to define machine code parameters as well.

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

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

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

Currently, there is no way to pass a C structure by value. You can only pass a pointer to a structure.

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

### Example:

```
atom user32
integer ShowWindow

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

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

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

**See Also:**    [c\\_proc](#), [define\\_c\\_func](#), [c\\_func](#), [open\\_dll](#), [platform.doc](#)



## define\_c\_var

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

a1 = define\_c\_var(a2, s)

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

**Comments:** Once you have the address of a C variable, and you know its type, you can use peek() and poke() to read or write the value of the variable.

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

**See Also:** [c\\_proc](#), [define\\_c\\_func](#), [c\\_func](#), [open\\_dll](#), [platform.doc](#)

## dir

**Syntax:** include file.e

x = dir(st)

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

## display\_image

**Platform:** **DOS32**

**Syntax:** include image.e

display\_image(s1, s2)

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

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

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

**Example:**

```
display_image({20,30}, {{7,5,9,4,8},
                        {2,4,1,2},
                        {1,0,1,0,4,6,1},
                        {5,5,5,5,5,5}})
```

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

**Example Program:** [demo\dos32\bitmap.ex](#)

**See Also:** [save\\_image](#), [read\\_bitmap](#), [display\\_text\\_image](#)

## display\_text\_image

**Syntax:** include image.e

display\_text\_image(s1, s2)

**Description:** Display the 2-d sequence of characters and attributes contained in s2 at line s1[1], column s1[2]. s2 is a sequence of sequences, where each sequence is a string of characters and attributes to be displayed. The top-left character is displayed at s1. Other characters appear to the right or below this position. The attributes indicate the foreground and background color of the preceding character. On DOS32, the attribute should consist of the foreground color plus 16 times the background color.

**Comments:** s2 would normally be the result of a previous call to save\_text\_image(), although you could construct it yourself.

This routine only works in **text modes**.

You might use save\_text\_image()/display\_text\_image() in a text-mode graphical user interface, to allow "pop-up" dialog boxes, and drop-down menus to appear and disappear without losing what was previously on the screen.

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

**Example:**

```
clear_screen()
display_text_image({1,1}, {{ 'A', WHITE, 'B', GREEN},
                           { 'C', RED+16*WHITE},
                           { 'D', BLUE}})

-- displays:
  AB
  C
  D

-- at the top left corner of the screen.
-- 'A' will be white with black (0) background color,
-- 'B' will be green on black,
-- 'C' will be red on white, and
-- 'D' will be blue on black.
```

**See Also:** [save\\_text\\_image](#), [display\\_image](#), [put\\_screen\\_char](#)

## dos\_interrupt

**Platform:** **DOS32**

**Syntax:** `include machine.e`

`s2 = dos_interrupt(i, s1)`

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

```
global constant REG_DI = 1,
                REG_SI = 2,
                REG_BP = 3,
                REG_BX = 4,
                REG_DX = 5,
                REG_CX = 6,
                REG_AX = 7,
                REG_FLAGS = 8,
                REG_ES = 9,
                REG_DS = 10
```

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

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

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

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

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

**Example:**

```
sequence registers
```

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

```
-- call DOS interrupt 5: Print Screen
```

```
registers = dos_interrupt(#5, registers)
```

**Example Program:** [demoldos32ldosint.ex](#)

**See Also:** [allocate\\_low](#), [free\\_low](#)



## draw\_line

**Platform:** **DOS32**

**Syntax:** include graphics.e

draw\_line(i, s)

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

**Example:**

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

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

**See Also:** [polygon](#), [ellipse](#), [pixel](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)

## ellipse

**Platform:** **DOS32**

**Syntax:** `include graphics.e`

`ellipse(i1, i2, s1, s2)`

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

**Example:**

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

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

**Example Program:** [demo\dos32\s\\_b.ex](#)

**See Also:** [polygon](#), [draw\\_line](#)

## equal

**Syntax:** `i = equal(x1, x2)`

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

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

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](#), [equals operator \(=\)](#)

## find

**Syntax:** `i = find(x, s)`

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

**Example 1:**

```
location = find(11, {5, 8, 11, 2, 3})
-- location is set to 3
```

### Example 2:

```
names = {"fred", "rob", "george", "mary", ""}
location = find("mary", names)
-- location is set to 4
```

See Also: [match](#), [compare](#)

## float32\_to\_atom

**Syntax:** include machine.e

a1 = float32\_to\_atom(s)

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

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

**Example:**

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

See Also: [float64\\_to\\_atom](#), [atom\\_to\\_float32](#)

## float64\_to\_atom

**Syntax:** include machine.e

a1 = float64\_to\_atom(s)

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

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

**Example:**

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

See Also: [float32\\_to\\_atom](#), [atom\\_to\\_float64](#)

## floor

**Syntax:** x2 = floor(x1)

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

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

**Example:**

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

**See Also:** [remainder](#)

## flush

**Syntax:** include file.e

flush(fn)

**Description:** When you write data to a file, Euphoria normally stores the data in a memory buffer until a large enough chunk of data has accumulated. This large chunk can then be written to disk very efficiently. Sometimes you may want to force, or flush, all data out immediately, even if the memory buffer is not full. To do this you must call flush(fn), where fn is the file number of a file open for writing or appending.

**Comments:** When a file is closed, (see close()), all buffered data is flushed out. When a program terminates, all open files are flushed and closed automatically.

Use flush() when another process may need to see all of the data written so far, but you aren't ready to close the file yet.

**Example:**

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

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

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

**See Also:** [close](#), [lock\\_file](#)

## free

**Syntax:** include machine.e

free(a)

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

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

Do not use free() to deallocate memory that was allocated using [allocate\\_low\(\)](#). Use free\_low() for this purpose.

**Example Program:** [demo\callmach.ex](#)

**See Also:** [allocate](#), [free\\_low](#)

## free\_console

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

free\_console()

**Description:** Free (delete) any console window associated with your program.

**Comments:** Euphoria will create a console **text** window for your program the first time that your program prints something to the screen, reads something from the keyboard, or in some way needs a console (similar to a DOS-prompt window). On WIN32 this window will automatically disappear when your program terminates, but you can call free\_console() to make it disappear sooner. On Linux or FreeBSD, the text mode console is always there, but an xterm window will disappear after Euphoria issues a "Press Enter" prompt at the end of execution.

On Linux or FreeBSD, free\_console() will set the terminal parameters back to normal, undoing the effect that curses has on the screen.

In a Linux or FreeBSD xterm window, a call to free\_console(), without any further printing to the screen or reading from the keyboard, will eliminate the "Press Enter" prompt that Euphoria normally issues at the end of execution.

After freeing the console window, you can create a new console window by printing something to the screen, or simply calling clear\_screen(), [position\(\)](#) or any other routine that needs a console.

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

There's a WIN32 API routine, FreeConsole() that does something similar to free\_console(). You should use free\_console(), because it lets the interpreter know that there is no longer a console.

**See Also:** [clear\\_screen](#), [platform.doc](#)

## free\_low

**Platform:** DOS32

**Syntax:** include machine.e

free\_low(i)

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

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

Do not use free\_low() to deallocate memory that was allocated using [allocate\(\)](#). Use free() for this purpose.

**Example Program:** [demo\dos32\dosint.ex](#)

**See Also:** [allocate\\_low](#), [dos\\_interrupt](#), [free](#)

## get

**Syntax:** include get.e

s = get(fn)

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

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

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

Each call to get() picks up where the previous call left off. For instance, a series of 5 calls to get() would be needed to read in:

```
99 5.2 {1,2,3} "Hello" -1
```

On the sixth and any subsequent call to get() you would see a GET\_EOF status. If you had something like:

```
{1, 2, xxx}
```

in the input stream you would see a GET\_FAIL error status because xxx is not a Euphoria object.

Multiple "top-level" objects in the input stream must be separated from each other with one or more "whitespace" characters (blank, tab, \r or \n). Whitespace is not necessary *within* a top-level object. A call to get() will read one entire top-level object, plus one additional (whitespace) character.

**Comments:** The combination of print() and get() can be used to save a Euphoria object to disk and later read it back. This technique could be used to implement a database as one or more large Euphoria sequences stored in disk files. The sequences could be read into memory, updated and then written back to disk after each series of transactions is complete. Remember to write out a whitespace character (using puts()) after each call to print().

The value returned is not meaningful unless you have a GET\_SUCCESS status.

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

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

```
{GET_SUCCESS, 77.5}
```

```
-- whereas gets(0) would return:
```

```
"77.5\n"
```

**Example Program:** [demo\mydata.ex](#)

**See Also:** [print](#), [value](#), [gets](#), [getc](#), [prompt\\_number](#), [prompt\\_string](#)

## get\_active\_page

**Platform:** DOS32

**Syntax:** include image.e

`i = get_active_page()`

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

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

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

**See Also:** [set\\_active\\_page](#), [get\\_display\\_page](#), [video\\_config](#)

## get\_all\_palette

**Platform:** DOS32

**Syntax:** `include image.e`

`s = get_all_palette()`

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

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

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

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

**See Also:** [palette](#), [all\\_palette](#), [read\\_bitmap](#), [save\\_bitmap](#), [save\\_screen](#)

## get\_bytes

**Syntax:** `include get.e`

`s = get_bytes(fn, i)`

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

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

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

**Example:**

```
include get.e

integer fn
fn = open("temp", "rb")  -- an existing file

sequence whole_file
whole_file = {}

sequence chunk

while 1 do
```

```

    chunk = get_bytes(fn, 100) -- read 100 bytes at a time
    whole_file &= chunk        -- chunk might be empty, that's ok
    if length(chunk) < 100 then
        exit
    end if
end while

close(fn)
? length(whole_file) -- should match DIR size of "temp"

```

**See Also:**    [getc](#), [gets](#)

## get\_display\_page

**Platform:**    **DOS32**

**Syntax:**      include image.e

i = get\_display\_page()

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

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

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

**See Also:**    [set\\_display\\_page](#), [get\\_active\\_page](#), [video\\_config](#)

## get\_key

**Syntax:**      i = get\_key()

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

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

Run the [key.bat](#) program to see what key code is generated for each key on your keyboard.

**See Also:**    [wait\\_key](#), [getc](#)

## get\_mouse

**Platform:**    **DOS32, Linux**

**Syntax:**      include mouse.e

x1 = get\_mouse()

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

Constants have been defined in [mouse.e](#) for the possible mouse events:

```

global constant MOVE = 1,
    LEFT_DOWN = 2,
    LEFT_UP = 4,

```



```

RIGHT_DOWN = 8,
RIGHT_UP = 16,
MIDDLE_DOWN = 32,
MIDDLE_UP = 64

```

x and y are the coordinates of the mouse pointer at the time that the event occurred. `get_mouse()` returns immediately with either a -1 or a mouse event. It does not wait for an event to occur. You must check it frequently enough to avoid missing an event. When the next event occurs, the current event will be lost, if you haven't read it. In practice it is not hard to catch almost all events. Losing a MOVE event is generally not too serious, as the next MOVE will tell you where the mouse pointer is.

Sometimes multiple events will be reported. For example, if the mouse is moving when the left button is clicked, `get_mouse()` will report an event value of LEFT\_DOWN+MOVE, i.e. 2+1 or 3. For this reason you should test for a particular event using `and_bits()`. See examples below.

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

In DOS32 **text modes** you need to scale the x and y coordinates to get line and column positions. In Linux, no scaling is required - x and y correspond to the line and column on the screen, with (1,1) at the top left.

In DOS32, you need a DOS mouse driver to use this routine. In Linux, GPM Server must be running.

In Linux, mouse movement events are not reported in an xterm window, only in the text console.

In Linux, LEFT\_UP, RIGHT\_UP and MIDDLE\_UP are not distinguishable from one another.

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

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

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

The x,y coordinate returned could be that of the very tip of the mouse pointer or might refer to the pixel pointed-to by the mouse pointer. Test this if you are trying to read the pixel color using [get\\_pixel\(\)](#). You may have to read x-1,y-1 instead.

**Example 1:** a return value of:

```
{2, 100, 50}
```

would indicate that the left button was pressed down when the mouse pointer was at location x=100, y=50 on the screen.

**Example 2:** To test for LEFT\_DOWN, write something like the following:

```
object event
```

```

while 1 do
  event = get_mouse()
  if sequence(event) then
    if and_bits(event[1], LEFT_DOWN) then
      -- left button was pressed
      exit
    end if
  end if
end if

```

end while

**See Also:** [mouse\\_events](#), [mouse\\_pointer](#), and [bits](#)

## get\_pixel

**Platform:** DOS32

**Syntax:** x = get\_pixel(s)

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

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

Points off the screen have unpredictable color values.

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

**Example:**

object x

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

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

**See Also:** [pixel](#), [graphics\\_mode](#), [get\\_position](#)

## get\_position

**Syntax:** include graphics.e

s = get\_position()

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

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

The coordinate system for displaying text is different from the one for displaying pixels. Pixels are displayed such that the top-left is (x=0,y=0) and the first coordinate controls the horizontal, left-right location. In pixel-graphics modes you can display both text and pixels. get\_position() returns the current line and column for the text that you are displaying, not the pixels that you may be plotting. There is no corresponding routine for getting the current pixel position.

**See Also:** [position](#), [get\\_pixel](#)

## get\_screen\_char

**Syntax:** include image.e

s = get\_screen\_char(i1, i2)

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

**Comments:** With get\_screen\_char() and put\_screen\_char() you can save and restore a character on the screen along with its attributes.

**Example:**

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

**See Also:** [put\\_screen\\_char](#), [save\\_text\\_image](#)

## get\_vector

**Platform:** DOS32

**Syntax:** include machine.e

s = get\_vector(i)

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

**Example:**

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

**Example Program:** [demo\dos32\hardint.ex](#)

**See Also:** [set\\_vector](#), [lock\\_memory](#)

## getc

**Syntax:** i = getc(fn)

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

**Comments:** File input using getc() is buffered, i.e. getc() does not actually go out to the disk for each character. Instead, a large block of characters will be read in at one time and returned to you one by one from a memory buffer.

When getc() reads from the keyboard, it will not see any characters until the user presses Enter. Note that the user can type control-Z, which the operating system treats as "end of file". -1 will be returned.

**See Also:** [gets](#), [get\\_key](#), [wait\\_key](#), [open](#)

## getenv

**Syntax:**      `x = getenv(s)`

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

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

Both the argument and the return value, may, or may not be, case sensitive. You might need to test this on your system.

**Example:**

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

**See Also:**    [command\\_line](#)

## gets

**Syntax:**      `x = gets(fn)`

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

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

After reading a line of text from the keyboard, you should normally output a '\n' character, e.g. `puts(1, '\n')`, before printing something. Only on the last line of the screen does the operating system automatically scroll the screen and advance to the next line.

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

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

In SVGA modes, DOS might set the wrong cursor position, after a call to `gets(0)` to read the keyboard.

You should set it yourself using [position\(\)](#).

**Example 1:**

```
    sequence buffer
object line
integer fn

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

buffer = {}
while 1 do
    line = gets(fn)
    if atom(line) then
        exit -- -1 is returned at end of file
    end if
    buffer = append(buffer, line)
```

```
end while
```

### Example 2:

```
object line

puts(1, "What is your name?\n")
line = gets(0) -- read standard input (keyboard)
line = line[1..length(line)-1] -- get rid of \n character at end
puts(1, '\n') -- necessary
puts(1, line & " is a nice name.\n")
```

**See Also:** [getc](#), [puts](#), [open](#)

## graphics\_mode

**Platform:** **DOS32**

**Syntax:** include graphics.e

i1 = graphics\_mode(i2)

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

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

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

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

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

### Example:

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

**See Also:** [text\\_rows](#), [video\\_config](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)

## ellipse

**Platform:** **DOS32**

**Syntax:** include graphics.e

ellipse(i1, i2, s1, s2)

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

**Example:**

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

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

**Example Program:** [demo\dos32\sb.ex](#)

**See Also:** [polygon](#), [draw\\_line](#)

## equal

**Syntax:**        `i = equal(x1, x2)`

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

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

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](#), [equals operator \(=\)](#)

## find

**Syntax:** `i = find(x, s)`

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

**Example 1:**

```
location = find(11, {5, 8, 11, 2, 3})  
-- location is set to 3
```

**Example 2:**

```
names = {"fred", "rob", "george", "mary", ""}  
location = find("mary", names)  
-- location is set to 4
```

**See Also:** [match](#), [compare](#)



## float32\_to\_atom

**Syntax:**           include machine.e

a1 = float32\_to\_atom(s)

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

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

**Example:**

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

**See Also:**     [float64\\_to\\_atom](#), [atom\\_to\\_float32](#)

## float64\_to\_atom

**Syntax:**           include machine.e

a1 = float64\_to\_atom(s)

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

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

**Example:**

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

**See Also:**     [float32\\_to\\_atom](#), [atom\\_to\\_float64](#)

## floor

**Syntax:** `x2 = floor(x1)`

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

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

**Example:**

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

**See Also:** [remainder](#)

## flush

**Syntax:** include file.e

flush(fn)

**Description:** When you write data to a file, Euphoria normally stores the data in a memory buffer until a large enough chunk of data has accumulated. This large chunk can then be written to disk very efficiently. Sometimes you may want to force, or flush, all data out immediately, even if the memory buffer is not full. To do this you must call flush(fn), where fn is the file number of a file open for writing or appending.

**Comments:** When a file is closed, (see close()), all buffered data is flushed out. When a program terminates, all open files are flushed and closed automatically.

Use flush() when another process may need to see all of the data written so far, but you aren't ready to close the file yet.

**Example:**

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

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

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

**See Also:** [close](#), [lock\\_file](#)

## free

**Syntax:**       include machine.h  
free(a)

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

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

Do not use free() to deallocate memory that was allocated using [allocate\\_low\(\)](#). Use free\_low() for this purpose.

**Example Program:**   [demo\callmach.ex](#)

**See Also:**       [allocate](#), [free\\_low](#)

## free\_console

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

free\_console()

**Description:** Free (delete) any console window associated with your program.

**Comments:** Euphoria will create a console **text** window for your program the first time that your program prints something to the screen, reads something from the keyboard, or in some way needs a console (similar to a DOS-prompt window). On WIN32 this window will automatically disappear when your program terminates, but you can call free\_console() to make it disappear sooner. On Linux or FreeBSD, the text mode console is always there, but an xterm window will disappear after Euphoria issues a "Press Enter" prompt at the end of execution.

On Linux or FreeBSD, free\_console() will set the terminal parameters back to normal, undoing the effect that curses has on the screen.

In a Linux or FreeBSD xterm window, a call to free\_console(), without any further printing to the screen or reading from the keyboard, will eliminate the "Press Enter" prompt that Euphoria normally issues at the end of execution.

After freeing the console window, you can create a new console window by printing something to the screen, or simply calling clear\_screen(), [position\(\)](#) or any other routine that needs a console.

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

There's a WIN32 API routine, FreeConsole() that does something similar to free\_console(). You should use free\_console(), because it lets the interpreter know that there is no longer a console.

**See Also:** [clear\\_screen](#), [platform.doc](#)

## free\_low

**Platform:** **DOS32**

**Syntax:** include machine.e

free\_low(i)

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

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

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

**Example Program:** [demo\dos32\dosint.ex](#)

**See Also:** [allocate\\_low](#), [dos\\_interrupt](#), [free](#)

## get

**Syntax:** include get.e

s = get(fn)

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

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

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

Each call to get() picks up where the previous call left off. For instance, a series of 5 calls to get() would be needed to read in:

```
99 5.2 {1,2,3} "Hello" -1
```

On the sixth and any subsequent call to get() you would see a GET\_EOF status. If you had something like:

```
{1, 2, xxx}
```

in the input stream you would see a GET\_FAIL error status because xxx is not a Euphoria object.

Multiple "top-level" objects in the input stream must be separated from each other with one or more "whitespace" characters (blank, tab, \r or \n). Whitespace is not necessary *within* a top-level object. A call to get() will read one entire top-level object, plus one additional (whitespace) character.

**Comments:** The combination of print() and get() can be used to save a Euphoria object to disk and later read it back. This technique could be used to implement a database as one or more large Euphoria sequences stored in disk files. The sequences could be read into memory, updated and then written back to disk after each series of transactions is complete. Remember to write out a whitespace character (using puts()) after each call to print().

The value returned is not meaningful unless you have a GET\_SUCCESS status.

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

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

```
{GET_SUCCESS, 77.5}
```

```
-- whereas gets(0) would return:
```

```
"77.5\n"
```

**Example Program:** [demo\mydata.ex](#)

**See Also:** [print](#), [value](#), [gets](#), [getc](#), [prompt\\_number](#), [prompt\\_string](#)



## get\_active\_page

**Platform:** DOS32

**Syntax:** include image.e

i = get\_active\_page()

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

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

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

**See Also:** [set\\_active\\_page](#), [get\\_display\\_page](#), [video\\_config](#)

## get\_all\_palette

**Platform:** DOS32

**Syntax:** include image.e

s = get\_all\_palette()

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

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

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

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

**See Also:** [palette](#), [all\\_palette](#), [read\\_bitmap](#), [save\\_bitmap](#), [save\\_screen](#)

## get\_bytes

**Syntax:**       include get.e

s = get\_bytes(fn, i)

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

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

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

**Example:**

```
include get.e

integer fn
fn = open("temp", "rb")  -- an existing file

sequence whole_file
whole_file = {}

sequence chunk

while 1 do
  chunk = get_bytes(fn, 100) -- read 100 bytes at a time
  whole_file &= chunk        -- chunk might be empty, that's ok
  if length(chunk) < 100 then
    exit
  end if
end while

close(fn)
? length(whole_file)  -- should match DIR size of "temp"
```

**See Also:**   [getc](#), [gets](#)

## get\_display\_page

**Platform:** **DOS32**

**Syntax:** include image.e

i = get\_display\_page()

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

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

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

**See Also:** [set\\_display\\_page](#), [get\\_active\\_page](#), [video\\_config](#)

## get\_key

**Syntax:** `i = get_key()`

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

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

Run the [key.bat](#) program to see what key code is generated for each key on your keyboard.

**See Also:** [wait\\_key](#), [getc](#)

## get\_mouse

**Platform:** DOS32, Linux

**Syntax:** include mouse.e

x1 = get\_mouse()

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

Constants have been defined in [mouse.e](#) for the possible mouse events:

```
global constant MOVE = 1,
    LEFT_DOWN = 2,
    LEFT_UP = 4,
    RIGHT_DOWN = 8,
    RIGHT_UP = 16,
    MIDDLE_DOWN = 32,
    MIDDLE_UP = 64
```

x and y are the coordinates of the mouse pointer at the time that the event occurred. get\_mouse() returns immediately with either a -1 or a mouse event. It does not wait for an event to occur. You must check it frequently enough to avoid missing an event. When the next event occurs, the current event will be lost, if you haven't read it. In practice it is not hard to catch almost all events. Losing a MOVE event is generally not too serious, as the next MOVE will tell you where the mouse pointer is.

Sometimes multiple events will be reported. For example, if the mouse is moving when the left button is clicked, get\_mouse() will report an event value of LEFT\_DOWN+MOVE, i.e. 2+1 or 3. For this reason you should test for a particular event using and\_bits(). See examples below.

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

In DOS32 [text modes](#) you need to scale the x and y coordinates to get line and column positions. In Linux, no scaling is required - x and y correspond to the line and column on the screen, with (1,1) at the top left.

In DOS32, you need a DOS mouse driver to use this routine. In Linux, GPM Server must be running.

In Linux, mouse movement events are not reported in an xterm window, only in the text console.

In Linux, LEFT\_UP, RIGHT\_UP and MIDDLE\_UP are not distinguishable from one another.

You can use get\_mouse() in [most text and pixel-graphics modes](#).

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

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

The x,y coordinate returned could be that of the very tip of the mouse pointer or might refer to the pixel pointed-to by the mouse pointer. Test this if you are trying to read the pixel color using [get\\_pixel\(\)](#). You may have to read x-1,y-1 instead.

**Example 1:** a return value of:

```
{2, 100, 50}
```

would indicate that the left button was pressed down when the mouse pointer was at location x=100, y=50 on the screen.

**Example 2:** To test for LEFT\_DOWN, write something like the following:

```
object event

while 1 do
  event = get_mouse()
  if sequence(event) then
    if and_bits(event[1], LEFT_DOWN) then
      -- left button was pressed
      exit
    end if
  end if
end while
```

**See Also:** [mouse\\_events](#), [mouse\\_pointer](#), [and\\_bits](#)

## get\_pixel

**Platform:** DOS32

**Syntax:** `x = get_pixel(s)`

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

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

Points off the screen have unpredictable color values.

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

**Example:**

`object x`

```
x = get_pixel({30,40})
```

```
-- x is set to the color value of point x=30, y=40
```

```
x = get_pixel({30,40,100})
```

```
-- x is set to a sequence of 100 integer values, representing
```

```
-- the colors starting at {30,40} and going to the right
```

**See Also:** [pixel](#), [graphics\\_mode](#), [get\\_position](#)



## get\_position

**Syntax:**           include graphics.e

s = get\_position()

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

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

The coordinate system for displaying text is different from the one for displaying pixels. Pixels are displayed such that the top-left is (x=0,y=0) and the first coordinate controls the horizontal, left-right location. In pixel-graphics modes you can display both text and pixels. get\_position() returns the current line and column for the text that you are displaying, not the pixels that you may be plotting. There is no corresponding routine for getting the current pixel position.

**See Also:**       [position](#), [get\\_pixel](#)

## get\_screen\_char

**Syntax:**           include image.e

s = get\_screen\_char(i1, i2)

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

**Comments:**     With get\_screen\_char() and put\_screen\_char() you can save and restore a character on the screen along with its attributes.

**Example:**

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

**See Also:**     [put\\_screen\\_char](#), [save\\_text\\_image](#)

## get\_vector

**Platform:** **DOS32**

**Syntax:** include machine.e

s = get\_vector(i)

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

**Example:**

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

**Example Program:** [demo\dos32\hardint.ex](#)

**See Also:** [set\\_vector](#), [lock\\_memory](#)

## getc

**Syntax:** `i = getc(fn)`

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

**Comments:** File input using `getc()` is buffered, i.e. `getc()` does not actually go out to the disk for each character. Instead, a large block of characters will be read in at one time and returned to you one by one from a memory buffer.

When `getc()` reads from the keyboard, it will not see any characters until the user presses Enter. Note that the user can type control-Z, which the operating system treats as "end of file". -1 will be returned.

**See Also:** [gets](#), [get\\_key](#), [wait\\_key](#), [open](#)

## getenv

**Syntax:** x = getenv(s)

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

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

Both the argument and the return value, may, or may not be, case sensitive. You might need to test this on your system.

**Example:**

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

**See Also:** [command\\_line](#)

## gets

**Syntax:** x = gets(fn)

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

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

After reading a line of text from the keyboard, you should normally output a '\n' character, e.g. puts(1, '\n'), before printing something. Only on the last line of the screen does the operating system automatically scroll the screen and advance to the next line.

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

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

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

You should set it yourself using [position\(\)](#).

### Example 1:

```
sequence buffer
object line
integer fn

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

buffer = {}
while 1 do
    line = gets(fn)
    if atom(line) then
        exit -- -1 is returned at end of file
    end if
    buffer = append(buffer, line)
end while
```

### Example 2:

```
object line

puts(1, "What is your name?\n")
line = gets(0) -- read standard input (keyboard)
line = line[1..length(line)-1] -- get rid of \n character at end
puts(1, '\n') -- necessary
puts(1, line & " is a nice name.\n")
```

**See Also:** [getc](#), [puts](#), [open](#)

## graphics\_mode

**Platform:** **DOS32**

**Syntax:** include graphics.e

i1 = graphics\_mode(i2)

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

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

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

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

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

**Example:**

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

**See Also:** [text\\_rows](#), [video\\_config](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)

## instance

**Platform:** WIN32

**Syntax:** include misc.e

i = instance()

**Description:** Return a handle to the current program.

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

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

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

**See Also:** [platform.doc](#)

## int\_to\_bits

**Syntax:** include machine.e

s = int\_to\_bits(a, i)

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

**Comments:** You can use [subscripting](#), [slicing](#), [and/or/xor/not](#) of entire sequences etc. to manipulate sequences of bits. Shifting of bits and rotating of bits are easy to perform.

**Example:**

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

**See Also:** [bits to int](#), [and bits](#), [or bits](#), [xor bits](#), [not bits](#), [operations on sequences](#)

## int\_to\_bytes

**Syntax:** include machine.e

s = int\_to\_bytes(a)

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

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

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

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

**Example 1:**

```
s = int_to_bytes(999)
```



```
-- s is {231, 3, 0, 0}
```

#### Example 2:

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

See Also: [bytes\\_to\\_int](#), [int\\_to\\_bits](#), [bits\\_to\\_int](#), [peek](#), [poke](#), [poke4](#)

## integer

**Syntax:** `i = integer(x)`

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

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

#### Example 1:

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

#### Example 2:

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

See Also: [atom](#), [sequence](#), [floor](#)

## length

**Syntax:** `i = length(s)`

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

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

#### Example 1:

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

#### Example 2:

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

#### Example 3:

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

See Also: [sequence](#)

## lock\_file

**Syntax:** `include file.e`

`i1 = lock_file(fn, i2, s)`

**Description:** When multiple processes can simultaneously access a file, some kind of locking mechanism

may be needed to avoid mangling the contents of the file, or causing erroneous data to be read from the file.

`lock_file()` attempts to place a lock on an open file, `fn`, to stop other processes from using the file while your program is reading it or writing it. Under Linux/FreeBSD, there are two types of locks that you can request using the `i2` parameter. (Under DOS32 and WIN32 the `i2` parameter is ignored, but should be an integer.) Ask for a *shared* lock when you intend to read a file, and you want to temporarily block other processes from writing it. Ask for an *exclusive* lock when you intend to write to a file and you want to temporarily block other processes from reading or writing it. It's ok for many processes to simultaneously have shared locks on the same file, but only one process can have an exclusive lock, and that can happen only when no other process has any kind of lock on the file. `file.e` contains the following declaration:

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

On DOS32 and WIN32 you can lock a specified portion of a file using the `s` parameter. `s` is a sequence of the form: {first\_byte, last\_byte}. It indicates the first byte and last byte in the file, that the lock applies to. Specify the empty sequence {}, if you want to lock the whole file. In the current release for Linux/FreeBSD, locks always apply to the whole file, and you should specify {} for this parameter.

If it is successful in obtaining the desired lock, `lock_file()` will return 1. If unsuccessful, it will return 0. `lock_file()` does not wait for other processes to relinquish their locks. You may have to call it repeatedly, before the lock request is granted.

**Comments:** On Linux/FreeBSD, these locks are called advisory locks, which means they aren't enforced by the operating system. It is up to the processes that use a particular file to cooperate with each other. A process can access a file without first obtaining a lock on it. On WIN32 and DOS32, locks are enforced by the operating system.

On DOS32, `lock_file()` is more useful when file sharing is enabled. It will typically return 0 (unsuccessful) under plain MS-DOS, outside of Windows.

**Example:**

```
include misc.e
include file.e
integer v
atom t
v = open("visitor_log", "a") -- open for append
t = time()
while not lock_file(v, LOCK_EXCLUSIVE, {}) do
  if time() > t + 60 then
    puts(1, "One minute already ... I can't wait forever!\n")
    abort(1)
  end if
  sleep(5) -- let other processes run
end while
puts(v, "Yet another visitor\n")
unlock_file(v, {})
close(v)
```

**See Also:** [unlock\\_file](#), [flush](#), [sleep](#)

## lock\_memory

**Platform:** DOS32

**Syntax:** include machine.e

`lock_memory(a, i)`

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

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

**Example Program:** [demo\dos32\hardint.ex](#)

**See Also:** [get\\_vector](#), [set\\_vector](#)

## log

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

**Description:** Return the natural logarithm of x1.

**Comments:** This function may be applied to an atom or to all elements of a sequence. Note that log is only defined for positive numbers. Your program will abort with a message if you try to take the log of a negative number or zero.

**Example:**

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

**See Also:** [sin](#), [cos](#), [tan](#), [sqrt](#)

## lower

**Syntax:** include wildcard.e

x2 = lower(x1)

**Description:** Convert an atom or sequence to lower case.

**Example:**

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

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

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

**See Also:** [upper](#)

## machine\_func

**Syntax:** x1 = machine\_func(a, x)

**Description:** see [machine\\_proc\(\)](#) below

## machine\_proc

**Syntax:** machine\_proc(a, x)

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

**See Also:** [machine\\_func](#)

## match

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

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

**Example:**

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

**See Also:** [find](#), [compare](#), [wildcard\\_match](#)

## mem\_copy

**Syntax:** mem\_copy(a1, a2, i)

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

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

mem\_copy(a1, a2, i) is equivalent to: **poke(a1, peek({a2, i}))** but is much faster.

**Example:**

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

**See Also:** [mem\\_set](#), [peek](#), [poke](#), [allocate](#), [allocate\\_low](#)

## mem\_set

**Syntax:** mem\_set(a1, i1, i2)

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

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

mem\_set(a1, i1, i2) is equivalent to: **poke(a1, repeat(i1, i2))** but is much faster.

**Example:**

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

**See Also:** [mem\\_copy](#), [peek](#), [poke](#), [allocate](#), [allocate\\_low](#)

## message\_box

**Platform:** WIN32

**Syntax:** include msgbox.e

i = message\_box(s1, s2, x)

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

**Comments:** See [msgbox.e](#) for a complete list of possible values for x and i.

**Example:**

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

**Example Program:** [demo\win32\email.exw](#)

## mouse\_events

**Platform:** DOS32, Linux

**Syntax:** include mouse.e

mouse\_events(i)

**Description:** Use this procedure to select the mouse events that you want get\_mouse() to report. By default, get\_mouse() will report all events. mouse\_events() can be called at various stages of the execution of your program, as the need to detect events changes. Under Linux, mouse\_events() currently has no effect.

**Comments:** It is good practice to ignore events that you are not interested in, particularly the very frequent MOVE event, in order to reduce the chance that you will miss a significant event.

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

**Example:**

```
        mouse_events(LEFT_DOWN + LEFT_UP + RIGHT_DOWN)
-- will restrict get_mouse() to reporting the left button
-- being pressed down or released, and the right button
-- being pressed down. All other events will be ignored.
```

**See Also:** [get\\_mouse](#), [mouse\\_pointer](#)

## mouse\_pointer

**Platform:** DOS32, Linux

**Syntax:** include mouse.e

mouse\_pointer(i)

**Description:** If i is 0 hide the mouse pointer, otherwise turn on the mouse pointer. Multiple calls to hide the pointer will require multiple calls to turn it back on. The first call to either get\_mouse() or mouse\_events(),

will also turn the pointer on (once). Under Linux, `mouse_pointer()` currently has no effect

**Comments:** It may be necessary to hide the mouse pointer temporarily when you update the screen.

After a call to [text\\_rows\(\)](#) you may have to call `mouse_pointer(1)` to see the mouse pointer again.

**See Also:** [get\\_mouse](#), [mouse\\_events](#)

## not\_bits

**Syntax:** `x2 = not_bits(x1)`

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

**Comments:** The argument to this function may be an atom or a sequence. The rules for [operations on sequences](#) apply.

The argument must be representable as a 32-bit number, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

**Example:**

```
a = not_bits(#000000F7)
-- a is -248 (i.e. FFFFFFF08 interpreted as a negative number)
```

**See Also:** [and\\_bits](#), [or\\_bits](#), [xor\\_bits](#), [int\\_to\\_bits](#)

## object

**Syntax:** `i = object(x)`

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

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

**Example:**

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

**See Also:** [integer](#), [atom](#), [sequence](#)

## open

**Syntax:** `fn = open(st1, st2)`

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

"r" - open text file for reading

"rb" - open binary file for reading

"w" - create text file for writing

"wb" - create binary file for writing

"u" - open text file for update (reading and writing)  
"ub" - open binary file for update  
"a" - open text file for appending  
"ab" - open binary file for appending

Files opened for read or update must already exist. Files opened for write or append will be created if necessary. A file opened for write will be set to 0 bytes. Output to a file opened for append will start at the end of file.

Output to **text files** will have carriage-return characters automatically added before linefeed characters. On input, these carriage-return characters are removed. A control-Z character (ASCII 26) will signal an immediate end of file. Note: on some versions of DOS, a control-Z typed by the user might cause standard input to permanently appear to be at the end-of-file, until the DOS window is closed.

I/O to **binary files** is not modified in any way. Any byte values from 0 to 255 can be read or written.

Some typical devices that you can open on DOS or 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

Currently, files up to 2 Gb in size can be handled. Beyond that, some file operations may not work correctly. This limit will likely be increased in the future.

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

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

**DOS32:** Be careful not to use the special device names in a file name, even if you add an extension. e.g. CON.TXT, CON.DAT, CON.JPG etc. all refer to the CON device, not a file.

**Example:**

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

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

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

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

**See Also:**    [close](#), [lock\\_file](#)

## open\_dll

**Platform:**    **WIN32, Linux, FreeBSD**

**Syntax:**    include dll.e

**a = open\_dll(st)**

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

**Comments:**    The value returned by open\_dll() can be passed to define\_c\_proc(), define\_c\_func(), or define\_c\_var().

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

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

### Example:

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

**See Also:**    [define\\_c\\_func](#), [define\\_c\\_proc](#), [define\\_c\\_var](#), [c\\_func](#), [c\\_proc](#), [platform.doc](#)

## or\_bits

**Syntax:**    x3 = or\_bits(x1, x2)

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

**Comments:**    The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

### Example 1:



```
    a = or_bits(#0F0F0000, #12345678)
-- a is #1F3F5678
```

**Example 2:**

```
    a = or_bits(#FF, {#123456, #876543, #2211})
-- a is {#1234FF, #8765FF, #22FF}
```

**See Also:**    [and\\_bits](#), [xor\\_bits](#), [not\\_bits](#), [int\\_to\\_bits](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)

## instance

**Platform:** WIN32

**Syntax:** include misc.e

i = instance()

**Description:** Return a handle to the current program.

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

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

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

**See Also:** [platform.doc](#)

## int\_to\_bits

**Syntax:** `include machine.e`

`s = int_to_bits(a, i)`

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

**Comments:** You can use [subscripting](#), [slicing](#), [and/or/xor/not](#) of entire sequences etc. to manipulate sequences of bits. Shifting of bits and rotating of bits are easy to perform.

**Example:**

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

**See Also:** [bits\\_to\\_int](#), [and\\_bits](#), [or\\_bits](#), [xor\\_bits](#), [not\\_bits](#), [operations on sequences](#)

## int\_to\_bytes

**Syntax:**       include machine.e

s = int\_to\_bytes(a)

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

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

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

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

### Example 1:

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

### Example 2:

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

**See Also:**     [bytes\\_to\\_int](#), [int\\_to\\_bits](#), [bits\\_to\\_int](#), [peek](#), [poke](#), [poke4](#)

# integer

**Syntax:** i = integer(x)

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

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

**Example 1:**

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

**Example 2:**

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

**See Also:** [atom](#), [sequence](#), [floor](#)

## length

**Syntax:** `i = length(s)`

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

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

**Example 1:**

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

**Example 2:**

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

**Example 3:**

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

**See Also:** [sequence](#)

## lock\_file

**Syntax:** include file.e

i1 = lock\_file(fn, i2, s)

**Description:** When multiple processes can simultaneously access a file, some kind of locking mechanism may be needed to avoid mangling the contents of the file, or causing erroneous data to be read from the file.

lock\_file() attempts to place a lock on an open file, fn, to stop other processes from using the file while your program is reading it or writing it. Under Linux/FreeBSD, there are two types of locks that you can request using the i2 parameter. (Under DOS32 and WIN32 the i2 parameter is ignored, but should be an integer.) Ask for a *shared* lock when you intend to read a file, and you want to temporarily block other processes from writing it. Ask for an *exclusive* lock when you intend to write to a file and you want to temporarily block other processes from reading or writing it. It's ok for many processes to simultaneously have shared locks on the same file, but only one process can have an exclusive lock, and that can happen only when no other process has any kind of lock on the file. file.e contains the following declaration:

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

On DOS32 and WIN32 you can lock a specified portion of a file using the s parameter. s is a sequence of the form: {first\_byte, last\_byte}. It indicates the first byte and last byte in the file, that the lock applies to. Specify the empty sequence {}, if you want to lock the whole file. In the current release for Linux/FreeBSD, locks always apply to the whole file, and you should specify {} for this parameter.

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

lock\_file() does not wait for other processes to relinquish their locks. You may have to call it repeatedly, before the lock request is granted.

**Comments:** On Linux/FreeBSD, these locks are called advisory locks, which means they aren't enforced by the operating system. It is up to the processes that use a particular file to cooperate with each other. A process can access a file without first obtaining a lock on it. On WIN32 and DOS32, locks are enforced by the operating system.

On DOS32, lock\_file() is more useful when file sharing is enabled. It will typically return 0 (unsuccessful) under plain MS-DOS, outside of Windows.

**Example:**

```
include misc.e  
include file.e  
integer v  
atom t  
v = open("visitor_log", "a") -- open for append  
t = time()  
while not lock_file(v, LOCK_EXCLUSIVE, {}) do  
  if time() > t + 60 then  
    puts(1, "One minute already ... I can't wait forever!\n")  
    abort(1)  
  end if  
  sleep(5) -- let other processes run  
end while  
puts(v, "Yet another visitor\n")  
unlock_file(v, {})  
close(v)
```

**See Also:** [unlock\\_file](#), [flush](#), [sleep](#)

## lock\_memory

**Platform:** **DOS32**

**Syntax:** include machine.e

lock\_memory(a, i)

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

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

**Example Program:** [demo\dos32\hardint.ex](#)

**See Also:** [get\\_vector](#), [set\\_vector](#)



## log

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

**Description:** Return the natural logarithm of x1.

**Comments:** This function may be applied to an atom or to all elements of a sequence. Note that log is only defined for positive numbers. Your program will abort with a message if you try to take the log of a negative number or zero.

**Example:**

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

**See Also:** [sin](#), [cos](#), [tan](#), [sqrt](#)

## lower

**Syntax:**           include wildcard.e

x2 = lower(x1)

**Description:**   Convert an atom or sequence to lower case.

**Example:**

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

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

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

**See Also:**   [upper](#)

## machine\_func

**Syntax:** x1 = machine\_func(a, x)  
**Description:** see [machine\\_proc\(\)](#) below

## machine\_proc

**Syntax:** machine\_proc(a, x)

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

**See Also:** [machine\\_func](#)

## match

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

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

**Example:**

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

**See Also:** [find](#), [compare](#), [wildcard\\_match](#)

## mem\_copy

**Syntax:** `mem_copy(a1, a2, i)`

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

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

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

**Example:**

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

**See Also:** [mem\\_set](#), [peek](#), [poke](#), [allocate](#), [allocate\\_low](#)

## mem\_set

**Syntax:** `mem_set(a1, i1, i2)`

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

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

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

**Example:**

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

**See Also:** [mem\\_copy](#), [peek](#), [poke](#), [allocate](#), [allocate\\_low](#)

## message\_box

**Platform:** WIN32

**Syntax:** include msgbox.e

i = message\_box(s1, s2, x)

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

**Comments:** See [msgbox.e](#) for a complete list of possible values for x and i.

**Example:**

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

**Example Program:** [demo\win32\email.exw](#)



## mouse\_events

**Platform:** **DOS32, Linux**

**Syntax:** include mouse.e

mouse\_events(i)

**Description:** Use this procedure to select the mouse events that you want get\_mouse() to report. By default, get\_mouse() will report all events. mouse\_events() can be called at various stages of the execution of your program, as the need to detect events changes. Under Linux, mouse\_events() currently has no effect.

**Comments:** It is good practice to ignore events that you are not interested in, particularly the very frequent MOVE event, in order to reduce the chance that you will miss a significant event.

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

**Example:**

```
mouse_events(LEFT_DOWN + LEFT_UP + RIGHT_DOWN)
-- will restrict get_mouse() to reporting the left button
-- being pressed down or released, and the right button
-- being pressed down. All other events will be ignored.
```

**See Also:** [get\\_mouse](#), [mouse\\_pointer](#)

## mouse\_pointer

**Platform:** **DOS32, Linux**

**Syntax:** include mouse.e

mouse\_pointer(i)

**Description:** If i is 0 hide the mouse pointer, otherwise turn on the mouse pointer. Multiple calls to hide the pointer will require multiple calls to turn it back on. The first call to either `get_mouse()` or `mouse_events()`, will also turn the pointer on (once). Under Linux, `mouse_pointer()` currently has no effect

**Comments:** It may be necessary to hide the mouse pointer temporarily when you update the screen.

After a call to [text\\_rows\(\)](#) you may have to call `mouse_pointer(1)` to see the mouse pointer again.

**See Also:** [get\\_mouse](#), [mouse\\_events](#)

## not\_bits

**Syntax:** `x2 = not_bits(x1)`

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

**Comments:** The argument to this function may be an atom or a sequence. The rules for [operations on sequences](#) apply.

The argument must be representable as a 32-bit number, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

**Example:**

```
a = not_bits(#000000F7)
-- a is -248 (i.e. FFFFFFF08 interpreted as a negative number)
```

**See Also:** [and\\_bits](#), [or\\_bits](#), [xor\\_bits](#), [int\\_to\\_bits](#)

## object

**Syntax:** `i = object(x)`

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

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

**Example:**

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

**See Also:** [integer](#), [atom](#), [sequence](#)

## open

**Syntax:** fn = open(st1, st2)

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

"r" - open text file for reading  
"rb" - open binary file for reading  
"w" - create text file for writing  
"wb" - create binary file for writing  
"u" - open text file for update (reading and writing)  
"ub" - open binary file for update  
"a" - open text file for appending  
"ab" - open binary file for appending

Files opened for read or update must already exist. Files opened for write or append will be created if necessary. A file opened for write will be set to 0 bytes. Output to a file opened for append will start at the end of file.

Output to **text files** will have carriage-return characters automatically added before linefeed characters. On input, these carriage-return characters are removed. A control-Z character (ASCII 26) will signal an immediate end of file. Note: on some versions of DOS, a control-Z typed by the user might cause standard input to permanently appear to be at the end-of-file, until the DOS window is closed.

I/O to **binary files** is not modified in any way. Any byte values from 0 to 255 can be read or written.

Some typical devices that you can open on DOS or 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

Currently, files up to 2 Gb in size can be handled. Beyond that, some file operations may not work correctly. This limit will likely be increased in the future.

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

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

**DOS32:** Be careful not to use the special device names in a file name, even if you add an extension. e.g. CON.TXT, CON.DAT, CON.JPG etc. all refer to the CON device, not a file.

**Example:**

```
integer file_num, file_num95
```

```

sequence first_line
constant ERROR = 2

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

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

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

```

**See Also:**    [close](#), [lock\\_file](#)

## open\_dll

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

a = open\_dll(st)

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

**Comments:** The value returned by open\_dll() can be passed to define\_c\_proc(), define\_c\_func(), or define\_c\_var().

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

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

**Example:**

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

**See Also:** [define\\_c\\_func](#), [define\\_c\\_proc](#), [define\\_c\\_var](#), [c\\_func](#), [c\\_proc](#), [platform.doc](#)

## or\_bits

**Syntax:** x3 = or\_bits(x1, x2)

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

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

**Example 1:**

```
a = or_bits(#0F0F0000, #12345678)
-- a is #1F3F5678
```

**Example 2:**

```
a = or_bits(#FF, {#123456, #876543, #2211})
-- a is {#1234FF, #8765FF, #22FF}
```

**See Also:** [and\\_bits](#), [xor\\_bits](#), [not\\_bits](#), [int\\_to\\_bits](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)



## palette

**Platform:** **DOS32**

**Syntax:** include graphics.e

x = palette(i, s)

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

**Example:**

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

**See Also:** [all\\_palette](#)

## peek

**Syntax:** i = peek(a)

or ...

s = peek({a, i})

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

**Comments:** Since addresses are 32-bit numbers, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as **atoms**.

It is faster to read several bytes at once using the second form of peek() than it is to read one byte at a time in a loop.

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

**Example:** The following are equivalent:

```
-- method 1  
s = {peek(100), peek(101), peek(102), peek(103)}  
  
-- method 2  
s = peek({100, 4})
```

**See Also:** [poke](#), [peek4s](#), [peek4u](#), [allocate](#), [free](#), [allocate\\_low](#), [free\\_low](#), [call](#)

## peek4s

**Syntax:** a2 = peek4s(a1)

or ...

s = peek4s({a1, i})

**Description:** Return a 4-byte (32-bit) signed value in the range -2147483648 to +2147483647 from

machine address a1, or return a sequence containing i consecutive 4-byte signed values starting at address a1 in memory.

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

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

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

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

**Example:** The following are equivalent:

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

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

**See Also:** [peek4u](#), [peek](#), [poke4](#), [allocate](#), [free](#), [allocate\\_low](#), [free\\_low](#), [call](#)

## peek4u

**Syntax:** a2 = peek4u(a1)

or ...

s = peek4u({a1, i})

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

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

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

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

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

**Example:** The following are equivalent:

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

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

**See Also:** [peek4s](#), [peek](#), [poke4](#), [allocate](#), [free](#), [allocate\\_low](#), [free\\_low](#), [call](#)

## PI

**Syntax:** include misc.e

PI

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

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

**Example:**

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

**See Also:** [sin](#), [cos](#), [tan](#)

## pixel

**Platform:** DOS32

**Syntax:** pixel(x1, s)

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

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

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

Any off-screen pixels will be safely clipped.

**Example 1:**

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

**Example 2:**

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

**See Also:** [get\\_pixel](#), [graphics\\_mode](#)

## platform

**Syntax:** i = platform()

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

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

The include file [misc.e](#) contains the following constants:

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

Use `platform()` when you want to execute different code depending on which platform the program is running on.

Additional platforms will be added as Euphoria is ported to new machines and operating environments.

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

#### Example 1:

```
    if platform() = WIN32 then
        -- call system Beep routine
        err = c_func(Beep, {0,0})
    elsif platform() = DOS32 then
        -- make beep
        sound(500)
        t = time()
        while time() < t + 0.5 do
            end while
        sound(0)
    else
        -- do nothing (Linux/FreeBSD)
    end if
```

See Also: [platform.doc](#)

## poke

**Syntax:** `poke(a, x)`

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

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

**Comments:** The lower 8-bits of each byte value, i.e. **remainder(x, 256)**, is actually stored in memory.

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

Writing to the screen memory with `poke()` can be much faster than using `puts()` or `printf()`, but the programming is more difficult. In most cases the speed is not needed. For example, the Euphoria editor, **ed**, never uses `poke()`.

#### Example:

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

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

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

**Example Program:** [demo\callmach.ex](#)

**See Also:** [peek](#), [poke4](#), [allocate](#), [free](#), [allocate\\_low](#), [free\\_low](#), [call](#), [safe.e](#)

## poke4

**Syntax:** `poke4(a, x)`

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

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

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

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

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

**Example:**

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

```
-- poke one 4-byte value at a time:
poke4(a, 9712345)
poke4(a+4, #FF00FF00)
poke4(a+8, -12345)

-- poke 3 4-byte values at once:
poke4(a, {9712345, #FF00FF00, -12345})
```

**See Also:** [peek4u](#), [peek4s](#), [poke](#), [allocate](#), [allocate\\_low](#), [call](#)

## polygon

**Platform:** **DOS32**

**Syntax:** `include graphics.e`

`polygon(i1, i2, s)`

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

**Example:**

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

**See Also:** [draw\\_line](#), [ellipse](#)

## position

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

**Description:** Set the cursor to line i1, column i2, where the top left corner of the screen is line 1, column 1.

The next character displayed on the screen will be printed at this location. `position()` will report an error if the location is off the screen.

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

The coordinate system for displaying text is different from the one for displaying pixels. Pixels are displayed such that the top-left is (x=0,y=0) and the first coordinate controls the horizontal, left-right location. In **pixel-graphics modes** you can display both text and pixels. `position()` only sets the line and column for the text that you display, not the pixels that you plot. There is no corresponding routine for setting the next pixel position.

**Example:**

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

**See Also:** [get\\_position](#), [puts](#), [print](#), [printf](#)

## power

**Syntax:** `x3 = power(x1, x2)`

**Description:** Raise `x1` to the power `x2`

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

Powers of 2 are calculated very efficiently.

**Example 1:**

```
? power(5, 2)
-- 25 is printed
```

**Example 2:**

```
? power({5, 4, 3.5}, {2, 1, -0.5})
-- {25, 4, 0.534522} is printed
```

**Example 3:**

```
? power(2, {1, 2, 3, 4})
-- {2, 4, 8, 16}
```

**Example 4:**

```
? power({1, 2, 3, 4}, 2)
-- {1, 4, 9, 16}
```

**See Also:** [log](#), [sqrt](#)

## prepend

**Syntax:** `s2 = prepend(s1, x)`

**Description:** Prepend `x` to the start of sequence `s1`. The length of `s2` will be [length\(s1\)](#) + 1.

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

The case where `s1` and `s2` are the same variable is handled very efficiently.

**Example 1:**

```
prepend({1,2,3}, {0,0})    -- {{0,0}, 1, 2, 3}

-- Compare with concatenation:

{0,0} & {1,2,3}           -- {0, 0, 1, 2, 3}
```

**Example 2:**

```
s = {}
for i = 1 to 10 do
    s = prepend(s, i)
end for
-- s is {10,9,8,7,6,5,4,3,2,1}
```

**See Also:**    [append](#), [concatenation operator &](#), [sequence-formation operator](#)

## pretty\_print

**Syntax:**       include misc.e

pretty\_print(fn, x, s)

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

Several options may be supplied in s to control the presentation. Pass { } to select the defaults, or set options as below:

[1] display ASCII characters:

\* 0: never

\* 1: alongside any integers in the printable ASCII range 32..127 (default)

\* 2: like 1, plus display as "string" when all integers of a sequence are in the printable ASCII range

\* 3: like 2, but show *\*only\** quoted characters, not numbers, for any integers in the printable ASCII range, as well as the whitespace 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

If the length of s is less than 8, unspecified options at the end of the sequence will keep the default values. e.g. {0, 5} will choose "never display ASCII", plus 5-character indentation, with defaults for everything else.

**Comments:**    The display will start at the current cursor position. Normally you will want to call pretty\_print() when the cursor is in column 1 (after printing a \n character). If you want to start in a different column, you should call position() and specify a value for option [3]. This will ensure that the first and last braces in a sequence line up vertically.

When specifying the format to use for integers and floating-point numbers, you can add some decoration,

e.g. "(%d)" or "\$ %.2f"

#### Example 1:

```
pretty_print(1, "ABC", {})
```

```
{65'A',66'B',67'C'}
```

#### Example 2:

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

```
{
  {1,2,3},
  {4,5,6}
}
```

#### Example 3:

```
pretty_print(1, {"Euphoria", "Programming", "Language"}, {2})
```

```
{
  "Euphoria",
  "Programming",
  "Language"
}
```

#### Example 4:

```
puts(1, "word_list = ") -- moves cursor to column 13
pretty_print(1,
  {"Euphoria", 8, 5.3},
  {"Programming", 11, -2.9},
  {"Language", 8, 9.8}},
  {2, 4, 13, 78, "%03d", "%.3f"}) -- first 6 of 8 options

word_list = {
  {
    "Euphoria",
    008,
    5.300
  },
  {
    "Programming",
    011,
    -2.900
  },
  {
    "Language",
    008,
    9.800
  }
}
```

**See Also:** [?](#), [print](#), [puts](#), [printf](#)

## print

**Syntax:** print(fn, x)



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

**Example 1:**

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

**Example 2:**

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

**See Also:** [?](#), [pretty\\_print](#), [puts](#), [printf](#), [get](#)

## printf

**Syntax:** printf(fn, st, x)

**Description:** Print x, to file or device fn, using format string st. If x is a sequence, then format specifiers from st are matched with corresponding elements of x. If x is an atom, then normally st will contain just one format specifier and it will be applied to x, however if st contains multiple format specifiers, each one will be applied to the same value x. Thus printf() always takes exactly 3 arguments. Only the length of the last argument, containing the values to be printed, will vary. The basic format specifiers are:

%d - print an atom as a decimal integer

%x - print an atom as a hexadecimal integer. Negative numbers are printed in two's complement, so -1 will print as FFFFFFFF

%o - print an atom as an octal integer

%s - print a sequence as a string of characters, or print an atom as a single character

%e - print an atom as a floating-point number with exponential notation

%f - print an atom as a floating-point number with a decimal point but no exponent

%g - print an atom as a floating-point number using whichever format seems appropriate, given the magnitude of the number

%% - print the '%' character itself

Field widths can be added to the basic formats, e.g. %5d, %8.2f, %10.4s. The number before the decimal point is the minimum field width to be used. The number after the decimal point is the precision to be used.

If the field width is negative, e.g. %-5d then the value will be left-justified within the field. Normally it will be right-justified. If the field width starts with a leading 0, e.g. %08d then leading zeros will be supplied to fill up the field. If the field width starts with a '+' e.g. %+7d then a plus sign will be printed for positive values.

**Comments:** Watch out for the following common mistake:

```
name="John Smith"
printf(1, "%s", name) -- error!
```

This will print only the first character, J, of name, as each element of name is taken to be a separate value to be formatted. You must say this instead:

```
name="John Smith"
printf(1, "%s", {name}) -- correct
```

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

**Example 1:**

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

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

### Example 2:

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

John Smith,      97
```

### Example 3:

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

ABCD           $ XXX
```

### Example 4:

```
printf(1, "%d %e %f %g", 7.75) -- same value in different formats

7  7.750000e+000  7.750000  7.75
```

See Also: [sprintf](#), [puts](#), [open](#)

## profile

**Syntax:** `profile(i)`

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

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

### Example 1:

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

See Also: [trace](#), [profiling](#), [special top-level statements](#)

## prompt\_number

**Syntax:** `include get.e`

`a = prompt_number(st, s)`

**Description:** Prompt the user to enter a number. *st* is a string of text that will be displayed on the screen. *s* is a sequence of two values {lower, upper} which determine the range of values that the user may enter. If the user enters a number that is less than lower or greater than upper, he will be prompted again. *s* can be

[empty](#), {}, if there are no restrictions.

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

**Example 1:**

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

**Example 2:**

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

**See Also:** [get](#), [prompt\\_string](#)

## prompt\_string

**Syntax:** include get.e

s = prompt\_string(st)

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

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

**Example:**

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

**See Also:** [gets](#), [prompt\\_number](#)

## put\_screen\_char

**Syntax:** include image.e

put\_screen\_char(i1, i2, s)

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

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

It's faster to write several characters to the screen with a single call to put\_screen\_char() than it is to write one character at a time.

**Example:**

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

**See Also:** [get\\_screen\\_char](#), [display\\_text\\_image](#)

## puts

**Syntax:** puts(fn, x)

**Description:** Output, to file or device fn, a single byte (atom) or sequence of bytes. The low order 8-bits

of each value is actually sent out. If `fn` is the screen you will see text characters displayed.

**Comments:** When you output a sequence of bytes it must not have any (sub)sequences within it. It must be a **sequence of atoms** only. (Typically a string of ASCII codes).

Avoid outputting 0's to the screen or to standard output. Your output might get truncated.

Remember that if the output file was opened in text mode, DOS and Windows will change `\n` (10) to `\r\n` (13 10). Open the file in binary mode if this is not what you want.

**Example 1:**

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

**Example 2:**

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

**See Also:** [printf](#), [gets](#), [open](#)

## rand

**Syntax:** `x2 = rand(x1)`

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

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

**Example:**

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

**See Also:** [set\\_rand](#)

## read\_bitmap

**Syntax:** `include image.e`

`x = read_bitmap(st)`

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

Bitmaps of 2, 4, 16 or 256 colors are supported. If the file is not in a good format, an error code (atom) is returned instead:

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

**Comments:** You can create your own bitmap picture files using Windows Paintbrush and many other graphics programs. You can then incorporate these pictures into your Euphoria programs.

**Example:**

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

-- a string

**Example Program:** [demo\dos32\bitmap.ex](#)

**See Also:** [palette](#), [all\\_palette](#), [display\\_image](#), [save\\_bitmap](#)

## register\_block

**Syntax:** include machine.e (or safe.e)

register\_block(a, i)

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

**Comments:** This routine is only meant to be used for **debugging purposes**. [safe.e](#) tracks the blocks of memory that your program is allowed to [peek\(\)](#), [poke\(\)](#), [mem\\_copy\(\)](#) etc. These are normally just the blocks that you have allocated using Euphoria's [allocate\(\)](#) or [allocate\\_low\(\)](#) routines, and which you have not yet freed using Euphoria's [free\(\)](#) or [free\\_low\(\)](#). In some cases, you may acquire additional, external, blocks of memory, perhaps as a result of calling a C routine. If you are debugging your program using [safe.e](#), you must register these external blocks of memory or [safe.e](#) will prevent you from accessing them. When you are finished using an external block you can unregister it using [unregister\\_block\(\)](#).

When you include [machine.e](#), you'll get different versions of [register\\_block\(\)](#) and [unregister\\_block\(\)](#) that do nothing. This makes it easy to switch back and forth between debug and non-debug runs of your program.

**Example 1:**

```
atom addr
```

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

**See Also:** [unregister\\_block](#), [safe.e](#)

## remainder

**Syntax:** x3 = remainder(x1, x2)

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

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

**Example 1:**

```
a = remainder(9, 4)
```

```
-- a is 1
```

**Example 2:**

```
s = remainder({81, -3.5, -9, 5.5}, {8, -1.7, 2, -4})
```

```
-- s is {1, -0.1, -1, 1.5}
```

**Example 3:**

```
s = remainder({17, 12, 34}, 16)
```

```
-- s is {1, 12, 2}
```

**Example 4:**

```
s = remainder(16, {2, 3, 5})
-- s is {0, 1, 1}
```

See Also: [floor](#)

## repeat

**Syntax:** s = repeat(x, a)

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

**Comments:** When you repeat a sequence or a floating-point number the interpreter does not actually make multiple copies in memory. Rather, a single copy is "pointed to" a number of times.

**Example 1:**

```
repeat(0, 10)      -- {0,0,0,0,0,0,0,0,0,0}
```

**Example 2:**

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

See Also: [append](#), [prepend](#), [sequence-formation operator](#)

## reverse

**Syntax:** include misc.e

s2 = reverse(s1)

**Description:** Reverse the order of elements in a sequence.

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

**Example 1:**

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

**Example 2:**

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

**Example 3:**

```
reverse({99})      -- {99}
```

**Example 4:**

```
reverse({})        -- {}
```

See Also: [append](#), [prepend](#), [repeat](#)

## routine\_id

**Syntax:** i = routine\_id(st)

**Description:** Return an integer id number, known as a **routine id**, for a user-defined Euphoria procedure or function. The name of the procedure or function is given by the string sequence st. -1 is returned if the named routine can't be found.

**Comments:** The id number can be passed to call\_proc() or call\_func(), to indirectly call the routine named by st.

The routine named by st must be visible, i.e. callable, at the place where routine\_id() is used to get the id number. Indirect calls to the routine can appear earlier in the program than the definition of the routine, but the id number can only be obtained in code that comes *after* the definition of the routine - see example 2 below.

Once obtained, a valid **routine id** can be used at *any* place in the program to call a routine indirectly via call\_proc()/call\_func().

Some typical uses of routine\_id() are:

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

Note that C routines, callable by Euphoria, also have routine id's. See define\_c\_proc() and define\_c\_func().

#### Example 1:

```
procedure foo()
    puts(1, "Hello World\n")
end procedure

integer foo_num
foo_num = routine_id("foo")

call_proc(foo_num, {}) -- same as calling foo()
```

#### Example 2:

```
function apply_to_all(sequence s, integer f)
    -- apply a function to all elements of a sequence
    sequence result
    result = {}
    for i = 1 to length(s) do
        -- we can call add1() here although it comes later in the program
        result = append(result, call_func(f, {s[i]}))
    end for
    return result
end function

function add1(atom x)
    return x + 1
end function

-- add1() is visible here, so we can ask for its routine id
? apply_to_all({1, 2, 3}, routine_id("add1"))
-- displays {2,3,4}
```

**See Also:**    [call\\_proc](#), [call\\_func](#), [call\\_back](#), [define\\_c\\_func](#), [define\\_c\\_proc](#), [platform.doc](#)

... [continue](#)

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)



## palette

**Platform:** **DOS32**

**Syntax:** include graphics.e

x = palette(i, s)

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

**Example:**

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

**See Also:** [all\\_palette](#)

## peek

**Syntax:** `i = peek(a)`

or ...

`s = peek({a, i})`

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

**Comments:** Since addresses are 32-bit numbers, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as **atoms**.

It is faster to read several bytes at once using the second form of `peek()` than it is to read one byte at a time in a loop.

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

**Example:** The following are equivalent:

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

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

**See Also:** [poke](#), [peek4s](#), [peek4u](#), [allocate](#), [free](#), [allocate\\_low](#), [free\\_low](#), [call](#)

## peek4s

**Syntax:**        a2 = peek4s(a1)

or ...

s = peek4s({a1, i})

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

**Comments:**      The 32-bit values returned by peek4s() may be too large for the Euphoria integer type (31-bits), so you should use **atom** variables.

Since machine addresses are 32-bit numbers, they can also be too large for Euphoria's integer type.

Variables that hold an address should therefore be declared as **atoms**.

It is faster to read several 4-byte values at once using the second form of peek4s() than it is to read one 4-byte value at a time in a loop.

Remember that peek4s() takes just one argument, which in the second form is actually a 2-element sequence.

**Example:**        The following are equivalent:

```
-- method 1
s = {peek4s(100), peek4s(104), peek4s(108), peek4s(112)}
```

```
-- method 2
s = peek4s({100, 4})
```

**See Also:**        [peek4u](#), [peek](#), [poke4](#), [allocate](#), [free](#), [allocate\\_low](#), [free\\_low](#), [call](#)

## peek4u

**Syntax:**        a2 = peek4u(a1)

or ...

s = peek4u({a1, i})

**Description:**   Return a 4-byte (32-bit) unsigned value in the range 0 to 4294967295 from machine address a1, or return a sequence containing i consecutive 4-byte unsigned values starting at address a1 in memory.

**Comments:**     The 32-bit values returned by peek4u() may be too large for the Euphoria integer type (31-bits), so you should use **atom** variables.

Since machine addresses are 32-bit numbers, they can also be too large for Euphoria's integer type. Variables that hold an address should therefore be declared as **atoms**.

It is faster to read several 4-byte values at once using the second form of peek4u() than it is to read one 4-byte value at a time in a loop.

Remember that peek4u() takes just one argument, which in the second form is actually a 2-element sequence.

**Example:**        The following are equivalent:

```
-- method 1
s = {peek4u(100), peek4u(104), peek4u(108), peek4u(112)}
```

```
-- method 2
s = peek4u({100, 4})
```

**See Also:**        [peek4s](#), [peek](#), [poke4](#), [allocate](#), [free](#), [allocate\\_low](#), [free\\_low](#), [call](#)

# PI

**Syntax:** include misc.e

PI

**Description:** PI (3.14159...) has been defined as a global constant.

**Comments:** Enough digits have been used to attain the maximum accuracy possible for a Euphoria atom.

**Example:**

```
x = PI  -- x is 3.14159...
```

**See Also:** [sin](#), [cos](#), [tan](#)

# pixel

**Platform:** DOS32

**Syntax:** pixel(x1, s)

**Description:** Set one or more pixels on a [pixel-graphics](#) screen starting at point s, where s is a 2-element screen coordinate {x, y}. If x1 is an atom, one pixel will be set to the color indicated by x1. If x1 is a sequence then a number of pixels will be set, starting at s and moving to the right (increasing x value, same y value).

**Comments:** When x1 is a sequence, a very fast algorithm is used to put the pixels on the screen. It is much faster to call pixel() once, with a sequence of pixel colors, than it is to call it many times, plotting one pixel color at a time.

In graphics mode 19, pixel() is highly optimized.

Any off-screen pixels will be safely clipped.

## Example 1:

```
pixel(BLUE, {50, 60})  
-- the point {50,60} is set to the color BLUE
```

## Example 2:

```
pixel({BLUE, GREEN, WHITE, RED}, {50,60})  
-- {50,60} set to BLUE  
-- {51,60} set to GREEN  
-- {52,60} set to WHITE  
-- {53,60} set to RED
```

**See Also:** [get\\_pixel](#), [graphics\\_mode](#)

## platform

**Syntax:** `i = platform()`

**Description:** `platform()` is a function built-in to the interpreter. It indicates the platform that the program is being executed on: **DOS32**, **WIN32**, **Linux** or **FreeBSD**.

**Comments:** When **ex.exe** is running, the platform is DOS32. When **exw.exe** is running the platform is WIN32. When **exu** is running the platform is LINUX (or FreeBSD).

The include file **misc.e** contains the following constants:

```
global constant DOS32 = 1,
                WIN32 = 2,
                LINUX = 3,
                FREEBSD = 3
```

Use `platform()` when you want to execute different code depending on which platform the program is running on.

Additional platforms will be added as Euphoria is ported to new machines and operating environments.

The call to `platform()` costs nothing. It is optimized at compile-time into the appropriate integer value: 1, 2 or 3.

### Example 1:

```
        if platform() = WIN32 then
            -- call system Beep routine
            err = c_func(Beep, {0,0})
elseif platform() = DOS32 then
            -- make beep
            sound(500)
            t = time()
            while time() < t + 0.5 do
            end while
            sound(0)
else
            -- do nothing (Linux/FreeBSD)
end if
```

**See Also:** [platform.doc](#)

## poke

**Syntax:**        `poke(a, x)`

**Description:**    If `x` is an atom, write a single byte value to memory address `a`.

If `x` is a sequence, write a sequence of byte values to consecutive memory locations starting at location `a`.

**Comments:**     The lower 8-bits of each byte value, i.e. `remainder(x, 256)`, is actually stored in memory.

It is faster to write several bytes at once by poking a sequence of values, than it is to write one byte at a time in a loop.

Writing to the screen memory with `poke()` can be much faster than using `puts()` or `printf()`, but the programming is more difficult. In most cases the speed is not needed. For example, the Euphoria editor, **ed**, never uses `poke()`.

**Example:**

```
a = allocate(100)     -- allocate 100 bytes in memory

-- poke one byte at a time:
poke(a, 97)
poke(a+1, 98)
poke(a+2, 99)

-- poke 3 bytes at once:
poke(a, {97, 98, 99})
```

**Example Program:**    [demo\callmach.ex](#)

**See Also:**        [peek](#), [poke4](#), [allocate](#), [free](#), [allocate\\_low](#), [free\\_low](#), [call](#), [safe.e](#)



## poke4

**Syntax:**        `poke4(a, x)`

**Description:**    If `x` is an atom, write a 4-byte (32-bit) value to memory address `a`.

If `x` is a sequence, write a sequence of 4-byte values to consecutive memory locations starting at location `a`.

**Comments:**     The value or values to be stored must not exceed 32-bits in size.

It is faster to write several 4-byte values at once by poking a sequence of values, than it is to write one 4-byte value at a time in a loop.

The 4-byte values to be stored can be negative or positive. You can read them back with either `peek4s()` or `peek4u()`.

**Example:**

```
a = allocate(100)    -- allocate 100 bytes in memory
```

```
-- poke one 4-byte value at a time:
```

```
poke4(a, 9712345)
```

```
poke4(a+4, #FF00FF00)
```

```
poke4(a+8, -12345)
```

```
-- poke 3 4-byte values at once:
```

```
poke4(a, {9712345, #FF00FF00, -12345})
```

**See Also:**     [peek4u](#), [peek4s](#), [poke](#), [allocate](#), [allocate\\_low](#), [call](#)

# polygon

**Platform:** **DOS32**

**Syntax:** include graphics.e

polygon(i1, i2, s)

**Description:** Draw a polygon with 3 or more vertices given in s, on a **pixel-graphics** screen using a certain color i1. Fill the area if i2 is 1. Don't fill if i2 is 0.

**Example:**

```
    polygon(GREEN, 1, {{100, 100}, {200, 200}, {900, 700}})
-- makes a solid green triangle.
```

**See Also:** [draw\\_line](#), [ellipse](#)

## position

**Syntax:**        `position(i1, i2)`

**Description:**    Set the cursor to line i1, column i2, where the top left corner of the screen is line 1, column 1. The next character displayed on the screen will be printed at this location. `position()` will report an error if the location is off the screen.

**Comments:**      `position()` works in both **text and pixel-graphics modes**.

The coordinate system for displaying text is different from the one for displaying pixels. Pixels are displayed such that the top-left is (x=0,y=0) and the first coordinate controls the horizontal, left-right location. In **pixel-graphics modes** you can display both text and pixels. `position()` only sets the line and column for the text that you display, not the pixels that you plot. There is no corresponding routine for setting the next pixel position.

**Example:**

```
        position(2,1)
-- the cursor moves to the beginning of the second line from
-- the top
```

**See Also:**      [get\\_position](#), [puts](#), [print](#), [printf](#)

## power

**Syntax:** x3 = power(x1, x2)

**Description:** Raise x1 to the power x2

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

Powers of 2 are calculated very efficiently.

### Example 1:

```
? power(5, 2)
-- 25 is printed
```

### Example 2:

```
? power({5, 4, 3.5}, {2, 1, -0.5})
-- {25, 4, 0.534522} is printed
```

### Example 3:

```
? power(2, {1, 2, 3, 4})
-- {2, 4, 8, 16}
```

### Example 4:

```
? power({1, 2, 3, 4}, 2)
-- {1, 4, 9, 16}
```

**See Also:** [log](#), [sqrt](#)

## prepend

**Syntax:** `s2 = prepend(s1, x)`

**Description:** Prepend x to the start of sequence s1. The length of s2 will be [length\(s1\)](#) + 1.

**Comments:** If x is an atom this is the same as **s2 = x & s1**. If x is a sequence it is definitely not the same.

The case where s1 and s2 are the same variable is handled very efficiently.

**Example 1:**

```
prepend({1,2,3}, {0,0})  -- {{0,0}, 1, 2, 3}
```

-- Compare with concatenation:

```
{0,0} & {1,2,3}          -- {0, 0, 1, 2, 3}
```

**Example 2:**

```
s = {}  
for i = 1 to 10 do  
  s = prepend(s, i)  
end for  
-- s is {10,9,8,7,6,5,4,3,2,1}
```

**See Also:** [append](#), [concatenation operator &](#), [sequence-formation operator](#)

## pretty\_print

**Syntax:** include misc.e

pretty\_print(fn, x, s)

**Description:** Print, to file or device fn, an object x, using braces { , , , }, indentation, and multiple lines to show the structure.

Several options may be supplied in s to control the presentation. Pass {} to select the defaults, or set options as below:

[1] display ASCII characters:

\* 0: never

\* 1: alongside any integers in the printable ASCII range 32..127 (default)

\* 2: like 1, plus display as "string" when all integers of a sequence are in the printable ASCII range

\* 3: like 2, but show \*only\* quoted characters, not numbers, for any integers in the printable ASCII range, as well as the whitespace 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

If the length of s is less than 8, unspecified options at the end of the sequence will keep the default values. e.g. {0, 5} will choose "never display ASCII", plus 5-character indentation, with defaults for everything else.

**Comments:** The display will start at the current cursor position. Normally you will want to call pretty\_print() when the cursor is in column 1 (after printing a \n character). If you want to start in a different column, you should call position() and specify a value for option [3]. This will ensure that the first and last braces in a sequence line up vertically.

When specifying the format to use for integers and floating-point numbers, you can add some decoration, e.g. "(%d)" or "\$ %.2f"

### Example 1:

```
pretty_print(1, "ABC", {})
```

```
{65'A',66'B',67'C'}
```

### Example 2:

```
pretty_print(1, {{1,2,3}, {4,5,6}}, {})
```

```
{
  {1,2,3},
  {4,5,6}
}
```

### Example 3:

```
pretty_print(1, {"Euphoria", "Programming", "Language"}, {2})
```

```
{
  "Euphoria",
```

```

    "Programming",
    "Language"
}

```

#### Example 4:

```

    puts(1, "word_list = ") -- moves cursor to column 13
pretty_print(1,
    {"Euphoria", 8, 5.3},
    {"Programming", 11, -2.9},
    {"Language", 8, 9.8}},
    {2, 4, 13, 78, "%03d", "%.3f"}) -- first 6 of 8 options

word_list = {
    {
        "Euphoria",
        008,
        5.300
    },
    {
        "Programming",
        011,
        -2.900
    },
    {
        "Language",
        008,
        9.800
    }
}

```

**See Also:**    [2](#), [print](#), [puts](#), [printf](#)

## print

**Syntax:** `print(fn, x)`

**Description:** Print, to file or device `fn`, an object `x` with braces `{ , , }` to show the structure.

**Example 1:**

```
print(1, "ABC")  -- output is: {65, 66, 67}
puts(1, "ABC")   -- output is: ABC
```

**Example 2:**

```
print(1, repeat({10,20}, 3))
-- output is: {{10,20},{10,20},{10,20}}
```

**See Also:** [?](#), [pretty\\_print](#), [puts](#), [printf](#), [get](#)



# printf

**Syntax:** `printf(fn, st, x)`

**Description:** Print x, to file or device fn, using format string st. If x is a sequence, then format specifiers from st are matched with corresponding elements of x. If x is an atom, then normally st will contain just one format specifier and it will be applied to x, however if st contains multiple format specifiers, each one will be applied to the same value x. Thus printf() always takes exactly 3 arguments. Only the length of the last argument, containing the values to be printed, will vary. The basic format specifiers are:

%d - print an atom as a decimal integer

%x - print an atom as a hexadecimal integer. Negative numbers are printed in two's complement, so -1 will print as FFFFFFFF

%o - print an atom as an octal integer

%s - print a sequence as a string of characters, or print an atom as a single character

%e - print an atom as a floating-point number with exponential notation

%f - print an atom as a floating-point number with a decimal point but no exponent

%g - print an atom as a floating-point number using whichever format seems appropriate, given the magnitude of the number

%% - print the '%' character itself

Field widths can be added to the basic formats, e.g. %5d, %8.2f, %10.4s. The number before the decimal point is the minimum field width to be used. The number after the decimal point is the precision to be used.

If the field width is negative, e.g. %-5d then the value will be left-justified within the field. Normally it will be right-justified. If the field width starts with a leading 0, e.g. %08d then leading zeros will be supplied to fill up the field. If the field width starts with a '+' e.g. %+7d then a plus sign will be printed for positive values.

**Comments:** Watch out for the following common mistake:

```
name="John Smith"
printf(1, "%s", name)      -- error!
```

This will print only the first character, J, of name, as each element of name is taken to be a separate value to be formatted. You must say this instead:

```
name="John Smith"
printf(1, "%s", {name})    -- correct
```

Now, the third argument of printf() is a one-element sequence containing the item to be formatted.

**Example 1:**

```
rate = 7.875
printf(myfile, "The interest rate is: %8.2f\n", rate)
```

```
The interest rate is:      7.88
```

**Example 2:**

```
name="John Smith"
score=97
printf(1, "%15s, %5d\n", {name, score})
```

```
John Smith,      97
```

**Example 3:**

```
printf(1, "%-10.4s $ %s", {"ABCDEFGHJKLMNOP", "XXX"})
```

ABCD            \$ XXX

**Example 4:**

```
printf(1, "%d %e %f %g", 7.75) -- same value in different formats
```

```
7 7.750000e+000 7.750000 7.75
```

**See Also:**    [sprintf](#), [puts](#), [open](#)

## profile

**Syntax:**        `profile(i)`

**Description:**    Enable or disable profiling at run-time. This works for both **execution-count** and **time-profiling**. If *i* is 1 then profiling will be enabled, and samples/counts will be recorded. If *i* is 0 then profiling will be disabled and samples/counts will not be recorded.

**Comments:**     After a "**with profile**" or "**with profile\_time**" statement, profiling is turned on automatically. Use `profile(0)` to turn it off. Use `profile(1)` to turn it back on when execution reaches the code that you wish to focus the profile on.

**Example 1:**

```
          with profile_time
profile(0)
...
procedure slow_routine()
profile(1)
...
profile(0)
end procedure
```

**See Also:**     [trace](#), [profiling](#), [special top-level statements](#)

## prompt\_number

**Syntax:**           include get.e

a = prompt\_number(st, s)

**Description:**   Prompt the user to enter a number. st is a string of text that will be displayed on the screen. s is a sequence of two values {lower, upper} which determine the range of values that the user may enter. If the user enters a number that is less than lower or greater than upper, he will be prompted again. s can be [empty](#), {}, if there are no restrictions.

**Comments:**     If this routine is too simple for your needs, feel free to copy it and make your own more specialized version.

**Example 1:**

```
age = prompt_number("What is your age? ", {0, 150})
```

**Example 2:**

```
t = prompt_number("Enter a temperature in Celcius:\n", {})
```

**See Also:**     [get](#), [prompt\\_string](#)

## prompt\_string

**Syntax:**           include get.e

s = prompt\_string(st)

**Description:**   Prompt the user to enter a string of text. st is a string that will be displayed on the screen. The string that the user types will be returned as a sequence, minus any new-line character.

**Comments:**     If the user happens to type control-Z (indicates end-of-file), "" will be returned.

**Example:**

```
name = prompt_string("What is your name? ")
```

**See Also:**     [gets](#), [prompt\\_number](#)

## put\_screen\_char

**Syntax:**       include image.e

put\_screen\_char(i1, i2, s)

**Description:** Write zero or more characters onto the screen along with their attributes. i1 specifies the line, and i2 specifies the column where the first character should be written. The sequence s looks like: {ascii-code1, attribute1, ascii-code2, attribute2, ...}. Each pair of elements in s describes one character. The ascii-code atom contains the ASCII code of the character. The attributes atom contains the foreground color, background color, and possibly other platform-dependent information controlling how the character is displayed on the screen.

**Comments:**     The length of s must be a multiple of 2. If s has 0 length, nothing will be written to the screen.

It's faster to write several characters to the screen with a single call to put\_screen\_char() than it is to write one character at a time.

**Example:**

```
-- write AZ to the top left of the screen
-- (attributes are platform-dependent)
put_screen_char(1, 1, {'A', 152, 'Z', 131})
```

**See Also:**     [get\\_screen\\_char](#), [display\\_text\\_image](#)

## puts

**Syntax:** puts(fn, x)

**Description:** Output, to file or device fn, a single byte (atom) or sequence of bytes. The low order 8-bits of each value is actually sent out. If fn is the screen you will see text characters displayed.

**Comments:** When you output a sequence of bytes it must not have any (sub)sequences within it. It must be a **sequence of atoms** only. (Typically a string of ASCII codes).

Avoid outputting 0's to the screen or to standard output. Your output might get truncated.

Remember that if the output file was opened in text mode, DOS and Windows will change \n (10) to \r\n (13 10). Open the file in binary mode if this is not what you want.

**Example 1:**

```
puts(SCREEN, "Enter your first name: ")
```

**Example 2:**

```
puts(output, 'A') -- the single byte 65 will be sent to output
```

**See Also:** [printf](#), [gets](#), [open](#)

## rand

**Syntax:** `x2 = rand(x1)`

**Description:** Return a random integer from 1 to x1, where x1 may be from 1 to the largest positive value of type integer (1073741823).

**Comments:** This function may be applied to an atom or to all elements of a sequence.

**Example:**

```
s = rand({10, 20, 30})  
-- s might be: {5, 17, 23} or {9, 3, 12} etc.
```

**See Also:** [set\\_rand](#)



## read\_bitmap

**Syntax:**       include image.e

x = read\_bitmap(st)

**Description:**   st is the name of a .bmp "bitmap" file. The file should be in the bitmap format. The most common variations of the format are supported. If the file is read successfully the result will be a 2-element sequence. The first element is the palette, containing intensity values in the range 0 to 255. The second element is a 2-d sequence of sequences containing a pixel-graphics image. You can pass the palette to all\_palette() (after dividing it by 4 to scale it). The image can be passed to display\_image().

Bitmaps of 2, 4, 16 or 256 colors are supported. If the file is not in a good format, an error code (atom) is returned instead:

```
global constant BMP_OPEN_FAILED = 1,
                BMP_UNEXPECTED_EOF = 2,
                BMP_UNSUPPORTED_FORMAT = 3
```

**Comments:**    You can create your own bitmap picture files using Windows Paintbrush and many other graphics programs. You can then incorporate these pictures into your Euphoria programs.

**Example:**

```
x = read_bitmap("c:\\windows\\arcade.bmp")
-- note: double backslash needed to get single backslash in
-- a string
```

**Example Program:**   [demo\\dos32\\bitmap.ex](#)

**See Also:**       [palette](#), [all\\_palette](#), [display\\_image](#), [save\\_bitmap](#)

## register\_block

**Syntax:** include machine.e (or safe.e)

register\_block(a, i)

**Description:** Add a block of memory to the list of safe blocks maintained by [safe.e](#) (the debug version of [machine.e](#)). The block starts at address a. The length of the block is i bytes.

**Comments:** This routine is only meant to be used for **debugging purposes**. [safe.e](#) tracks the blocks of memory that your program is allowed to [peek\(\)](#), [poke\(\)](#), [mem\\_copy\(\)](#) etc. These are normally just the blocks that you have allocated using Euphoria's [allocate\(\)](#) or [allocate\\_low\(\)](#) routines, and which you have not yet freed using Euphoria's [free\(\)](#) or [free\\_low\(\)](#). In some cases, you may acquire additional, external, blocks of memory, perhaps as a result of calling a C routine. If you are debugging your program using [safe.e](#), you must register these external blocks of memory or [safe.e](#) will prevent you from accessing them. When you are finished using an external block you can unregister it using [unregister\\_block\(\)](#).

When you include [machine.e](#), you'll get different versions of [register\\_block\(\)](#) and [unregister\\_block\(\)](#) that do nothing. This makes it easy to switch back and forth between debug and non-debug runs of your program.

### Example 1:

```
atom addr
```

```
addr = c_func(x, {})  
register_block(addr, 5)  
poke(addr, "ABCDE")  
unregister_block(addr)
```

**See Also:** [unregister\\_block](#), [safe.e](#)

## remainder

**Syntax:** `x3 = remainder(x1, x2)`

**Description:** Compute the remainder after dividing x1 by x2. The result will have the same sign as x1, and the magnitude of the result will be less than the magnitude of x2.

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

**Example 1:**

```
a = remainder(9, 4)
-- a is 1
```

**Example 2:**

```
s = remainder({81, -3.5, -9, 5.5}, {8, -1.7, 2, -4})
-- s is {1, -0.1, -1, 1.5}
```

**Example 3:**

```
s = remainder({17, 12, 34}, 16)
-- s is {1, 12, 2}
```

**Example 4:**

```
s = remainder(16, {2, 3, 5})
-- s is {0, 1, 1}
```

**See Also:** [floor](#)

## repeat

**Syntax:** `s = repeat(x, a)`

**Description:** Create a sequence of length `a` where each element is `x`.

**Comments:** When you repeat a sequence or a floating-point number the interpreter does not actually make multiple copies in memory. Rather, a single copy is "pointed to" a number of times.

**Example 1:**

```
repeat(0, 10)      -- {0,0,0,0,0,0,0,0,0,0}
```

**Example 2:**

```
repeat("JOHN", 4)  -- {"JOHN", "JOHN", "JOHN", "JOHN"}
-- The interpreter will create only one copy of "JOHN"
-- in memory
```

**See Also:** [append](#), [prepend](#), [sequence-formation operator](#)

## reverse

**Syntax:** include misc.e

s2 = reverse(s1)

**Description:** Reverse the order of elements in a sequence.

**Comments:** A new sequence is created where the top-level elements appear in reverse order compared to the original sequence.

**Example 1:**

```
reverse({1, 3, 5, 7})      -- {7, 5, 3, 1}
```

**Example 2:**

```
reverse({{1, 2, 3}, {4, 5, 6}}) -- {{4, 5, 6}, {1, 2, 3}}
```

**Example 3:**

```
reverse({99})             -- {99}
```

**Example 4:**

```
reverse({})               -- {}
```

**See Also:** [append](#), [prepend](#), [repeat](#)

## routine\_id

**Syntax:** i = routine\_id(st)

**Description:** Return an integer id number, known as a **routine id**, for a user-defined Euphoria procedure or function. The name of the procedure or function is given by the string sequence st. -1 is returned if the named routine can't be found.

**Comments:** The id number can be passed to call\_proc() or call\_func(), to indirectly call the routine named by st.

The routine named by st must be visible, i.e. callable, at the place where routine\_id() is used to get the id number. Indirect calls to the routine can appear earlier in the program than the definition of the routine, but the id number can only be obtained in code that comes *after* the definition of the routine - see example 2 below.

Once obtained, a valid **routine id** can be used at *any* place in the program to call a routine indirectly via call\_proc()/call\_func().

Some typical uses of routine\_id() are:

1. [Calling a routine that is defined later in a program.](#)
2. Creating a subroutine that takes another routine as a parameter. (See Example 2 below)
3. Using a sequence of **routine id's** to make a case (switch) statement.
4. Setting up an Object-Oriented system.
5. Getting a **routine id** so you can pass it to call\_back(). (See [platform.doc](#))

Note that C routines, callable by Euphoria, also have routine id's. See define\_c\_proc() and define\_c\_func().

### Example 1:

```
procedure foo()
  puts(1, "Hello World\n")
end procedure

integer foo_num
foo_num = routine_id("foo")

call_proc(foo_num, {}) -- same as calling foo()
```

### Example 2:

```
function apply_to_all(sequence s, integer f)
  -- apply a function to all elements of a sequence
  sequence result
  result = {}
  for i = 1 to length(s) do
    -- we can call add1() here although it comes later in the program
    result = append(result, call_func(f, {s[i]}))
  end for
  return result
end function

function add1(atom x)
  return x + 1
end function
```

```
-- add1() is visible here, so we can ask for its routine id
? apply_to_all({1, 2, 3}, routine_id("add1"))
-- displays {2,3,4}
```

**See Also:**     [call\\_proc](#), [call\\_func](#), [call\\_back](#), [define\\_c\\_func](#), [define\\_c\\_proc](#), [platform.doc](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | from P to R | [from S to T](#) | [from U to Z](#)

## save\_bitmap

**Syntax:**       include image.e

i = save\_bitmap(s, st)

**Description:** Create a bitmap (.bmp) file from a 2-element sequence s. st is the name of a .bmp "bitmap" file. s[1] is the palette:

{ {r,g,b}, {r,g,b}, ..., {r,g,b} }

Each red, green, or blue value is in the range 0 to 255. s[2] is a 2-d sequence of sequences containing a pixel-graphics image. The sequences contained in s[2] must all have the same length. s is in the same format as the value returned by read\_bitmap().

The result will be one of the following codes:

```
global constant BMP_SUCCESS = 0,
    BMP_OPEN_FAILED = 1,
    BMP_INVALID_MODE = 4 -- invalid graphics mode
                        -- or invalid argument
```

**Comments:** If you use get\_all\_palette() to get the palette before calling this function, you must multiply the returned intensity values by 4 before calling save\_bitmap().

You might use save\_image() to get the 2-d image for s[2].

save\_bitmap() produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with read\_bitmap(). Windows Paintbrush and some other tools do not support 4-color bitmaps.

**Example:**

```
paletteData = get_all_palette() * 4
code = save_bitmap({paletteData, imageData},
    "c:\\example\\a1.bmp")
```

**See Also:**   [save\\_image](#), [read\\_bitmap](#), [save\\_screen](#), [get\\_all\\_palette](#)

## save\_image

**Platform:**   **DOS32**

**Syntax:**       include image.e

s3 = save\_image(s1, s2)

**Description:** Save a rectangular image from a pixel-graphics screen. The result is a 2-d sequence of sequences containing all the pixels in the image. You can redisplay the image using display\_image(). s1 is a 2-element sequence {x1,y1} specifying the top-left pixel in the image. s2 is a sequence {x2,y2} specifying the bottom-right pixel.

**Example:**

```
s = save_image({0,0}, {50,50})
display_image({100,200}, s)
display_image({300,400}, s)
-- saves a 51x51 square image, then redisplay it at {100,200}
-- and at {300,400}
```



See Also: [display\\_image](#), [save\\_text\\_image](#)

## save\_screen

**Platform:** **DOS32**

**Syntax:** `include image.e`

`i = save_screen(x1, st)`

**Description:** Save the whole screen or a rectangular region of the screen as a Windows bitmap (.bmp) file. To save the whole screen, pass the integer 0 for x1. To save a rectangular region of the screen, x1 should be a sequence of 2 sequences: { {topLeftXPixel, topLeftYPixel}, {bottomRightXPixel, bottomRightYPixel} } st is the name of a .bmp "bitmap" file.

The result will be one of the following codes:

```
global constant BMP_SUCCESS = 0,
    BMP_OPEN_FAILED = 1,
    BMP_INVALID_MODE = 4 -- invalid graphics mode
                        -- or invalid argument
```

**Comments:** `save_screen()` produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with `read_bitmap()`. Windows Paintbrush and some other tools do not support 4-color bitmaps.

`save_screen()` only works in **pixel-graphics modes**, not text modes.

### Example 1:

```
-- save whole screen:
code = save_screen(0, "c:\\example\\a1.bmp")
```

### Example 2:

```
-- save part of screen:
err = save_screen({{0,0},{200, 15}}, "b1.bmp")
```

See Also: [save\\_image](#), [read\\_bitmap](#), [save\\_bitmap](#)

## save\_text\_image

**Syntax:** `include image.e`

`s3 = save_text_image(s1, s2)`

**Description:** Save a rectangular region of text from a **text-mode** screen. The result is a sequence of sequences containing ASCII characters and attributes from the screen. You can redisplay this text using `display_text_image()`. s1 is a 2-element sequence {line1, column1} specifying the top-left character. s2 is a sequence {line2, column2} specifying the bottom right character.

**Comments:** Because the character attributes are also saved, you will get the correct foreground color, background color and other properties for each character when you redisplay the text.

On DOS32, an attribute byte is made up of two 4-bit fields that encode the foreground and background color of a character. The high-order 4 bits determine the background color, while the low-order 4 bits determine the foreground color.

This routine only works in **text modes**.

You might use this function in a text-mode graphical user interface to save a portion of the screen before displaying a drop-down menu, dialog box, alert box etc.

On DOS32, if you are flipping video pages, note that this function reads from the current active page.

**Example:** If the top 2 lines of the screen have:

```
Hello
World
```

And you execute:

```
s = save_text_image({1,1}, {2,5})
```

Then s is something like:

```
{ "H-e-l-l-o-",
  "W-o-r-l-d-" }
```

where '-' indicates the attribute bytes

**See Also:** [display\\_text\\_image](#), [save\\_image](#), [set\\_active\\_page](#), [get\\_screen\\_char](#)

## scroll

**Syntax:** include graphics.e

scroll(i1, i2, i3)

**Description:** Scroll a region of text on the screen either up (i1 positive) or down (i1 negative) by i1 lines. The region is the series of lines on the screen from i2 (top line) to i3 (bottom line), inclusive. New blank lines will appear at the top or bottom.

**Comments:** You could perform the scrolling operation using a series of calls to puts(), but scroll() is much faster.

The position of the cursor after scrolling is not defined.

**Example Program:** [bin\ed.ex](#)

**See Also:** [clear\\_screen](#), [text\\_rows](#)

## seek

**Syntax:** include file.e

i1 = seek(fn, a1)

**Description:** Seek (move) to any byte position in the file fn or to the end of file if a1 is -1. For each open file there is a current byte position that is updated as a result of I/O operations on the file. The initial file position is 0 for files opened for read, write or update. The initial position is the end of file for files opened for append. The value returned by seek() is 0 if the seek was successful, and non-zero if it was unsuccessful. It is possible to seek past the end of a file. If you seek past the end of the file, and write some data, undefined bytes will be inserted into the gap between the original end of file and your new data.

**Comments:** After seeking and reading (writing) a series of bytes, you may need to call seek() explicitly before you switch to writing (reading) bytes, even though the file position should already be what you want.

This function is normally used with files opened in binary mode. In text mode, DOS converts CR LF to LF on input, and LF to CR LF on output, which can cause great confusion when you are trying to count bytes.

**Example:**

```
include file.e
```

```
integer fn
fn = open("mydata", "rb")
-- read and display first line of file 3 times:
for i = 1 to 3 do
    puts(1, gets(fn))
    if seek(fn, 0) then
        puts(1, "rewind failed!\n")
    end if
end for
```

**See Also:** [where](#), [open](#)

## sequence

**Syntax:** i = sequence(x)

**Description:** Return 1 if x is a sequence else return 0.

**Comments:** This serves to define the sequence type. You can also call it like an ordinary function to determine if an object is a sequence.

**Example 1:**

```
sequence s
s = {1,2,3}
```

**Example 2:**

```
if sequence(x) then
    sum = 0
    for i = 1 to length(x) do
        sum = sum + x[i]
    end for
else
    -- x must be an atom
    sum = x
end if
```

**See Also:** [atom](#), [object](#), [integer](#), [atoms and sequences](#)

## set\_active\_page

**Platform:** **DOS32**

**Syntax:** include image.e

set\_active\_page(i)

**Description:** Select video page i to send all screen output to.

**Comments:** With multiple pages you can instantaneously change the entire screen without causing any visible "flicker". You can also save the screen and bring it back quickly.

video\_config() will tell you how many pages are available in the current graphics mode.

By default, the active page and the display page are both 0.

This works under DOS, or in a full-screen DOS window. In a partial-screen window you cannot change the

active page.

**Example:**

```
include image.e

-- active & display pages are initially both 0
puts(1, "\nThis is page 0\n")
set_active_page(1)      -- screen output will now go to page 1
clear_screen()
puts(1, "\nNow we've flipped to page 1\n")
if getc(0) then          -- wait for key-press
end if
set_display_page(1)      -- "Now we've ..." becomes visible
if getc(0) then          -- wait for key-press
end if
set_display_page(0)      -- "This is ..." becomes visible again
set_active_page(0)
```

**See Also:** [get\\_active\\_page](#), [set\\_display\\_page](#), [video\\_config](#)

## set\_display\_page

**Platform:** **DOS32**

**Syntax:** include image.e

set\_display\_page(i)

**Description:** Set video page i to be mapped to the visible screen.

**Comments:** With multiple pages you can instantaneously change the entire screen without causing any visible "flicker". You can also save the screen and bring it back quickly.

video\_config() will tell you how many pages are available in the current graphics mode.

By default, the active page and the display page are both 0.

This works under DOS, or in a full-screen DOS window. In a partial-screen window you cannot change the active page.

**Example:** See set\_active\_page() example.

**See Also:** [get\\_display\\_page](#), [set\\_active\\_page](#), [video\\_config](#)

## set\_rand

**Syntax:** include machine.e

set\_rand(i1)

**Description:** Set the random number generator to a certain state, i1, so that you will get a known series of random numbers on subsequent calls to rand().

**Comments:** Normally the numbers returned by the rand() function are totally unpredictable, and will be different each time you run your program. Sometimes however you may wish to repeat the same series of numbers, perhaps because you are trying to debug your program, or maybe you want the ability to generate the same output (e.g. a random picture) for your user upon request.

**Example:**

```

sequence s, t
s = repeat(0, 3)
t = s

set_rand(12345)
s[1] = rand(10)
s[2] = rand(100)
s[3] = rand(1000)

set_rand(12345) -- same value for set_rand()
t[1] = rand(10) -- same arguments to rand() as before
t[2] = rand(100)
t[3] = rand(1000)
-- at this point s and t will be identical

```

**See Also:** [rand](#)

## set\_vector

**Platform:** **DOS32**

**Syntax:** include machine.e

set\_vector(i, s)

**Description:** Set s as the new address for handling interrupt number i. s must be a protected mode **far address** in the form: { 16-bit segment, 32-bit offset}.

**Comments:** Before calling set\_vector() you must store a machine-code interrupt handling routine at location s in memory.

The 16-bit segment can be the code segment used by Euphoria. To get the value of this segment see [demo\dos32\hardint.ex](#). The offset can be the 32-bit value returned by allocate(). Euphoria runs in **protected mode** with the code segment and data segment pointing to the same physical memory, but with different access modes.

Interrupts occurring in either **real mode** or **protected mode** will be passed to your handler. Your interrupt handler should immediately load the correct data segment before it tries to reference memory.

Your handler might return from the interrupt using the iretd instruction, or jump to the original interrupt handler. It should save and restore any registers that it modifies.

You should lock the memory used by your handler to ensure that it will never be swapped out. See lock\_memory().

It is highly recommended that you study [demo\dos32\hardint.ex](#) before trying to set up your own interrupt handler.

You should have a good knowledge of machine-level programming before attempting to write your own handler.

You can call set\_vector() with the far address returned by get\_vector(), when you want to restore the original handler.

**Example:**

```
set_vector(#1C, {code_segment, my_handler_address})
```

**Example Program:** [demo\dos32\hardint.ex](#)

**See Also:** [get\\_vector](#), [lock\\_memory](#), [allocate](#)

## sin

**Syntax:** `x2 = sin(x1)`

**Description:** Return the sine of x1, where x1 is in radians.

**Comments:** This function may be applied to an atom or to all elements of a sequence.

**Example:**

```
sin_x = sin({.5, .9, .11})
-- sin_x is {.479, .783, .110}
```

**See Also:** [cos](#), [tan](#)

## sleep

**Syntax:** `include misc.e`

`sleep(i)`

**Description:** Suspend execution for i seconds.

**Comments:** On WIN32 and Linux/FreeBSD, the operating system will suspend your process and schedule other processes. On DOS32, your program will go into a busy loop for i seconds, during which time other processes may run, but they will compete with your process for the CPU.

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()`.

**Example:**

```
puts(1, "Waiting 15 seconds...\n")
sleep(15)
puts(1, "Done.\n")
```

**See Also:** [lock\\_file](#), [abort](#), [time](#)

## sort

**Syntax:** `include sort.e`

`s2 = sort(s1)`

**Description:** Sort s1 into ascending order using a fast sorting algorithm. The elements of s1 can be any mix of atoms or sequences. Atoms come before sequences, and sequences are sorted "alphabetically" where the first elements are more significant than the later elements.

**Example 1:**

```
x = 0 & sort({7,5,3,8}) & 0
-- x is set to {0, 3, 5, 7, 8, 0}
```

**Example 2:**

```
y = sort({"Smith", "Jones", "Doe", 5.5, 4, 6})
-- y is {4, 5.5, 6, "Doe", "Jones", "Smith"}
```

### Example 3:

```
database = sort({{"Smith", 95.0, 29},
                 {"Jones", 77.2, 31},
                 {"Clinton", 88.7, 44}})

-- The 3 database "records" will be sorted by the first "field"
-- i.e. by name. Where the first field (element) is equal it
-- will be sorted by the second field etc.

-- after sorting, database is:
    {"Clinton", 88.7, 44},
    {"Jones", 77.2, 31},
    {"Smith", 95.0, 29}
```

See Also: [custom\\_sort](#), [compare](#), [match](#), [find](#)

## sound

**Platform:** **DOS32**

**Syntax:** include graphics.e

sound(i)

**Description:** Turn on the PC speaker at frequency i. If i is 0 the speaker will be turned off.

**Comments:** On **WIN32** and **Linux/FreeBSD** no sound will be made.

**Example:**

```
sound(1000) -- starts a fairly high pitched sound
```

## sprint

**Syntax:** include misc.e

s = sprint(x)

**Description:** The representation of x as a string of characters is returned. This is exactly the same as **print(fn, x)**, except that the output is returned as a sequence of characters, rather than being sent to a file or device. x can be any Euphoria object.

**Comments:** The atoms contained within x will be displayed to a maximum of 10 significant digits, just as with print().

**Example 1:**

```
s = sprint(12345)
-- s is "12345"
```

**Example 2:**

```
s = sprint({10,20,30}+5)
-- s is "{15,25,35}"
```

See Also: [print](#), [sprintf](#), [value](#), [get](#)

## sprintf

**Syntax:** `s = sprintf(st, x)`

**Description:** This is exactly the same as **printf()**, except that the output is returned as a sequence of characters, rather than being sent to a file or device. `st` is a format string, `x` is the value or sequence of values to be formatted. **printf(fn, st, x)** is equivalent to **puts(fn, sprintf(st, x))**.

**Comments:** Some typical uses of `sprintf()` are:

1. Converting numbers to strings.
2. Creating strings to pass to `system()`.
3. Creating formatted error messages that can be passed to a common error message handler.

**Example:**

```
s = sprintf("%08d", 12345)
-- s is "00012345"
```

**See Also:** [printf](#), [value](#), [sprint](#), [get](#), [system](#)

## sqrt

**Syntax:** `x2 = sqrt(x1)`

**Description:** Calculate the square root of `x1`.

**Comments:** This function may be applied to an atom or to all elements of a sequence.

Taking the square root of a negative number will abort your program with a run-time error message.

**Example:**

```
r = sqrt(16)
-- r is 4
```

**See Also:** [log](#), [power](#)

## system

**Syntax:** `system(st, i)`

**Description:** Pass a command string `st` to the operating system command interpreter. The argument `i` indicates the manner in which to return from the call to `system()`:

When `i` is 0, the previous graphics mode is restored and the screen is cleared.

When `i` is 1, a beep sound will be made and the program will wait for the user to press a key before the previous graphics mode is restored.

When `i` is 2, the graphics mode is not restored and the screen is not cleared.

**Comments:** `i = 2` should only be used when it is known that the command executed by `system()` will not change the graphics mode.

You can use Euphoria as a sophisticated "batch" (.bat) language by making calls to `system()` and `system_exec()`.

`system()` will start a new DOS or Linux/FreeBSD shell.

`system()` allows you to use command-line redirection of standard input and output in the command string `st`.



Under **DOS32**, a Euphoria program will start off using extended memory. If extended memory runs out the program will consume conventional memory. If conventional memory runs out it will use virtual memory, i.e. swap space on disk. The DOS command run by `system()` will fail if there is not enough conventional memory available. To avoid this situation you can reserve some conventional (low) memory by typing:

```
SET CAUSEWAY=LOWMEM:xxx
```

where xxx is the number of K of conventional memory to reserve. Type this before running your program. You can also put this in **autoexec.bat**, or in a **.bat** file that runs your program. For example:

```
SET CAUSEWAY=LOWMEM:80
```

```
ex myprog.ex
```

This will reserve 80K of conventional memory, which should be enough to run simple DOS commands like COPY, MOVE, MKDIR etc.

#### Example 1:

```
system("copy temp.txt a:\\temp.bak", 2)
-- note use of double backslash in literal string to get
-- single backslash
```

#### Example 2:

```
system("ex \\test\\myprog.ex < indata > outdata", 2)
-- executes myprog by redirecting standard input and
-- standard output
```

**See Also:**    [system\\_exec](#), [dir](#), [current\\_dir](#), [getenv](#), [command\\_line](#)

## system\_exec

**Syntax:**     `i1 = system_exec(st, i2)`

**Description:** Try to run the command given by st. st must be a command to run an executable program, possibly with some command-line arguments. If the program can be run, i1 will be the exit code from the program. If it is not possible to run the program, `system_exec()` will return -1. i2 is a code that indicates what to do about the graphics mode when `system_exec()` is finished. These codes are the same as for `system()`:

When i2 is 0, the previous graphics mode is restored and the screen is cleared.

When i2 is 1, a beep sound will be made and the program will wait for the user to press a key before the previous graphics mode is restored.

When i2 is 2, the graphics mode is not restored and the screen is not cleared.

**Comments:** On DOS32 or WIN32, `system_exec()` will only run **.exe** and **.com** programs. To run **.bat** files, or built-in DOS commands, you need `system()`. Some commands, such as DEL, are not programs, they are actually built-in to the command interpreter.

On DOS32 and WIN32, `system_exec()` does not allow the use of command-line redirection in the command string st, nor does it allow you to quote strings that contain blanks, such as file names.

exit codes from DOS or Windows programs are normally in the range 0 to 255, with 0 indicating "success".

You can run a Euphoria program using `system_exec()`. A Euphoria program can return an exit code using `abort()`.

system\_exec() does not start a new DOS shell.

#### Example 1:

```
integer exit_code
exit_code = system_exec("xcopy temp1.dat temp2.dat", 2)

if exit_code = -1 then
    puts(2, "\n couldn't run xcopy.exe\n")
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("ex \\test\\myprog.ex indata outdata", 2) then
    puts(2, "failure!\n")
end if
```

See Also: [system](#), [abort](#)

## tan

**Syntax:** x2 = tan(x1)

**Description:** Return the tangent of x1, where x1 is in radians.

**Comments:** This function may be applied to an atom or to all elements of a sequence.

#### Example:

```
t = tan(1.0)
-- t is 1.55741
```

See Also: [sin](#), [cos](#), [arctan](#)

## task\_clock\_start

**Syntax:** task\_clock\_start()

**Description:** Restart the clock used for scheduling real-time tasks. Call this routine, some time after calling task\_clock\_stop(), when you want scheduling of real-time tasks to continue.

**Comments:** 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:

```
-- 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\_clock\_stop

**Syntax:** `task_clock_stop()`

**Description:** Stop the scheduling of real-time tasks. Scheduling will resume when `task_clock_start()` is called. Time-shared tasks can continue. The current task can also continue, unless it's a real-time task and it yields.

**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.

The `time()` function is not affected by this.

**See Also:** [task\\_clock\\_start](#), [task\\_schedule](#), [task\\_yield](#), [task\\_suspend](#)

## task\_create

**Syntax:** `a2 = task_create(i1, s)`

**Description:** Create a new task. `i1` is the routine id of a user-defined Euphoria procedure. `s` is the list of arguments that will be passed to this procedure when the task starts executing. The result, `a2`, is a task identifier, created by the system. It can be used to identify this task to the other Euphoria multitasking routines.

**Comments:** `task_create()` creates a new task, but does not start it executing.

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:**

```
mytask = task_create(routine_id("myproc"), {5, 9, "ABC"})
```

**See Also:** [task\\_schedule](#), [task\\_yield](#), [task\\_suspend](#), [task\\_self](#)

## task\_list

**Syntax:** `s = task_list()`

**Description:** Get a sequence containing the task id's for all active or suspended tasks.

**Comments:** This function lets you find out which tasks currently exist. Tasks that have terminated naturally, or have been killed are not included. You can pass a task id to `task_status()` to find out more

about a particular task.

**Example:**

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

**Syntax:** task\_schedule(a, x)

**Description:** Schedule task a to run using scheduling parameter x. a must be a task identifier returned by task\_create(). If x is a positive atom, then x tells the task scheduler how many times it should run time-shared task a in one burst before it considers running other tasks. If x is a sequence, it must be a 2-element sequence {min, max}, both values greater than or equal to 0, where min and max indicate the minimum and maximum times, in seconds, to wait before running real-time task a. min and max also set the time interval for subsequent runs of task a, unless overridden by another call to task\_schedule() or task\_suspend().

**Comments:** The task scheduler, which is built-in to the Euphoria run-time system, will use the scheduling parameter x 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.

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), 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 and Linux/FreeBSD, and 0.55 seconds on DOS unless tick\_rate() is called) then accurate time scheduling cannot be performed, but the scheduler will try to run the task several times in a row to approximate what is desired. For example, if you ask for a min time of 0.002 seconds, then the scheduler will try to run your task  $.01/.002 = 5$  times in a row before waiting for the clock to "click" ahead by .01. During the next 0.01 seconds it will run your task (up to) another 5 times etc. provided your task can be completed 5 times in one clock period.

At program start-up there is a single task running. Its task id is 0, and initially it's a time-shared task allowed 1 run per task\_yield(). No other task can run until task 0 executes a task\_yield().

If task 0 (top-level) runs off the end of the main file, the whole program terminates, regardless of what other

tasks may still be active.

If the scheduler finds that no task is active, i.e. no task will ever run again (not even task 0), it terminates the program with a 0 exit code code, similar to abort(0).

**Example:**

```
-- 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\\_suspend](#)

## task\_self

**Syntax:** a = task\_self()

**Description:** Return the task id of the current task.

**Comments:** This value may be needed, if a task wants to schedule, kill or suspend itself.

**Example:**

```
-- schedule self
task_schedule(task_self(), {5.9, 6.0})
```

**See Also:** [task\\_create](#), [task\\_schedule](#), [task\\_yield](#), [task\\_suspend](#)

## task\_status

**Syntax:** i2 = task\_status(a)

**Description:** Return the status of a task. Status values are 1 (active), 0 (suspended), and -1 (task does not exist)

**Comments:** A task might want to know the status of one or more other tasks when deciding whether to proceed with some processing.

A -1 value could indicate a task that never existed, or a task that terminated naturally or was killed.

**Example:**

```
integer s

s = task_status(tid)
if s = 1 then
    puts(1, "ACTIVE\n")
elsif s = 0 then
    puts(1, "SUSPENDED\n")
else
    puts(1, "DOESN'T EXIST\n")
end if
```

**See Also:** [task\\_list](#), [task\\_create](#), [task\\_schedule](#), [task\\_suspend](#)

## task\_suspend

**Syntax:** `task_suspend(a)`

**Description:** Suspend execution of task a.

**Comments:** This causes task a to be suspended. Task a will not be executed again unless there is a call to `task_schedule()` for task a. a 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()`.

**Example:**

```
-- 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

**Syntax:** `task_yield()`

**Description:** Yield control to the scheduler. The scheduler can then choose another task to run, or perhaps let the current task continue running.

**Comments:** Tasks should call `task_yield()` periodically so other tasks will have a chance to run. Only when `task_yield()` is called, is there a way for the scheduler to take back control from a task. This is what's known as cooperative multitasking.

A task can have calls to `task_yield()` in many different places in it's code, and at any depth of subroutine call.

The scheduler will use the current value of { min, max } or the current number of consecutive runs remaining, 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 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:**

```

-- 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.

global 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](#) [task\\_suspend](#)

## text\_color

**Syntax:** include graphics.e

text\_color(i)

**Description:** Set the foreground text color. Add 16 to get blinking text in some modes. See [graphics.e](#) for a list of possible colors.

**Comments:** Text that you print *after* calling text\_color() will have the desired color.

When your program terminates, the last color that you selected and actually printed on the screen will remain in effect. Thus you may have to print something, maybe just '\n', in WHITE to restore white text, especially if you are at the bottom line of the screen, ready to scroll up.

**Example:**

```
text_color(BRIGHT_BLUE)
```

**See Also:** [bk\\_color](#)

## text\_rows

**Platform:** DOS32, WIN32

**Syntax:** include graphics.e

i2 = text\_rows(i1)

**Description:** Set the number of lines on a **text-mode** screen to i1 if possible. i2 will be set to the actual new number of lines.

**Comments:** Values of 25, 28, 43 and 50 lines are supported by most video cards.

**See Also:** [graphics\\_mode](#)

## tick\_rate

**Platform:** DOS32

**Syntax:** include machine.e

tick\_rate(a)

**Description:** Specify the number of clock-tick interrupts per second. This determines the precision of the time() library routine. It also affects the sampling rate for time profiling.

**Comments:** tick\_rate() is ignored on WIN32 and Linux/FreeBSD. The time resolution on WIN32 is always 100 ticks/second.

On a PC the clock-tick interrupt normally occurs at 18.2 interrupts per second. tick\_rate() lets you increase that rate, but not decrease it.

tick\_rate(0) will restore the rate to the normal 18.2 rate. Euphoria will also restore the rate automatically when it exits, even when it finds an error in your program.

If a program runs in a DOS window with a tick rate other than 18.2, the time() function will not advance unless the window is the active window.

With a tick rate other than 18.2, the time() function on DOS takes about 1/100 the usual time that it needs to execute. On Windows and FreeBSD, time() normally executes very quickly.

While **ex.exe** is running, the system will maintain the correct time of day. However if **ex.exe** should crash (e.g. you see a "CauseWay..." error) while the tick rate is high, you (or your user) may need to reboot the machine to restore the proper rate. If you don't, the system time may advance too quickly. This problem does not occur on Windows 95/98/NT, only on DOS or Windows 3.1. You will always get back the correct time of day from the battery-operated clock in your system when you boot up again.

**Example:**

```
tick_rate(100)
-- time() will now advance in steps of .01 seconds
-- instead of the usual .055 seconds
```

**See Also:** [time](#), [time profiling](#)

## time

**Syntax:** a = time()

**Description:** Return the number of seconds since some fixed point in the past.

**Comments:** Take the difference between two readings of time(), to measure, for example, how long a section of code takes to execute.

The resolution with DOS32 is normally about 0.05 seconds. On WIN32 and Linux/FreeBSD it's about 0.01 seconds.

Under DOS32 you can improve the resolution by calling tick\_rate().

Under DOS32 the period of time that you can normally measure is limited to 24 hours. After that, the value returned by time() will reset and start over. If however, you have called tick\_rate(), and clock ticks are



happening at a rate that is higher than the usual 18.2/sec, time() will continue much longer, since in that case, Euphoria handles the clock-tick interrupt directly, and accumulates the ticks in a larger, 32-bit variable.

DOS emulation under Windows XP is not perfect. When you do time profiling, (with profile\_time) the time() function might be off by several percent. This problem does not occur on Windows ME/98/95.

**Example:**

```
constant ITERATIONS = 1000000
integer p
atom t0, loop_overhead

t0 = time()
for i = 1 to ITERATIONS do
    -- time an empty loop
end for
loop_overhead = time() - t0

t0 = time()
for i = 1 to ITERATIONS do
    p = power(2, 20)
end for
? (time() - t0 - loop_overhead) / ITERATIONS
-- calculates time (in seconds) for one call to power
```

**See Also:** [date](#), [tick\\_rate](#)

## trace

**Syntax:** with trace

trace(i)

**Description:** If i is 1 or 2, turn on full-screen interactive statement tracing/debugging. If i is 3, turn on tracing of statements to a file called **ctrace.out**. If i is 0, turn off tracing. When i is 1 a color display appears. When i is 2 a monochrome trace display appears. Tracing can only occur in routines that were compiled "with trace", and trace() has no effect unless it is executed in a "**with trace**" section of your program.

See [Part I - 3.1 Debugging](#) for a full discussion of tracing / debugging.

**Comments:** Use trace(2) if the color display is hard to view on your system.

All forms of trace() are supported by the Interpreter.

Only trace(3) is supported by the Euphoria To C Translator.

**Example:**

```
if x < 0 then
    -- ok, here's the case I want to debug...
    trace(1)
    -- etc.
...
end if
```

**See Also:** [profile](#), [debugging and profiling](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | from S to T | [from U to Z](#)

## save\_bitmap

**Syntax:** include image.e

i = save\_bitmap(s, st)

**Description:** Create a bitmap (.bmp) file from a 2-element sequence s. st is the name of a .bmp "bitmap" file. s[1] is the palette:

{ {r,g,b}, {r,g,b}, ..., {r,g,b} }

Each red, green, or blue value is in the range 0 to 255. s[2] is a 2-d sequence of sequences containing a pixel-graphics image. The sequences contained in s[2] must all have the same length. s is in the same format as the value returned by read\_bitmap().

The result will be one of the following codes:

```
global constant BMP_SUCCESS = 0,
    BMP_OPEN_FAILED = 1,
    BMP_INVALID_MODE = 4 -- invalid graphics mode
                        -- or invalid argument
```

**Comments:** If you use get\_all\_palette() to get the palette before calling this function, you must multiply the returned intensity values by 4 before calling save\_bitmap().

You might use save\_image() to get the 2-d image for s[2].

save\_bitmap() produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with read\_bitmap(). Windows Paintbrush and some other tools do not support 4-color bitmaps.

**Example:**

```
paletteData = get_all_palette() * 4
code = save_bitmap({paletteData, imageData},
    "c:\\example\\a1.bmp")
```

**See Also:** [save\\_image](#), [read\\_bitmap](#), [save\\_screen](#), [get\\_all\\_palette](#)

## save\_image

**Platform:** DOS32

**Syntax:** include image.e

s3 = save\_image(s1, s2)

**Description:** Save a rectangular image from a pixel-graphics screen. The result is a 2-d sequence of sequences containing all the pixels in the image. You can redisplay the image using display\_image(). s1 is a 2-element sequence {x1,y1} specifying the top-left pixel in the image. s2 is a sequence {x2,y2} specifying the bottom-right pixel.

**Example:**

```
s = save_image({0,0}, {50,50})
display_image({100,200}, s)
display_image({300,400}, s)
-- saves a 51x51 square image, then redisplay it at {100,200}
-- and at {300,400}
```

**See Also:** [display\\_image](#), [save\\_text\\_image](#)

## save\_screen

**Platform:** **DOS32**

**Syntax:** include image.e

i = save\_screen(x1, st)

**Description:** Save the whole screen or a rectangular region of the screen as a Windows bitmap (.bmp) file. To save the whole screen, pass the integer 0 for x1. To save a rectangular region of the screen, x1 should be a sequence of 2 sequences: {{topLeftXPixel, topLeftYPixel}, {bottomRightXPixel, bottomRightYPixel}}

st is the name of a .bmp "bitmap" file.

The result will be one of the following codes:

```
global constant BMP_SUCCESS = 0,
    BMP_OPEN_FAILED = 1,
    BMP_INVALID_MODE = 4 -- invalid graphics mode
                        -- or invalid argument
```

**Comments:** save\_screen() produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with read\_bitmap(). Windows Paintbrush and some other tools do not support 4-color bitmaps.

save\_screen() only works in **pixel-graphics modes**, not text modes.

### Example 1:

```
-- save whole screen:
code = save_screen(0, "c:\\example\\a1.bmp")
```

### Example 2:

```
-- save part of screen:
err = save_screen({{0,0},{200, 15}}, "b1.bmp")
```

**See Also:** [save\\_image](#), [read\\_bitmap](#), [save\\_bitmap](#)

## save\_text\_image

**Syntax:**           include image.e

s3 = save\_text\_image(s1, s2)

**Description:**   Save a rectangular region of text from a **text-mode** screen. The result is a sequence of sequences containing ASCII characters and attributes from the screen. You can redisplay this text using `display_text_image()`. s1 is a 2-element sequence {line1, column1} specifying the top-left character. s2 is a sequence {line2, column2} specifying the bottom right character.

**Comments:**      Because the character attributes are also saved, you will get the correct foreground color, background color and other properties for each character when you redisplay the text.

On DOS32, an attribute byte is made up of two 4-bit fields that encode the foreground and background color of a character. The high-order 4 bits determine the background color, while the low-order 4 bits determine the foreground color.

This routine only works in **text modes**.

You might use this function in a text-mode graphical user interface to save a portion of the screen before displaying a drop-down menu, dialog box, alert box etc.

On DOS32, if you are flipping video pages, note that this function reads from the current active page.

**Example:**        If the top 2 lines of the screen have:

```
      Hello
World
```

And you execute:

```
s = save_text_image({1,1}, {2,5})
```

Then s is something like:

```
      {"H-e-l-l-o-",
      "W-o-r-l-d-"}
```

where '-' indicates the attribute bytes

**See Also:**       [display\\_text\\_image](#), [save\\_image](#), [set\\_active\\_page](#), [get\\_screen\\_char](#)

## scroll

**Syntax:** include graphics.h

scroll(i1, i2, i3)

**Description:** Scroll a region of text on the screen either up (i1 positive) or down (i1 negative) by i1 lines. The region is the series of lines on the screen from i2 (top line) to i3 (bottom line), inclusive. New blank lines will appear at the top or bottom.

**Comments:** You could perform the scrolling operation using a series of calls to puts(), but scroll() is much faster.

The position of the cursor after scrolling is not defined.

**Example Program:** [binled.ex](#)

**See Also:** [clear\\_screen](#), [text\\_rows](#)

## seek

**Syntax:**           include file.e

i1 = seek(fn, a1)

**Description:**   Seek (move) to any byte position in the file fn or to the end of file if a1 is -1. For each open file there is a current byte position that is updated as a result of I/O operations on the file. The initial file position is 0 for files opened for read, write or update. The initial position is the end of file for files opened for append. The value returned by seek() is 0 if the seek was successful, and non-zero if it was unsuccessful. It is possible to seek past the end of a file. If you seek past the end of the file, and write some data, undefined bytes will be inserted into the gap between the original end of file and your new data.

**Comments:**      After seeking and reading (writing) a series of bytes, you may need to call seek() explicitly before you switch to writing (reading) bytes, even though the file position should already be what you want.

This function is normally used with files opened in binary mode. In text mode, DOS converts CR LF to LF on input, and LF to CR LF on output, which can cause great confusion when you are trying to count bytes.

**Example:**

```
include file.e

integer fn
fn = open("mydata", "rb")
-- read and display first line of file 3 times:
for i = 1 to 3 do
    puts(1, gets(fn))
    if seek(fn, 0) then
        puts(1, "rewind failed!\n")
    end if
end for
```

**See Also:**      [where](#), [open](#)



## sequence

**Syntax:** `i = sequence(x)`

**Description:** Return 1 if x is a sequence else return 0.

**Comments:** This serves to define the sequence type. You can also call it like an ordinary function to determine if an object is a sequence.

**Example 1:**

```
sequence s
s = {1,2,3}
```

**Example 2:**

```
if sequence(x) then
  sum = 0
  for i = 1 to length(x) do
    sum = sum + x[i]
  end for
else
  -- x must be an atom
  sum = x
end if
```

**See Also:** [atom](#), [object](#), [integer](#), [atoms and sequences](#)

## set\_active\_page

**Platform:** DOS32

**Syntax:** include image.e

set\_active\_page(i)

**Description:** Select video page i to send all screen output to.

**Comments:** With multiple pages you can instantaneously change the entire screen without causing any visible "flicker". You can also save the screen and bring it back quickly.

video\_config() will tell you how many pages are available in the current graphics mode.

By default, the active page and the display page are both 0.

This works under DOS, or in a full-screen DOS window. In a partial-screen window you cannot change the active page.

**Example:**

```
include image.e
```

```
-- active & display pages are initially both 0
puts(1, "\nThis is page 0\n")
set_active_page(1)      -- screen output will now go to page 1
clear_screen()
puts(1, "\nNow we've flipped to page 1\n")
if getc(0) then         -- wait for key-press
end if
set_display_page(1)     -- "Now we've ..." becomes visible
if getc(0) then         -- wait for key-press
end if
set_display_page(0)     -- "This is ..." becomes visible again
set_active_page(0)
```

**See Also:** [get\\_active\\_page](#), [set\\_display\\_page](#), [video\\_config](#)

## set\_display\_page

**Platform:** DOS32

**Syntax:** include image.e

set\_display\_page(i)

**Description:** Set video page i to be mapped to the visible screen.

**Comments:** With multiple pages you can instantaneously change the entire screen without causing any visible "flicker". You can also save the screen and bring it back quickly.

video\_config() will tell you how many pages are available in the current graphics mode.

By default, the active page and the display page are both 0.

This works under DOS, or in a full-screen DOS window. In a partial-screen window you cannot change the active page.

**Example:** See set\_active\_page() example.

**See Also:** [get\\_display\\_page](#), [set\\_active\\_page](#), [video\\_config](#)

## set\_rand

**Syntax:** include machine.e

set\_rand(i1)

**Description:** Set the random number generator to a certain state, i1, so that you will get a known series of random numbers on subsequent calls to rand().

**Comments:** Normally the numbers returned by the rand() function are totally unpredictable, and will be different each time you run your program. Sometimes however you may wish to repeat the same series of numbers, perhaps because you are trying to debug your program, or maybe you want the ability to generate the same output (e.g. a random picture) for your user upon request.

**Example:**

```
sequence s, t
s = repeat(0, 3)
t = s

set_rand(12345)
s[1] = rand(10)
s[2] = rand(100)
s[3] = rand(1000)

set_rand(12345)  -- same value for set_rand()
t[1] = rand(10)  -- same arguments to rand() as before
t[2] = rand(100)
t[3] = rand(1000)
-- at this point s and t will be identical
```

**See Also:** [rand](#)

## set\_vector

**Platform:** **DOS32**

**Syntax:** include machine.e

set\_vector(i, s)

**Description:** Set s as the new address for handling interrupt number i. s must be a protected mode **far address** in the form: {16-bit segment, 32-bit offset}.

**Comments:** Before calling set\_vector() you must store a machine-code interrupt handling routine at location s in memory.

The 16-bit segment can be the code segment used by Euphoria. To get the value of this segment see [demo\dos32\hardint.ex](#). The offset can be the 32-bit value returned by allocate(). Euphoria runs in **protected mode** with the code segment and data segment pointing to the same physical memory, but with different access modes.

Interrupts occurring in either **real mode** or **protected mode** will be passed to your handler. Your interrupt handler should immediately load the correct data segment before it tries to reference memory.

Your handler might return from the interrupt using the iretd instruction, or jump to the original interrupt handler. It should save and restore any registers that it modifies.

You should lock the memory used by your handler to ensure that it will never be swapped out. See lock\_memory().

It is highly recommended that you study [demo\dos32\hardint.ex](#) before trying to set up your own interrupt handler.

You should have a good knowledge of machine-level programming before attempting to write your own handler.

You can call set\_vector() with the far address returned by get\_vector(), when you want to restore the original handler.

**Example:**

```
set_vector(#1C, {code_segment, my_handler_address})
```

**Example Program:** [demo\dos32\hardint.ex](#)

**See Also:** [get\\_vector](#), [lock\\_memory](#), [allocate](#)

## sin

**Syntax:** `x2 = sin(x1)`

**Description:** Return the sine of x1, where x1 is in radians.

**Comments:** This function may be applied to an atom or to all elements of a sequence.

**Example:**

```
sin_x = sin({.5, .9, .11})  
-- sin_x is {.479, .783, .110}
```

**See Also:** [cos](#), [tan](#)

## sleep

**Syntax:** include misc.e

sleep(i)

**Description:** Suspend execution for i seconds.

**Comments:** On WIN32 and Linux/FreeBSD, the operating system will suspend your process and schedule other processes. On DOS32, your program will go into a busy loop for i seconds, during which time other processes may run, but they will compete with your process for the CPU.

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()`.

**Example:**

```
puts(1, "Waiting 15 seconds...\n")
sleep(15)
puts(1, "Done.\n")
```

**See Also:** [lock\\_file](#), [abort](#), [time](#)

## sort

**Syntax:** include sort.e

s2 = sort(s1)

**Description:** Sort s1 into ascending order using a fast sorting algorithm. The elements of s1 can be any mix of atoms or sequences. Atoms come before sequences, and sequences are sorted "alphabetically" where the first elements are more significant than the later elements.

**Example 1:**

```
x = 0 & sort({7,5,3,8}) & 0
-- x is set to {0, 3, 5, 7, 8, 0}
```

**Example 2:**

```
y = sort({"Smith", "Jones", "Doe", 5.5, 4, 6})
-- y is {4, 5.5, 6, "Doe", "Jones", "Smith"}
```

**Example 3:**

```
database = sort({{"Smith", 95.0, 29},
                  {"Jones", 77.2, 31},
                  {"Clinton", 88.7, 44}})

-- The 3 database "records" will be sorted by the first "field"
-- i.e. by name. Where the first field (element) is equal it
-- will be sorted by the second field etc.

-- after sorting, database is:
      {{ "Clinton", 88.7, 44 },
        { "Jones", 77.2, 31 },
        { "Smith", 95.0, 29 }}
```

**See Also:** [custom\\_sort](#), [compare](#), [match](#), [find](#)



## sound

**Platform:** DOS32

**Syntax:** include graphics.e

sound(i)

**Description:** Turn on the PC speaker at frequency i. If i is 0 the speaker will be turned off.

**Comments:** On WIN32 and Linux/FreeBSD no sound will be made.

**Example:**

```
sound(1000) -- starts a fairly high pitched sound
```

# sprint

**Syntax:** include misc.e

`s = sprint(x)`

**Description:** The representation of x as a string of characters is returned. This is exactly the same as `print(fn, x)`, except that the output is returned as a sequence of characters, rather than being sent to a file or device. x can be any Euphoria object.

**Comments:** The atoms contained within x will be displayed to a maximum of 10 significant digits, just as with `print()`.

**Example 1:**

```
s = sprint(12345)
-- s is "12345"
```

**Example 2:**

```
s = sprint({10,20,30}+5)
-- s is "{15,25,35}"
```

**See Also:** [print](#), [sprintf](#), [value](#), [get](#)

## sprintf

**Syntax:** `s = sprintf(st, x)`

**Description:** This is exactly the same as `printf()`, except that the output is returned as a sequence of characters, rather than being sent to a file or device. `st` is a format string, `x` is the value or sequence of values to be formatted. `printf(fn, st, x)` is equivalent to `puts(fn, sprintf(st, x))`.

**Comments:** Some typical uses of `sprintf()` are:

1. Converting numbers to strings.
2. Creating strings to pass to `system()`.
3. Creating formatted error messages that can be passed to a common error message handler.

**Example:**

```
s = sprintf("%08d", 12345)
-- s is "00012345"
```

**See Also:** [printf](#), [value](#), [sprintf](#), [get](#), [system](#)

## sqrt

**Syntax:** `x2 = sqrt(x1)`

**Description:** Calculate the square root of x1.

**Comments:** This function may be applied to an atom or to all elements of a sequence.

Taking the square root of a negative number will abort your program with a run-time error message.

**Example:**

```
r = sqrt(16)
-- r is 4
```

**See Also:** [log](#), [power](#)

## system

**Syntax:** system(st, i)

**Description:** Pass a command string st to the operating system command interpreter. The argument i indicates the manner in which to return from the call to system():

When i is 0, the previous graphics mode is restored and the screen is cleared.

When i is 1, a beep sound will be made and the program will wait for the user to press a key before the previous graphics mode is restored.

When i is 2, the graphics mode is not restored and the screen is not cleared.

**Comments:** i = 2 should only be used when it is known that the command executed by system() will not change the graphics mode.

You can use Euphoria as a sophisticated "batch" (.bat) language by making calls to system() and system\_exec().

system() will start a new DOS or Linux/FreeBSD shell.

system() allows you to use command-line redirection of standard input and output in the command string st.

Under **DOS32**, a Euphoria program will start off using extended memory. If extended memory runs out the program will consume conventional memory. If conventional memory runs out it will use virtual memory, i.e. swap space on disk. The DOS command run by system() will fail if there is not enough conventional memory available. To avoid this situation you can reserve some conventional (low) memory by typing:

```
SET CAUSEWAY=LOWMEM:xxx
```

where xxx is the number of K of conventional memory to reserve. Type this before running your program. You can also put this in **autoexec.bat**, or in a **.bat** file that runs your program. For example:

```
SET CAUSEWAY=LOWMEM:80
```

```
ex myprog.ex
```

This will reserve 80K of conventional memory, which should be enough to run simple DOS commands like COPY, MOVE, MKDIR etc.

**Example 1:**

```
system("copy temp.txt a:\\temp.bak", 2)
-- note use of double backslash in literal string to get
-- single backslash
```

**Example 2:**

```
system("ex \\test\\myprog.ex < indata > outdata", 2)
-- executes myprog by redirecting standard input and
-- standard output
```

**See Also:** [system\\_exec](#), [dir](#), [current\\_dir](#), [getenv](#), [command\\_line](#)

## system\_exec

**Syntax:** i1 = system\_exec(st, i2)

**Description:** Try to run the command given by st. st must be a command to run an executable program, possibly with some command-line arguments. If the program can be run, i1 will be the exit code from the program. If it is not possible to run the program, system\_exec() will return -1. i2 is a code that indicates what to do about the graphics mode when system\_exec() is finished. These codes are the same as for system():

When i2 is 0, the previous graphics mode is restored and the screen is cleared.

When i2 is 1, a beep sound will be made and the program will wait for the user to press a key before the previous graphics mode is restored.

When i2 is 2, the graphics mode is not restored and the screen is not cleared.

**Comments:** On DOS32 or WIN32, system\_exec() will only run **.exe** and **.com** programs. To run **.bat** files, or built-in DOS commands, you need system(). Some commands, such as DEL, are not programs, they are actually built-in to the command interpreter.

On DOS32 and WIN32, system\_exec() does not allow the use of command-line redirection in the command string st, nor does it allow you to quote strings that contain blanks, such as file names.

exit codes from DOS or Windows programs are normally in the range 0 to 255, with 0 indicating "success".

You can run a Euphoria program using system\_exec(). A Euphoria program can return an exit code using abort().

system\_exec() does not start a new DOS shell.

### Example 1:

```
integer exit_code
exit_code = system_exec("xcopy temp1.dat temp2.dat", 2)

if exit_code = -1 then
    puts(2, "\n couldn't run xcopy.exe\n")
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("ex \\test\\myprog.ex indata outdata", 2) then
    puts(2, "failure!\n")
end if
```

**See Also:** [system](#), [abort](#)

## tan

**Syntax:** `x2 = tan(x1)`

**Description:** Return the tangent of x1, where x1 is in radians.

**Comments:** This function may be applied to an atom or to all elements of a sequence.

**Example:**

```
t = tan(1.0)
-- t is 1.55741
```

**See Also:** [sin](#), [cos](#), [arctan](#)

## task\_clock\_start

**Syntax:** task\_clock\_start()

**Description:** Restart the clock used for scheduling real-time tasks. Call this routine, some time after calling task\_clock\_stop(), when you want scheduling of real-time tasks to continue.

**Comments:** 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:**

```
-- 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\_clock\_stop

**Syntax:** task\_clock\_stop()

**Description:** Stop the scheduling of real-time tasks. Scheduling will resume when task\_clock\_start() is called. Time-shared tasks can continue. The current task can also continue, unless it's a real-time task and it yields.

**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.

The time() function is not affected by this.

**See Also:** [task\\_clock\\_start](#), [task\\_schedule](#), [task\\_yield](#), [task\\_suspend](#)

## task\_create

**Syntax:**       a2 = task\_create(i1, s)

**Description:**   Create a new task. i1 is the routine id of a user-defined Euphoria procedure. s is the list of arguments that will be passed to this procedure when the task starts executing. The result, a2, is a task identifier, created by the system. It can be used to identify this task to the other Euphoria multitasking routines.

**Comments:**     task\_create() creates a new task, but does not start it executing.

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:**

```
mytask = task_create(routine_id("myproc"), {5, 9, "ABC"})
```

**See Also:**     [task\\_schedule](#), [task\\_yield](#), [task\\_suspend](#), [task\\_self](#)

## task\_list

**Syntax:** s = task\_list()

**Description:** Get a sequence containing the task id's for all active or suspended tasks.

**Comments:** This function lets you find out which tasks currently exist. Tasks that have terminated naturally, or have been killed are not included. You can pass a task id to task\_status() to find out more about a particular task.

**Example:**

```
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

**Syntax:** task\_schedule(a, x)

**Description:** Schedule task a to run using scheduling parameter x. a must be a task identifier returned by task\_create(). If x is a positive atom, then x tells the task scheduler how many times it should run time-shared task a in one burst before it considers running other tasks. If x is a sequence, it must be a 2-element sequence {min, max}, both values greater than or equal to 0, where min and max indicate the minimum and maximum times, in seconds, to wait before running real-time task a. min and max also set the time interval for subsequent runs of task a, unless overridden by another call to task\_schedule() or task\_suspend().

**Comments:** The task scheduler, which is built-in to the Euphoria run-time system, will use the scheduling parameter x 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.

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), 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 and Linux/FreeBSD, and 0.55 seconds on DOS unless tick\_rate() is called) then accurate time scheduling cannot be performed, but the scheduler will try to run the task several times in a row to approximate what is desired. For example, if you ask for a min time of 0.002 seconds, then the scheduler will try to run your task  $.01/.002 = 5$  times in a row before waiting for the clock to "click" ahead by .01. During the next 0.01 seconds it will run your task (up to) another 5 times etc. provided your task can be completed 5 times in one clock period.

At program start-up there is a single task running. Its task id is 0, and initially it's a time-shared task allowed 1 run per task\_yield(). No other task can run until task 0 executes a task\_yield().

If task 0 (top-level) runs off the end of the main file, the whole program terminates, regardless of what other tasks may still be active.

If the scheduler finds that no task is active, i.e. no task will ever run again (not even task 0), it terminates the program with a 0 exit code code, similar to abort(0).

### Example:

```
-- 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\\_suspend](#)

## task\_self

**Syntax:**       a = task\_self()

**Description:** Return the task id of the current task.

**Comments:**    This value may be needed, if a task wants to schedule, kill or suspend itself.

**Example:**

```
        -- schedule self
task_schedule(task_self(), {5.9, 6.0})
```

**See Also:**     [task\\_create](#), [task\\_schedule](#), [task\\_yield](#), [task\\_suspend](#)

## task\_status

**Syntax:** i2 = task\_status(a)

**Description:** Return the status of a task. Status values are 1 (active), 0 (suspended), and -1 (task does not exist)

**Comments:** A task might want to know the status of one or more other tasks when deciding whether to proceed with some processing.

A -1 value could indicate a task that never existed, or a task that terminated naturally or was killed.

**Example:**

```
integer s

s = task_status(tid)
if s = 1 then
    puts(1, "ACTIVE\n")
elsif s = 0 then
    puts(1, "SUSPENDED\n")
else
    puts(1, "DOESN'T EXIST\n")
end if
```

**See Also:** [task\\_list](#), [task\\_create](#), [task\\_schedule](#), [task\\_suspend](#)

## task\_suspend

**Syntax:** task\_suspend(a)

**Description:** Suspend execution of task a.

**Comments:** This causes task a to be suspended. Task a will not be executed again unless there is a call to task\_schedule() for task a. a 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().

**Example:**

```
-- 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

**Syntax:** task\_yield()

**Description:** Yield control to the scheduler. The scheduler can then choose another task to run, or perhaps let the current task continue running.

**Comments:** Tasks should call task\_yield() periodically so other tasks will have a chance to run. Only when task\_yield() is called, is there a way for the scheduler to take back control from a task. This is what's known as cooperative multitasking.

A task can have calls to task\_yield() in many different places in it's code, and at any depth of subroutine call.

The scheduler will use the current value of {min, max} or the current number of consecutive runs remaining, 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 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:**

```
-- 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.
```

```
global 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](#) [task\\_suspend](#)

## text\_color

**Syntax:**       include graphics.e

text\_color(i)

**Description:**   Set the foreground text color. Add 16 to get blinking text in some modes. See [graphics.e](#) for a list of possible colors.

**Comments:**     Text that you print *after* calling text\_color() will have the desired color.

When your program terminates, the last color that you selected and actually printed on the screen will remain in effect. Thus you may have to print something, maybe just '\n', in WHITE to restore white text, especially if you are at the bottom line of the screen, ready to scroll up.

**Example:**

```
text_color(BRIGHT_BLUE)
```

**See Also:**     [bk\\_color](#)

## text\_rows

**Platform:** DOS32, WIN32

**Syntax:** include graphics.e

i2 = text\_rows(i1)

**Description:** Set the number of lines on a **text-mode** screen to i1 if possible. i2 will be set to the actual new number of lines.

**Comments:** Values of 25, 28, 43 and 50 lines are supported by most video cards.

**See Also:** [graphics\\_mode](#)

## tick\_rate

**Platform:** DOS32

**Syntax:** include machine.e

tick\_rate(a)

**Description:** Specify the number of clock-tick interrupts per second. This determines the precision of the time() library routine. It also affects the sampling rate for time profiling.

**Comments:** tick\_rate() is ignored on WIN32 and Linux/FreeBSD. The time resolution on WIN32 is always 100 ticks/second.

On a PC the clock-tick interrupt normally occurs at 18.2 interrupts per second. tick\_rate() lets you increase that rate, but not decrease it.

tick\_rate(0) will restore the rate to the normal 18.2 rate. Euphoria will also restore the rate automatically when it exits, even when it finds an error in your program.

If a program runs in a DOS window with a tick rate other than 18.2, the time() function will not advance unless the window is the active window.

With a tick rate other than 18.2, the time() function on DOS takes about 1/100 the usual time that it needs to execute. On Windows and FreeBSD, time() normally executes very quickly.

While **ex.exe** is running, the system will maintain the correct time of day. However if **ex.exe** should crash (e.g. you see a "CauseWay..." error) while the tick rate is high, you (or your user) may need to reboot the machine to restore the proper rate. If you don't, the system time may advance too quickly. This problem does not occur on **Windows 95/98/NT**, only on **DOS** or **Windows 3.1**. You will always get back the correct time of day from the battery-operated clock in your system when you boot up again.

**Example:**

```
tick_rate(100)
-- time() will now advance in steps of .01 seconds
-- instead of the usual .055 seconds
```

**See Also:** [time](#), [time profiling](#)

## time

**Syntax:** a = time()

**Description:** Return the number of seconds since some fixed point in the past.

**Comments:** Take the difference between two readings of time(), to measure, for example, how long a section of code takes to execute.

The resolution with **DOS32** is normally about 0.05 seconds. On **WIN32 and Linux/FreeBSD** it's about 0.01 seconds.

Under **DOS32** you can improve the resolution by calling tick\_rate().

Under **DOS32** the period of time that you can normally measure is limited to 24 hours. After that, the value returned by time() will reset and start over. If however, you have called tick\_rate(), and clock ticks are happening at a rate that is higher than the usual 18.2/sec, time() will continue much longer, since in that case, Euphoria handles the clock-tick interrupt directly, and accumulates the ticks in a larger, 32-bit variable.

DOS emulation under Windows XP is not perfect. When you do time profiling, (with profile\_time) the time() function might be off by several percent. This problem does not occur on Windows ME/98/95.

**Example:**

```
constant ITERATIONS = 1000000
integer p
atom t0, loop_overhead

t0 = time()
for i = 1 to ITERATIONS do
    -- time an empty loop
end for
loop_overhead = time() - t0

t0 = time()
for i = 1 to ITERATIONS do
    p = power(2, 20)
end for
? (time() - t0 - loop_overhead)/ITERATIONS
-- calculates time (in seconds) for one call to power
```

**See Also:** [date](#), [tick\\_rate](#)

## trace

**Syntax:**       with trace  
trace(i)

**Description:**   If i is 1 or 2, turn on full-screen interactive statement tracing/debugging. If i is 3, turn on tracing of statements to a file called **ctrace.out**. If i is 0, turn off tracing. When i is 1 a color display appears. When i is 2 a monochrome trace display appears. Tracing can only occur in routines that were compiled "**with trace**", and trace() has no effect unless it is executed in a "**with trace**" section of your program.

See [Part I - 3.1 Debugging](#) for a full discussion of tracing / debugging.

**Comments:**     Use trace(2) if the color display is hard to view on your system.

All forms of trace() are supported by the Interpreter.

Only trace(3) is supported by the Euphoria To C Translator.

**Example:**

```
      if x < 0 then
        -- ok, here's the case I want to debug...
        trace(1)
        -- etc.
        ...
end if
```

**See Also:**     [profile](#), [debugging and profiling](#)

## ... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | from S to T | [from U to Z](#)

## unlock\_file

**Syntax:** include file.e

unlock\_file(fn, s)

**Description:** Unlock an open file fn, or a portion of file fn. You must have previously locked the file using lock\_file(). On DOS32 and WIN32 you can unlock a range of bytes within a file by specifying the s parameter as {first\_byte, last\_byte}. The same range of bytes must have been locked by a previous call to lock\_file(). On Linux/FreeBSD you can currently only lock or unlock an entire file. The s parameter should be {} when you want to unlock an entire file. On Linux/FreeBSD, s must always be {}.

**Comments:** You should unlock a file as soon as possible so other processes can use it.  
Any files that you have locked, will automatically be unlocked when your program terminates.

See lock\_file() for further comments and an example.

**See Also:** [lock\\_file](#)

## unregister\_block

**Syntax:** include machine.e (or safe.e)

unregister\_block(a)

**Description:** Remove a block of memory from the list of safe blocks maintained by [safe.e](#) (the debug version of [machine.e](#)). The block starts at address a.

**Comments:** This routine is only meant to be used for **debugging purposes**. Use it to unregister blocks of memory that you have previously registered using register\_block(). By unregistering a block, you remove it from the list of safe blocks maintained by [safe.e](#). This prevents your program from performing any further reads or writes of memory within the block.

See register\_block() for further comments and an example.

**See Also:** [register\\_block](#), [safe.e](#)

## upper

**Syntax:** include wildcard.e

x2 = upper(x1)

**Description:** Convert an atom or sequence to upper case.

**Example:**

```
s = upper("Euphoria")
-- s is "EUPHORIA"

a = upper('g')
-- a is 'G'

s = upper({"Euphoria", "Programming"})
-- s is {"EUPHORIA", "PROGRAMMING"}
```

**See Also:** [lower](#)

## use\_vesa

**Platform:** **DOS32**

**Syntax:** include machine.e

use\_vesa(i)

**Description:** use\_vesa(1) will force Euphoria to use the VESA graphics standard. This may cause Euphoria programs to work better in SVGA graphics modes with certain video cards. use\_vesa(0) will restore Euphoria's original method of using the video card.

**Comments:** Most people can ignore this. However if you experience difficulty in SVGA graphics modes you should try calling use\_vesa(1) at the start of your program before any calls to graphics\_mode().

Arguments to use\_vesa() other than 0 or 1 should not be used.

**Example:**

```
        use_vesa(1)
fail = graphics_mode(261)
```

**See Also:** [graphics\\_mode](#)

## value

**Syntax:** include get.e

s = value(st)

**Description:** Read the string representation of a Euphoria object, and compute the value of that object. A 2-element sequence, {**error\_status**, **value**} is actually returned, where error\_status can be one of:

```
GET_SUCCESS -- a valid object representation was found
GET_EOF     -- end of string reached too soon
GET_FAIL    -- syntax is wrong
```

**Comments:** This works the same as **get()**, but it reads from a string that you supply, rather than from a file or device.

After reading one valid representation of a Euphoria object, value() will stop reading and ignore any additional characters in the string. For example, "36" and "36P" will both give you {GET\_SUCCESS, 36}.

**Example 1:**

```
s = value("12345")
-- s is {GET_SUCCESS, 12345}
```

**Example 2:**

```
s = value("{0, 1, -99.9}")
-- s is {GET_SUCCESS, {0, 1, -99.9}}
```

**Example 3:**

```
s = value("+++")
-- s is {GET_FAIL, 0}
```

**See Also:** [get](#), [sprintf](#), [print](#)



## video\_config

**Syntax:**       include graphics.e

s = video\_config()

**Description:** Return a sequence of values describing the current video configuration:

{color monitor?, graphics mode, text rows, text columns, xpixels, ypixels, number of colors, number of pages}

The following constants are defined in [graphics.e](#):

```
global constant VC_COLOR    = 1,
                  VC_MODE    = 2,
                  VC_LINES   = 3,
                  VC_COLUMNS = 4,
                  VC_XPIXELS = 5,
                  VC_YPIXELS = 6,
                  VC_NCOLORS = 7,
                  VC_PAGES   = 8
```

**Comments:** This routine makes it easy for you to parameterize a program so it will work in many different graphics modes.

On the PC there are two types of graphics mode. The first type, **text mode**, lets you print text only. The second type, **pixel-graphics mode**, lets you plot pixels, or points, in various colors, as well as text. You can tell that you are in a **text mode**, because the VC\_XPIXELS and VC\_YPIXELS fields will be 0. Library routines such as [polygon\(\)](#), [draw\\_line\(\)](#), and [ellipse\(\)](#) only work in a **pixel-graphics mode**.

**Example:**

```
vc = video_config()  -- in mode 3 with 25-lines of text:
-- vc is {1, 3, 25, 80, 0, 0, 32, 8}
```

**See Also:**   [graphics\\_mode](#)

## wait\_key

**Syntax:**       include get.e

i = wait\_key()

**Description:** Return the next key pressed by the user. Don't return until a key is pressed.

**Comments:** You could achieve the same result using [get\\_key\(\)](#) as follows:

```
while 1 do
  k = get_key()
  if k != -1 then
    exit
  end if
end while
```

However, on multi-tasking systems like **Windows** or **Linux/FreeBSD**, this "busy waiting" would tend to slow the system down. [wait\\_key\(\)](#) lets the operating system do other useful work while your program is waiting for the user to press a key.

You could also use [getc\(0\)](#), assuming file number 0 was input from the keyboard, except that you wouldn't pick up the special codes for function keys, arrow keys etc.

**See Also:**   [get\\_key](#), [getc](#)

## walk\_dir

**Syntax:** include file.e

i1 = walk\_dir(st, i2, i3)

**Description:** This routine will "walk" through a directory with path name given by st. i2 is the **routine id** of a routine that you supply. walk\_dir() will call your routine once for each file and subdirectory in st. If i3 is non-zero (TRUE), then the subdirectories in st will be walked through recursively.

The routine that you supply should accept the path name and [dir\(\) entry](#) for each file and subdirectory. It should return 0 to keep going, or non-zero to stop walk\_dir().

**Comments:** This mechanism allows you to write a simple function that handles one file at a time, while walk\_dir() handles the process of walking through all the files and subdirectories.

By default, the files and subdirectories will be visited in alphabetical order. To use a different order, set the global integer **my\_dir** to the **routine id** of your own *modified* dir() function that sorts the directory entries differently. See the default dir() function in [file.e](#).

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.

**Example:**

```
function look_at(sequence path_name, sequence entry)
-- this function accepts two sequences as arguments
    printf(1, "%s\\%s: %d\\n",
        {path_name, entry[D_NAME], entry[D_SIZE]})
    return 0 -- keep going
end function

exit_code = walk_dir("C:\\MYFILES", routine_id("look_at"), TRUE)
```

**Example Program:** [euphoria/bin/search.ex](#)

**See Also:** [dir](#), [current\\_dir](#)

## where

**Syntax:** include file.e

a1 = where(fn)

**Description:** This function returns the current byte position in the file fn. This position is updated by reads, writes and seeks on the file. It is the place in the file where the next byte will be read from, or written to.

**See Also:** [seek](#), [open](#)

## wildcard\_file

**Syntax:** include wildcard.e

i = wildcard\_file(st1, st2)

**Description:** Return 1 (true) if the filename st2 matches the wild card pattern st1. Return 0 (false) otherwise. This is similar to DOS wildcard matching, but better in some cases. \* matches any 0 or more characters, ? matches any single character. On Linux and FreeBSD the character comparisons are case

sensitive. On DOS and Windows they are not.

**Comments:** You might use this function to check the output of the `dir()` routine for file names that match a pattern supplied by the user of your program.

In DOS `"*ABC.*"` will match *all* files. `wildcard_file("*ABC.*", s)` will only match when the file name part has "ABC" at the end (as you would expect).

**Example 1:**

```
i = wildcard_file("AB*CD.?", "aB123cD.e")
-- i is set to 1 on DOS or Windows, 0 on Linux or FreeBSD
```

**Example 2:**

```
i = wildcard_file("AB*CD.?", "abcd.ex")
-- i is set to 0 on all systems,
-- because the file type has 2 letters not 1
```

**Example Program:** [bin\search.ex](#)

**See Also:** [wildcard\\_match](#), [dir](#)

## wildcard\_match

**Syntax:** `include wildcard.e`

`i = wildcard_match(st1, st2)`

**Description:** This function performs a general matching of a string against a pattern containing \* and ? wildcards. It returns 1 (true) if string st2 matches pattern st1. It returns 0 (false) otherwise. \* matches any 0 or more characters. ? matches any single character. Character comparisons are case sensitive.

**Comments:** If you want case insensitive comparisons, pass both st1 and st2 through `upper()`, or both through `lower()` before calling `wildcard_match()`.

If you want to detect a pattern anywhere within a string, add \* to each end of the pattern:

```
i = wildcard_match('*' & pattern & '*', string)
```

There is currently no way to treat \* or ? literally in a pattern.

**Example 1:**

```
i = wildcard_match("A?B*", "AQBXXYY")
-- i is 1 (TRUE)
```

**Example 2:**

```
i = wildcard_match("*xyz*", "AAAbbbxyz")
-- i is 1 (TRUE)
```

**Example 3:**

```
i = wildcard_match("A*B*C", "a111b222c")
-- i is 0 (FALSE) because upper/lower case doesn't match
```

**Example Program:** [bin\search.ex](#)

**See Also:** [wildcard\\_file](#), [match](#), [upper](#), [lower](#), [compare](#)

## wrap

**Syntax:** include graphics.e

wrap(i)

**Description:** Allow text to wrap at the right margin (i = 1) or get truncated (i = 0).

**Comments:** By default text will wrap.

Use wrap() in **text modes** or **pixel-graphics modes** when you are displaying long lines of text.

**Example:**

```
puts(1, repeat('x', 100) & "\n\n")
-- now have a line of 80 'x' followed a line of 20 more 'x'
wrap(0)
puts(1, repeat('x', 100) & "\n\n")
-- creates just one line of 80 'x'
```

**See Also:** [puts](#), [position](#)

## xor\_bits

**Syntax:** x3 = xor\_bits(x1, x2)

**Description:** Perform the logical XOR (exclusive OR) operation on corresponding bits in x1 and x2. A bit in x3 will be 1 when one of the two corresponding bits in x1 or x2 is 1, and the other is 0.

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

**Example 1:**

```
a = xor_bits(#0110, #1010)
-- a is #1100
```

**See Also:** [and\\_bits](#), [or\\_bits](#), [not\\_bits](#), [int\\_to\\_bits](#), [int\\_to\\_bytes](#)

## ... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | from U to Z

## unlock\_file

**Syntax:**       include file.e

unlock\_file(fn, s)

**Description:**   Unlock an open file fn, or a portion of file fn. You must have previously locked the file using lock\_file(). On DOS32 and WIN32 you can unlock a range of bytes within a file by specifying the s parameter as {first\_byte, last\_byte}. The same range of bytes must have been locked by a previous call to lock\_file(). On Linux/FreeBSD you can currently only lock or unlock an entire file. The s parameter should be {} when you want to unlock an entire file. On Linux/FreeBSD, s must always be {}.

**Comments:**     You should unlock a file as soon as possible so other processes can use it.  
Any files that you have locked, will automatically be unlocked when your program terminates.  
See lock\_file() for further comments and an example.

**See Also:**      [lock\\_file](#)

## unregister\_block

**Syntax:**       include machine.e (or safe.e)

unregister\_block(a)

**Description:**   Remove a block of memory from the list of safe blocks maintained by [safe.e](#) (the debug version of [machine.e](#)). The block starts at address a.

**Comments:**     This routine is only meant to be used for **debugging purposes**. Use it to unregister blocks of memory that you have previously registered using register\_block(). By unregistering a block, you remove it from the list of safe blocks maintained by [safe.e](#). This prevents your program from performing any further reads or writes of memory within the block.

See register\_block() for further comments and an example.

**See Also:**       [register\\_block](#), [safe.e](#)

## upper

**Syntax:**       include wildcard.e

x2 = upper(x1)

**Description:**   Convert an atom or sequence to upper case.

**Example:**

```
s = upper("Euphoria")
-- s is "EUPHORIA"

a = upper('g')
-- a is 'G'

s = upper({"Euphoria", "Programming"})
-- s is {"EUPHORIA", "PROGRAMMING"}
```

**See Also:**     [lower](#)

## use\_vesa

**Platform:** **DOS32**

**Syntax:** include machine.e

use\_vesa(i)

**Description:** use\_vesa(1) will force Euphoria to use the VESA graphics standard. This may cause Euphoria programs to work better in SVGA graphics modes with certain video cards. use\_vesa(0) will restore Euphoria's original method of using the video card.

**Comments:** Most people can ignore this. However if you experience difficulty in SVGA graphics modes you should try calling use\_vesa(1) at the start of your program before any calls to graphics\_mode().

Arguments to use\_vesa() other than 0 or 1 should not be used.

**Example:**

```
        use_vesa(1)
fail = graphics_mode(261)
```

**See Also:** [graphics\\_mode](#)



## value

**Syntax:** include get.e

s = value(st)

**Description:** Read the string representation of a Euphoria object, and compute the value of that object. A 2-element sequence, **{error\_status, value}** is actually returned, where error\_status can be one of:

```
GET_SUCCESS -- a valid object representation was found
GET_EOF     -- end of string reached too soon
GET_FAIL    -- syntax is wrong
```

**Comments:** This works the same as **get()**, but it reads from a string that you supply, rather than from a file or device.

After reading one valid representation of a Euphoria object, value() will stop reading and ignore any additional characters in the string. For example, "36" and "36P" will both give you {GET\_SUCCESS, 36}.

**Example 1:**

```
s = value("12345")
-- s is {GET_SUCCESS, 12345}
```

**Example 2:**

```
s = value("{0, 1, -99.9}")
-- s is {GET_SUCCESS, {0, 1, -99.9}}
```

**Example 3:**

```
s = value("+++")
-- s is {GET_FAIL, 0}
```

**See Also:** [get](#), [sprintf](#), [print](#)

## video\_config

**Syntax:** include graphics.e

s = video\_config()

**Description:** Return a sequence of values describing the current video configuration:  
{color monitor?, graphics mode, text rows, text columns, xpixels, ypixels, number of colors, number of pages}

The following constants are defined in [graphics.e](#):

```
global constant VC_COLOR    = 1,
                  VC_MODE    = 2,
                  VC_LINES   = 3,
                  VC_COLUMNS = 4,
                  VC_XPIXELS = 5,
                  VC_YPIXELS = 6,
                  VC_NCOLORS = 7,
                  VC_PAGES   = 8
```

**Comments:** This routine makes it easy for you to parameterize a program so it will work in many different graphics modes.

On the PC there are two types of graphics mode. The first type, **text mode**, lets you print text only. The second type, **pixel-graphics mode**, lets you plot pixels, or points, in various colors, as well as text. You can tell that you are in a **text mode**, because the VC\_XPIXELS and VC\_YPIXELS fields will be 0. Library routines such as [polygon\(\)](#), [draw\\_line\(\)](#), and [ellipse\(\)](#) only work in a **pixel-graphics mode**.

**Example:**

```
vc = video_config()  -- in mode 3 with 25-lines of text:
-- vc is {1, 3, 25, 80, 0, 0, 32, 8}
```

**See Also:** [graphics\\_mode](#)

## wait\_key

**Syntax:** include get.e

i = wait\_key()

**Description:** Return the next key pressed by the user. Don't return until a key is pressed.

**Comments:** You could achieve the same result using **get\_key()** as follows:

```
while 1 do
  k = get_key()
  if k != -1 then
    exit
  end if
end while
```

However, on multi-tasking systems like **Windows** or **Linux/FreeBSD**, this "busy waiting" would tend to slow the system down. **wait\_key()** lets the operating system do other useful work while your program is waiting for the user to press a key.

You could also use **getc(0)**, assuming file number 0 was input from the keyboard, except that you wouldn't pick up the special codes for function keys, arrow keys etc.

**See Also:** [get\\_key](#), [getc](#)

## walk\_dir

**Syntax:** include file.e

i1 = walk\_dir(st, i2, i3)

**Description:** This routine will "walk" through a directory with path name given by st. i2 is the **routine id** of a routine that you supply. walk\_dir() will call your routine once for each file and subdirectory in st. If i3 is non-zero (TRUE), then the subdirectories in st will be walked through recursively.

The routine that you supply should accept the path name and [dir\(\) entry](#) for each file and subdirectory. It should return 0 to keep going, or non-zero to stop walk\_dir().

**Comments:** This mechanism allows you to write a simple function that handles one file at a time, while walk\_dir() handles the process of walking through all the files and subdirectories.

By default, the files and subdirectories will be visited in alphabetical order. To use a different order, set the global integer **my\_dir** to the **routine id** of your own *modified* dir() function that sorts the directory entries differently. See the default dir() function in [file.e](#).

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.

**Example:**

```
function look_at(sequence path_name, sequence entry)
-- this function accepts two sequences as arguments
    printf(1, "%s\\%s: %d\\n",
           {path_name, entry[D_NAME], entry[D_SIZE]})
    return 0 -- keep going
end function

exit_code = walk_dir("C:\\MYFILES", routine_id("look_at"), TRUE)
```

**Example Program:** [euphoria/bin/search.ex](#)

**See Also:** [dir](#), [current\\_dir](#)

## where

**Syntax:** include file.e

a1 = where(fn)

**Description:** This function returns the current byte position in the file fn. This position is updated by reads, writes and seeks on the file. It is the place in the file where the next byte will be read from, or written to.

**See Also:** [seek](#), [open](#)

## wildcard\_file

**Syntax:**       include wildcard.e

i = wildcard\_file(st1, st2)

**Description:**   Return 1 (true) if the filename st2 matches the wild card pattern st1. Return 0 (false) otherwise. This is similar to DOS wildcard matching, but better in some cases. \* matches any 0 or more characters, ? matches any single character. On Linux and FreeBSD the character comparisons are case sensitive. On DOS and Windows they are not.

**Comments:**     You might use this function to check the output of the dir() routine for file names that match a pattern supplied by the user of your program.

In DOS "\*ABC.\*" will match *all* files. wildcard\_file("\*ABC.\*", s) will only match when the file name part has "ABC" at the end (as you would expect).

### Example 1:

```
i = wildcard_file("AB*CD.?", "aB123cD.e")
-- i is set to 1 on DOS or Windows, 0 on Linux or FreeBSD
```

### Example 2:

```
i = wildcard_file("AB*CD.?", "abcd.ex")
-- i is set to 0 on all systems,
-- because the file type has 2 letters not 1
```

**Example Program:**   [bin\search.ex](#)

**See Also:**       [wildcard\\_match](#), [dir](#)

## wildcard\_match

**Syntax:**       include wildcard.e

i = wildcard\_match(st1, st2)

**Description:**   This function performs a general matching of a string against a pattern containing \* and ? wildcards. It returns 1 (true) if string st2 matches pattern st1. It returns 0 (false) otherwise. \* matches any 0 or more characters. ? matches any single character. Character comparisons are case sensitive.

**Comments:**     If you want case insensitive comparisons, pass both st1 and st2 through upper(), or both through lower() before calling wildcard\_match().

If you want to detect a pattern anywhere within a string, add \* to each end of the pattern:

```
i = wildcard_match('*' & pattern & '*', string)
```

There is currently no way to treat \* or ? literally in a pattern.

**Example 1:**

```
i = wildcard_match("A?B*", "AQBXXYY")
-- i is 1 (TRUE)
```

**Example 2:**

```
i = wildcard_match("*xyz*", "AAAbbbxyz")
-- i is 1 (TRUE)
```

**Example 3:**

```
i = wildcard_match("A*B*C", "a111b222c")
-- i is 0 (FALSE) because upper/lower case doesn't match
```

**Example Program:**   [bin\search.ex](#)

**See Also:**       [wildcard\\_file](#), [match](#), [upper](#), [lower](#), [compare](#)

## wrap

**Syntax:** include graphics.e

wrap(i)

**Description:** Allow text to wrap at the right margin (i = 1) or get truncated (i = 0).

**Comments:** By default text will wrap.

Use wrap() in **text modes** or **pixel-graphics modes** when you are displaying long lines of text.

**Example:**

```
    puts(1, repeat('x', 100) & "\n\n")
-- now have a line of 80 'x' followed a line of 20 more 'x'
wrap(0)
puts(1, repeat('x', 100) & "\n\n")
-- creates just one line of 80 'x'
```

**See Also:** [puts](#), [position](#)



## xor\_bits

**Syntax:** `x3 = xor_bits(x1, x2)`

**Description:** Perform the logical XOR (exclusive OR) operation on corresponding bits in x1 and x2. A bit in x3 will be 1 when one of the two corresponding bits in x1 or x2 is 1, and the other is 0.

**Comments:** The arguments to this function may be atoms or sequences. The rules for [operations on sequences](#) apply.

The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

**Example 1:**

```
a = xor_bits(#0110, #1010)
-- a is #1100
```

**See Also:** [and\\_bits](#), [or\\_bits](#), [not\\_bits](#), [int\\_to\\_bits](#), [int\\_to\\_bytes](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | from U to Z

```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- routines and constants for dynamic linking to C functions

-- C types for .dll arguments and return value:
global constant
    C_CHAR      = #01000001,
    C_UCHAR     = #02000001,
    C_SHORT     = #01000002,
    C_USHORT    = #02000002,
    C_INT       = #01000004,
    C_UINT      = #02000004,
    C_LONG      = C_INT,
    C_ULONG     = C_UINT,
    C_POINTER   = C_ULONG,
    C_FLOAT     = #03000004,
    C_DOUBLE    = #03000008

-- Euphoria types for .dll arguments and return value:
global constant
    E_INTEGER = #06000004,
    E_ATOM    = #07000004,
    E_SEQUENCE = #08000004,
    E_OBJECT  = #09000004

global constant NULL = 0 -- NULL pointer

constant M_OPEN_DLL = 50,
    M_DEFINE_C = 51,
    M_CALL_BACK = 52,
    M_FREE_CONSOLE = 54,
    M_DEFINE_VAR = 56

global function open_dll(sequence file_name)
-- Open a .DLL file
    return machine_func(M_OPEN_DLL, file_name)
end function

global function define_c_var(atom lib, sequence variable_name)
-- get the memory address where a global C variable is stored
    return machine_func(M_DEFINE_VAR, {lib, variable_name})
end function

global function define_c_proc(object lib, object routine_name,
    sequence arg_types)
-- Define a C function with VOID return type, or where the
-- return value will always be ignored.
-- Alternatively, define a machine-code routine at a given address.
    return machine_func(M_DEFINE_C, {lib, routine_name, arg_types, 0})
end function

global function define_c_func(object lib, object routine_name,
    sequence arg_types, atom return_type)

```

```

-- define a C function (or machine code routine)
    return machine_func(M_DEFINE_C, {lib, routine_name, arg_types, return_type})
end function

global function call_back(object id)
-- return a 32-bit call-back address for a Euphoria routine
-- id can be of the form:
--     routine_id          - for Linux or Windows stdcall calls,
-- or
--     {'+', routine_id}   - for Windows cdecl calls
    return machine_func(M_CALL_BACK, id)
end function

global procedure free_console()
-- Delete the console text-window (if one currently exists).
-- Call this if you are getting an unwanted "Press Enter" message
-- at the end of execution of your program on Linux/FreeBSD or Windows.
    machine_proc(M_FREE_CONSOLE, 0)
end procedure

```

## open\_dll

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

a = open\_dll(st)

**Description:** Open a Windows dynamic link library (.dll) file, or a Linux or FreeBSD shared library (.so) file. A 32-bit address will be returned, or 0 if the .dll can't be found. st can be a relative or an absolute file name. Windows will use the normal search path for locating .dll files.

**Comments:** The value returned by open\_dll() can be passed to define\_c\_proc(), define\_c\_func(), or define\_c\_var().

You can open the same .dll or .so file multiple times. No extra memory is used and you'll get the same number returned each time.

Euphoria will close the .dll for you automatically at the end of execution.

**Example:**

```
atom user32
user32 = open_dll("user32.dll")
if user32 = 0 then
    puts(1, "Couldn't open user32.dll!\n")
end if
```

**See Also:** [define\\_c\\_func](#), [define\\_c\\_proc](#), [define\\_c\\_var](#), [c\\_func](#), [c\\_proc](#), [platform.doc](#)

## define\_c\_var

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

a1 = define\_c\_var(a2, s)

**Description:** a2 is the address of a Linux or FreeBSD shared library, or Windows .dll, as returned by open\_dll(). s is the name of a global C variable defined within the library. a1 will be the memory address of variable s.

**Comments:** Once you have the address of a C variable, and you know its type, you can use peek() and poke() to read or write the value of the variable.

**Example Program:** euphoria/demo/linux/mylib.exu

**See Also:** [c\\_proc](#), [define\\_c\\_func](#), [c\\_func](#), [open\\_dll](#), [platform.doc](#)

## define\_c\_proc

**Syntax:**           include dll.e  
i1 = define\_c\_proc(x1, x2, s1)

**Description:**    Define the characteristics of either a C function, or a machine-code routine that you wish to call as a procedure from your Euphoria program. A small integer, known as a **routine id**, will be returned. Use this routine id as the first argument to c\_proc() when you wish to call the routine from Euphoria.

When defining a C function, x1 is the address of the library containing the C function, while x2 is the name of the C function. x1 is a value returned by open\_dll(). If the C function can't be found, -1 will be returned as the routine id. On Windows, you can add a '+' character as a prefix to the function name. This tells Euphoria that the function uses the **cdecl** calling convention. By default, Euphoria assumes that C routines accept the **stdcall** convention.

When defining a machine code routine, x1 must be the empty sequence, "" or {}, and x2 indicates the address of the machine code routine. You can poke the bytes of machine code into a block of memory reserved using allocate(). On Windows, the machine code routine is normally expected to follow the **stdcall** calling convention, but if you wish to use the **cdecl** convention instead, you can code {'+', **address**} instead of **address**.

s1 is a list of the parameter types for the function. A list of C types is contained in [dll.e](#), and [shown above](#). These can be used to define machine code parameters as well.

The C function that you define could be one created by the Euphoria To C Translator, in which case you can pass Euphoria data to it, and receive Euphoria data back. A list of Euphoria types is contained in [dll.e](#), and [shown above](#).

**Comments:**       You can pass any C integer type or pointer type. You can also pass a Euphoria atom as a C double or float.

Parameter types which use 4 bytes or less are all passed the same way, so it is not necessary to be exact.

Currently, there is no way to pass a C structure by value. You can only pass a pointer to a structure.

The C function can return a value but it will be ignored. If you want to use the value returned by the C function, you must instead define it with define\_c\_func() and call it with c\_func().

### Example:

```
atom user32
integer ShowWindow

-- open user32.dll - it contains the ShowWindow C function
user32 = open_dll("user32.dll")

-- It has 2 parameters that are both C int.
ShowWindow = define_c_proc(user32, "ShowWindow", {C_INT, C_INT})
-- If ShowWindow used the cdecl convention,
-- we would have coded "+ShowWindow" here

if ShowWindow = -1 then
    puts(1, "ShowWindow not found!\n")
end if
```

**See Also:**       [c\\_proc](#), [define\\_c\\_func](#), [c\\_func](#), [open\\_dll](#), [platform.doc](#)



## define\_c\_func

**Syntax:**       include dll.e  
i1 = define\_c\_func(x1, x2, s1, i2)

**Description:**   Define the characteristics of either a C function, or a machine-code routine that returns a value. A small integer, i1, known as a **routine id**, will be returned. Use this routine id as the first argument to c\_func() when you wish to call the function from Euphoria.

When defining a C function, x1 is the address of the library containing the C function, while x2 is the name of the C function. x1 is a value returned by open\_dll(). If the C function can't be found, -1 will be returned as the routine id. On Windows, you can add a '+' character as a prefix to the function name. This indicates to Euphoria that the function uses the **cdecl** calling convention. By default, Euphoria assumes that C routines accept the **stdcall** convention.

When defining a machine code routine, x1 must be the empty sequence, "" or {}, and x2 indicates the address of the machine code routine. You can poke the bytes of machine code into a block of memory reserved using allocate(). On Windows, the machine code routine is normally expected to follow the **stdcall** calling convention, but if you wish to use the **cdecl** convention instead, you can code {'+', **address**} instead of **address** for x2.

s1 is a list of the parameter types for the function. i2 is the return type of the function. A list of C types is contained in [dll.e](#), and these can be used to define machine code parameters as well:

```
global constant C_CHAR = #01000001,  
                C_UCHAR = #02000001,  
                C_SHORT = #01000002,  
                C_USHORT = #02000002,  
                C_INT = #01000004,  
                C_UINT = #02000004,  
                C_LONG = C_INT,  
                C_ULONG = C_UINT,  
                C_POINTER = C_ULONG,  
                C_FLOAT = #03000004,  
                C_DOUBLE = #03000008
```

The C function that you define could be one created by the Euphoria To C Translator, in which case you can pass Euphoria data to it, and receive Euphoria data back. A list of Euphoria types is contained in [dll.e](#):

```
global constant  
    E_INTEGER = #06000004,  
    E_ATOM    = #07000004,  
    E_SEQUENCE = #08000004,  
    E_OBJECT  = #09000004
```

**Comments:**    You can pass or return any C integer type or pointer type. You can also pass a Euphoria atom as a C double or float, and get a C double or float returned to you as a Euphoria atom.

Parameter types which use 4 bytes or less are all passed the same way, so it is not necessary to be exact 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.

If you are not interested in using the value returned by the C function, you should instead define it with



define\_c\_proc() and call it with c\_proc().

If you use exw to call a cdecl C routine that returns a floating-point value, it might not work. This is because the Watcom C compiler (used to build exw) has a non-standard way of handling cdecl floating-point return values.

Passing floating-point values to a machine code routine will be faster if you use c\_func() rather than call() to call the routine, since you won't have to use atom\_to\_float64() and poke() to get the floating-point values into memory.

ex.exe (DOS) uses calls to WATCOM floating-point routines (which then use hardware floating-point instructions if available), so floating-point values are generally passed and returned in integer register-pairs rather than floating-point registers. You'll have to disassemble some Watcom code to see how it works.

**Example:**

```
atom user32
integer LoadIcon

-- open user32.dll - it contains the LoadIconA C function
user32 = open_dll("user32.dll")

-- It takes a C pointer and a C int as parameters.
-- It returns a C int as a result.
LoadIcon = define_c_func(user32, "LoadIconA",
                        {C_POINTER, C_INT}, C_INT)

-- We use "LoadIconA" here because we know that LoadIconA
-- needs the stdcall convention, as do
-- all standard .dll routines in the WIN32 API.
-- To specify the cdecl convention, we would have used "+LoadIconA".

if LoadIcon = -1 then
    puts(1, "LoadIconA could not be found!\n")
end if
```

**See Also:**     euphoria\demo\callmach.ex, [c\\_func](#), [define\\_c\\_proc](#), [c\\_proc](#), [open\\_dll](#), [platform.doc](#)

## call\_back

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

a = call\_back(i)

or

a = call\_back({i1, i})

**Description:** Get a machine address for the Euphoria routine with **routine id** i. This address can be used by Windows, or an external C routine in a Windows .dll or Linux/FreeBSD shared library (.so), as a 32-bit "call-back" address for calling your Euphoria routine. On Windows, you can specify i1, which determines the C calling convention that can be used to call your routine. If i1 is '+', then your routine will work with the **cdecl** calling convention. By default it will work with the **stdcall** convention. On Linux and FreeBSD you should only use the first form, as there is just one standard calling convention

**Comments:** You can set up as many call-back functions as you like, but they must all be Euphoria functions (or types) with 0 to 9 arguments. If your routine has nothing to return (it should really be a procedure), just return 0 (say), and the calling C routine can ignore the result.

When your routine is called, the argument values will all be 32-bit unsigned (positive) values. You should declare each parameter of your routine as **atom**, unless you want to impose tighter checking. Your routine must return a 32-bit integer value.

You can also use a call-back address to specify a Euphoria routine as an exception handler in the Linux/FreeBSD signal() function. For example, you might want to catch the SIGTERM signal, and do a graceful shutdown. Some Web hosts send a SIGTERM to a CGI process that has used too much CPU time.

A call-back routine that uses the cdecl convention and returns a floating-point result, might not work with exw. This is because the Watcom C compiler (used to build exw) has a non-standard way of handling cdecl floating-point return values.

**Example Program:** [demo\win32\window.exw](#), [demo\linux\qsort.exu](#)

**See Also:** [routine\\_id](#), [platform.doc](#)

## free\_console

**Platform:** WIN32, Linux, FreeBSD

**Syntax:** include dll.e

free\_console()

**Description:** Free (delete) any console window associated with your program.

**Comments:** Euphoria will create a console **text** window for your program the first time that your program prints something to the screen, reads something from the keyboard, or in some way needs a console (similar to a DOS-prompt window). On WIN32 this window will automatically disappear when your program terminates, but you can call free\_console() to make it disappear sooner. On Linux or FreeBSD, the text mode console is always there, but an xterm window will disappear after Euphoria issues a "Press Enter" prompt at the end of execution.

On Linux or FreeBSD, free\_console() will set the terminal parameters back to normal, undoing the effect that curses has on the screen.

In a Linux or FreeBSD xterm window, a call to free\_console(), without any further printing to the screen or reading from the keyboard, will eliminate the "Press Enter" prompt that Euphoria normally issues at the end of execution.

After freeing the console window, you can create a new console window by printing something to the screen, or simply calling clear\_screen(), [position\(\)](#) or any other routine that needs a console.

When you use the **trace** facility, or when your program has an error, Euphoria will automatically create a console window to display trace information, error messages etc.

There's a WIN32 API routine, FreeConsole() that does something similar to free\_console(). You should use free\_console(), because it lets the interpreter know that there is no longer a console.

**See Also:** [clear\\_screen](#), [platform.doc](#)

```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- Directory and File Operations --

include sort.e
include misc.e

constant M_SEEK = 19,
         M_WHERE = 20,
         M_DIR = 22,
         M_CURRENT_DIR = 23,
         M_ALLOW_BREAK = 42,
         M_CHECK_BREAK = 43,
         M_FLUSH = 60,
         M_LOCK_FILE = 61,
         M_UNLOCK_FILE = 62,
         M_CHDIR = 63

type file_number(integer f)
    return f >= 0
end type

type file_position(atom p)
    return p >= -1
end type

type boolean(integer b)
    return b = 0 or b = 1
end type

global function seek(file_number fn, file_position pos)
-- Seeks to a byte position in the file,
-- or to end of file if pos is -1.
-- This function is normally used with
-- files opened in binary mode.
    return machine_func(M_SEEK, {fn, pos})
end function

global function where(file_number fn)
-- Returns the current byte position in the file.
-- This function is normally used with
-- files opened in binary mode.
    return machine_func(M_WHERE, fn)
end function

global procedure flush(file_number fn)
-- flush out the buffer associated with file fn
    machine_proc(M_FLUSH, fn)
end procedure

global constant LOCK_SHARED = 1,
                LOCK_EXCLUSIVE = 2

```

```

type lock_type(integer t)
  if platform() = LINUX then
    return t = LOCK_SHARED or t = LOCK_EXCLUSIVE
  else
    return 1
  end if
end type

type byte_range(sequence r)
  if length(r) = 0 then
    return 1
  elsif length(r) = 2 and r[1] <= r[2] then
    return 1
  else
    return 0
  end if
end type

global function lock_file(file_number fn, lock_type t, byte_range r)
-- Attempt to lock a file so other processes won't interfere with it.
-- The byte range can be {} if you want to lock the whole file
  return machine_func(M_LOCK_FILE, {fn, t, r})
end function

global procedure unlock_file(file_number fn, byte_range r)
-- The byte range can be {} if you want to unlock the whole file.
  machine_proc(M_UNLOCK_FILE, {fn, r})
end procedure

global constant
  D_NAME = 1,
  D_ATTRIBUTES = 2,
  D_SIZE = 3,

  D_YEAR = 4,
  D_MONTH = 5,
  D_DAY = 6,

  D_HOUR = 7,
  D_MINUTE = 8,
  D_SECOND = 9

global function dir(sequence name)
-- returns directory information, given the name
-- of a file or directory. Format returned is:
-- {
--   {"name1", attributes, size, year, month, day, hour, minute, second},
--   {"name2", ...
-- }
  return machine_func(M_DIR, name)
end function

global function current_dir()
-- returns name of current working directory
  return machine_func(M_CURRENT_DIR, 0)
end function

```

```

global function chdir(sequence newdir)
-- Changes the current directory. Returns 1 - success, 0 - fail.
  return machine_func(M_CHDIR, newdir)
end function

global procedure allow_break(boolean b)
-- If b is TRUE then allow control-c/control-break to
-- terminate the program. If b is FALSE then don't allow it.
-- Initially they *will* terminate the program, but only when it
-- tries to read input from the keyboard.
  machine_proc(M_ALLOW_BREAK, b)
end procedure

global function check_break()
-- returns the number of times that control-c or control-break
-- were pressed since the last time check_break() was called
  return machine_func(M_CHECK_BREAK, 0)
end function

-- Generalized recursive directory walker

global constant W_BAD_PATH = -1 -- error code

function default_dir(sequence path)
-- Default directory sorting function for walk_dir().
-- * sorts by name *
  object d

  d = dir(path)
  if atom(d) then
    return d
  else
    -- sort by name
    return sort(d)
  end if
end function

integer SLASH
if platform() = LINUX then
  SLASH='/'
else
  SLASH='\\'
end if

-- override the dir sorting function with your own routine id
constant DEFAULT = -2
global integer my_dir
my_dir = DEFAULT -- it's better not to use routine_id() here,
                  -- or else users will have to bind with clear routine names

global function walk_dir(sequence path_name, integer your_function,
                        integer scan_subdirs)
-- Generalized Directory Walker
-- Walk through a directory and (optionally) its subdirectories,
-- "visiting" each file and subdirectory. Your function will be called
-- via its routine id. The visits will occur in alphabetical order.

```

```

-- Your function should accept the path name and dir() entry for
-- each file and subdirectory. It should return 0 to keep going,
-- or an error code (greater than 0) to quit, or it can return
-- any sequence or atom other than 0 as a useful diagnostic value.
    object d, abort_now

-- get the full directory information
if my_dir = DEFAULT then
    d = default_dir(path_name)
else
    d = call_func(my_dir, {path_name})
end if
if atom(d) then
    return W_BAD_PATH
end if

-- trim any trailing blanks or '\' characters from the path
while length(path_name) > 0 and
    find(path_name[$], {' ', SLASH}) do
    path_name = path_name[1..$-1]
end while

for i = 1 to length(d) do
    if find('d', d[i][D_ATTRIBUTES]) then
        -- a directory
        if not find(d[i][D_NAME], {".", ".."}) then
            abort_now = call_func(your_function, {path_name, d[i]})
            if not equal(abort_now, 0) then
                return abort_now
            end if
            if scan_subdirs then
                abort_now = walk_dir(path_name & SLASH & d[i][D_NAME],
                    your_function, scan_subdirs)

                if not equal(abort_now, 0) and
                    not equal(abort_now, W_BAD_PATH) then
                    -- allow BAD PATH, user might delete a file or directory
                    return abort_now
                end if
            end if
        end if
    else
        -- a file
        abort_now = call_func(your_function, {path_name, d[i]})
        if not equal(abort_now, 0) then
            return abort_now
        end if
    end if
end for
return 0
end function

```

## seek

**Syntax:**           include file.e

i1 = seek(fn, a1)

**Description:**   Seek (move) to any byte position in the file fn or to the end of file if a1 is -1. For each open file there is a current byte position that is updated as a result of I/O operations on the file. The initial file position is 0 for files opened for read, write or update. The initial position is the end of file for files opened for append. The value returned by seek() is 0 if the seek was successful, and non-zero if it was unsuccessful. It is possible to seek past the end of a file. If you seek past the end of the file, and write some data, undefined bytes will be inserted into the gap between the original end of file and your new data.

**Comments:**       After seeking and reading (writing) a series of bytes, you may need to call seek() explicitly before you switch to writing (reading) bytes, even though the file position should already be what you want.

This function is normally used with files opened in binary mode. In text mode, DOS converts CR LF to LF on input, and LF to CR LF on output, which can cause great confusion when you are trying to count bytes.

**Example:**

```
include file.e

integer fn
fn = open("mydata", "rb")
-- read and display first line of file 3 times:
for i = 1 to 3 do
    puts(1, gets(fn))
    if seek(fn, 0) then
        puts(1, "rewind failed!\n")
    end if
end for
```

**See Also:**       [where](#), [open](#)



## where

**Syntax:** include file.e

a1 = where(fn)

**Description:** This function returns the current byte position in the file fn. This position is updated by reads, writes and seeks on the file. It is the place in the file where the next byte will be read from, or written to.

**See Also:** [seek](#), [open](#)

## flush

**Syntax:** include file.e

flush(fn)

**Description:** When you write data to a file, Euphoria normally stores the data in a memory buffer until a large enough chunk of data has accumulated. This large chunk can then be written to disk very efficiently. Sometimes you may want to force, or flush, all data out immediately, even if the memory buffer is not full. To do this you must call flush(fn), where fn is the file number of a file open for writing or appending.

**Comments:** When a file is closed, (see close()), all buffered data is flushed out. When a program terminates, all open files are flushed and closed automatically.

Use flush() when another process may need to see all of the data written so far, but you aren't ready to close the file yet.

**Example:**

```
f = open("logfile", "w")
puts(f, "Record#1\n")
puts(1, "Press Enter when ready\n")

flush(f)  -- This forces "Record #1" into "logfile" on disk.
          -- Without this, "logfile" will appear to have
          -- 0 characters when we stop for keyboard input.

s = gets(0) -- wait for keyboard input
```

**See Also:** [close](#), [lock\\_file](#)

## lock\_file

**Syntax:** include file.e

i1 = lock\_file(fn, i2, s)

**Description:** When multiple processes can simultaneously access a file, some kind of locking mechanism may be needed to avoid mangling the contents of the file, or causing erroneous data to be read from the file.

lock\_file() attempts to place a lock on an open file, fn, to stop other processes from using the file while your program is reading it or writing it. Under Linux/FreeBSD, there are two types of locks that you can request using the i2 parameter. (Under DOS32 and WIN32 the i2 parameter is ignored, but should be an integer.) Ask for a *shared* lock when you intend to read a file, and you want to temporarily block other processes from writing it. Ask for an *exclusive* lock when you intend to write to a file and you want to temporarily block other processes from reading or writing it. It's ok for many processes to simultaneously have shared locks on the same file, but only one process can have an exclusive lock, and that can happen only when no other process has any kind of lock on the file. file.e contains the following declaration:

```
global constant LOCK_SHARED = 1,  
                LOCK_EXCLUSIVE = 2
```

On DOS32 and WIN32 you can lock a specified portion of a file using the s parameter. s is a sequence of the form: {first\_byte, last\_byte}. It indicates the first byte and last byte in the file, that the lock applies to. Specify the empty sequence {}, if you want to lock the whole file. In the current release for Linux/FreeBSD, locks always apply to the whole file, and you should specify {} for this parameter.

If it is successful in obtaining the desired lock, lock\_file() will return 1. If unsuccessful, it will return 0.

lock\_file() does not wait for other processes to relinquish their locks. You may have to call it repeatedly, before the lock request is granted.

**Comments:** On Linux/FreeBSD, these locks are called advisory locks, which means they aren't enforced by the operating system. It is up to the processes that use a particular file to cooperate with each other. A process can access a file without first obtaining a lock on it. On WIN32 and DOS32, locks are enforced by the operating system.

On DOS32, lock\_file() is more useful when file sharing is enabled. It will typically return 0 (unsuccessful) under plain MS-DOS, outside of Windows.

**Example:**

```
include misc.e  
include file.e  
integer v  
atom t  
v = open("visitor_log", "a") -- open for append  
t = time()  
while not lock_file(v, LOCK_EXCLUSIVE, {}) do  
  if time() > t + 60 then  
    puts(1, "One minute already ... I can't wait forever!\n")  
    abort(1)  
  end if  
  sleep(5) -- let other processes run  
end while  
puts(v, "Yet another visitor\n")  
unlock_file(v, {})  
close(v)
```

**See Also:** [unlock\\_file](#), [flush](#), [sleep](#)

## unlock\_file

**Syntax:**       include file.e

unlock\_file(fn, s)

**Description:**   Unlock an open file fn, or a portion of file fn. You must have previously locked the file using lock\_file(). On DOS32 and WIN32 you can unlock a range of bytes within a file by specifying the s parameter as {first\_byte, last\_byte}. The same range of bytes must have been locked by a previous call to lock\_file(). On Linux/FreeBSD you can currently only lock or unlock an entire file. The s parameter should be {} when you want to unlock an entire file. On Linux/FreeBSD, s must always be {}.

**Comments:**     You should unlock a file as soon as possible so other processes can use it.  
Any files that you have locked, will automatically be unlocked when your program terminates.  
See lock\_file() for further comments and an example.

**See Also:**      [lock\\_file](#)

## dir

**Syntax:** include file.e

x = dir(st)

**Description:** Return directory information for the file or directory named by st. If there is no file or directory with this name then -1 is returned. On Windows and DOS st can contain \* and ? wildcards to select multiple files.

## current\_dir

**Syntax:**           include file.e

s = current\_dir()

**Description:**   Return the name of the current working directory.

**Comments:**     There will be no slash or backslash on the end of the current directory, except under DOS/Windows, at the top-level of a drive, e.g. C:\

**Example:**

```
sequence s
s = current_dir()
-- s would have "C:\EUPHORIA\DOC" if you were in that directory
```

**See Also:**     [dir](#), [chdir](#), [getenv](#)

## chdir

**Syntax:** include file.e

i = chdir(s)

**Description:** Set the current directory to the path given by sequence s. s must name an existing directory on the system. If successful, chdir() returns 1. If unsuccessful, chdir() returns 0.

**Comments:** By setting the current directory, you can refer to files in that directory using just the file name.

The function `current_dir()` will return the name of the current directory.

On DOS32 and WIN32 the current directory is a global property shared by all the processes running under one shell. On Linux/FreeBSD, a subprocess can change the current directory for itself, but this won't affect the current directory of its parent process.

**Example:**

```
if chdir("c:\\euphoria") then
  f = open("readme.doc", "r")
else
  puts(1, "Error: No euphoria directory?\n")
end if
```

**See Also:** [current\\_dir](#)

## allow\_break

**Syntax:** include file.e

allow\_break(i)

**Description:** When i is 1 (true) control-c and control-Break can terminate your program when it tries to read input from the keyboard. When i is 0 (false) your program will not be terminated by control-c or control-Break.

**Comments:** DOS will display ^C on the screen, even when your program cannot be terminated.

Initially your program can be terminated at any point where it tries to read from the keyboard. It could also be terminated by other input/output operations depending on options the user has set in his **config.sys** file. (Consult an MS-DOS manual for the BREAK command.) For some types of program this sudden termination could leave things in a messy state and might result in loss of data. allow\_break(0) lets you avoid this situation.

You can find out if the user has pressed control-c or control-Break by calling check\_break().

**Example:**

```
allow_break(0)  -- don't let the user kill me!
```

**See Also:** [check\\_break](#)



## check\_break

**Syntax:**           include file.e

i = check\_break()

**Description:**   Return the number of times that control-c or control-Break have been pressed since the last call to check\_break(), or since the beginning of the program if this is the first call.

**Comments:**      This is useful after you have called allow\_break(0) which prevents control-c or control-Break from terminating your program. You can use check\_break() to find out if the user has pressed one of these keys. You might then perform some action such as a graceful shutdown of your program.

Neither control-c nor control-Break will be returned as input characters when you read the keyboard. You can only detect them by calling check\_break().

**Example:**

```
        k = get_key()
if check_break() then
    temp = graphics_mode(-1)
    puts(1, "Shutting down...")
    save_all_user_data()
    abort(1)
end if
```

**See Also:**       [allow\\_break](#), [get\\_key](#)

## walk\_dir

**Syntax:** include file.e

i1 = walk\_dir(st, i2, i3)

**Description:** This routine will "walk" through a directory with path name given by st. i2 is the **routine id** of a routine that you supply. walk\_dir() will call your routine once for each file and subdirectory in st. If i3 is non-zero (TRUE), then the subdirectories in st will be walked through recursively.

The routine that you supply should accept the path name and [dir\(\) entry](#) for each file and subdirectory. It should return 0 to keep going, or non-zero to stop walk\_dir().

**Comments:** This mechanism allows you to write a simple function that handles one file at a time, while walk\_dir() handles the process of walking through all the files and subdirectories.

By default, the files and subdirectories will be visited in alphabetical order. To use a different order, set the global integer **my\_dir** to the **routine id** of your own *modified* dir() function that sorts the directory entries differently. See the default dir() function in [file.e](#).

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.

**Example:**

```
function look_at(sequence path_name, sequence entry)
-- this function accepts two sequences as arguments
    printf(1, "%s\\%s: %d\n",
           {path_name, entry[D_NAME], entry[D_SIZE]})
    return 0 -- keep going
end function

exit_code = walk_dir("C:\\MYFILES", routine_id("look_at"), TRUE)
```

**Example Program:** [euphoria/bin/search.ex](#)

**See Also:** [dir](#), [current\\_dir](#)

```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- Input and Conversion Routines:
-- get()
-- value()
-- wait_key()

-- error status values returned from get() and value():
global constant GET_SUCCESS = 0,
                GET_EOF = -1,
                GET_FAIL = 1

constant M_WAIT_KEY = 26

constant DIGITS = "0123456789",
        HEX_DIGITS = DIGITS & "ABCDEF",
        START_NUMERIC = DIGITS & "-+.#"

constant TRUE = 1

type natural(integer x)
    return x >= 0
end type

type char(integer x)
    return x >= -1 and x <= 255
end type

natural input_file  -- file to be read from

object input_string -- string to be read from
natural string_next

char ch  -- the current character

global function wait_key()
-- Get the next key pressed by the user.
-- Wait until a key is pressed.
    return machine_func(M_WAIT_KEY, 0)
end function

procedure get_ch()
-- set ch to the next character in the input stream (either string or file)

    if sequence(input_string) then
        if string_next <= length(input_string) then
            ch = input_string[string_next]
            string_next += 1
        else
            ch = GET_EOF
        end if
    else
        ch = getc(input_file)
    end if
end procedure

```

```

        end if
    end procedure

procedure skip_blanks()
-- skip white space
-- ch is "live" at entry and exit

    while find(ch, " \t\n\r") do
        get_ch()
    end while
end procedure

constant ESCAPE_CHARS = "nt'\"\\r",
          ESCAPED_CHARS = "\n\t'\"\\r"

function escape_char(char c)
-- return escape character
    natural i

    i = find(c, ESCAPE_CHARS)
    if i = 0 then
        return GET_FAIL
    else
        return ESCAPED_CHARS[i]
    end if
end function

function get_qchar()
-- get a single-quoted character
-- ch is "live" at exit
    char c

    get_ch()
    c = ch
    if ch = '\\' then
        get_ch()
        c = escape_char(ch)
        if c = GET_FAIL then
            return {GET_FAIL, 0}
        end if
    elsif ch = '\'' then
        return {GET_FAIL, 0}
    end if
    get_ch()
    if ch != '\'' then
        return {GET_FAIL, 0}
    else
        get_ch()
        return {GET_SUCCESS, c}
    end if
end function

function get_string()
-- get a double-quoted character string
-- ch is "live" at exit
    sequence text

```

```

text = ""
while TRUE do
    get_ch()
    if ch = GET_EOF or ch = '\n' then
        return {GET_FAIL, 0}
    elseif ch = '"' then
        get_ch()
        return {GET_SUCCESS, text}
    elseif ch = '\\' then
        get_ch()
        ch = escape_char(ch)
        if ch = GET_FAIL then
            return {GET_FAIL, 0}
        end if
    end if
    text = text & ch
end while
end function

type plus_or_minus(integer x)
    return x = -1 or x = +1
end type

function get_number()
-- read a number
-- ch is "live" at entry and exit
plus_or_minus sign, e_sign
natural ndigits
integer hex_digit
atom mantissa, dec, e_mag

sign = +1
mantissa = 0
ndigits = 0

-- process sign
if ch = '-' then
    sign = -1
    get_ch()
elseif ch = '+' then
    get_ch()
end if

-- get mantissa
if ch = '#' then
    -- process hex integer and return
    get_ch()
    while TRUE do
        hex_digit = find(ch, HEX_DIGITS)-1
        if hex_digit >= 0 then
            ndigits += 1
            mantissa = mantissa * 16 + hex_digit
            get_ch()
        else
            if ndigits > 0 then

```

```

        return {GET_SUCCESS, sign * mantissa}
    else
        return {GET_FAIL, 0}
    end if
end if
end while
end if

-- decimal integer or floating point
while ch >= '0' and ch <= '9' do
    ndigits += 1
    mantissa = mantissa * 10 + (ch - '0')
    get_ch()
end while

if ch = '.' then
    -- get fraction
    get_ch()
    dec = 10
    while ch >= '0' and ch <= '9' do
        ndigits += 1
        mantissa += (ch - '0') / dec
        dec *= 10
        get_ch()
    end while
end if

if ndigits = 0 then
    return {GET_FAIL, 0}
end if

mantissa = sign * mantissa

if ch = 'e' or ch = 'E' then
    -- get exponent sign
    e_sign = +1
    e_mag = 0
    get_ch()
    if ch = '-' then
        e_sign = -1
        get_ch()
    elseif ch = '+' then
        get_ch()
    end if
    -- get exponent magnitude
    if ch >= '0' and ch <= '9' then
        e_mag = ch - '0'
        get_ch()
        while ch >= '0' and ch <= '9' do
            e_mag = e_mag * 10 + ch - '0'
            get_ch()
        end while
    else
        return {GET_FAIL, 0} -- no exponent
    end if
    e_mag *= e_sign
end if

```

```

    if e_mag > 308 then
        -- rare case: avoid power() overflow
        mantissa *= power(10, 308)
        if e_mag > 1000 then
            e_mag = 1000
        end if
        for i = 1 to e_mag - 308 do
            mantissa *= 10
        end for
    else
        mantissa *= power(10, e_mag)
    end if
end if

    return {GET_SUCCESS, mantissa}
end function

function Get()
-- read a Euphoria data object as a string of characters
-- and return {error_flag, value}
-- Note: ch is "live" at entry and exit of this routine
    sequence s, e

    skip_blanks()

    if find(ch, START_NUMERIC) then
        return get_number()

    elsif ch = '{' then
        -- process a sequence
        s = {}
        get_ch()
        skip_blanks()
        if ch = '}' then
            get_ch()
            return {GET_SUCCESS, s} -- empty sequence
        end if

        while TRUE do
            e = Get() -- read next element
            if e[1] != GET_SUCCESS then
                return e
            end if
            s = append(s, e[2])
            skip_blanks()
            if ch = '}' then
                get_ch()
                return {GET_SUCCESS, s}
            elsif ch != ',' then
                return {GET_FAIL, 0}
            end if
            get_ch() -- skip comma
        end while

    elsif ch = '\"' then
        return get_string()
    end if
end function

```

```

elseif ch = '\\' then
    return get_qchar()

elseif ch = -1 then
    return {GET_EOF, 0}

else
    return {GET_FAIL, 0}

end if
end function

global function get(integer file)
-- Read the string representation of a Euphoria object
-- from a file. Convert to the value of the object.
-- Return {error_status, value}.
    input_file = file
    input_string = 0
    get_ch()
    return Get()
end function

global function value(sequence string)
-- Read the representation of a Euphoria object
-- from a sequence of characters. Convert to the value of the object.
-- Return {error_status, value}.
    input_string = string
    string_next = 1
    get_ch()
    return Get()
end function

global function prompt_number(sequence prompt, sequence range)
-- Prompt the user to enter a number.
-- A range of allowed values may be specified.
    object answer

    while 1 do
        puts(1, prompt)
        answer = gets(0) -- make sure whole line is read
        puts(1, '\n')

        answer = value(answer)
        if answer[1] != GET_SUCCESS or sequence(answer[2]) then
            puts(1, "A number is expected - try again\n")
        else
            if length(range) = 2 then
                if range[1] <= answer[2] and answer[2] <= range[2] then
                    return answer[2]
                else
                    printf(1,
                        "A number from %g to %g is expected here - try again\n",
                        range)
                end if
            else

```



```

        return answer[2]
    end if
end if
end while
end function

global function prompt_string(sequence prompt)
-- Prompt the user to enter a string
    object answer

    puts(1, prompt)
    answer = gets(0)
    puts(1, '\n')
    if sequence(answer) and length(answer) > 0 then
        return answer[1..$-1] -- trim the \n
    else
        return ""
    end if
end function

constant CHUNK = 100

global function get_bytes(integer fn, integer n)
-- Return a sequence of n bytes (maximum) from an open file.
-- If n > 0 and fewer than n bytes are returned,
-- you've reached the end of file.
-- This function is normally used with files opened in binary mode.
    sequence s
    integer c, first, last

    if n = 0 then
        return {}
    end if

    c = getc(fn)
    if c = GET_EOF then
        return {}
    end if

    s = repeat(c, n)

    last = 1
    while last < n do
        -- for speed, read a chunk without checking for EOF
        first = last+1
        last = last+CHUNK
        if last > n then
            last = n
        end if
        for i = first to last do
            s[i] = getc(fn)
        end for
        -- check for EOF after each chunk
        if s[last] = GET_EOF then
            -- trim the EOF's and return
            while s[last] = GET_EOF do

```

```
        last -= 1
    end while
    return s[1..last]
end if
end while
return s
end function
```

## wait\_key

**Syntax:** include get.e

i = wait\_key()

**Description:** Return the next key pressed by the user. Don't return until a key is pressed.

**Comments:** You could achieve the same result using **get\_key()** as follows:

```
while 1 do
  k = get_key()
  if k != -1 then
    exit
  end if
end while
```

However, on multi-tasking systems like **Windows** or **Linux/FreeBSD**, this "busy waiting" would tend to slow the system down. wait\_key() lets the operating system do other useful work while your program is waiting for the user to press a key.

You could also use **getc(0)**, assuming file number 0 was input from the keyboard, except that you wouldn't pick up the special codes for function keys, arrow keys etc.

**See Also:** [get\\_key](#), [getc](#)

## get

**Syntax:** include get.e

s = get(fn)

**Description:** Input, from file fn, a human-readable string of characters representing a Euphoria object. Convert the string into the numeric value of that object. s will be a 2-element sequence: **{error status, value}**. Error status codes are:

```
GET_SUCCESS -- object was read successfully
GET_EOF     -- end of file before object was read
GET_FAIL    -- object is not syntactically correct
```

get() can read arbitrarily complicated Euphoria objects. You could have a long sequence of values in braces and separated by commas, e.g. {23, {49, 57}, 0.5, -1, 99, 'A', "john"}. **A single call to get() will read in this entire sequence and return it's value as a result.**

Each call to get() picks up where the previous call left off. For instance, a series of 5 calls to get() would be needed to read in:

```
99 5.2 {1,2,3} "Hello" -1
```

On the sixth and any subsequent call to get() you would see a GET\_EOF status. If you had something like:

```
{1, 2, xxx}
```

in the input stream you would see a GET\_FAIL error status because xxx is not a Euphoria object.

Multiple "top-level" objects in the input stream must be separated from each other with one or more "whitespace" characters (blank, tab, \r or \n). Whitespace is not necessary *within* a top-level object. A call to get() will read one entire top-level object, plus one additional (whitespace) character.

**Comments:** The combination of print() and get() can be used to save a Euphoria object to disk and later read it back. This technique could be used to implement a database as one or more large Euphoria sequences stored in disk files. The sequences could be read into memory, updated and then written back to disk after each series of transactions is complete. Remember to write out a whitespace character (using puts()) after each call to print().

The value returned is not meaningful unless you have a GET\_SUCCESS status.

**Example:** Suppose your program asks the user to enter a number from the keyboard.

```
-- If he types 77.5, get(0) would return:
```

```
{GET_SUCCESS, 77.5}
```

```
-- whereas gets(0) would return:
```

```
"77.5\n"
```

**Example Program:** [demo\mydata.ex](#)

**See Also:** [print](#), [value](#), [gets](#), [getc](#), [prompt\\_number](#), [prompt\\_string](#)

## value

**Syntax:** include get.e

s = value(st)

**Description:** Read the string representation of a Euphoria object, and compute the value of that object. A 2-element sequence, **{error\_status, value}** is actually returned, where error\_status can be one of:

```
GET_SUCCESS -- a valid object representation was found
GET_EOF     -- end of string reached too soon
GET_FAIL    -- syntax is wrong
```

**Comments:** This works the same as **get()**, but it reads from a string that you supply, rather than from a file or device.

After reading one valid representation of a Euphoria object, value() will stop reading and ignore any additional characters in the string. For example, "36" and "36P" will both give you {GET\_SUCCESS, 36}.

**Example 1:**

```
s = value("12345")
-- s is {GET_SUCCESS, 12345}
```

**Example 2:**

```
s = value("{0, 1, -99.9}")
-- s is {GET_SUCCESS, {0, 1, -99.9}}
```

**Example 3:**

```
s = value("+++")
-- s is {GET_FAIL, 0}
```

**See Also:** [get](#), [sprintf](#), [print](#)

## prompt\_number

**Syntax:**           include get.e

a = prompt\_number(st, s)

**Description:**   Prompt the user to enter a number. st is a string of text that will be displayed on the screen. s is a sequence of two values {lower, upper} which determine the range of values that the user may enter. If the user enters a number that is less than lower or greater than upper, he will be prompted again. s can be [empty](#), {}, if there are no restrictions.

**Comments:**     If this routine is too simple for your needs, feel free to copy it and make your own more specialized version.

**Example 1:**

```
age = prompt_number("What is your age? ", {0, 150})
```

**Example 2:**

```
t = prompt_number("Enter a temperature in Celcius:\n", {})
```

**See Also:**     [get](#), [prompt\\_string](#)

## prompt\_string

**Syntax:**           include get.e

s = prompt\_string(st)

**Description:**   Prompt the user to enter a string of text. st is a string that will be displayed on the screen. The string that the user types will be returned as a sequence, minus any new-line character.

**Comments:**     If the user happens to type control-Z (indicates end-of-file), "" will be returned.

**Example:**

```
name = prompt_string("What is your name? ")
```

**See Also:**     [gets](#), [prompt\\_number](#)

## get\_bytes

**Syntax:**       include get.e

s = get\_bytes(fn, i)

**Description:**   Read the next i bytes from file number fn. Return the bytes as a sequence. The sequence will be of length i, except when there are fewer than i bytes remaining to be read in the file.

**Comments:**     When i > 0 and [length\(s\)](#) < i you know you've reached the end of file. Eventually, an [empty sequence](#) will be returned for s.

This function is normally used with files opened in binary mode, "rb". This avoids the confusing situation in text mode where DOS will convert CR LF pairs to LF.

**Example:**

```
include get.e

integer fn
fn = open("temp", "rb")  -- an existing file

sequence whole_file
whole_file = {}

sequence chunk

while 1 do
  chunk = get_bytes(fn, 100) -- read 100 bytes at a time
  whole_file &= chunk        -- chunk might be empty, that's ok
  if length(chunk) < 100 then
    exit
  end if
end while

close(fn)
? length(whole_file)  -- should match DIR size of "temp"
```

**See Also:**   [getc](#), [gets](#)



```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- Graphics & Sound Routines

--    GRAPHICS MODES --  argument to graphics_mode()

-- mode  description
-- ----  -
--  -1  restore to original default mode
--   0  40 x 25 text, 16 grey
--   1  40 x 25 text, 16/8 color
--   2  80 x 25 text, 16 grey
--   3  80 x 25 text, 16/8 color
--   4  320 x 200, 4 color
--   5  320 x 200, 4 grey
--   6  640 x 200, BW
--   7  80 x 25 text, BW
--  11  720 x 350, BW  (many video cards are lacking this one)
--  13  320 x 200, 16 color
--  14  640 x 200, 16 color
--  15  640 x 350, BW  (may be 4-color with blinking)
--  16  640 x 350, 4 or 16 color
--  17  640 x 480, BW
--  18  640 x 480, 16 color
--  19  320 x 200, 256 color
-- 256  640 x 400, 256 color  (some cards are missing this one)
-- 257  640 x 480, 256 color  (some cards are missing this one)
-- 258  800 x 600, 16 color
-- 259  800 x 600, 256 color
-- 260 1024 x 768, 16 color
-- 261 1024 x 768, 256 color

-- COLOR values -- for characters and pixels
global constant
    BLACK = 0,  -- in graphics modes this is "transparent"
    GREEN = 2,
    MAGENTA = 5,
    WHITE = 7,
    GRAY = 8,
    BRIGHT_GREEN = 10,
    BRIGHT_MAGENTA = 13,
    BRIGHT_WHITE = 15
global integer
    BLUE, CYAN, RED, BROWN, BRIGHT_BLUE, BRIGHT_CYAN, BRIGHT_RED, YELLOW

include misc.e

if platform() = LINUX then
    BLUE = 4
    CYAN = 6
    RED = 1
    BROWN = 3
    BRIGHT_BLUE = 12

```

```

    BRIGHT_CYAN = 14
    BRIGHT_RED = 9
    YELLOW = 11
else
    BLUE = 1
    CYAN = 3
    RED = 4
    BROWN = 6
    BRIGHT_BLUE = 9
    BRIGHT_CYAN = 11
    BRIGHT_RED = 12
    YELLOW = 14
end if

global constant BLINKING = 16  -- add to color to get blinking text

-- machine() commands
constant M_SOUND = 1,
        M_LINE = 2,
        M_PALETTE = 3,
        M_GRAPHICS_MODE = 5,
        M_CURSOR = 6,
        M_WRAP = 7,
        M_SCROLL = 8,
        M_SET_T_COLOR = 9,
        M_SET_B_COLOR = 10,
        M_POLYGON = 11,
        M_TEXTROWS = 12,
        M_VIDEO_CONFIG = 13,
        M_ELLIPSE = 18,
        M_GET_POSITION = 25,
        M_ALL_PALETTE = 27

type mode(integer x)
    return (x >= -3 and x <= 19) or (x >= 256 and x <= 263)
end type

type color(integer x)
    return x >= 0 and x <= 255
end type

type boolean(integer x)
    return x = 0 or x = 1
end type

type positive_int(integer x)
    return x >= 1
end type

type point(sequence x)
    return length(x) = 2
end type

type point_sequence(sequence x)
    return length(x) >= 2
end type

```

```

global procedure draw_line(color c, point_sequence xyarray)
-- draw a line connecting the 2 or more points
-- in xyarray: {{x1, y1}, {x2, y2}, ...}
-- using a certain color
    machine_proc(M_LINE, {c, 0, xyarray})
end procedure

global procedure polygon(color c,
                        boolean fill,
                        point_sequence xyarray)
-- draw a polygon using a certain color
-- fill the area if fill is TRUE
-- 3 or more vertices are given in xyarray
    machine_proc(M_POLYGON, {c, fill, xyarray})
end procedure

global procedure ellipse(color c, boolean fill, point p1, point p2)
-- draw an ellipse with a certain color that fits in the
-- rectangle defined by diagonal points p1 and p2, i.e.
-- {x1, y1} and {x2, y2}. The ellipse may be filled or just an outline.
    machine_proc(M_ELLIPSE, {c, fill, p1, p2})
end procedure

global function graphics_mode(mode m)
-- try to set up a new graphics mode
-- return 0 if successful, non-zero if failed
    return machine_func(M_GRAPHICS_MODE, m)
end function

global constant VC_COLOR = 1,
                VC_MODE   = 2,
                VC_LINES  = 3,
                VC_COLUMNS = 4,
                VC_XPIXELS = 5,
                VC_YPIXELS = 6,
                VC_NCOLORS = 7,
                VC_PAGES  = 8

global function video_config()
-- return sequence of information on video configuration
-- {color?, mode, text lines, text columns, xpixels, ypixels, #colors, pages}
    return machine_func(M_VIDEO_CONFIG, 0)
end function

-- cursor styles:
global constant NO_CURSOR      = #2000,
                UNDERLINE_CURSOR = #0607,
                THICK_UNDERLINE_CURSOR = #0507,
                HALF_BLOCK_CURSOR = #0407,
                BLOCK_CURSOR     = #0007

global procedure cursor(integer style)
-- choose a cursor style
    machine_proc(M_CURSOR, style)
end procedure

```

```

global function get_position()
-- return {line, column} of current cursor position
  return machine_func(M_GET_POSITION, 0)
end function

global function text_rows(positive_int rows)
  return machine_func(M_TEXTROWS, rows)
end function

global procedure wrap(boolean on)
-- on = 1: characters will wrap at end of long line
-- on = 0: lines will be truncated
  machine_proc(M_WRAP, on)
end procedure

global procedure scroll(integer amount,
                        positive_int top_line,
                        positive_int bottom_line)
-- scroll lines of text on screen between top_line and bottom_line
-- amount > 0: scroll text up by amount lines
-- amount < 0: scroll text down by amount lines
-- (had only the first parameter in v1.2)
  machine_proc(M_SCROLL, {amount, top_line, bottom_line})
end procedure

global procedure text_color(color c)
-- set the foreground text color to c - text or graphics modes
-- add 16 to get blinking
  machine_proc(M_SET_T_COLOR, c)
end procedure

global procedure bk_color(color c)
-- set the background color to c - text or graphics modes
  machine_proc(M_SET_B_COLOR, c)
end procedure

type mixture(sequence s)
  return length(s) = 3 -- {red, green, blue}
end type

global function palette(color c, mixture s)
-- choose a new mix of {red, green, blue} to be shown on the screen for
-- color number c. Returns previous mixture as {red, green, blue}.
  return machine_func(M_PALETTE, {c, s})
end function

global procedure all_palette(sequence s)
-- s is a sequence of the form: {{r,g,b},{r,g,b}, ...{r,g,b}}
-- that specifies new color intensities for the entire set of
-- colors in the current graphics mode.
  machine_proc(M_ALL_PALETTE, s)
end procedure

-- Sound Effects --

```

```
type frequency(integer x)
  return x >= 0
end type

global procedure sound(frequency f)
-- turn on speaker at frequency f
-- turn off speaker if f is 0
  machine_proc(M_SOUND, f)
end procedure
```

## draw\_line

**Platform:** DOS32

**Syntax:** include graphics.e

draw\_line(i, s)

**Description:** Draw a line on a pixel-graphics screen connecting two or more points in s, using color i.

**Example:**

```
draw_line(WHITE, {{100, 100}, {200, 200}, {900, 700}})
```

```
-- This would connect the three points in the sequence using  
-- a white line, i.e. a line would be drawn from {100, 100} to  
-- {200, 200} and another line would be drawn from {200, 200} to  
-- {900, 700}.
```

**See Also:** [polygon](#), [ellipse](#), [pixel](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)

# polygon

**Platform:** **DOS32**

**Syntax:** include graphics.e

polygon(i1, i2, s)

**Description:** Draw a polygon with 3 or more vertices given in s, on a pixel-graphics screen using a certain color i1. Fill the area if i2 is 1. Don't fill if i2 is 0.

**Example:**

```
    polygon(GREEN, 1, {{100, 100}, {200, 200}, {900, 700}})
-- makes a solid green triangle.
```

**See Also:** [draw\\_line](#), [ellipse](#)

## ellipse

**Platform:** **DOS32**

**Syntax:** include graphics.e

ellipse(i1, i2, s1, s2)

**Description:** Draw an ellipse with color i1 on a **pixel-graphics** screen. The ellipse will neatly fit inside the rectangle defined by diagonal points s1 {x1, y1} and s2 {x2, y2}. If the rectangle is a square then the ellipse will be a circle. Fill the ellipse when i2 is 1. Don't fill when i2 is 0.

**Example:**

```
ellipse(MAGENTA, 0, {10, 10}, {20, 20})
```

```
-- This would make a magenta colored circle just fitting
-- inside the square:
--      {10, 10}, {10, 20}, {20, 20}, {20, 10}.
```

**Example Program:** [demo\dos32\sb.ex](#)

**See Also:** [polygon](#), [draw\\_line](#)



## graphics\_mode

**Platform:** **DOS32**

**Syntax:** include graphics.e

i1 = graphics\_mode(i2)

**Description:** Select graphics mode i2. See [graphics.e](#) for a list of valid graphics modes. If successful, i1 is set to 0, otherwise i1 is set to 1.

**Comments:** Some modes are referred to as **text modes** because they only let you display text. Other modes are referred to as **pixel-graphics modes** because you can display pixels, lines, ellipses etc., as well as text.

As a convenience to your users, it is usually a good idea to switch back from a pixel-graphics mode to the standard text mode before your program terminates. You can do this with `graphics_mode(-1)`. If a pixel-graphics program leaves your screen in a mess, you can clear it up with the DOS CLS command, or by running **ex** or **ed**.

Some graphics cards will be unable to enter some SVGA modes, under some conditions. You can't always tell from the i1 value, whether the graphics mode was set up successfully.

On the **WIN32** and **Linux/FreeBSD** platforms, `graphics_mode()` will allocate a plain, text mode console if one does not exist yet. It will then return 0, no matter what value is passed as i2.

**Example:**

```
    if graphics_mode(18) then
      puts(SCREEN, "need VGA graphics!\n")
      abort(1)
    end if
    draw_line(BLUE, {{0,0}}, {{50,50}})
```

**See Also:** [text\\_rows](#), [video\\_config](#)

... continue

[from A to B](#) | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)

## video\_config

**Syntax:** include graphics.e

s = video\_config()

**Description:** Return a sequence of values describing the current video configuration:  
{color monitor?, graphics mode, text rows, text columns, xpixels, ypixels, number of colors, number of pages}

The following constants are defined in [graphics.e](#):

```
global constant VC_COLOR    = 1,
                  VC_MODE    = 2,
                  VC_LINES   = 3,
                  VC_COLUMNS = 4,
                  VC_XPIXELS = 5,
                  VC_YPIXELS = 6,
                  VC_NCOLORS = 7,
                  VC_PAGES   = 8
```

**Comments:** This routine makes it easy for you to parameterize a program so it will work in many different graphics modes.

On the PC there are two types of graphics mode. The first type, **text mode**, lets you print text only. The second type, **pixel-graphics mode**, lets you plot pixels, or points, in various colors, as well as text. You can tell that you are in a **text mode**, because the VC\_XPIXELS and VC\_YPIXELS fields will be 0. Library routines such as [polygon\(\)](#), [draw\\_line\(\)](#), and [ellipse\(\)](#) only work in a **pixel-graphics mode**.

**Example:**

```
vc = video_config()  -- in mode 3 with 25-lines of text:
-- vc is {1, 3, 25, 80, 0, 0, 32, 8}
```

**See Also:** [graphics\\_mode](#)

## cursor

**Platform:** WIN32, DOS32

**Syntax:** include graphics.e

cursor(i)

**Description:** Select a style of cursor. [graphics.e](#) contains:

```
global constant NO_CURSOR = #2000,  
    UNDERLINE_CURSOR = #0607,  
    THICK_UNDERLINE_CURSOR = #0507,  
    HALF_BLOCK_CURSOR = #0407,  
    BLOCK_CURSOR = #0007
```

The second and fourth hex digits (from the left) determine the top and bottom rows of pixels in the cursor. The first digit controls whether the cursor will be visible or not. For example, #0407 turns on the 4th through 7th rows.

**Comments:** In [pixel-graphics modes](#) no cursor is displayed.

**Example:**

```
cursor(BLOCK_CURSOR)
```

**See Also:** [graphics\\_mode](#), [text\\_rows](#)

## get\_position

**Syntax:**           include graphics.e

s = get\_position()

**Description:**   Return the current line and column position of the cursor as a 2-element sequence {**line**, **column**}.

**Comments:**      get\_position() works in both **text and pixel-graphics modes**. In **pixel-graphics modes** no cursor will be displayed, but get\_position() will return the line and column where the next character will be displayed.

The coordinate system for displaying text is different from the one for displaying pixels. Pixels are displayed such that the top-left is (x=0,y=0) and the first coordinate controls the horizontal, left-right location. In pixel-graphics modes you can display both text and pixels. get\_position() returns the current line and column for the text that you are displaying, not the pixels that you may be plotting. There is no corresponding routine for getting the current pixel position.

**See Also:**       [position](#), [get\\_pixel](#)

## text\_rows

**Platform:** DOS32, WIN32

**Syntax:** include graphics.e

i2 = text\_rows(i1)

**Description:** Set the number of lines on a **text-mode** screen to i1 if possible. i2 will be set to the actual new number of lines.

**Comments:** Values of 25, 28, 43 and 50 lines are supported by most video cards.

**See Also:** [graphics\\_mode](#)

## wrap

**Syntax:** include graphics.e

wrap(i)

**Description:** Allow text to wrap at the right margin (i = 1) or get truncated (i = 0).

**Comments:** By default text will wrap.

Use wrap() in **text modes** or **pixel-graphics modes** when you are displaying long lines of text.

**Example:**

```
    puts(1, repeat('x', 100) & "\n\n")
-- now have a line of 80 'x' followed a line of 20 more 'x'
wrap(0)
puts(1, repeat('x', 100) & "\n\n")
-- creates just one line of 80 'x'
```

**See Also:** [puts](#), [position](#)

## scroll

**Syntax:**           include graphics.h

scroll(i1, i2, i3)

**Description:**   Scroll a region of text on the screen either up (i1 positive) or down (i1 negative) by i1 lines. The region is the series of lines on the screen from i2 (top line) to i3 (bottom line), inclusive. New blank lines will appear at the top or bottom.

**Comments:**      You could perform the scrolling operation using a series of calls to puts(), but scroll() is much faster.

The position of the cursor after scrolling is not defined.

**Example Program:**   [binled.ex](#)

**See Also:**       [clear\\_screen](#), [text\\_rows](#)

## text\_color

**Syntax:**       include graphics.e

text\_color(i)

**Description:**   Set the foreground text color. Add 16 to get blinking text in some modes. See [graphics.e](#) for a list of possible colors.

**Comments:**     Text that you print *after* calling text\_color() will have the desired color.

When your program terminates, the last color that you selected and actually printed on the screen will remain in effect. Thus you may have to print something, maybe just '\n', in WHITE to restore white text, especially if you are at the bottom line of the screen, ready to scroll up.

**Example:**

```
text_color(BRIGHT_BLUE)
```

**See Also:**     [bk\\_color](#)



## bk\_color

**Syntax:** include graphics.e

bk\_color(i)

**Description:** Set the background color to one of the 16 standard colors. In **pixel-graphics modes** the whole screen is affected immediately. In **text modes** any new characters that you print will have the new background color. In some text modes there might only be 8 distinct background colors available.

**Comments:** The 16 standard colors are defined as constants in [graphics.e](#)

In **pixel-graphics modes**, color 0 which is normally BLACK, will be set to the same {r,g,b} palette value as color number i.

In some **pixel-graphics modes**, there is a *border* color that appears at the edges of the screen. In 256-color modes, this is the 17th color in the palette. You can control it as you would any other color.

In **text modes**, to restore the original background color when your program finishes, e.g. 0 - BLACK, you must call bk\_color(0). If the cursor is at the bottom line of the screen, you may have to actually print something before terminating your program. Printing '\n' may be enough.

**Example:**

```
bk_color(BLACK)
```

**See Also:** [text\\_color](#), [palette](#)

## palette

**Platform:** **DOS32**

**Syntax:** include graphics.e

x = palette(i, s)

**Description:** Change the color for color number i to s, where s is a sequence of color intensities: {red, green, blue}. Each value in s can be from 0 to 63. If successful, a 3-element sequence containing the previous color for i will be returned, and all pixels on the screen with value i will be set to the new color. If unsuccessful, the atom -1 will be returned.

**Example:**

```
x = palette(0, {15, 40, 10})  
-- color number 0 (normally black) is changed to a shade  
-- of mainly green.
```

**See Also:** [all\\_palette](#)

## all\_palette

**Platform:** DOS32

**Syntax:** include graphics.e

all\_palette(s)

**Description:** Specify new color intensities for the entire set of colors in the current graphics mode. s is a sequence of the form:

{ {r,g,b}, {r,g,b}, ..., {r,g,b} }

Each element specifies a new color intensity {red, green, blue} for the corresponding color number, starting with color number 0. The values for red, green and blue must be in the range 0 to 63.

**Comments:** This executes much faster than if you were to use palette() to set the new color intensities one by one. This procedure can be used with read\_bitmap() to quickly display a picture on the screen.

**Example Program:** [demo\dos32\bitmap.ex](#)

**See Also:** [get\\_all\\_palette](#), [palette](#), [read\\_bitmap](#), [video\\_config](#), [graphics\\_mode](#)

## sound

**Platform:** DOS32

**Syntax:** include graphics.e

sound(i)

**Description:** Turn on the PC speaker at frequency i. If i is 0 the speaker will be turned off.

**Comments:** On WIN32 and Linux/FreeBSD no sound will be made.

**Example:**

```
sound(1000) -- starts a fairly high pitched sound
```

```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- Graphical Image routines

include machine.e
include graphics.e
include misc.e

constant BMPFILEHDRSIZE = 14
constant OLDHDRSIZE = 12, NEWHDRSIZE = 40
constant EOF = -1

-- error codes returned by read_bitmap(), save_bitmap() and save_screen()
global constant BMP_SUCCESS = 0,
                BMP_OPEN_FAILED = 1,
                BMP_UNEXPECTED_EOF = 2,
                BMP_UNSUPPORTED_FORMAT = 3,
                BMP_INVALID_MODE = 4

integer fn, error_code

function get_word()
-- read 2 bytes
    integer lower, upper

    lower = getc(fn)
    upper = getc(fn)
    if upper = EOF then
        error_code = BMP_UNEXPECTED_EOF
    end if
    return upper * 256 + lower
end function

function get_dword()
-- read 4 bytes
    integer lower, upper

    lower = get_word()
    upper = get_word()
    return upper * 65536 + lower
end function

function get_c_block(integer num_bytes)
-- read num_bytes bytes
    sequence s

    s = repeat(0, num_bytes)
    for i = 1 to num_bytes do
        s[i] = getc(fn)
    end for
    if s[$] = EOF then
        error_code = BMP_UNEXPECTED_EOF
    end if

```

```

    return s
end function

function get_rgb(integer set_size)
-- get red, green, blue palette values
    integer red, green, blue

    blue = getc(fn)
    green = getc(fn)
    red = getc(fn)
    if set_size = 4 then
        if getc(fn) then
            end if
        end if
    return {red, green, blue}
end function

function get_rgb_block(integer num_dwords, integer set_size)
-- reads palette
    sequence s

    s = {}
    for i = 1 to num_dwords do
        s = append(s, get_rgb(set_size))
    end for
    if s[$][3] = EOF then
        error_code = BMP_UNEXPECTED_EOF
    end if
    return s
end function

function row_bytes(atom BitCount, atom Width)
-- number of bytes per row of pixel data
    return floor(((BitCount * Width) + 31) / 32) * 4
end function

function unpack(sequence image, integer BitCount, integer Width, integer Height)
-- unpack the 1-d byte sequence into a 2-d sequence of pixels
    sequence pic_2d, row, bits
    integer bytes, next_byte, byte

    pic_2d = {}
    bytes = row_bytes(BitCount, Width)
    next_byte = 1
    for i = 1 to Height do
        row = {}
        if BitCount = 1 then
            for j = 1 to bytes do
                byte = image[next_byte]
                next_byte += 1
                bits = repeat(0, 8)
                for k = 8 to 1 by -1 do
                    bits[k] = and_bits(byte, 1)
                    byte = floor(byte/2)
                end for
                row &= bits
            end for
        end if
    end for

```

```

        end for
    elseif BitCount = 2 then
        for j = 1 to bytes do
            byte = image[next_byte]
            next_byte += 1
            bits = repeat(0, 4)
            for k = 4 to 1 by -1 do
                bits[k] = and_bits(byte, 3)
                byte = floor(byte/4)
            end for
            row &= bits
        end for
    elseif BitCount = 4 then
        for j = 1 to bytes do
            byte = image[next_byte]
            row = append(row, floor(byte/16))
            row = append(row, and_bits(byte, 15))
            next_byte += 1
        end for
    elseif BitCount = 8 then
        row = image[next_byte..next_byte+bytes-1]
        next_byte += bytes
    else
        error_code = BMP_UNSUPPORTED_FORMAT
        exit
    end if
    pic_2d = prepend(pic_2d, row[1..Width])
end for
return pic_2d
end function

global function read_bitmap(sequence file_name)
-- read a bitmap (.BMP) file into a 2-d sequence of sequences (image)
-- return {palette,image}
    atom Size
    integer Type, Xhot, Yhot, Planes, BitCount
    atom Width, Height, Compression, OffBits, SizeHeader,
        SizeImage, XPelsPerMeter, YPelsPerMeter, ClrUsed,
        ClrImportant, NumColors
    sequence Palette, Bits, two_d_bits

    error_code = 0
    fn = open(file_name, "rb")
    if fn = -1 then
        return BMP_OPEN_FAILED
    end if
    Type = get_word()
    Size = get_dword()
    Xhot = get_word()
    Yhot = get_word()
    OffBits = get_dword()
    SizeHeader = get_dword()

    if SizeHeader = NEWHDRSIZE then
        Width = get_dword()
        Height = get_dword()

```

```

    Planes = get_word()
    BitCount = get_word()
    Compression = get_dword()
    if Compression != 0 then
        close(fn)
        return BMP_UNSUPPORTED_FORMAT
    end if
    SizeImage = get_dword()
    XPelsPerMeter = get_dword()
    YPelsPerMeter = get_dword()
    ClrUsed = get_dword()
    ClrImportant = get_dword()
    NumColors = (OffBits - SizeHeader - BMPFILEHDRSIZE) / 4
    if NumColors < 2 or NumColors > 256 then
        close(fn)
        return BMP_UNSUPPORTED_FORMAT
    end if
    Palette = get_rgb_block(NumColors, 4)

elseif SizeHeader = OLDDHDRSIZE then
    Width = get_word()
    Height = get_word()
    Planes = get_word()
    BitCount = get_word()
    NumColors = (OffBits - SizeHeader - BMPFILEHDRSIZE) / 3
    SizeImage = row_bytes(BitCount, Width) * Height
    Palette = get_rgb_block(NumColors, 3)
else
    close(fn)
    return BMP_UNSUPPORTED_FORMAT
end if
if Planes != 1 or Height <= 0 or Width <= 0 then
    close(fn)
    return BMP_UNSUPPORTED_FORMAT
end if
Bits = get_c_block(row_bytes(BitCount, Width) * Height)
close(fn)
two_d_bits = unpack(Bits, BitCount, Width, Height)
if error_code then
    return error_code
end if
return {Palette, two_d_bits}
end function

type graphics_point(sequence p)
    return length(p) = 2 and p[1] >= 0 and p[2] >= 0
end type

type text_point(sequence p)
    return length(p) = 2 and p[1] >= 1 and p[2] >= 1
        and p[1] <= 200 and p[2] <= 500 -- rough sanity check
end type

global procedure display_image(graphics_point xy, sequence pixels)
-- display a 2-d sequence of pixels at location xy
-- N.B. coordinates are {x, y} with {0,0} at top left of screen

```



```

-- and x values increasing towards the right,
-- and y values increasing towards the bottom of the screen
    for i = 1 to length(pixels) do
        pixel(pixels[i], xy)
        xy[2] += 1
    end for
end procedure

global function save_image(graphics_point top_left, graphics_point bottom_right)
-- Save a rectangular region on a graphics screen,
-- given the {x, y} coordinates of the top-left and bottom-right
-- corner pixels. The result is a 2-d sequence of pixels suitable
-- for use in display_image() above.
    integer x, width
    sequence save

    x = top_left[1]
    width = bottom_right[1] - x + 1
    save = {}
    for y = top_left[2] to bottom_right[2] do
        save = append(save, get_pixel({x, y, width}))
    end for
    return save
end function

constant COLOR_TEXT_MEMORY = #B8000,
        MONO_TEXT_MEMORY = #B0000

constant M_GET_DISPLAY_PAGE = 28,
        M_SET_DISPLAY_PAGE = 29,
        M_GET_ACTIVE_PAGE = 30,
        M_SET_ACTIVE_PAGE = 31

constant BYTES_PER_CHAR = 2

type page_number(integer p)
    return p >= 0 and p <= 7
end type

global function get_display_page()
-- return current page# mapped to the monitor
    return machine_func(M_GET_DISPLAY_PAGE, 0)
end function

global procedure set_display_page(page_number page)
-- select a page to be displayed
    machine_proc(M_SET_DISPLAY_PAGE, page)
end procedure

global function get_active_page()
-- return current page# that screen output is sent to
    return machine_func(M_GET_ACTIVE_PAGE, 0)
end function

global procedure set_active_page(page_number page)
-- select a page for screen output

```

```

    machine_proc(M_SET_ACTIVE_PAGE, page)
end procedure

constant M_GET_SCREEN_CHAR = 58,
       M_PUT_SCREEN_CHAR = 59

type positive_atom(atom x)
    return x >= 1
end type

function DOS_scr_addr(sequence vc, text_point xy)
-- calculate address in DOS screen memory for a given line, column
    atom screen_memory
    integer page_size

    if vc[VC_MODE] = 7 then
        screen_memory = MONO_TEXT_MEMORY
    else
        screen_memory = COLOR_TEXT_MEMORY
    end if
    page_size = vc[VC_LINES] * vc[VC_COLUMNS] * BYTES_PER_CHAR
    page_size = 1024 * floor((page_size + 1023) / 1024)
    screen_memory = screen_memory + get_active_page() * page_size
    return screen_memory + ((xy[1]-1) * vc[VC_COLUMNS] + (xy[2]-1))
        * BYTES_PER_CHAR
end function

global function get_screen_char(positive_atom line, positive_atom column)
-- returns {character, attributes} of the single character
-- at the given (line, column) position on the screen
    atom scr_addr
    sequence vc

    if platform() = DOS32 then
        vc = video_config()
        if line >= 1 and line <= vc[VC_LINES] and
            column >= 1 and column <= vc[VC_COLUMNS] then
            scr_addr = DOS_scr_addr(vc, {line, column})
            return peek({scr_addr, 2})
        else
            return {0,0}
        end if
    else
        return machine_func(M_GET_SCREEN_CHAR, {line, column})
    end if
end function

global procedure put_screen_char(positive_atom line, positive_atom column,
                                sequence char_attr)
-- stores {character, attributes, character, attributes, ...}
-- of 1 or more characters at position (line, column) on the screen
    atom scr_addr
    sequence vc
    integer overflow

    if platform() = DOS32 then

```

```

vc = video_config()
if line <= vc[VC_LINES] and column <= vc[VC_COLUMNS] then
    scr_addr = DOS_scr_addr(vc, {line, column})
    overflow = length(char_attr) - 2 * (vc[VC_COLUMNS] - column + 1)
    if overflow > 0 then
        poke(scr_addr, char_attr[1..$ - overflow])
    else
        poke(scr_addr, char_attr)
    end if
end if
else
    machine_proc(M_PUT_SCREEN_CHAR, {line, column, char_attr})
end if
end procedure

global procedure display_text_image(text_point xy, sequence text)
-- Display a text image at line xy[1], column xy[2] in any text mode.
-- N.B. coordinates are {line, column} with {1,1} at the top left of screen
-- Displays to the active text page.
    atom scr_addr
    integer screen_width, extra_col2, extra_lines
    sequence vc, one_row

    vc = video_config()
    if platform() = DOS32 then
        screen_width = vc[VC_COLUMNS] * BYTES_PER_CHAR
        scr_addr = DOS_scr_addr(vc, xy)
    end if
    if xy[1] < 1 or xy[2] < 1 then
        return -- bad starting point
    end if
    extra_lines = vc[VC_LINES] - xy[1] + 1
    if length(text) > extra_lines then
        if extra_lines <= 0 then
            return -- nothing to display
        end if
        text = text[1..extra_lines] -- truncate
    end if
    extra_col2 = 2 * (vc[VC_COLUMNS] - xy[2] + 1)
    for row = 1 to length(text) do
        one_row = text[row]
        if length(one_row) > extra_col2 then
            if extra_col2 <= 0 then
                return -- nothing to display
            end if
            one_row = one_row[1..extra_col2] -- truncate
        end if
        if platform() = DOS32 then
            poke(scr_addr, one_row)
            scr_addr += screen_width
        else
            machine_proc(M_PUT_SCREEN_CHAR, {xy[1]+row-1, xy[2], one_row})
        end if
    end for
end procedure

```

```

global function save_text_image(text_point top_left, text_point bottom_right)
-- Copy a rectangular block of text out of screen memory,
-- given the coordinates of the top-left and bottom-right corners.
-- Reads from the active text page.
sequence image, row_chars, vc
atom scr_addr, screen_memory
integer screen_width, image_width
integer page_size

vc = video_config()
screen_width = vc[VC_COLUMNS] * BYTES_PER_CHAR
if platform() = DOS32 then
    if vc[VC_MODE] = 7 then
        screen_memory = MONO_TEXT_MEMORY
    else
        screen_memory = COLOR_TEXT_MEMORY
    end if
    page_size = vc[VC_LINES] * screen_width
    page_size = 1024 * floor((page_size + 1023) / 1024)
    screen_memory = screen_memory + get_active_page() * page_size
    scr_addr = screen_memory +
        (top_left[1]-1) * screen_width +
        (top_left[2]-1) * BYTES_PER_CHAR
end if
image = {}
image_width = (bottom_right[2] - top_left[2] + 1) * BYTES_PER_CHAR
for row = top_left[1] to bottom_right[1] do
    if platform() = DOS32 then
        row_chars = peek({scr_addr, image_width})
        scr_addr += screen_width
    else
        row_chars = {}
        for col = top_left[2] to bottom_right[2] do
            row_chars &= machine_func(M_GET_SCREEN_CHAR, {row, col})
        end for
    end if
    image = append(image, row_chars)
end for
return image
end function

```

```

-- save_screen() and related functions were written by
-- Junko C. Miura of Rapid Deployment Software.

```

```

integer numXPixels, numYPixels, bitCount, numRowsBytes
integer startXPixel, startYPixel, endYPixel

type region(object r)
-- a region on the screen
if atom(r) then
    return r = 0
else
    return length(r) = 2 and graphics_point(r[1]) and
        graphics_point(r[2])
end if

```

```

end type

type two_seq(sequence s)
    -- a two element sequence, both elements are sequences
    return length(s) = 2 and sequence(s[1]) and sequence(s[2])
end type

procedure putBmpFileHeader(integer numColors)
    integer offBytes

    -- calculate bitCount, ie, color bits per pixel, (1, 2, 4, 8, or error)
    if numColors = 256 then
        bitCount = 8          -- 8 bits per pixel
    elsif numColors = 16 then
        bitCount = 4          -- 4 bits per pixel
    elsif numColors = 4 then
        bitCount = 2          -- 2 bits per pixel
    elsif numColors = 2 then
        bitCount = 1          -- 1 bit per pixel
    else
        error_code = BMP_INVALID_MODE
        return
    end if

    puts(fn, "BM") -- file-type field in the file header
    offBytes = 4 * numColors + BMPFILEHDRSIZE + NEWHDRSIZE
    numRowsBytes = row_bytes(bitCount, numXPixels)
    -- put total size of the file
    puts(fn, int_to_bytes(offBytes + numRowsBytes * numYPixels))

    puts(fn, {0, 0, 0, 0}) -- reserved fields, must be 0
    puts(fn, int_to_bytes(offBytes)) -- offBytes is the offset to the start
                                     -- of the bitmap information
    puts(fn, int_to_bytes(NEWHDRSIZE)) -- size of the secondary header
    puts(fn, int_to_bytes(numXPixels)) -- width of the bitmap in pixels
    puts(fn, int_to_bytes(numYPixels)) -- height of the bitmap in pixels

    puts(fn, {1, 0}) -- planes, must be a word of value 1

    puts(fn, {bitCount, 0}) -- bitCount

    puts(fn, {0, 0, 0, 0}) -- compression scheme
    puts(fn, {0, 0, 0, 0}) -- size image, not required
    puts(fn, {0, 0, 0, 0}) -- XPelsPerMeter, not required
    puts(fn, {0, 0, 0, 0}) -- YPelsPerMeter, not required
    puts(fn, int_to_bytes(numColors)) -- num colors used in the image
    puts(fn, int_to_bytes(numColors)) -- num important colors in the image
end procedure

procedure putOneRowImage(sequence x, integer numPixelsPerByte, integer shift)
    -- write out one row of image data
    integer j, byte, numBytesFilled

    x &= repeat(0, 7) -- 7 zeros is safe enough

    numBytesFilled = 0

```

```

j = 1
while j <= numXPixels do
    byte = x[j]
    for k = 1 to numPixelsPerByte - 1 do
        byte = byte * shift + x[j + k]
    end for

    puts(fn, byte)
    numBytesFilled += 1
    j += numPixelsPerByte
end while

for m = 1 to numRowsBytes - numBytesFilled do
    puts(fn, 0)
end for
end procedure

procedure putImage()
-- Write image data packed according to the bitCount information, in the order
-- last row ... first row. Data for each row is padded to a 4-byte boundary.
sequence x
integer numPixelsPerByte, shift

numPixelsPerByte = 8 / bitCount
shift = power(2, bitCount)
for i = endYPixel to startYPixel by -1 do
    x = get_pixel({startXPixel, i, numXPixels})
    putOneRowImage(x, numPixelsPerByte, shift)
end for
end procedure

global function get_all_palette()
-- Get color intensities for the entire set of colors in the current
-- graphics mode. Returned sequence is {{r,g,b},{r,g,b},...,{r,g,b}}.
-- Intensity values are in the range 0 to 63.
integer mem, numColors
sequence vc, reg, colors

vc = video_config()
numColors = vc[VC_NCOLORS]
reg = repeat(0, REG_LIST_SIZE)
mem = allocate_low(numColors*3)
if mem then
    reg[REG_AX] = #1017
    reg[REG_BX] = 0
    reg[REG_CX] = numColors
    reg[REG_ES] = floor(mem/16)
    reg[REG_DX] = and_bits(mem, 15)
    reg = dos_interrupt(#10, reg)
    colors = {}
    for col = mem to mem+(numColors-1)*3 by 3 do
        colors = append(colors, peek({col,3}))
    end for
    free_low(mem)
    return colors
else

```

```

        return {} -- unlikely
    end if
end function

procedure putColorTable(integer numColors, sequence pal)
-- Write color table information to the .BMP file.
-- palette data is given as a sequence {{r,g,b},...,{r,g,b}}, where each
-- r, g, or b value is 0 to 255.

    for i = 1 to numColors do
        puts(fn, pal[i][3]) -- blue first in .BMP file
        puts(fn, pal[i][2]) -- green second
        puts(fn, pal[i][1]) -- red third
        puts(fn, 0)         -- reserved, must be 0
    end for
end procedure

global function save_screen(region r, sequence file_name)
-- Capture the whole screen or a region of the screen, and create a Windows
-- bitmap (.BMP) file. The file name is given as a parameter. Region r is
-- either a sequence of 2 sequences: {{topLeftXPixel, topLeftYPixel},
-- {bottomRightXPixel, bottomRightYPixel}} defining a region,
-- or the integer 0 if you want to save the whole screen.
    sequence vc
    integer numColors

    error_code = BMP_SUCCESS
    fn = open(file_name, "wb")
    if fn = -1 then
        return BMP_OPEN_FAILED
    end if

    vc = video_config()
    if sequence(r) then
        numXPixels = r[2][1] - r[1][1] + 1
        numYPixels = r[2][2] - r[1][2] + 1
        if r[2][1] >= vc[VC_XPIXELS] or r[2][2] >= vc[VC_YPIXELS] then
            close(fn)
            return BMP_INVALID_MODE -- not a valid argument
        end if
        startXPixel = r[1][1]
        startYPixel = r[1][2]
        endYPixel = r[2][2]
    else
        numXPixels = vc[VC_XPIXELS]
        numYPixels = vc[VC_YPIXELS]
        startXPixel = 0
        startYPixel = 0
        endYPixel = numYPixels - 1
    end if

    if numXPixels <= 0 or numYPixels <= 0 then
        -- not a valid graphics mode or not a valid argument
        close(fn)
        return BMP_INVALID_MODE
    end if

```

```

numColors = vc[VC_NCOLORS]
putBmpFileHeader(numColors)

if error_code = BMP_SUCCESS then
    putColorTable(numColors, get_all_palette()*4)
end if
if error_code = BMP_SUCCESS then
    putImage()
end if
close(fn)
return error_code
end function

procedure putImage1(sequence image)
-- Write image data packed according to the bitCount information, in the order
-- last row ... first row. Data for each row is padded to a 4-byte boundary.
-- Image data is given as a 2-d sequence in the order first row... last row.
    object    x
    integer    numPixelsPerByte, shift

    numPixelsPerByte = 8 / bitCount
    shift = power(2, bitCount)
    for i = numYPixels to 1 by -1 do
        x = image[i]
        if atom(x) then
            error_code = BMP_INVALID_MODE
            return
        elsif length(x) != numXPixels then
            error_code = BMP_INVALID_MODE
            return
        end if
        putOneRowImage(x, numPixelsPerByte, shift)
    end for
end procedure

global 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.
-- The opposite of read_bitmap().
    sequence color, image
    integer numColors

    error_code = BMP_SUCCESS
    fn = open(file_name, "wb")
    if fn = -1 then
        return BMP_OPEN_FAILED
    end if

    color = palette_n_image[1]
    image = palette_n_image[2]
    numYPixels = length(image)
    numXPixels = length(image[1])    -- assume the same length with each row
    numColors = length(color)

    putBmpFileHeader(numColors)

```



```
    if error_code = BMP_SUCCESS then
        putColorTable(numColors, color)
        putImage1(image)
    end if
    close(fn)
    return error_code
end function
```

## read\_bitmap

**Syntax:**       include image.e

x = read\_bitmap(st)

**Description:**   st is the name of a .bmp "bitmap" file. The file should be in the bitmap format. The most common variations of the format are supported. If the file is read successfully the result will be a 2-element sequence. The first element is the palette, containing intensity values in the range 0 to 255. The second element is a 2-d sequence of sequences containing a pixel-graphics image. You can pass the palette to all\_palette() (after dividing it by 4 to scale it). The image can be passed to display\_image().

Bitmaps of 2, 4, 16 or 256 colors are supported. If the file is not in a good format, an error code (atom) is returned instead:

```
global constant BMP_OPEN_FAILED = 1,
               BMP_UNEXPECTED_EOF = 2,
               BMP_UNSUPPORTED_FORMAT = 3
```

**Comments:**    You can create your own bitmap picture files using Windows Paintbrush and many other graphics programs. You can then incorporate these pictures into your Euphoria programs.

**Example:**

```
x = read_bitmap("c:\\windows\\arcade.bmp")
-- note: double backslash needed to get single backslash in
-- a string
```

**Example Program:**   [demo\\dos32\\bitmap.ex](#)

**See Also:**       [palette](#), [all\\_palette](#), [display\\_image](#), [save\\_bitmap](#)

## display\_image

**Platform:** **DOS32**

**Syntax:** include image.e

display\_image(s1, s2)

**Description:** Display at point s1 on a **pixel-graphics** screen the 2-d sequence of pixels contained in s2. s1 is a two-element sequence {x, y}. s2 is a sequence of sequences, where each sequence is one horizontal row of pixel colors to be displayed. The first pixel of the first sequence is displayed at s1. It is the top-left pixel. All other pixels appear to the right or below of this point.

**Comments:** s2 might be the result of a previous call to `save_image()`, or `read_bitmap()`, or it could be something you have created.

The sequences (rows) of the image do not have to all be the same length.

**Example:**

```
display_image({20,30}, {{7,5,9,4,8},
                        {2,4,1,2},
                        {1,0,1,0,4,6,1},
                        {5,5,5,5,5,5}})
```

```
-- This will display a small image containing 4 rows of
-- pixels. The first pixel (7) of the top row will be at
-- {20,30}. The top row contains 5 pixels. The last row
-- contains 6 pixels ending at {25,33}.
```

**Example Program:** [demo\dos32\bitmap.ex](#)

**See Also:** [save\\_image](#), [read\\_bitmap](#), [display\\_text\\_image](#)

## save\_image

**Platform:** **DOS32**

**Syntax:** include image.e

s3 = save\_image(s1, s2)

**Description:** Save a rectangular image from a **pixel-graphics** screen. The result is a 2-d sequence of sequences containing all the pixels in the image. You can redisplay the image using `display_image()`. s1 is a 2-element sequence {x1,y1} specifying the top-left pixel in the image. s2 is a sequence {x2,y2} specifying the bottom-right pixel.

**Example:**

```
s = save_image({0,0}, {50,50})
display_image({100,200}, s)
display_image({300,400}, s)
-- saves a 51x51 square image, then redisplay it at {100,200}
-- and at {300,400}
```

**See Also:** [display\\_image](#), [save\\_text\\_image](#)

## get\_display\_page

**Platform:** **DOS32**

**Syntax:** include image.e

i = get\_display\_page()

**Description:** Some graphics modes on most video cards have multiple pages of memory. This lets you write screen output to one page while displaying another. get\_display\_page() returns the current page number that is being displayed on the monitor.

**Comments:** The active and display pages are both 0 by default.

video\_config() will tell you how many pages are available in the current graphics mode.

**See Also:** [set\\_display\\_page](#), [get\\_active\\_page](#), [video\\_config](#)

## set\_display\_page

**Platform:** DOS32

**Syntax:** include image.e

set\_display\_page(i)

**Description:** Set video page i to be mapped to the visible screen.

**Comments:** With multiple pages you can instantaneously change the entire screen without causing any visible "flicker". You can also save the screen and bring it back quickly.

video\_config() will tell you how many pages are available in the current graphics mode.

By default, the active page and the display page are both 0.

This works under DOS, or in a full-screen DOS window. In a partial-screen window you cannot change the active page.

**Example:** See set\_active\_page() example.

**See Also:** [get\\_display\\_page](#), [set\\_active\\_page](#), [video\\_config](#)

## get\_active\_page

**Platform:** DOS32

**Syntax:** include image.e

i = get\_active\_page()

**Description:** Some graphics modes on most video cards have multiple pages of memory. This lets you write screen output to one page while displaying a different page. get\_active\_page() returns the current page number that screen output is being sent to.

**Comments:** The active and display pages are both 0 by default.

video\_config() will tell you how many pages are available in the current graphics mode.

**See Also:** [set\\_active\\_page](#), [get\\_display\\_page](#), [video\\_config](#)

## set\_active\_page

**Platform:** DOS32

**Syntax:** include image.e

set\_active\_page(i)

**Description:** Select video page i to send all screen output to.

**Comments:** With multiple pages you can instantaneously change the entire screen without causing any visible "flicker". You can also save the screen and bring it back quickly.

video\_config() will tell you how many pages are available in the current graphics mode.

By default, the active page and the display page are both 0.

This works under DOS, or in a full-screen DOS window. In a partial-screen window you cannot change the active page.

**Example:**

```
include image.e
```

```
-- active & display pages are initially both 0
puts(1, "\nThis is page 0\n")
set_active_page(1)      -- screen output will now go to page 1
clear_screen()
puts(1, "\nNow we've flipped to page 1\n")
if getc(0) then         -- wait for key-press
end if
set_display_page(1)     -- "Now we've ..." becomes visible
if getc(0) then         -- wait for key-press
end if
set_display_page(0)     -- "This is ..." becomes visible again
set_active_page(0)
```

**See Also:** [get\\_active\\_page](#), [set\\_display\\_page](#), [video\\_config](#)



## get\_screen\_char

**Syntax:**           include image.e

s = get\_screen\_char(i1, i2)

**Description:**   Return a 2-element sequence s, of the form **{ascii-code, attributes}** for the character on the screen at line i1, column i2. s consists of two atoms. The first is the ASCII code for the character. The second is an atom that contains the foreground and background color of the character, and possibly other information describing the appearance of the character on the screen.

**Comments:**     With get\_screen\_char() and put\_screen\_char() you can save and restore a character on the screen along with its attributes.

**Example:**

```
      -- read character and attributes at top left corner
s = get_screen_char(1,1)
-- store character and attributes at line 25, column 10
put_screen_char(25, 10, {s})
```

**See Also:**     [put\\_screen\\_char](#), [save\\_text\\_image](#)

## put\_screen\_char

**Syntax:**       include image.e

put\_screen\_char(i1, i2, s)

**Description:** Write zero or more characters onto the screen along with their attributes. i1 specifies the line, and i2 specifies the column where the first character should be written. The sequence s looks like: {ascii-code1, attribute1, ascii-code2, attribute2, ...}. Each pair of elements in s describes one character. The ascii-code atom contains the ASCII code of the character. The attributes atom contains the foreground color, background color, and possibly other platform-dependent information controlling how the character is displayed on the screen.

**Comments:**     The length of s must be a multiple of 2. If s has 0 length, nothing will be written to the screen.

It's faster to write several characters to the screen with a single call to put\_screen\_char() than it is to write one character at a time.

**Example:**

```
-- write AZ to the top left of the screen
-- (attributes are platform-dependent)
put_screen_char(1, 1, {'A', 152, 'Z', 131})
```

**See Also:**     [get\\_screen\\_char](#), [display\\_text\\_image](#)

## display\_text\_image

**Syntax:**       include image.e

display\_text\_image(s1, s2)

**Description:**   Display the 2-d sequence of characters and attributes contained in s2 at line s1[1], column s1[2]. s2 is a sequence of sequences, where each sequence is a string of characters and attributes to be displayed. The top-left character is displayed at s1. Other characters appear to the right or below this position. The attributes indicate the foreground and background color of the preceding character. On DOS32, the attribute should consist of the foreground color plus 16 times the background color.

**Comments:**     s2 would normally be the result of a previous call to save\_text\_image(), although you could construct it yourself.

This routine only works in **text modes**.

You might use save\_text\_image()/display\_text\_image() in a text-mode graphical user interface, to allow "pop-up" dialog boxes, and drop-down menus to appear and disappear without losing what was previously on the screen.

The sequences of the text image do not have to all be the same length.

**Example:**

```
clear_screen()
display_text_image({1,1}, {{ 'A', WHITE, 'B', GREEN},
                           { 'C', RED+16*WHITE},
                           { 'D', BLUE}})

-- displays:
  AB
  C
  D

-- at the top left corner of the screen.
-- 'A' will be white with black (0) background color,
-- 'B' will be green on black,
-- 'C' will be red on white, and
-- 'D' will be blue on black.
```

**See Also:**     [save\\_text\\_image](#), [display\\_image](#), [put\\_screen\\_char](#)

## save\_text\_image

**Syntax:**           include image.e

s3 = save\_text\_image(s1, s2)

**Description:**   Save a rectangular region of text from a **text-mode** screen. The result is a sequence of sequences containing ASCII characters and attributes from the screen. You can redisplay this text using `display_text_image()`. s1 is a 2-element sequence {line1, column1} specifying the top-left character. s2 is a sequence {line2, column2} specifying the bottom right character.

**Comments:**     Because the character attributes are also saved, you will get the correct foreground color, background color and other properties for each character when you redisplay the text.

On DOS32, an attribute byte is made up of two 4-bit fields that encode the foreground and background color of a character. The high-order 4 bits determine the background color, while the low-order 4 bits determine the foreground color.

This routine only works in **text modes**.

You might use this function in a text-mode graphical user interface to save a portion of the screen before displaying a drop-down menu, dialog box, alert box etc.

On DOS32, if you are flipping video pages, note that this function reads from the current active page.

**Example:**       If the top 2 lines of the screen have:

```
      Hello
World
```

And you execute:

```
s = save_text_image({1,1}, {2,5})
```

Then s is something like:

```
      {"H-e-l-l-o-",
      "W-o-r-l-d-"}
```

where '-' indicates the attribute bytes

**See Also:**       [display\\_text\\_image](#), [save\\_image](#), [set\\_active\\_page](#), [get\\_screen\\_char](#)

## get\_all\_palette

**Platform:** DOS32

**Syntax:** include image.e

s = get\_all\_palette()

**Description:** Retrieve color intensities for the entire set of colors in the current graphics mode. s is a sequence of the form:

{{r,g,b}, {r,g,b}, ..., {r,g,b}}

Each element specifies a color intensity {red, green, blue} for the corresponding color number, starting with color number 0. The values for red, green and blue will be in the range 0 to 63.

**Comments:** This function might be used to get the palette values needed by save\_bitmap(). Remember to multiply these values by 4 before calling save\_bitmap(), since save\_bitmap() expects values in the range 0 to 255.

**See Also:** [palette](#), [all\\_palette](#), [read\\_bitmap](#), [save\\_bitmap](#), [save\\_screen](#)

## save\_screen

**Platform:** DOS32

**Syntax:** include image.e

i = save\_screen(x1, st)

**Description:** Save the whole screen or a rectangular region of the screen as a Windows bitmap (.bmp) file. To save the whole screen, pass the integer 0 for x1. To save a rectangular region of the screen, x1 should be a sequence of 2 sequences: {{topLeftXPixel, topLeftYPixel}, {bottomRightXPixel, bottomRightYPixel}}

st is the name of a .bmp "bitmap" file.

The result will be one of the following codes:

```
global constant BMP_SUCCESS = 0,
    BMP_OPEN_FAILED = 1,
    BMP_INVALID_MODE = 4 -- invalid graphics mode
                        -- or invalid argument
```

**Comments:** save\_screen() produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with read\_bitmap(). Windows Paintbrush and some other tools do not support 4-color bitmaps.

save\_screen() only works in **pixel-graphics modes**, not text modes.

### Example 1:

```
-- save whole screen:
code = save_screen(0, "c:\\example\\a1.bmp")
```

### Example 2:

```
-- save part of screen:
err = save_screen({{0,0},{200, 15}}, "b1.bmp")
```

**See Also:** [save\\_image](#), [read\\_bitmap](#), [save\\_bitmap](#)

## save\_bitmap

**Syntax:** include image.e

i = save\_bitmap(s, st)

**Description:** Create a bitmap (.bmp) file from a 2-element sequence s. st is the name of a .bmp "bitmap" file. s[1] is the palette:

{ {r,g,b}, {r,g,b}, ..., {r,g,b} }

Each red, green, or blue value is in the range 0 to 255. s[2] is a 2-d sequence of sequences containing a pixel-graphics image. The sequences contained in s[2] must all have the same length. s is in the same format as the value returned by read\_bitmap().

The result will be one of the following codes:

```
global constant BMP_SUCCESS = 0,
    BMP_OPEN_FAILED = 1,
    BMP_INVALID_MODE = 4 -- invalid graphics mode
                        -- or invalid argument
```

**Comments:** If you use get\_all\_palette() to get the palette before calling this function, you must multiply the returned intensity values by 4 before calling save\_bitmap().

You might use save\_image() to get the 2-d image for s[2].

save\_bitmap() produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with read\_bitmap(). Windows Paintbrush and some other tools do not support 4-color bitmaps.

**Example:**

```
paletteData = get_all_palette() * 4
code = save_bitmap({paletteData, imageData},
    "c:\\example\\a1.bmp")
```

**See Also:** [save\\_image](#), [read\\_bitmap](#), [save\\_screen](#), [get\\_all\\_palette](#)

```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- Machine Level Programming (386/486/Pentium)

-- Warning: Some of these routines require a knowledge of
-- machine-level programming. You could crash your system!

-- These routines, along with peek(), poke() and call(), let you access all
-- of the features of your computer. You can read and write to any memory
-- location, and you can create and execute machine code subroutines.

-- If you are manipulating 32-bit addresses or values, remember to use
-- variables declared as atom. The integer type only goes up to 31 bits.

-- Writing characters to screen memory with poke() is much faster than
-- using puts(). Address of start of text screen memory:
--     mono: #B0000
--     color: #B8000

-- If you choose to call machine_proc() or machine_func() directly (to save
-- a bit of overhead) you *must* pass valid arguments or Euphoria could crash.

-- Some example programs to look at:
--     demo\callmach.ex      - calling a machine language routine
--     demo\dos32\hardint.ex - setting up a hardware interrupt handler
--     demo\dos32\dosint.ex  - calling a DOS software interrupt

-- See also safe.e in this directory. It's a safe, debugging version of this
-- file.

constant M_ALLOC = 16,
         M_FREE = 17,
         M_ALLOC_LOW = 32,
         M_FREE_LOW = 33,
         M_INTERRUPT = 34,
         M_SET_RAND = 35,
         M_USE_VESA = 36,
         M_CRASH_MESSAGE = 37,
         M_TICK_RATE = 38,
         M_GET_VECTOR = 39,
         M_SET_VECTOR = 40,
         M_LOCK_MEMORY = 41,
         M_A_TO_F64 = 46,
         M_F64_TO_A = 47,
         M_A_TO_F32 = 48,
         M_F32_TO_A = 49,
         M_CRASH_FILE = 57,
         M_CRASH_ROUTINE = 66

-- biggest address on a 32-bit machine
constant MAX_ADDR = power(2, 32)-1

-- biggest address accessible to 16-bit real mode

```



```

constant LOW_ADDR = power(2, 20)-1

type positive_int(integer x)
    return x >= 1
end type

type machine_addr(atom a)
    -- a 32-bit non-null machine address
    return a > 0 and a <= MAX_ADDR and floor(a) = a
end type

type far_addr(sequence a)
    -- protected mode far address {seg, offset}
    return length(a) = 2 and integer(a[1]) and machine_addr(a[2])
end type

type low_machine_addr(integer a)
    -- a legal low machine address
    return a > 0 and a <= LOW_ADDR
end type

type sequence_8(sequence s)
    -- an 8-element sequence
    return length(s) = 8
end type

type sequence_4(sequence s)
    -- a 4-element sequence
    return length(s) = 4
end type

global constant REG_LIST_SIZE = 10
global constant REG_DI = 1,
                REG_SI = 2,
                REG_BP = 3,
                REG_BX = 4,
                REG_DX = 5,
                REG_CX = 6,
                REG_AX = 7,
                REG_FLAGS = 8, -- on input: ignored
                                -- on output: low bit has carry flag for
                                -- success/fail
                REG_ES = 9,
                REG_DS = 10

type register_list(sequence r)
    -- a list of register values
    return length(r) = REG_LIST_SIZE
end type

global function allocate(positive_int n)
    -- Allocate n bytes of memory and return the address.
    -- Free the memory using free() below.
    return machine_func(M_ALLOC, n)
end function

```

```

global procedure free(machine_addr a)
-- free the memory at address a
    machine_proc(M_FREE, a)
end procedure

global function allocate_low(positive_int n)
-- Allocate n bytes of low memory (address less than 1Mb)
-- and return the address. Free this memory using free_low() below.
-- Addresses in this range can be passed to DOS during software interrupts.
    return machine_func(M_ALLOC_LOW, n)
end function

global procedure free_low(low_machine_addr a)
-- free the low memory at address a
    machine_proc(M_FREE_LOW, a)
end procedure

global function dos_interrupt(integer int_num, register_list input_regs)
-- call the DOS operating system via software interrupt int_num, using the
-- register values in input_regs. A similar register_list is returned.
-- It contains the register values after the interrupt.
    return machine_func(M_INTERRUPT, {int_num, input_regs})
end function

global function int_to_bytes(atom x)
-- returns value of x as a sequence of 4 bytes
-- that you can poke into memory
--     {bits 0-7,  (least significant)
--     bits 8-15,
--     bits 16-23,
--     bits 24-31} (most significant)
-- This is the order of bytes in memory on 386+ machines.
    integer a,b,c,d

    a = remainder(x, #100)
    x = floor(x / #100)
    b = remainder(x, #100)
    x = floor(x / #100)
    c = remainder(x, #100)
    x = floor(x / #100)
    d = remainder(x, #100)
    return {a,b,c,d}
end function

atom mem
mem = allocate(4)

global function bytes_to_int(sequence s)
-- converts 4-byte peek() sequence into an integer value
    if length(s) = 4 then
        poke(mem, s)
    else
        poke(mem, s[1..4]) -- avoid breaking old code
    end if
    return peek4u(mem)
end function

```

```

global function int_to_bits(atom x, integer nbits)
-- Returns the low-order nbits bits of x as a sequence of 1's and 0's.
-- Note that the least significant bits come first. You can use Euphoria's
-- and/or/not operators on sequences of bits. You can also subscript,
-- slice, concatenate etc. to manipulate bits.
    sequence bits
    integer mask

    bits = repeat(0, nbits)
    if integer(x) and nbits < 30 then
        -- faster method
        mask = 1
        for i = 1 to nbits do
            bits[i] = and_bits(x, mask) and 1
            mask *= 2
        end for
    else
        -- slower, but works for large x and large nbits
        if x < 0 then
            x += power(2, nbits) -- for 2's complement bit pattern
        end if
        for i = 1 to nbits do
            bits[i] = remainder(x, 2)
            x = floor(x / 2)
        end for
    end if
    return bits
end function

global function bits_to_int(sequence bits)
-- get the (positive) value of a sequence of "bits"
    atom value, p

    value = 0
    p = 1
    for i = 1 to length(bits) do
        if bits[i] then
            value += p
        end if
        p += p
    end for
    return value
end function

global procedure set_rand(integer seed)
-- Reset the random number generator.
-- A given value of seed will cause the same series of
-- random numbers to be generated from the rand() function
    machine_proc(M_SET_RAND, seed)
end procedure

global procedure use_vesa(integer code)
-- If code is 1 then force Euphoria to use the VESA graphics standard.
-- This may let Euphoria work better in SVGA modes with certain graphics cards.
-- If code is 0 then Euphoria's normal use of the graphics card is restored.

```

```

-- Values of code other than 0 or 1 should not be used.
    machine_proc(M_USE_VESA, code)
end procedure

-- Crash handling routines:

global procedure crash_message(sequence msg)
-- Specify a final message to display for your user, in the event
-- that Euphoria has to shut down your program due to an error.
    machine_proc(M_CRASH_MESSAGE, msg)
end procedure

global procedure crash_file(sequence file_path)
-- Specify a file path name in place of "ex.err" where you want
-- any diagnostic information to be written.
    machine_proc(M_CRASH_FILE, file_path)
end procedure

global procedure crash_routine(integer proc)
-- specify the routine id of a 1-parameter Euphoria function to call in the
-- event that Euphoria must shut down your program due to an error.
    machine_proc(M_CRASH_ROUTINE, proc)
end procedure

global procedure tick_rate(atom rate)
-- Specify the number of clock-tick interrupts per second.
-- This determines the precision of the time() library routine,
-- and also the sampling rate for time profiling.
    machine_proc(M_TICK_RATE, rate)
end procedure

global function get_vector(integer int_num)
-- returns the current (far) address of the interrupt handler
-- for interrupt vector number int_num as a 2-element sequence:
-- {16-bit segment, 32-bit offset}
    return machine_func(M_GET_VECTOR, int_num)
end function

global procedure set_vector(integer int_num, far_addr a)
-- sets a new interrupt handler address for vector int_num
    machine_proc(M_SET_VECTOR, {int_num, a})
end procedure

global procedure lock_memory(machine_addr a, positive_int n)
-- Prevent a chunk of code or data from ever being swapped out to disk.
-- You should lock any code or data used by an interrupt handler.
    machine_proc(M_LOCK_MEMORY, {a, n})
end procedure

global function atom_to_float64(atom a)
-- Convert an atom to a sequence of 8 bytes in IEEE 64-bit format
    return machine_func(M_A_TO_F64, a)
end function

global function atom_to_float32(atom a)
-- Convert an atom to a sequence of 4 bytes in IEEE 32-bit format

```

```

    return machine_func(M_A_TO_F32, a)
end function

global function float64_to_atom(sequence_8 ieee64)
-- Convert a sequence of 8 bytes in IEEE 64-bit format to an atom
    return machine_func(M_F64_TO_A, ieee64)
end function

global function float32_to_atom(sequence_4 ieee32)
-- Convert a sequence of 4 bytes in IEEE 32-bit format to an atom
    return machine_func(M_F32_TO_A, ieee32)
end function

global function allocate_string(sequence s)
-- create a C-style null-terminated string in memory
    atom mem

    mem = machine_func(M_ALLOC, length(s) + 1) -- Thanks to Igor
    if mem then
        poke(mem, s)
        poke(mem+length(s), 0) -- Thanks to Aku
    end if
    return mem
end function

-- variables and routines used in safe.e
without warning
integer check_calls
check_calls = 1

global procedure register_block(atom block_addr, atom block_len)
end procedure

global procedure unregister_block(atom block_addr)
end procedure

global procedure check_all_blocks()
end procedure
with warning

```

## allocate

**Syntax:**           include machine.e

a = allocate(i)

**Description:**   Allocate i contiguous bytes of memory. Return the address of the block of memory, or return 0 if the memory can't be allocated. The address returned will be at least 4-byte aligned.

**Comments:**      When you are finished using the block, you should pass the address of the block to free(). This will free the block and make the memory available for other purposes. Euphoria will never free or reuse your block until you explicitly call free(). When your program terminates, the operating system will reclaim all memory for use with other programs.

**Example:**

```
        buffer = allocate(100)
for i = 0 to 99 do
    poke(buffer+i, 0)
end for
```

**See Also:**       [free](#), [allocate\\_low](#), [peek](#), [poke](#), [mem\\_set](#), [call](#)

## free

**Syntax:**       include machine.h  
free(a)

**Description:**   Free up a previously allocated block of memory by specifying the address of the start of the block, i.e. the address that was returned by allocate().

**Comments:**     Use free() to recycle blocks of memory during execution. This will reduce the chance of running out of memory or getting into excessive virtual memory swapping to disk. Do not reference a block of memory that has been freed. When your program terminates, all allocated memory will be returned to the system.

Do not use free() to deallocate memory that was allocated using [allocate\\_low\(\)](#). Use free\_low() for this purpose.

**Example Program:**   [demo\callmach.ex](#)

**See Also:**       [allocate](#), [free\\_low](#)

## allocate\_low

**Platform:** **DOS32**

**Syntax:** include machine.e

i2 = allocate\_low(i1)

**Description:** Allocate i1 contiguous bytes of low memory, i.e. conventional memory (address below 1 megabyte). Return the address of the block of memory, or return 0 if the memory can't be allocated.

**Comments:** Some DOS software interrupts require that you pass one or more addresses in registers. These addresses must be conventional memory addresses for DOS to be able to read or write to them.

**Example Program:** [demo\dos32\dosint.ex](#)

**See Also:** [dos\\_interrupt](#), [free\\_low](#), [allocate](#), [peek](#), [poke](#)



## free\_low

**Platform:** DOS32

**Syntax:** include machine.e

free\_low(i)

**Description:** Free up a previously allocated block of conventional memory by specifying the address of the start of the block, i.e. the address that was returned by `allocate_low()`.

**Comments:** Use `free_low()` to recycle blocks of conventional memory during execution. This will reduce the chance of running out of conventional memory. Do not reference a block of memory that has been freed. When your program terminates, all allocated memory will be returned to the system.

Do not use `free_low()` to deallocate memory that was allocated using [allocate\(\)](#). Use `free()` for this purpose.

**Example Program:** [demo\dos32\dosint.ex](#)

**See Also:** [allocate\\_low](#), [dos\\_interrupt](#), [free](#)

## dos\_interrupt

**Platform:** **DOS32**

**Syntax:** `include machine.e`

`s2 = dos_interrupt(i, s1)`

**Description:** Call DOS software interrupt number *i*. *s1* is a 10-element sequence of 16-bit register values to be used as input to the interrupt routine. *s2* is a similar 10-element sequence containing output register values after the call returns. **machine.e** has the following declaration which shows the order of the register values in the input and output sequences.

```
global constant REG_DI = 1,
                REG_SI = 2,
                REG_BP = 3,
                REG_BX = 4,
                REG_DX = 5,
                REG_CX = 6,
                REG_AX = 7,
                REG_FLAGS = 8,
                REG_ES = 9,
                REG_DS = 10
```

**Comments:** The register values returned in *s2* are always positive values between 0 and #FFFF (65535).

The flags value in *s1*[REG\_FLAGS] is ignored on input. On output the least significant bit of *s2*[REG\_FLAGS] has the carry flag, which usually indicates failure if it is set to 1.

Certain interrupts require that you supply addresses of blocks of memory. These addresses must be conventional, low-memory addresses. You can allocate/deallocate low-memory using `allocate_low()` and `free_low()`.

With DOS software interrupts you can perform a wide variety of specialized operations, anything from formatting your floppy drive to rebooting your computer. For documentation on these interrupts consult a technical manual such as Peter Norton's *"PC Programmer's Bible"*, or download Ralf Brown's *Interrupt List* from the Web:

<http://www.cs.cmu.edu/afs/cs.cmu.edu/user/ralf/pub/WWW/files.html>

**Example:**

```
sequence registers
```

```
registers = repeat(0, 10)  -- no registers need to be set
```

```
-- call DOS interrupt 5: Print Screen
```

```
registers = dos_interrupt(#5, registers)
```

**Example Program:** [demoldos32ldosint.ex](#)

**See Also:** [allocate\\_low](#), [free\\_low](#)

## int\_to\_bytes

**Syntax:** `include machine.e`

`s = int_to_bytes(a)`

**Description:** Convert an integer into a sequence of 4 bytes. These bytes are in the order expected on the 386+, i.e. least-significant byte first.

**Comments:** You might use this routine prior to poking the 4 bytes into memory for use by a machine language program.

The integer can be negative. Negative byte-values will be returned, but after poking them into memory you will have the correct (two's complement) representation for the 386+.

This function will correctly convert integer values up to 32-bits. For larger values, only the low-order 32-bits are converted. Euphoria's integer type only allows values up to 31-bits, so declare your variables as **atom** if you need a larger range.

**Example 1:**

```
s = int_to_bytes(999)
-- s is {231, 3, 0, 0}
```

**Example 2:**

```
s = int_to_bytes(-999)
-- s is {-231, -4, -1, -1}
```

**See Also:** [bytes\\_to\\_int](#), [int\\_to\\_bits](#), [bits\\_to\\_int](#), [peek](#), [poke](#), [poke4](#)

## bytes\_to\_int

**Syntax:** `include machine.e`

`a = bytes_to_int(s)`

**Description:** Convert a 4-element sequence of byte values to an atom. The elements of `s` are in the order expected for a 32-bit integer on the 386+, i.e. least-significant byte first.

**Comments:** The result could be greater than the integer type allows, so you should assign it to an **atom**.

`s` would normally contain positive values that have been read using `peek()` from 4 consecutive memory locations.

**Example:**

```
atom int32
```

```
int32 = bytes_to_int({37,1,0,0})
```

```
-- int32 is 37 + 256*1 = 293
```

**See Also:** [int\\_to\\_bytes](#), [bits\\_to\\_int](#), [peek](#), [peek4s](#), [peek4u](#), [poke](#)

... continue

from A to B | [from C to D](#) | [from E to G](#) | [from H to O](#) | [from P to R](#) | [from S to T](#) | [from U to Z](#)

## int\_to\_bits

**Syntax:** `include machine.e`

`s = int_to_bits(a, i)`

**Description:** Return the low-order *i* bits of *a*, as a sequence of 1's and 0's. The least significant bits come first. For negative numbers the two's complement bit pattern is returned.

**Comments:** You can use [subscripting](#), [slicing](#), [and/or/xor/not](#) of entire sequences etc. to manipulate sequences of bits. Shifting of bits and rotating of bits are easy to perform.

**Example:**

```
s = int_to_bits(177, 8)
-- s is {1,0,0,0,1,1,0,1} -- "reverse" order
```

**See Also:** [bits\\_to\\_int](#), [and\\_bits](#), [or\\_bits](#), [xor\\_bits](#), [not\\_bits](#), [operations on sequences](#)

## bits\_to\_int

**Syntax:**           include machine.e

a = bits\_to\_int(s)

**Description:**   Convert a sequence of binary 1's and 0's into a positive number. The least-significant bit is s[1].

**Comments:**     If you print s the bits will appear in "reverse" order, but it is convenient to have increasing subscripts access bits of increasing significance.

**Example:**

```
        a = bits_to_int({1,1,1,0,1})
-- a is 23 (binary 10111)
```

**See Also:**     [int\\_to\\_bits](#), [operations on sequences](#)

## set\_rand

**Syntax:**           include machine.e

set\_rand(i1)

**Description:**   Set the random number generator to a certain state, i1, so that you will get a known series of random numbers on subsequent calls to rand().

**Comments:**      Normally the numbers returned by the rand() function are totally unpredictable, and will be different each time you run your program. Sometimes however you may wish to repeat the same series of numbers, perhaps because you are trying to debug your program, or maybe you want the ability to generate the same output (e.g. a random picture) for your user upon request.

**Example:**

```
sequence s, t
s = repeat(0, 3)
t = s

set_rand(12345)
s[1] = rand(10)
s[2] = rand(100)
s[3] = rand(1000)

set_rand(12345)  -- same value for set_rand()
t[1] = rand(10)  -- same arguments to rand() as before
t[2] = rand(100)
t[3] = rand(1000)
-- at this point s and t will be identical
```

**See Also:**     [rand](#)

## use\_vesa

**Platform:** **DOS32**

**Syntax:** include machine.e

use\_vesa(i)

**Description:** use\_vesa(1) will force Euphoria to use the VESA graphics standard. This may cause Euphoria programs to work better in SVGA graphics modes with certain video cards. use\_vesa(0) will restore Euphoria's original method of using the video card.

**Comments:** Most people can ignore this. However if you experience difficulty in SVGA graphics modes you should try calling use\_vesa(1) at the start of your program before any calls to graphics\_mode().

Arguments to use\_vesa() other than 0 or 1 should not be used.

**Example:**

```
        use_vesa(1)
fail = graphics_mode(261)
```

**See Also:** [graphics\\_mode](#)



## crash\_message

**Syntax:**           include machine.e  
crash\_message(s)

**Description:**   Specify a string, s, to be printed on the screen in the event that Euphoria must stop your program due to a run-time error.

**Comments:**      Normally Euphoria prints a diagnostic message such as "subscript out of bounds", or "divide by zero" on the screen, as well as dumping debugging information into **ex.err**. Euphoria's error messages will not be meaningful for your users unless they happen to be Euphoria programmers. By calling crash\_message() you can control the message that will appear on the screen. Debugging information will still be stored in **ex.err**. You won't lose any information by doing this.

s may contain '\n', new-line characters, so your message can span several lines on the screen. Euphoria will switch to the top of a clear **text-mode** screen before printing your message.

You can call crash\_message() as many times as you like from different parts of your program. The message specified by the last call will be the one displayed.

**Example:**

```
crash_message("An unexpected error has occurred!\n" &  
             "Please contact john_doe@whoops.com\n" &  
             "Do not delete the file \"ex.err\".\n")
```

**See Also:**       [abort](#), [crash\\_file](#), [crash\\_routine](#), [debugging and profiling](#)

## crash\_file

**Syntax:** include machine.e

crash\_file(s)

**Description:** Specify a file name, s, for holding error diagnostics if Euphoria must stop your program due to a compile-time or run-time error.

**Comments:** Normally Euphoria prints a diagnostic message such as "syntax error" or "divide by zero" on the screen, as well as dumping debugging information into **ex.err** in the current directory. By calling crash\_file() you can control the directory and file name where the debugging information will be written.

s may be empty, i.e. "". In this case no diagnostics or debugging information will be written to either a file or the screen. s might also be "NUL" or "/dev/null", in which case diagnostics will be written to the screen, but the ex.err information will be discarded.

You can call crash\_file() as many times as you like from different parts of your program. The file specified by the last call will be the one used.

**Example:**

```
crash_file("\\tmp\\mybug")
```

**See Also:** [abort](#), [crash\\_message](#), [crash\\_routine](#), [debugging and profiling](#)

## crash\_routine

**Syntax:** include machine.e

crash\_routine(i)

**Description:** Pass the routine id of a function that you want Euphoria to call in the event that a run-time error is detected and your program must be shut down. Your function should take one argument of type object. The object that is passed to your function is currently always 0. In future releases of Euphoria, a more meaningful value may be passed. You can call crash\_routine many times with many different routine id's. When a crash occurs, Euphoria will call your crash routines, the most recently specified first, working back to the first one specified. Normally each routine should return 0. If any routine returns a non-zero value, the chain of calls will terminate immediately.

**Comments:** By specifying a crash routine, you give your program a chance to handle fatal run-time errors, such as subscript out of bounds, in a more graceful way. You might save some critical data to disk. You might inform the user about what has happened, and what he can do about it. You might also save some key debugging information. In fact, when your crash routine is called, ex.err will have already been written. Your crash routine could save ex.err somewhere, or even open it and extract information from it, such as the error message.

crash\_routine can be used with the Interpreter or the Translator. Translated code does not check for as many run-time errors, and does not provide a full ex.err dump, but machine-level exceptions are caught, and a crash routine will give you an excellent opportunity to save some variable values to disk for debugging.

The developer of a library might want to specify a crash routine for his library. It could tidy things up by unlocking and closing files, releasing resources etc. The developer of the main program could have his own crash routine. Both routines would be called by Euphoria, unless the first one called (the last one specified) returned non-zero.

A crash routine can't resume execution at the point of the crash, but there is no limitation on what else it can do. It doesn't have to return. It could even reinitialize global variables and effectively restart the program.

If another error occurs while a crash routine is running, a new error dump will occur, but the file name this time will be ex\_crash.err, rather than ex.err. At this point no more calls to crash routines will be allowed. You will have to look at both ex.err and ex\_crash.err to fully understand what took place.

**Example:**

```
function crash(object x)
-- in case of fire ...

-- (on Linux) send an e-mail containing ex.err
system("mail -s \"crash!\" myname@xxx.com < ex.err > /dev/null", 2)

return 0
end function

crash_routine(routine_id("crash"))
```

**See Also:** [abort](#), [crash\\_file](#), [crash\\_message](#), [debugging and profiling](#)

## tick\_rate

**Platform:** DOS32

**Syntax:** include machine.e

tick\_rate(a)

**Description:** Specify the number of clock-tick interrupts per second. This determines the precision of the time() library routine. It also affects the sampling rate for time profiling.

**Comments:** tick\_rate() is ignored on WIN32 and Linux/FreeBSD. The time resolution on WIN32 is always 100 ticks/second.

On a PC the clock-tick interrupt normally occurs at 18.2 interrupts per second. tick\_rate() lets you increase that rate, but not decrease it.

tick\_rate(0) will restore the rate to the normal 18.2 rate. Euphoria will also restore the rate automatically when it exits, even when it finds an error in your program.

If a program runs in a DOS window with a tick rate other than 18.2, the time() function will not advance unless the window is the active window.

With a tick rate other than 18.2, the time() function on DOS takes about 1/100 the usual time that it needs to execute. On Windows and FreeBSD, time() normally executes very quickly.

While **ex.exe** is running, the system will maintain the correct time of day. However if **ex.exe** should crash (e.g. you see a "CauseWay..." error) while the tick rate is high, you (or your user) may need to reboot the machine to restore the proper rate. If you don't, the system time may advance too quickly. This problem does not occur on **Windows 95/98/NT**, only on **DOS** or **Windows 3.1**. You will always get back the correct time of day from the battery-operated clock in your system when you boot up again.

**Example:**

```
tick_rate(100)
-- time() will now advance in steps of .01 seconds
-- instead of the usual .055 seconds
```

**See Also:** [time](#), [time profiling](#)

## get\_vector

**Platform:** **DOS32**

**Syntax:** include machine.e

s = get\_vector(i)

**Description:** Return the current protected mode far address of the handler for interrupt number i. s will be a 2-element sequence: {16-bit segment, 32-bit offset}.

**Example:**

```
s = get_vector(#1C)
-- s will be set to the far address of the clock tick
-- interrupt handler, for example: {59, 808}
```

**Example Program:** [demo\dos32\hardint.ex](#)

**See Also:** [set\\_vector](#), [lock\\_memory](#)

## set\_vector

**Platform:** **DOS32**

**Syntax:** include machine.e

set\_vector(i, s)

**Description:** Set s as the new address for handling interrupt number i. s must be a protected mode **far address** in the form: {16-bit segment, 32-bit offset}.

**Comments:** Before calling set\_vector() you must store a machine-code interrupt handling routine at location s in memory.

The 16-bit segment can be the code segment used by Euphoria. To get the value of this segment see [demo\dos32\hardint.ex](#). The offset can be the 32-bit value returned by allocate(). Euphoria runs in **protected mode** with the code segment and data segment pointing to the same physical memory, but with different access modes.

Interrupts occurring in either **real mode** or **protected mode** will be passed to your handler. Your interrupt handler should immediately load the correct data segment before it tries to reference memory.

Your handler might return from the interrupt using the iretd instruction, or jump to the original interrupt handler. It should save and restore any registers that it modifies.

You should lock the memory used by your handler to ensure that it will never be swapped out. See lock\_memory().

It is highly recommended that you study [demo\dos32\hardint.ex](#) before trying to set up your own interrupt handler.

You should have a good knowledge of machine-level programming before attempting to write your own handler.

You can call set\_vector() with the far address returned by get\_vector(), when you want to restore the original handler.

**Example:**

```
set_vector(#1C, {code_segment, my_handler_address})
```

**Example Program:** [demo\dos32\hardint.ex](#)

**See Also:** [get\\_vector](#), [lock\\_memory](#), [allocate](#)

## lock\_memory

**Platform:** **DOS32**

**Syntax:** include machine.e

lock\_memory(a, i)

**Description:** Prevent the block of virtual memory starting at address a, of length i, from ever being swapped out to disk.

**Comments:** lock\_memory() should only be used in the highly-specialized situation where you have set up your own DOS hardware interrupt handler using machine code. When a hardware interrupt occurs, it is not possible for the operating system to retrieve any code or data that has been swapped out, so you need to protect any blocks of machine code or data that will be needed in servicing the interrupt.

**Example Program:** [demo\dos32\hardint.ex](#)

**See Also:** [get\\_vector](#), [set\\_vector](#)

## atom\_to\_float64

**Syntax:**           include machine.e

s = atom\_to\_float64(a1)

**Description:**   Convert a Euphoria atom to a sequence of 8 single-byte values. These 8 bytes contain the representation of an IEEE floating-point number in 64-bit format.

**Comments:**      All Euphoria atoms have values which can be represented as 64-bit IEEE floating-point numbers, so you can convert any atom to 64-bit format without losing any precision.

Integer values will also be converted to 64-bit floating-point format.

**Example:**

```
fn = open("numbers.dat", "wb")
puts(fn, atom_to_float64(157.82)) -- write 8 bytes to a file
```

**See Also:**       [atom\\_to\\_float32](#), [float64\\_to\\_atom](#)



## atom\_to\_float32

**Syntax:**           include machine.e

s = atom\_to\_float32(a1)

**Description:**   Convert a Euphoria atom to a sequence of 4 single-byte values. These 4 bytes contain the representation of an IEEE floating-point number in 32-bit format.

**Comments:**      Euphoria atoms can have values which are 64-bit IEEE floating-point numbers, so you may lose precision when you convert to 32-bits (16 significant digits versus 7). The range of exponents is much larger in 64-bit format (10 to the 308, versus 10 to the 38), so some atoms may be too large or too small to represent in 32-bit format. In this case you will get one of the special 32-bit values: [inf or -inf \(infinity or -infinity\)](#). To avoid this, you can use atom\_to\_float64().

Integer values will also be converted to 32-bit floating-point format.

**Example:**

```
fn = open("numbers.dat", "wb")
puts(fn, atom_to_float32(157.82)) -- write 4 bytes to a file
```

**See Also:**       [atom\\_to\\_float64](#), [float32\\_to\\_atom](#)

## float64\_to\_atom

**Syntax:**           include machine.e

a1 = float64\_to\_atom(s)

**Description:**   Convert a sequence of 8 bytes to an atom. These 8 bytes must contain an IEEE floating-point number in 64-bit format.

**Comments:**     Any 64-bit IEEE floating-point number can be converted to an atom.

**Example:**

```
        f = repeat(0, 8)
fn = open("numbers.dat", "rb")  -- read binary
for i = 1 to 8 do
    f[i] = getc(fn)
end for
a = float64_to_atom(f)
```

**See Also:**     [float32\\_to\\_atom](#), [atom\\_to\\_float64](#)

## float32\_to\_atom

**Syntax:**           include machine.e

a1 = float32\_to\_atom(s)

**Description:**   Convert a sequence of 4 bytes to an atom. These 4 bytes must contain an IEEE floating-point number in 32-bit format.

**Comments:**     Any 32-bit IEEE floating-point number can be converted to an atom.

**Example:**

```
        f = repeat(0, 4)
fn = open("numbers.dat", "rb") -- read binary
f[1] = getc(fn)
f[2] = getc(fn)
f[3] = getc(fn)
f[4] = getc(fn)
a = float32_to_atom(f)
```

**See Also:**     [float64\\_to\\_atom](#), [atom\\_to\\_float32](#)

## allocate\_string

**Syntax:**           include machine.e

a = allocate\_string(s)

**Description:**   Allocate space for string sequence s. Copy s into this space along with a 0 terminating character. This is the format expected for C strings. The memory address of the string will be returned. If there is not enough memory available, 0 will be returned.

**Comments:**      To free the string, use free().

**Example:**

```
atom title
```

```
title = allocate_string("The Wizard of Oz")
```

**Example Program:**   demo\win32\window.exw

**See Also:**       [allocate](#), [free](#)

## register\_block

**Syntax:** include machine.e (or safe.e)

register\_block(a, i)

**Description:** Add a block of memory to the list of safe blocks maintained by [safe.e](#) (the debug version of [machine.e](#)). The block starts at address a. The length of the block is i bytes.

**Comments:** This routine is only meant to be used for **debugging purposes**. [safe.e](#) tracks the blocks of memory that your program is allowed to [peek\(\)](#), [poke\(\)](#), [mem\\_copy\(\)](#) etc. These are normally just the blocks that you have allocated using Euphoria's [allocate\(\)](#) or [allocate\\_low\(\)](#) routines, and which you have not yet freed using Euphoria's [free\(\)](#) or [free\\_low\(\)](#). In some cases, you may acquire additional, external, blocks of memory, perhaps as a result of calling a C routine. If you are debugging your program using [safe.e](#), you must register these external blocks of memory or [safe.e](#) will prevent you from accessing them. When you are finished using an external block you can unregister it using [unregister\\_block\(\)](#).

When you include [machine.e](#), you'll get different versions of [register\\_block\(\)](#) and [unregister\\_block\(\)](#) that do nothing. This makes it easy to switch back and forth between debug and non-debug runs of your program.

### Example 1:

```
atom addr
```

```
addr = c_func(x, {})  
register_block(addr, 5)  
poke(addr, "ABCDE")  
unregister_block(addr)
```

**See Also:** [unregister\\_block](#), [safe.e](#)

## unregister\_block

**Syntax:** include machine.e (or safe.e)

unregister\_block(a)

**Description:** Remove a block of memory from the list of safe blocks maintained by [safe.e](#) (the debug version of [machine.e](#)). The block starts at address a.

**Comments:** This routine is only meant to be used for **debugging purposes**. Use it to unregister blocks of memory that you have previously registered using register\_block(). By unregistering a block, you remove it from the list of safe blocks maintained by [safe.e](#). This prevents your program from performing any further reads or writes of memory within the block.

See register\_block() for further comments and an example.

**See Also:** [register\\_block](#), [safe.e](#)

```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- Miscellaneous routines and constants

-- platform() values:
global constant DOS32 = 1,  -- ex.exe
                    WIN32 = 2,  -- exw.exe
                    LINUX = 3,  -- exu
                    FREEBSD = 3 -- exu

constant M_INSTANCE = 55, M_SLEEP = 64

global function instance()
-- WIN32: returns hInstance - handle to this instance of the program
-- DOS32: returns 0
    return machine_func(M_INSTANCE, 0)
end function

global procedure sleep(integer t)
-- go to sleep for t seconds
-- allowing (on WIN32 and Linux) other processes to run
    if t >= 0 then
        machine_proc(M_SLEEP, t)
    end if
end procedure

global function reverse(sequence s)
-- reverse the top-level elements of a sequence.
-- Thanks to Hawke' for helping to make this run faster.
    integer lower, n, n2
    sequence t

    n = length(s)
    n2 = floor(n/2)+1
    t = repeat(0, n)
    lower = 1
    for upper = n to n2 by -1 do
        t[upper] = s[lower]
        t[lower] = s[upper]
        lower += 1
    end for
    return t
end function

global function sprint(object x)
-- Return the string representation of any Euphoria data object.
-- This is the same as the output from print(1, x) or '?', but it's
-- returned as a string sequence rather than printed.
    sequence s

    if atom(x) then
        return sprintf("%.10g", x)
    else

```

```

        s = "{"
        for i = 1 to length(x) do
            s &= sprint(x[i])
            if i < length(x) then
                s &= ','
            end if
        end for
        s &= "}"
        return s
    end if
end function

-- pretty print variables
integer pretty_end_col, pretty_chars, pretty_start_col, pretty_level,
    pretty_file, pretty_ascii, pretty_indent, pretty_ascii_min,
    pretty_ascii_max, pretty_line_count, pretty_line_max, pretty_dots
sequence pretty_fp_format, pretty_int_format, pretty_line

procedure pretty_out(object text)
-- Output text, keeping track of line length.
-- Buffering lines speeds up Windows console output.
    pretty_line &= text
    if equal(text, '\n') then
        puts(pretty_file, pretty_line)
        pretty_line = ""
        pretty_line_count += 1
    end if
    if atom(text) then
        pretty_chars += 1
    else
        pretty_chars += length(text)
    end if
end procedure

procedure cut_line(integer n)
-- check for time to do line break
    if pretty_chars + n > pretty_end_col then
        pretty_out('\n')
        pretty_chars = 0
    end if
end procedure

procedure indent()
-- indent the display of a sequence
    if pretty_chars > 0 then
        pretty_out('\n')
        pretty_chars = 0
    end if
    pretty_out(repeat(' ', (pretty_start_col-1) +
        pretty_level * pretty_indent))
end procedure

function show(integer a)
-- show escaped characters
    if a = '\t' then
        return "\\t"
    end if
end function

```



```

elseif a = '\n' then
    return "\\n"
elseif a = '\r' then
    return "\\r"
else
    return a
end if
end function

procedure rPrint(object a)
-- recursively print a Euphoria object
sequence sbuff
integer multi_line, all_ascii

if atom(a) then
    if integer(a) then
        sbuff = sprintf(pretty_int_format, a)
        if pretty_ascii then
            if pretty_ascii >= 3 then
                -- replace number with display character?
                if (a >= pretty_ascii_min and a <= pretty_ascii_max) then
                    sbuff = '\\' & a & '\\' -- display char only

                elseif find(a, "\t\n\r") then
                    sbuff = '\\' & show(a) & '\\' -- display char only

                end if
            else -- pretty_ascii 1 or 2
                -- add display character to number?
                if (a >= pretty_ascii_min and a <= pretty_ascii_max) then
                    sbuff &= '\\' & a & '\\' -- add to numeric display
                end if
            end if
        end if
    else
        sbuff = sprintf(pretty_fp_format, a)
    end if
    pretty_out(sbuff)

else
    -- sequence
    cut_line(1)
    multi_line = 0
    all_ascii = pretty_ascii > 1
    for i = 1 to length(a) do
        if sequence(a[i]) and length(a[i]) > 0 then
            multi_line = 1
            all_ascii = 0
            exit
        end if
        if not integer(a[i]) or
            (a[i] < pretty_ascii_min and
             (pretty_ascii < 3 or not find(a[i], "\t\r\n"))) or
            a[i] > pretty_ascii_max then
            all_ascii = 0
        end if
    end if
end if

```

```

end for

if all_ascii then
    pretty_out('\\"')
else
    pretty_out('{')
end if
pretty_level += 1
for i = 1 to length(a) do
    if multi_line then
        indent()
    end if
    if all_ascii then
        pretty_out(show(a[i]))
    else
        rPrint(a[i])
    end if
    if pretty_line_count >= pretty_line_max then
        if not pretty_dots then
            pretty_out(" ...")
        end if
        pretty_dots = 1
        return
    end if
    if i != length(a) and not all_ascii then
        pretty_out(',')
        cut_line(6)
    end if
end for
pretty_level -= 1
if multi_line then
    indent()
end if
if all_ascii then
    pretty_out('\\"')
else
    pretty_out('{')
end if
end if
end procedure

```

```

global procedure pretty_print(integer fn, object x, sequence options)
-- Print any Euphoria object x, to file fn, in a form that shows
-- its structure.
--
-- argument 1: file number to write to
-- argument 2: the object to display
-- argument 3: is an (up to) 8-element options sequence:
--   Pass {} to select the defaults, or set options as below:
--   [1] display ASCII characters:
--       0: never
--       1: alongside any integers in printable ASCII range (default)
--       2: display as "string" when all integers of a sequence
--          are in ASCII range
--       3: show strings, and quoted characters (only) for any integers

```

```
--      in ASCII range 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
--
-- If the length is less than 8, unspecified options at
-- the end of the sequence will keep the default values.
-- e.g. {0, 5} will choose "never display ASCII",
-- plus 5-character indentation, with defaults for everything else
integer n

-- set option defaults
pretty_ascii = 1          --[1]
pretty_indent = 2         --[2]
pretty_start_col = 1      --[3]
pretty_end_col = 78       --[4]
pretty_int_format = "%d"  --[5]
pretty_fp_format = "%.10g" --[6]
pretty_ascii_min = 32     --[7]
pretty_ascii_max = 127    --[8]
    - (platform() = LINUX) -- DEL is a problem with ANSI code display
pretty_line_max = 1000000000 --[9]

n = length(options)
if n >= 1 then
    pretty_ascii = options[1]
    if n >= 2 then
        pretty_indent = options[2]
        if n >= 3 then
            pretty_start_col = options[3]
            if n >= 4 then
                pretty_end_col = options[4]
                if n >= 5 then
                    pretty_int_format = options[5]
                    if n >= 6 then
                        pretty_fp_format = options[6]
                        if n >= 7 then
                            pretty_ascii_min = options[7]
                            if n >= 8 then
                                pretty_ascii_max = options[8]
                                if n >= 9 then
                                    pretty_line_max = options[9]
                                end if
                            end if
                        end if
                    end if
                end if
            end if
        end if
    end if
end if
end if
end if
end if
```

```

    pretty_chars = pretty_start_col
    pretty_file = fn
    pretty_level = 0
    pretty_line = ""
    pretty_line_count = 0
    pretty_dots = 0
    rPrint(x)
    puts(pretty_file, pretty_line)
end procedure

-- trig formulas provided by Larry Gregg

global constant PI = 3.141592653589793238

constant PI_HALF = PI / 2.0 -- this is pi/2

type trig_range(object x)
-- values passed to arccos and arcsin must be [-1,+1]
  if atom(x) then
    return x >= -1 and x <= 1
  else
    for i = 1 to length(x) do
      if not trig_range(x[i]) then
        return 0
      end if
    end for
    return 1
  end if
end type

global function arccos(trig_range x)
-- returns angle in radians
  return PI_HALF - 2 * arctan(x / (1.0 + sqrt(1.0 - x * x)))
end function

global function arcsin(trig_range x)
-- returns angle in radians
  return 2 * arctan(x / (1.0 + sqrt(1.0 - x * x)))
end function

```

## instance

**Platform:** WIN32

**Syntax:** include misc.e

i = instance()

**Description:** Return a handle to the current program.

**Comments:** This handle value can be passed to various Windows routines to get information about the current program that is running, i.e. your program. Each time a user starts up your program, a different instance will be created.

In C, this is the first parameter to WinMain().

On **DOS32 and Linux/FreeBSD**, instance() always returns 0.

**See Also:** [platform.doc](#)

## sleep

**Syntax:** include misc.e

sleep(i)

**Description:** Suspend execution for i seconds.

**Comments:** On WIN32 and Linux/FreeBSD, the operating system will suspend your process and schedule other processes. On DOS32, your program will go into a busy loop for i seconds, during which time other processes may run, but they will compete with your process for the CPU.

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()`.

**Example:**

```
puts(1, "Waiting 15 seconds...\n")
sleep(15)
puts(1, "Done.\n")
```

**See Also:** [lock\\_file](#), [abort](#), [time](#)

## reverse

**Syntax:** include misc.e

s2 = reverse(s1)

**Description:** Reverse the order of elements in a sequence.

**Comments:** A new sequence is created where the top-level elements appear in reverse order compared to the original sequence.

**Example 1:**

```
reverse({1, 3, 5, 7})      -- {7, 5, 3, 1}
```

**Example 2:**

```
reverse({{1, 2, 3}, {4, 5, 6}}) -- {{4, 5, 6}, {1, 2, 3}}
```

**Example 3:**

```
reverse({99})             -- {99}
```

**Example 4:**

```
reverse({})               -- {}
```

**See Also:** [append](#), [prepend](#), [repeat](#)

## sprint

**Syntax:** include misc.e

`s = sprint(x)`

**Description:** The representation of `x` as a string of characters is returned. This is exactly the same as `print(fn, x)`, except that the output is returned as a sequence of characters, rather than being sent to a file or device. `x` can be any Euphoria object.

**Comments:** The atoms contained within `x` will be displayed to a maximum of 10 significant digits, just as with `print()`.

**Example 1:**

```
s = sprint(12345)
-- s is "12345"
```

**Example 2:**

```
s = sprint({10,20,30}+5)
-- s is "{15,25,35}"
```

**See Also:** [print](#), [sprintf](#), [value](#), [get](#)



## pretty\_print

**Syntax:** include misc.e

pretty\_print(fn, x, s)

**Description:** Print, to file or device fn, an object x, using braces { , , , }, indentation, and multiple lines to show the structure.

Several options may be supplied in s to control the presentation. Pass { } to select the defaults, or set options as below:

[1] display ASCII characters:

\* 0: never

\* 1: alongside any integers in the printable ASCII range 32..127 (default)

\* 2: like 1, plus display as "string" when all integers of a sequence are in the printable ASCII range

\* 3: like 2, but show \*only\* quoted characters, not numbers, for any integers in the printable ASCII range, as well as the whitespace 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

If the length of s is less than 8, unspecified options at the end of the sequence will keep the default values. e.g. {0, 5} will choose "never display ASCII", plus 5-character indentation, with defaults for everything else.

**Comments:** The display will start at the current cursor position. Normally you will want to call pretty\_print() when the cursor is in column 1 (after printing a \n character). If you want to start in a different column, you should call position() and specify a value for option [3]. This will ensure that the first and last braces in a sequence line up vertically.

When specifying the format to use for integers and floating-point numbers, you can add some decoration, e.g. "(%d)" or "\$ %.2f"

### Example 1:

```
pretty_print(1, "ABC", {})
```

```
{65'A',66'B',67'C'}
```

### Example 2:

```
pretty_print(1, {{1,2,3}, {4,5,6}}, {})
```

```
{
  {1,2,3},
  {4,5,6}
}
```

### Example 3:

```
pretty_print(1, {"Euphoria", "Programming", "Language"}, {2})
```

```
{
  "Euphoria",
```

```

    "Programming",
    "Language"
}

```

#### Example 4:

```

    puts(1, "word_list = ") -- moves cursor to column 13
pretty_print(1,
    {"Euphoria", 8, 5.3},
    {"Programming", 11, -2.9},
    {"Language", 8, 9.8}},
    {2, 4, 13, 78, "%03d", "%.3f"}) -- first 6 of 8 options

word_list = {
    {
        "Euphoria",
        008,
        5.300
    },
    {
        "Programming",
        011,
        -2.900
    },
    {
        "Language",
        008,
        9.800
    }
}

```

**See Also:**    [2](#), [print](#), [puts](#), [printf](#)

## arccos

**Syntax:** include misc.e

`x2 = arccos(x1)`

**Description:** Return an angle with cosine equal to x1.

**Comments:** The argument, x1, must be in the range -1 to +1 inclusive.

A value between 0 and [PI](#) radians will be returned.

This function may be applied to an atom or to all elements of a sequence.

`arccos()` is not as fast as `arctan()`.

**Example:**

```
s = arccos({-1,0,1})  
-- s is {3.141592654, 1.570796327, 0}
```

**See Also:** [cos](#), [arcsin](#), [arctan](#)

## arcsin

**Syntax:** include misc.e

`x2 = arcsin(x1)`

**Description:** Return an angle with sine equal to x1.

**Comments:** The argument, x1, must be in the range -1 to +1 inclusive.

A value between  $-\pi/2$  and  $+\pi/2$  (radians) will be returned.

This function may be applied to an atom or to all elements of a sequence.

`arcsin()` is not as fast as `arctan()`.

**Example:**

```
s = arcsin({-1,0,1})  
-- s is {-1.570796327, 0, 1.570796327}
```

**See Also:** [sin](#), [arccos](#), [arctan](#)

```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- Mouse Routines

-- DOS32 - you need a mouse driver
-- Linux - you need GPM server to be running
-- WIN32 - not implemented yet for the text console

include misc.e

-- Mouse Events:
global integer MOVE, LEFT_DOWN, LEFT_UP, RIGHT_DOWN, RIGHT_UP,
               MIDDLE_DOWN, MIDDLE_UP, ANY_UP

if platform() = LINUX then
    MOVE = 0
    LEFT_DOWN = 4
    LEFT_UP = 4
    RIGHT_DOWN = 1
    RIGHT_UP = 1
    MIDDLE_DOWN = 2
    MIDDLE_UP = 2
    ANY_UP = 35 -- LEFT, RIGHT or MIDDLE up (best you can do under xterm)
else
    MOVE = 1
    LEFT_DOWN = 2
    LEFT_UP = 4
    RIGHT_DOWN = 8
    RIGHT_UP = 16
    MIDDLE_DOWN = 32
    MIDDLE_UP = 64
end if

constant M_GET_MOUSE = 14,
         M_MOUSE_EVENTS = 15,
         M_MOUSE_POINTER = 24

global function get_mouse()
-- report mouse events,
-- returns -1 if no mouse event,
-- otherwise returns {event#, x-coord, y-coord}
    return machine_func(M_GET_MOUSE, 0)
end function

global procedure mouse_events(integer events)
-- select the mouse events to be reported by get_mouse()
-- e.g. mouse_events(LEFT_UP + LEFT_DOWN + RIGHT_DOWN)
    machine_proc(M_MOUSE_EVENTS, events)
end procedure

global procedure mouse_pointer(integer show_it)
-- show (1) or hide (0) the mouse pointer
    machine_proc(M_MOUSE_POINTER, show_it)

```

end procedure

## get\_mouse

**Platform:** DOS32, Linux

**Syntax:** include mouse.e

x1 = get\_mouse()

**Description:** Return the last mouse event in the form: {event, x, y} or return -1 if there has not been a mouse event since the last time get\_mouse() was called.

Constants have been defined in [mouse.e](#) for the possible mouse events:

```
global constant MOVE = 1,
    LEFT_DOWN = 2,
    LEFT_UP = 4,
    RIGHT_DOWN = 8,
    RIGHT_UP = 16,
    MIDDLE_DOWN = 32,
    MIDDLE_UP = 64
```

x and y are the coordinates of the mouse pointer at the time that the event occurred. get\_mouse() returns immediately with either a -1 or a mouse event. It does not wait for an event to occur. You must check it frequently enough to avoid missing an event. When the next event occurs, the current event will be lost, if you haven't read it. In practice it is not hard to catch almost all events. Losing a MOVE event is generally not too serious, as the next MOVE will tell you where the mouse pointer is.

Sometimes multiple events will be reported. For example, if the mouse is moving when the left button is clicked, get\_mouse() will report an event value of LEFT\_DOWN+MOVE, i.e. 2+1 or 3. For this reason you should test for a particular event using and\_bits(). See examples below.

**Comments:** In [pixel-graphics modes](#) that are 320 pixels wide, you need to divide the x value by 2 to get the correct position on the screen. (A strange feature of DOS.)

In DOS32 [text modes](#) you need to scale the x and y coordinates to get line and column positions. In Linux, no scaling is required - x and y correspond to the line and column on the screen, with (1,1) at the top left.

In DOS32, you need a DOS mouse driver to use this routine. In Linux, GPM Server must be running.

In Linux, mouse movement events are not reported in an xterm window, only in the text console.

In Linux, LEFT\_UP, RIGHT\_UP and MIDDLE\_UP are not distinguishable from one another.

You can use get\_mouse() in [most text and pixel-graphics modes](#).

The first call that you make to get\_mouse() will turn on a mouse pointer, or a highlighted character.

DOS generally does not support the use of a mouse in SVGA graphics modes (beyond 640x480 pixels). This restriction has been removed in Windows 95 (DOS 7.0). **Graeme Burke**, **Peter Blue** and others have contributed **mouse routines** that get around the problems with using a mouse in SVGA. See the [Euphoria Archive Web page](#).

The x,y coordinate returned could be that of the very tip of the mouse pointer or might refer to the pixel pointed-to by the mouse pointer. Test this if you are trying to read the pixel color using [get\\_pixel\(\)](#). You may have to read x-1,y-1 instead.

**Example 1:** a return value of:

```
{2, 100, 50}
```

would indicate that the left button was pressed down when the mouse pointer was at location x=100, y=50 on the screen.

**Example 2:** To test for LEFT\_DOWN, write something like the following:

```
object event

while 1 do
  event = get_mouse()
  if sequence(event) then
    if and_bits(event[1], LEFT_DOWN) then
      -- left button was pressed
      exit
    end if
  end if
end while
```

**See Also:** [mouse\\_events](#), [mouse\\_pointer](#), [and\\_bits](#)



## mouse\_events

**Platform:** **DOS32, Linux**

**Syntax:** include mouse.e

mouse\_events(i)

**Description:** Use this procedure to select the mouse events that you want get\_mouse() to report. By default, get\_mouse() will report all events. mouse\_events() can be called at various stages of the execution of your program, as the need to detect events changes. Under Linux, mouse\_events() currently has no effect.

**Comments:** It is good practice to ignore events that you are not interested in, particularly the very frequent MOVE event, in order to reduce the chance that you will miss a significant event.

The first call that you make to mouse\_events() will turn on a mouse pointer, or a highlighted character.

**Example:**

```
mouse_events(LEFT_DOWN + LEFT_UP + RIGHT_DOWN)
-- will restrict get_mouse() to reporting the left button
-- being pressed down or released, and the right button
-- being pressed down. All other events will be ignored.
```

**See Also:** [get\\_mouse](#), [mouse\\_pointer](#)

## mouse\_pointer

**Platform:** **DOS32, Linux**

**Syntax:** include mouse.e

mouse\_pointer(i)

**Description:** If i is 0 hide the mouse pointer, otherwise turn on the mouse pointer. Multiple calls to hide the pointer will require multiple calls to turn it back on. The first call to either `get_mouse()` or `mouse_events()`, will also turn the pointer on (once). Under Linux, `mouse_pointer()` currently has no effect

**Comments:** It may be necessary to hide the mouse pointer temporarily when you update the screen.

After a call to [text\\_rows\(\)](#) you may have to call `mouse_pointer(1)` to see the mouse pointer again.

**See Also:** [get\\_mouse](#), [mouse\\_events](#)

```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- Windows message_box() function

include dll.e
include machine.e
include misc.e

without warning

-- Possible style values for message_box() style sequence
global constant
  MB_ABORTRETRYIGNORE = #02, -- Abort, Retry, Ignore
  MB_APPLMODAL = #00,      -- User must respond before doing something else
  MB_DEFAULT_DESKTOP_ONLY = #20000,
  MB_DEFBUTTON1 = #00,    -- First button is default button
  MB_DEFBUTTON2 = #100,   -- Second button is default button
  MB_DEFBUTTON3 = #200,   -- Third button is default button
  MB_DEFBUTTON4 = #300,   -- Fourth button is default button
  MB_HELP = #4000,        -- Windows 95: Help button generates help event
  MB_ICONASTERISK = #40,
  MB_ICONERROR = #10,
  MB_ICONEXCLAMATION = #30, -- Exclamation-point appears in the box
  MB_ICONHAND = MB_ICONERROR, -- A hand appears
  MB_ICONINFORMATION = MB_ICONASTERISK, -- Lowercase letter i in a circle
appears
  MB_ICONQUESTION = #20, -- A question-mark icon appears
  MB_ICONSTOP = MB_ICONHAND,
  MB_ICONWARNING = MB_ICONEXCLAMATION,
  MB_OK = #00,          -- Message box contains one push button: OK
  MB_OKCANCEL = #01,    -- Message box contains OK and Cancel
  MB_RETRYCANCEL = #05, -- Message box contains Retry and Cancel
  MB_RIGHT = #80000,    -- Windows 95: The text is right-justified
  MB_RTLCREADING = #100000, -- Windows 95: For Hebrew and Arabic systems
  MB_SERVICE_NOTIFICATION = #40000, -- Windows NT: The caller is a service
  MB_SETFOREGROUND = #10000, -- Message box becomes the foreground window
  MB_SYSTEMMODAL = #1000, -- All applications suspended until user responds
  MB_TASKMODAL = #2000, -- Similar to MB_APPLMODAL
  MB_YESNO = #04,       -- Message box contains Yes and No
  MB_YESNOCANCEL = #03 -- Message box contains Yes, No, and Cancel

-- possible values returned by MessageBox()
-- 0 means failure
global constant IDABORT = 3, -- Abort button was selected.
  IDCANCEL = 2, -- Cancel button was selected.
  IDIGNORE = 5, -- Ignore button was selected.
  IDNO = 7, -- No button was selected.
  IDOK = 1, -- OK button was selected.
  IDRETRY = 4, -- Retry button was selected.
  IDYES = 6 -- Yes button was selected.

atom lib
integer msgbox_id, get_active_id

```

```

if platform() = WIN32 then
  lib = open_dll("user32.dll")
  msgbox_id = define_c_func(lib, "MessageBoxA", {C_POINTER, C_POINTER,
                                                    C_POINTER, C_INT}, C_INT)

  if msgbox_id = -1 then
    puts(2, "couldn't find MessageBoxA\n")
    abort(1)
  end if

  get_active_id = define_c_func(lib, "GetActiveWindow", {}, C_LONG)
  if get_active_id = -1 then
    puts(2, "couldn't find GetActiveWindow\n")
    abort(1)
  end if
end if

global function message_box(sequence text, sequence title, object style)
  integer or_style
  atom text_ptr, title_ptr, ret

  text_ptr = allocate_string(text)
  if not text_ptr then
    return 0
  end if
  title_ptr = allocate_string(title)
  if not title_ptr then
    free(text_ptr)
    return 0
  end if
  if atom(style) then
    or_style = style
  else
    or_style = 0
    for i = 1 to length(style) do
      or_style = or_bits(or_style, style[i])
    end for
  end if
  ret = c_func(msgbox_id, {c_func(get_active_id, {}),
                           text_ptr, title_ptr, or_style})

  free(text_ptr)
  free(title_ptr)
  return ret
end function

```

## message\_box

**Platform:** WIN32

**Syntax:** include msgbox.e

i = message\_box(s1, s2, x)

**Description:** Display a window with title s2, containing the message string s1. x determines the combination of buttons that will be available for the user to press, plus some other characteristics. x can be an atom or a sequence. A return value of 0 indicates a failure to set up the window.

**Comments:** See [msgbox.e](#) for a complete list of possible values for x and i.

**Example:**

```
        response = message_box("Do you wish to proceed?",
                                "My Application",
                                MB_YESNOCANCEL)
if response = IDCANCEL or response = IDNO then
    abort(1)
end if
```

**Example Program:** [demo\win32\email.exw](#)

```
-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- Sorting

-- Sort the elements of a sequence into ascending order, using "Shell" sort.
```

```
global function sort(sequence x)
-- Sort a sequence into ascending order. The elements can be atoms or
-- sequences. The standard compare() routine is used to compare elements.
integer gap, j, first, last
object tempi, tempj
```

```
last = length(x)
gap = floor(last / 10) + 1
while 1 do
    first = gap + 1
    for i = first to last do
        tempi = x[i]
        j = i - gap
        while 1 do
            tempj = x[j]
            if compare(tempi, tempj) >= 0 then
                j += gap
                exit
            end if
            x[j+gap] = tempj
            if j <= gap then
                exit
            end if
            j -= gap
        end while
        x[j] = tempi
    end for
    if gap = 1 then
        return x
    else
        gap = floor(gap / 3.5) + 1
    end if
end while
end function
```

```
global function custom_sort(integer custom_compare, sequence x)
-- Sort a sequence. A user-supplied comparison function is used
-- to compare elements. Note that this sort is not "stable", i.e.
-- elements that are considered equal might change position relative
-- to each other.
integer gap, j, first, last
object tempi, tempj

last = length(x)
gap = floor(last / 10) + 1
while 1 do
    first = gap + 1
```

```

for i = first to last do
    tempi = x[i]
    j = i - gap
    while 1 do
        tempj = x[j]
        if call_func(custom_compare, {tempi, tempj}) >= 0 then
            j += gap
            exit
        end if
        x[j+gap] = tempj
        if j <= gap then
            exit
        end if
        j -= gap
    end while
    x[j] = tempi
end for
if gap = 1 then
    return x
else
    gap = floor(gap / 3.5) + 1
end if
end while
end function

```

## sort

**Syntax:** include sort.e

s2 = sort(s1)

**Description:** Sort s1 into ascending order using a fast sorting algorithm. The elements of s1 can be any mix of atoms or sequences. Atoms come before sequences, and sequences are sorted "alphabetically" where the first elements are more significant than the later elements.

**Example 1:**

```
x = 0 & sort({7,5,3,8}) & 0
-- x is set to {0, 3, 5, 7, 8, 0}
```

**Example 2:**

```
y = sort({"Smith", "Jones", "Doe", 5.5, 4, 6})
-- y is {4, 5.5, 6, "Doe", "Jones", "Smith"}
```

**Example 3:**

```
database = sort({{"Smith", 95.0, 29},
                  {"Jones", 77.2, 31},
                  {"Clinton", 88.7, 44}})

-- The 3 database "records" will be sorted by the first "field"
-- i.e. by name. Where the first field (element) is equal it
-- will be sorted by the second field etc.

-- after sorting, database is:
      {{ "Clinton", 88.7, 44},
        {"Jones", 77.2, 31},
        {"Smith", 95.0, 29}}
```

**See Also:** [custom\\_sort](#), [compare](#), [match](#), [find](#)



## custom\_sort

**Syntax:**           include sort.e

s2 = custom\_sort(i, s1)

**Description:**   Sort the elements of sequence s1, using a compare function with **routine id** i.

**Comments:**     Your compare function must be a function of two arguments similar to Euphoria's compare(). It will compare two objects and return -1, 0 or +1.

**Example Program:**   [demo\csort.ex](#)

**See Also:**       [sort](#), [compare](#), [routine\\_id](#)

```

-- (c) Copyright 2006 Rapid Deployment Software - See License.txt
--
-- Euphoria 3.0
-- wild card matching for strings and file names

include misc.e

constant TO_LOWER = 'a' - 'A'

global function lower(object x)
-- convert atom or sequence to lower case
    return x + (x >= 'A' and x <= 'Z') * TO_LOWER
end function

global function upper(object x)
-- convert atom or sequence to upper case
    return x - (x >= 'a' and x <= 'z') * TO_LOWER
end function

function qmatch(sequence p, sequence s)
-- find pattern p in string s
-- p may have '?' wild cards (but not '*')
    integer k

    if not find('?', p) then
        return match(p, s) -- fast
    end if
    -- must allow for '?' wildcard
    for i = 1 to length(s) - length(p) + 1 do
        k = i
        for j = 1 to length(p) do
            if p[j] != s[k] and p[j] != '?' then
                k = 0
                exit
            end if
            k += 1
        end for
        if k != 0 then
            return i
        end if
    end for
    return 0
end function

constant END_MARKER = -1

global function wildcard_match(sequence pattern, sequence string)
-- returns TRUE if string matches pattern
-- pattern can include '*' and '?' "wildcard" characters
    integer p, f, t
    sequence match_string

    pattern = pattern & END_MARKER
    string = string & END_MARKER

```

```

p = 1
f = 1
while f <= length(string) do
    if not find(pattern[p], {string[f], '?'}) then
        if pattern[p] = '*' then
            while pattern[p] = '*' do
                p += 1
            end while
            if pattern[p] = END_MARKER then
                return 1
            end if
            match_string = ""
            while pattern[p] != '*' do
                match_string = match_string & pattern[p]
                if pattern[p] = END_MARKER then
                    exit
                end if
                p += 1
            end while
            if pattern[p] = '*' then
                p -= 1
            end if
            t = qmatch(match_string, string[f..$])
            if t = 0 then
                return 0
            else
                f += t + length(match_string) - 2
            end if
        else
            return 0
        end if
    end if
    p += 1
    f += 1
    if p > length(pattern) then
        return f > length(string)
    end if
end while
return 0
end function

global function wildcard_file(sequence pattern, sequence filename)
-- Return 1 (TRUE) if filename matches the wild card pattern.
-- Similar to DOS wild card matching but better. For example,
-- "*ABC.*" in DOS will match *all* files, where this function will
-- only match when the file name part has "ABC" at the end.

    if platform() != LINUX then
        pattern = upper(pattern)
        filename = upper(filename)
    end if
    if not find('.', pattern) then
        pattern = pattern & '.'
    end if
    if not find('.', filename) then
        filename = filename & '.'
    end if

```

```
    end if  
    return wildcard_match(pattern, filename)  
end function
```

## lower

**Syntax:**           include wildcard.e

x2 = lower(x1)

**Description:**   Convert an atom or sequence to lower case.

**Example:**

```
s = lower("Euphoria")
-- s is "euphoria"

a = lower('B')
-- a is 'b'

s = lower({"Euphoria", "Programming"})
-- s is {"euphoria", "programming"}
```

**See Also:**   [upper](#)

## upper

**Syntax:**       include wildcard.e

x2 = upper(x1)

**Description:**   Convert an atom or sequence to upper case.

**Example:**

```
s = upper("Euphoria")
-- s is "EUPHORIA"

a = upper('g')
-- a is 'G'

s = upper({"Euphoria", "Programming"})
-- s is {"EUPHORIA", "PROGRAMMING"}
```

**See Also:**     [lower](#)

## wildcard\_match

**Syntax:**       include wildcard.e

i = wildcard\_match(st1, st2)

**Description:**   This function performs a general matching of a string against a pattern containing \* and ? wildcards. It returns 1 (true) if string st2 matches pattern st1. It returns 0 (false) otherwise. \* matches any 0 or more characters. ? matches any single character. Character comparisons are case sensitive.

**Comments:**     If you want case insensitive comparisons, pass both st1 and st2 through upper(), or both through lower() before calling wildcard\_match().

If you want to detect a pattern anywhere within a string, add \* to each end of the pattern:

```
i = wildcard_match('*' & pattern & '*', string)
```

There is currently no way to treat \* or ? literally in a pattern.

**Example 1:**

```
i = wildcard_match("A?B*", "AQBXXYY")
-- i is 1 (TRUE)
```

**Example 2:**

```
i = wildcard_match("*xyz*", "AAAbbbxyz")
-- i is 1 (TRUE)
```

**Example 3:**

```
i = wildcard_match("A*B*C", "a111b222c")
-- i is 0 (FALSE) because upper/lower case doesn't match
```

**Example Program:**   [bin\search.ex](#)

**See Also:**       [wildcard\\_file](#), [match](#), [upper](#), [lower](#), [compare](#)

## wildcard\_file

**Syntax:**       include wildcard.e

i = wildcard\_file(st1, st2)

**Description:**   Return 1 (true) if the filename st2 matches the wild card pattern st1. Return 0 (false) otherwise. This is similar to DOS wildcard matching, but better in some cases. \* matches any 0 or more characters, ? matches any single character. On Linux and FreeBSD the character comparisons are case sensitive. On DOS and Windows they are not.

**Comments:**     You might use this function to check the output of the dir() routine for file names that match a pattern supplied by the user of your program.

In DOS "\*ABC.\*" will match *all* files. wildcard\_file("\*ABC.\*", s) will only match when the file name part has "ABC" at the end (as you would expect).

### Example 1:

```
i = wildcard_file("AB*CD.?", "aB123cD.e")
-- i is set to 1 on DOS or Windows, 0 on Linux or FreeBSD
```

### Example 2:

```
i = wildcard_file("AB*CD.?", "abcd.ex")
-- i is set to 0 on all systems,
-- because the file type has 2 letters not 1
```

**Example Program:**   [bin\search.ex](#)

**See Also:**       [wildcard\\_match](#), [dir](#)



# Euphoria Database System (EDS)

## Introduction

Many people have expressed an interest in accessing databases using Euphoria programs. Those people have either wanted to access a name-brand database management system from Euphoria, or they've wanted a simple, easy-to-use, Euphoria-oriented database for storing data. EDS is the latter. **It provides a simple, extremely flexible, database system for use by Euphoria programs.**

## Structure of an EDS database

In EDS, a **database** is a single file with **.edb** file type. An EDS database contains 0 or more **tables**. Each table has a **name**, and contains 0 or more **records**. Each record consists of a **key** part, and a **data** part. The key can be **any** Euphoria object - an atom, a sequence, a deeply-nested sequence, whatever. Similarly the 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 1 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.

**database.e** will work as is, on **Windows, DOS, Linux** or **FreeBSD**. **EDS database files can be copied and shared between programs running on Linux/FreeBSD and DOS/Windows.** 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's 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's easy to loop through the records looking for a particular field value.

## How to access the data

To reduce the number of parameters that you have to pass, there is a notion of the **current database**, and **current table**. 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 an argument. You can very quickly access any record, given it's number. You can access all the records by starting at record number 1 and looping through to the record number returned by [`db\_table\_size\(\)`](#).

## How does storage get recycled?

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 [`db\_compress\(\)`](#) to completely rewrite a database, removing the unused spaces.

## Security / 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 4 bytes of memory per record in the current table. So you'll need at least 4Mb 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's 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.

## Database Routines

In the descriptions below, to indicate what kind of **object** may be passed in and returned, the following prefixes are used:

- x** - a general [object](#) (atom or sequence)
- s** - a [sequence](#)
- a** - an [atom](#)
- i** - an [integer](#)
- fn** - an [integer](#) used as a file number
- st** - a [string sequence](#), or [single-character atom](#)

<a href="#">db_create</a>	- create a new database
<a href="#">db_open</a>	- open an existing database
<a href="#">db_select</a>	- select a database to be the current one
<a href="#">db_close</a>	- close a database
<a href="#">db_create_table</a>	- create a new table within a database
<a href="#">db_select_table</a>	- select a table to be the current one
<a href="#">db_rename_table</a>	- rename a table
<a href="#">db_delete_table</a>	- delete a table
<a href="#">db_table_list</a>	- get a list of all the table names in a database
<a href="#">db_table_size</a>	- get the number of records in the current table
<a href="#">db_find_key</a>	- quickly find the record with a certain key value
<a href="#">db_record_key</a>	- get the key portion of a record
<a href="#">db_record_data</a>	- get the data portion of a record
<a href="#">db_insert</a>	- insert a new record into the current table
<a href="#">db_delete_record</a>	- delete a record from the current table
<a href="#">db_replace_data</a>	- replace the data portion of a record
<a href="#">db_compress</a>	- compress a database
<a href="#">db_dump</a>	- print the contents of a database
<a href="#">db_fatal_id</a>	- handle fatal database errors

---

### db\_create

**Syntax:**       include database.e

i1 = db\_create(s, i2)

**Description:** Create a new database. A new database will be created in the file with path given by s. i2 indicates the type of lock that will be applied to the file as it is created. i1 is an error code that indicates success or failure. The values for i2 can be either DB\_LOCK\_NO (no lock) or DB\_LOCK\_EXCLUSIVE (exclusive lock). i1 is DB\_OK if the new database is successfully created. This database becomes the **current database** to which all other database operations will apply.

**Comments:**    If the path, s, does not end in .edb, it will be added automatically.

If the database already exists, it will not be overwritten. db\_create() will return DB\_EXISTS\_ALREADY.

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.

**Example:**

```
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\\_close](#)

## db\_open

**Syntax:** include database.e

i1 = db\_open(s, i2)

**Description:** Open an existing Euphoria database. The file containing the database is given by s. i1 is a return code indicating success or failure. i2 indicates the type of lock that you want to place on the database file while you have it open. This database becomes the **current database** to which all other database operations will apply.

The return codes are:

```
global constant DB_OK = 0    -- success
DB_OPEN_FAIL = -1    -- couldn't open the file
DB_LOCK_FAIL = -3    -- couldn't lock the file in the
                      -- manner requested
```

**Comments:** The types of lock that you can use are: DB\_LOCK\_NO (no lock), DB\_LOCK\_SHARED (shared lock for read-only access) and DB\_LOCK\_EXCLUSIVE (for read/write access).

DB\_LOCK\_SHARED is only supported on **Linux/FreeBSD**. It allows you to read the database, but not write anything to it. If you request DB\_LOCK\_SHARED on **WIN32** or **DOS32** 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.

DOS programs will typically get a "critical error" message if they try to access a database that is currently locked.

**Example:**

```
tries = 0
while 1 do
    err = db_open("mydata", DB_LOCK_SHARED)
    if err = DB_OK then
        exit
    elsif err = DB_LOCK_FAIL then
        tries += 1
        if tries > 10 then
            puts(2, "too many tries, giving up\n")
            abort(1)
        else
            sleep(5)
        end if
    else
        puts(2, "Couldn't open the database!\n")
    end if
end while
```

```

        abort(1)
    end if
end while

```

**See Also:**    [db\\_create](#), [db\\_close](#)

## db\_select

**Syntax:**        include database.e

i = db\_select(s)

**Description:** Choose a new, already open, database to be the **current database**. Subsequent database operations will apply to this database. s is the path of the database file as it was originally opened with db\_open() or db\_create(). i is a return code indicating success (DB\_OK) or failure.

**Comments:**    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().

**Example:**

```

    if db_select("employees") != DB_OK then
        puts(2, "Couldn't select employees database\n")
    end if

```

**See Also:**    [db\\_open](#)

## db\_close

**Syntax:**        include database.e

db\_close()

**Description:** Unlock and close the **current database**.

**Comments:**    Call this procedure when you are finished with the current database. Any lock will be removed, allowing other processes to access the database file.

**See Also:**    [db\\_open](#)

## db\_create\_table

**Syntax:**        include database.e

i = db\_create\_table(s)

**Description:** Create a new table within the **current database**. The name of the table is given by the sequence of characters, s, and may not be the same as any existing table in the current database.

**Comments:**    The table that you create will initially have 0 records. It becomes the **current table**.

**Example:**

```

    if db_create_table("my_new_table") != DB_OK then
        puts(2, "Couldn't create my_new_table!\n")
    end if

```

**See Also:**    [db\\_delete\\_table](#)

## db\_select\_table

**Syntax:** include database.e

i = db\_select\_table(s)

**Description:** The table with name given by s, becomes the **current table**. The return code, i, will be DB\_OK if the table exists in the **current database**, otherwise you'll get DB\_OPEN\_FAIL.

**Comments:** All record-level database operations apply automatically to the current table.

**Example:**

```
if db_select_table("salary") != DB_OK then
  puts(2, "Couldn't find salary table!\n")
  abort(1)
end if
```

**See Also:** [db\\_create\\_table](#), [db\\_delete\\_table](#)

## db\_rename\_table

**Syntax:** include database.e

db\_rename\_table(s1, s2)

**Description:** Rename a table in the **current database**. The current name of the table is given by s1. The new name of the table is s2.

**Comments:** The table to be renamed can be the **current table**, or some other table in the current database. An error will occur if s1 is not the name of a table in the current database, or if s2 is the name of an existing table in the current database.

**See Also:** [db\\_create\\_table](#) [db\\_select\\_table](#) [db\\_delete\\_table](#)

## db\_delete\_table

**Syntax:** include database.e

db\_delete\_table(s)

**Description:** Delete a table in the **current database**. The name of the table is given by s.

**Comments:** All records are deleted and all space used by the table is freed up. If the table is the **current table**, the **current table** becomes undefined.

If there is no table with the name given by s, then nothing happens.

**See Also:** [db\\_create\\_table](#) [db\\_select\\_table](#)

## db\_table\_list

**Syntax:** s = db\_table\_list()

**Description:** Return a sequence of all the table names in the **current database**. Each element of s is a sequence of characters containing the name of a table.

**Example:**

**sequence** names

```
names = db_table_list()
for i = 1 to length(names) do
  puts(1, names[i] & '\n')
end for
```

**See Also:**    [db\\_create\\_table](#)

## db\_table\_size

**Syntax:**        include database.e

i = db\_table\_size()

**Description:** Return the current number of records in the **current table**.

**Example:**

```
-- 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\\_select\\_table](#)

## db\_find\_key

**Syntax:**        include database.e

i = db\_find\_key(x)

**Description:** Find the record in the **current table** with key value x. If found, the record number will be returned. If not found, the record number that key would occupy, if inserted, is returned as a negative number.

**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.

You can select a range of records by searching for the first and last key values in the range. If those key values don't exist, you'll at least get a negative value showing where they would be, if they existed. e.g. Suppose you want to know which records have keys greater than "GGG" and less than "MMM". If -5 is returned for key "GGG", it means a record with "GGG" as a key would be inserted as record number 5. -27 for "MMM" means a record with "MMM" as its key would be inserted as record number 27. This quickly tells you that all records,  $\geq 5$  and  $< 27$  qualify.

**Example:**

```
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\\_record\\_key](#), [db\\_record\\_data](#), [db\\_insert](#)

## db\_record\_key

**Syntax:**        include database.e

x = db\_record\_key(i)

**Description:** Return the key portion of record number *i* in the **current table**.

**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:**

```
puts(1, "The 6th record has key value: ")
? db_record_key(6)
```

**See Also:** [db\\_record\\_data](#)

## db\_record\_data

**Syntax:** include database.e

*x* = db\_record\_data(*i*)

**Description:** Return the data portion of record number *i* in the **current table**.

**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:**

```
puts(1, "The 6th record has data value: ")
? db_record_data(6)
```

**See Also:** [db\\_record\\_key](#)

## db\_insert

**Syntax:** include database.e

*i* = db\_insert(*x1*, *x2*)

**Description:** Insert a new record into the **current table**. The record key is *x1* and the record data is *x2*. Both *x1* and *x2* can be any Euphoria data objects, atoms or sequences. The return code *i1* is DB\_OK if the record is inserted.

**Comments:** Within a table, all keys must be unique. db\_insert() will fail with DB\_EXISTS\_ALREADY if a record already exists with the same key value.

**Example:**

```
if db_insert("Smith", {"Peter", 100, 34.5}) != DB_OK then
    puts(2, "insert failed!\n")
end if
```

**See Also:** [db\\_find\\_key](#), [db\\_record\\_key](#), [db\\_record\\_data](#)

## db\_delete\_record

**Syntax:** include database.e

db\_delete\_record(*i*)

**Description:** Delete record number *i* from the **current table**.

**Comments:** The record number, *i*, must be an integer from 1 to the number of records in the current table.

**Example:**

```
db_delete_record(55)
```

**See Also:** [db\\_insert](#), [db\\_table\\_size](#)



## db\_replace\_data

**Syntax:** include database.e

db\_replace\_data(i, x)

**Description:** In the **current table**, replace the data portion of record number i, with x. x can be any Euphoria atom or sequence.

**Comments:** The record number, i, must be from 1 to the number of records in the current table.

**Example:**

```
db_replace_data(67, {"Peter", 150, 34.5})
```

**See Also:** [db\\_delete\\_record](#)

## db\_compress

**Syntax:** include database.e

i = db\_compress()

**Description:** Compress the **current database**. The current database is copied to a new file such that any blocks of unused space are eliminated. If successful, i 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.

**Comments:** 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:**

```
if db_compress() != DB_OK then
    puts(2, "compress failed!\n")
end if
```

**See Also:** [db\\_create](#)

## db\_dump

**Syntax:** include database.e

db\_dump(fn, i)

**Description:** Print the contents of an already-open Euphoria database. The contents are printed to file or device fn. All records in all tables are shown. If i 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:**

```
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, 0)`

**See Also:**    [db\\_open](#)

## **db\_fatal\_id**

**Syntax:**        `include database.e`

`db_fatal_id = i`

**Description:** You can catch certain fatal database errors by installing your own fatal error handler. Simply overwrite the global variable `db_fatal_id` with the routine id of one of your own procedures. The procedure must take a single argument which is a sequence. When certain errors occur your procedure will be called with an error message string as the argument. Your procedure should end by calling `abort()`.

**Example:**

```
procedure my_fatal(sequence msg)
  puts(2, "A fatal error occurred - " & msg & '\n')
  abort(1)
end procedure
```

```
db_fatal_id = routine_id("my_fatal")
```

**See Also:**    [db\\_close](#)

## db\_create

**Syntax:**           include database.e

i1 = db\_create(s, i2)

**Description:**   Create a new database. A new database will be created in the file with path given by s. i2 indicates the type of lock that will be applied to the file as it is created. i1 is an error code that indicates success or failure. The values for i2 can be either DB\_LOCK\_NO (no lock) or DB\_LOCK\_EXCLUSIVE (exclusive lock). i1 is DB\_OK if the new database is successfully created. This database becomes the **current database** to which all other database operations will apply.

**Comments:**       If the path, s, does not end in .edb, it will be added automatically.

If the database already exists, it will not be overwritten. db\_create() will return DB\_EXISTS\_ALREADY.

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.

**Example:**

```
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\\_close](#)

## db\_close

**Syntax:** include database.e

db\_close()

**Description:** Unlock and close the **current database**.

**Comments:** Call this procedure when you are finished with the current database. Any lock will be removed, allowing other processes to access the database file.

**See Also:** [db\\_open](#)

## db\_create\_table

**Syntax:** include database.e

i = db\_create\_table(s)

**Description:** Create a new table within the **current database**. The name of the table is given by the sequence of characters, s, and may not be the same as any existing table in the current database.

**Comments:** The table that you create will initially have 0 records. It becomes the **current table**.

**Example:**

```
if db_create_table("my_new_table") != DB_OK then
  puts(2, "Couldn't create my_new_table!\n")
end if
```

**See Also:** [db\\_delete\\_table](#)

## db\_compress

**Syntax:** include database.e

i = db\_compress()

**Description:** Compress the **current database**. The current database is copied to a new file such that any blocks of unused space are eliminated. If successful, i 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.

**Comments:** 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:**

```
if db_compress() != DB_OK then
  puts(2, "compress failed!\n")
end if
```

**See Also:** [db\\_create](#)

---> **Is this the latest version of Euphoria?**  
Visit: <http://www.RapidEuphoria.com>

---> To install/uninstall Euphoria, see [install.htm](#)

---> **What's new in this release?**  
See [relnotes.doc](#)

## Euphoria Programming Language version 3.0.1 November 3, 2006

### Welcome to Euphoria! ... **End User Programming with Hierarchical Objects for Robust Interpreted Applications**

Euphoria has come a long way since v1.0 was released in July 1993. There are now thousands of users around the world. There's an automated [discussion forum](#), managed and moderated by a Euphoria program, and supporting over 500 subscribers. The [Euphoria Web site](#) contains over 1600 contributed .zip files packed with Euphoria source programs and library routines. Dozens of people have set up their own independent Web pages with Euphoria-related content. Euphoria has been used in a variety of **commercial programs**. The **Windows** version has been used to create numerous **GUI, utility and Internet-related programs**. The **DOS** version has been used to create many exciting **high-speed action games**, complete with **Sound Blaster** sound effects. The **Linux** and **FreeBSD** versions have been used to write **X Windows GUI programs, Web-based (CGI) programs**, and lots of useful tools and utilities.

### Yet Another Programming Language?

Euphoria is a very-high-level programming language with several features that set it apart from the crowd:

- Euphoria programs run on **Windows, DOS, Linux, and FreeBSD**.
- Euphoria is **free** and **open source**. The complete source code for the Euphoria interpreter, translator and binder is included in the download package.
- The language is flexible, powerful, and easy to learn.
- There is no waiting for compiles and links - just edit and run.
- You can create and distribute a royalty-free, **stand-alone .exe** file.
- Dynamic storage allocation** is fundamental to Euphoria. Variables grow or shrink in size without the programmer having to worry about allocating and freeing chunks of memory. Elements of an array (Euphoria sequence) can be a dynamic mixture of different types and sizes of data.
- Euphoria provides **extensive run-time error checking** for: out-of-bounds subscripts, uninitialized variables, bad parameter values for library routines, illegal value assigned to a variable, and many

more. If something goes wrong you'll get a full error message, with a call traceback and a listing of variable values. With other languages you'll typically get protection faults with useless dumps of machine registers and addresses.

- The Euphoria interpreter is more than **30 times faster** than either Perl or Python, and it's considerably faster than all other interpreted languages, according to the "Great Computer Language Shootout" benchmark (see demo\bench\bench.doc).
- If that isn't enough, there's a **Euphoria To C Translator** that can translate any Euphoria program to C, and boost your speed even more. Why waste time debugging hand-coded C/C++? You can easily develop a Euphoria program, and then generate the C code.
- Euphoria programs are not constrained by any 640K memory restrictions for which MS-DOS is infamous. All versions of Euphoria let you use all the memory on your system, and if that isn't enough, a swap file on disk will provide additional virtual memory.
- An integrated, easy-to-use, **full-screen source-level debugger/tracer** is included.
- Both an **execution-count profiler**, and a **time profiler** are available.
- There is a large and rapidly growing collection of excellent 3rd party programs and libraries, most with full source code.
- RDS has developed an extremely flexible database system (**EDS**) that is portable across all Euphoria platforms.
- The **WIN32** implementation of Euphoria can access any WIN32 API routine, as well as C or Euphoria routines in .DLL files. A team of people has developed a Windows GUI library (**Win32Lib**), complete with a powerful Interactive Development Environment. You can design a user interface graphically, specify the Euphoria statements to be executed when someone clicks, and the IDE will create a complete Euphoria program for you. There are Windows Euphoria libraries for Internet access, 3-D games, and many other application areas.
- The **DOS32** implementation of Euphoria on MS-DOS contains a built-in graphics library. If necessary, you can access DOS software interrupts. You can call machine-code routines. You can even set up your own hardware interrupt handlers. Many high-speed action games, complete with Sound Blaster sound effects, have been developed 100% in Euphoria, without the need for any machine code.
- The **Linux** and **FreeBSD** implementations of Euphoria let you access C routines and variables in shared libraries, for tasks ranging from graphics, to X windows GUI programming, to Internet CGI programming. The good news is, you'll be programming in Euphoria, not C.

## Platforms and Products

Euphoria runs on four different platforms, **WIN32**, **DOS32**, **Linux**, and **FreeBSD**.

This **Euphoria Interpreter, Translator and Binder** package is free for anyone to use.

Using the Euphoria **Binder** you can **shroud** (encrypt) and **bind** any Euphoria program with a copy of the



interpreter back-end, to create a **single, stand-alone, tamper-resistant .exe** file for easy distribution. See [bind.doc](#))

The **Euphoria To C Translator** converts any Euphoria program into a stand-alone .exe file, but it has the added advantage of boosting the program's speed as well. To use it, you must have one of 7 free C compilers installed on your machine, but no knowledge of C is required.

The documentation contained in this package comes in both plain text and HTML form. The plain text (**.doc**) files can be viewed with any text editor, such as Windows NotePad or WordPad. The HTML (**.htm**) files can be viewed with your Web browser. A tool that we developed in Euphoria allows us to automatically generate both plain text and HTML files, from a common source. Thus the content of each file in the **doc** subdirectory should be identical to the content of the corresponding file in the **html** subdirectory, aside from the lack of links, fonts, colors, etc. See [doc\overview.doc](#) (or [html\overview.htm](#)) for a summary of the documentation files.

You can freely distribute the Euphoria interpreter, and any other files contained in this package, in whole or in part, so anyone can run a Euphoria program that you have developed. You are completely free to distribute any Euphoria programs that you write.

To run the **WIN32** version of Euphoria, you need Windows 95 or any later version of Windows. It runs fine on XP.

The **DOS32** version will run on any version of Windows, and will also run on plain DOS on any 386 or higher processor. Contrary to popular opinion, DOS is not dead. You can run DOS Euphoria programs on Windows XP in a command prompt window.

To run the **Linux** version of Euphoria you need any reasonably up-to-date Linux distribution, that has libc6 or later. For example, Red Hat 5.2 or later will work fine.

To run the **FreeBSD** version of Euphoria you need any reasonably up-to-date FreeBSD distribution.

## Getting Started

**0.** The Euphoria interpreter is an engine for running Euphoria programs. It does not have a fancy GUI interface. When you are ready to do Windows GUI programming, you should download Judith Evans' IDE (written in open source Euphoria code). It will provide you with a very nice graphical environment for Windows programming. For most other programming, all you really need is an editor, such as NotePad.

**1.** After you install Euphoria, the documentation files will be in the **doc** and **html** directories. [overview.doc](#) gives an overview of the documentation. [refman.htm](#) (or [refman.doc](#)) should be read first. If you want to search for information on any topic, type **guru**.

**2.** Have fun running the programs in the **demo** directory. Feel free to modify them, or run them in **trace** mode by adding:

```
with trace
trace(1)
```

as the first two lines in the **.ex** or **.exw** file.

**3.** Try typing in some simple statements and running them. You can use any text editor. Later you may want to use the Euphoria editor, **ed**, or download David Cuny's Euphoria editor from the [Euphoria Web site](#).

Don't be afraid to try things. Euphoria won't bite!

4. See [what2do.doc](#) for more ideas.

5. Visit the Euphoria Web site, download some files, and subscribe to the Euphoria **mailing list**.

If you are new to programming, and you find [refman.htm](#) hard to follow, download **David Gay**'s interactive tutorial called "*A Beginner's Guide To Euphoria*". It's in the Documentation section of our [Archive](#).

---

If you have any trouble installing, see [install.doc](#)

---

**Notice to Shareware Vendors:**

We encourage you to distribute this Euphoria Interpreter package. You can charge whatever you like for it. People can use Euphoria for as long as they like without obligation.

---

**DISCLAIMER:**

Euphoria is provided "as is" without warranty of any kind. In no event shall Rapid Deployment Software be held liable for any damages arising from the use of, or inability to use, this product.

---

## Guide to Documentation Files

The Euphoria documentation is spread over several plain text files in the **euphoria\doc** directory. For each **.doc** file in the **euphoria\doc** directory, there is a corresponding **.htm** file in the **euphoria\html** directory. You can also type **guru** to quickly search for relevant information from all directories under **\euphoria**.

Everyone should start with **euphoria\readme.doc** (**euphoria\readme.htm**).

If you are having trouble installing Euphoria read:

**install.doc** - Instructions for installing Euphoria

Read the following files first, if you want to learn to program in Euphoria:

**refman.doc** - The manual for the Euphoria core language

**library.doc** - Documentation for all of the Euphoria library routines

**what2do.doc** - Some things you can do to get started with Euphoria

**platform.doc** - A discussion of the differences between the four **platforms** that Euphoria runs on, i.e.

**DOS32 (ex.exe)** vs. **WIN32 (exw.exe)** vs. **Linux (exu)** vs. **FreeBSD (exu)**. Also included is a discussion of how to interface Euphoria programs with C libraries.

The rest of the files can be read any time, as needed:

**database.doc** - The Euphoria Database System (EDS)

**e2c.doc** - The Euphoria To C Translator

**tasking.doc** - Multitasking in Euphoria

**ed.doc** - Documentation for the standard Euphoria editor, **ed**

**trouble.doc** - A list of the most common problems and their solutions

**bind.doc** - Describes **bind.bat** and **shroud.bat** options for **binding** and **shrouding**

**perform.doc** - Performance tips

**relnotes.doc** - A description of what's new in the current release and a summary of previous releases

of Euphoria

**cgi.doc** - Web-based Programming using CGI and Euphoria

**c.doc** - Propaganda aimed at C/C++ programmers

Most subdirectories under **\euphoria** have a **.doc** file that describes the files in that subdirectory.

# How to Install Euphoria on Windows

Download and run the latest Euphoria setup program. It will install Euphoria on any Windows system from Windows 95 up.

After installing, see [doc\what2do.doc](#) (or [html\what2do.htm](http://html\what2do.htm)) for ideas on how to use this package. You should also read [readme.doc](#) (or [readme.htm](#)) if you haven't done so already.

## Possible problems ...

- If the install appeared to run ok, but example programs are not working, did you remember to shut down and restart your computer?
- 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 can be run under DOS on older Win 3.1 systems, but we no longer maintain an install program for this. You'll have to install Euphoria on a newer system and then copy your EUPHORIA directory to the old system. Set up AUTOEXEC.BAT manually on the old system as described below.

## How to manually edit autoexec.bat (WinME/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.

On WinNT/2000/XP and higher, if for some reason your **EUDIR** and **PATH** variables are not set correctly, then set them in whatever way your system allows. For example 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.

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.

If you have an **autoexec.bat** file, but it doesn't contain a PATH command, you will have to create one that includes **C:\EUPHORIA\BIN**.

### **How to Uninstall Euphoria**

1. If you wish to recover your previous version of Euphoria, or parts of it, the "backup" subdirectory contains a backup copy of each of your previous standard Euphoria subdirectories, plus any files that you may have added. However it does not contain any additional subdirectories that you may have created on your own.
2. If there are no files that you need, you can delete the EUPHORIA directory that you installed into.
3. Delete the EUDIR environment variable, and remove the EUPHORIA directory from your PATH, either in C:\AUTOEXEC.BAT, or in Control Panel/System/Advanced.
4. Delete the references to EUPHORIA in your Start Menu.

# What to Do?

Now that you have installed Euphoria, here are some things you can try:

- Run each of the demo programs in the **demo** directory. You just type **ex** or **exw** or **exu** followed by the name of the **.ex** or **.exw** or **.exu** file, e.g.
  - ```
ex buzz
```

will run the file **buzz.ex**. Depending on your graphics card you may have to edit a line in some of the **.ex** files to select a different graphics mode. Some demos try to use **SVGA** modes, which might not work with your video card. You need DOS mouse support to run **mouse.ex** and **tft.ex**.  
You can also double-click on a **.ex** (**.exw**) file from **Windows**, but you will have to "associate" **.ex** files with **ex.exe** and **.exw** files with **exw.exe**. A few of the demos are meant to be run from the command line, but most will look ok from Windows.
- Use the Euphoria editor, **ed**, to edit a Euphoria file. Notice the use of colors. **You can adjust these colors along with the cursor size and many other "user-modifiable" parameters by editing constant declarations in ed.ex.** Use **Esc q** to quit the editor or **Esc h** for help. There are several, even better, Euphoria-oriented editors in the Archive.
- Create some new benchmark tests. See **demo\bench**. Do you get the same speed ratios as we did in comparison with other popular languages?
- Read the manual in **doc\refman.doc** or [view the HTML version of the manual](#) by double-clicking it and starting your Web browser. The simple expressive power of Euphoria makes this manual much shorter than manuals for other languages. If you have a specific question, type **guru** followed by a list of words. The **guru** program will search all the **.doc** files as well as all the example programs and other files, and will present you with a *sorted* list of the most relevant chunks of text that might answer your enquiry.
- Try running a Euphoria program with **tracing** turned on. Add:
  - ```
with trace
```
  - ```
trace(1)
```at the beginning of any **.ex** or **.exw** file.
- Run some of the tutorial programs in **euphoria\tutorial**.
- Try modifying some of the demo programs.

First some **simple** modifications (takes less than a minute):

What if there were 100 C++ ships in **Language Wars**? What if **sb.ex** had to move 1000 balls instead of 125? Change some parameters in **polygon.ex**. Can you get prettier pictures to appear? Add some funny phrases to **buzz.ex**.

Then, some **slightly harder** ones (takes a few minutes):

Define a new function of x and y in [plot3d.ex](#).

Then a *challenging* one (takes an hour or more):

Set up your own customized database by defining the fields in [mydata.ex](#).

Then a *major* project (several days or weeks):

Write a *smarter* 3D TicTacToe algorithm.

- Try writing your own program in Euphoria. A program can be as simple as:

- `? 2+2`

**Remember that after any error you can simply type `ed` to jump into the editor at the offending file and line.**

Once you get used to it, you'll be developing programs *much* faster in Euphoria than you could in Perl, Java, C/C++ or any other language that we are aware of.

# Platform-Specific Issues for Euphoria

## 1. Introduction

Rapid Deployment Software currently supports Euphoria on four different *platforms*. More platforms will be added in the future.

The first platform is called **DOS32**, since it depends on the DOS operating system, but with the CPU operating in 32-bit (protected) mode.

The second platform is called **WIN32**, since the underlying operating system is Microsoft Windows, in particular, the 32-bit version of Windows that is used on Windows 95/98/ME, as well as NT/2000/XP and later systems.

The third platform is **Linux**. Linux is based on the UNIX operating system. It has recently become very popular on PCs. There are many distributors of Linux, including Red Hat, Debian, Caldera, etc. Linux can be obtained on a CD for a very low price. Linux is an open-source operating system.

The fourth platform is **FreeBSD**. FreeBSD is also based on the UNIX operating system. It is very popular on Internet server machines. It's also open source.

Users who have purchased the Euphoria source code have ported Euphoria to other platforms, such as HP Unix and Sun Unix.

The Euphoria for DOS32+WIN32 installation file contains two **.exe** files. The first is called **ex.exe**. It runs Euphoria programs on the DOS32 platform. The second is **exw.exe**. It runs Euphoria programs on the WIN32 platform. Euphoria programs that are meant to be run on the WIN32 platform have a **.exw** file type, while programs that are meant to be run on the DOS32 platform have a **.ex** file type.

The Euphoria for Linux .tar file contains only **exu**. It runs Euphoria programs on the Linux platform. Euphoria programs intended for Linux or FreeBSD have a **.exu** file type.

The FreeBSD version of Euphoria is installed by first installing the Linux version of Euphoria, and then replacing **exu**, by the version of **exu** for FreeBSD.

Many Euphoria programs can be run on two, three, or all four platforms without change. The file type should indicate the preferred platform for the program. Any Euphoria interpreter can try to run any Euphoria file, but you might have to specify the full name of the file, including the type, since each interpreter looks for its own preferred file type (**.ex**, **.exw** or **.exu**).

Sometimes you'll find that the majority of your code will be the same on all platforms, but some small parts will have to be written differently for each platform. Use the [platform\(\)](#) built-in function to tell you which platform you are currently running on. Note that `platform()` returns the same value (3) on both Linux and FreeBSD, since those systems are so similar.

## 2. The DOS32 Platform



If you are new to programming, you might want to start with **ex.exe** on the DOS32 platform. You should try to understand the essential core of Euphoria, before you leap into Windows GUI programming. All versions of Windows (even XP), let you open a Command Prompt text window, and run DOS programs.

**Euphoria programs run in 32-bit (protected) mode and have access to all of the megabytes of memory on the machine.** Many programming languages for DOS limit you to 16-bit **real** mode. This makes it impossible to access more than 640K of memory at one time. Your machine might have 256Mb of memory, but your program will run out of memory after using just 640K. QBasic is even worse. It limits you to just 160K.

DOS32 programs can flip the screen into either **text mode** or **pixel graphics mode**, and Euphoria provides you with library routines for both modes. There is rarely any need to call DOS directly, but you can do this using the [dos\\_interrupt\(\)](#) routine. You can also [peek](#) and [poke](#) into special memory locations to achieve high-speed graphics and get access to low-level details of the system.

Under **DOS32 for Windows 95 and later systems**, Euphoria files can have long filenames, and programs can open long filename files for reading and writing, but not for creating a new file.

Under **pure DOS, outside of Windows**, there is no system [swap file](#) so the DOS-extender built in to **ex.exe** will create one for possible use by your program. This file is created when your Euphoria program starts up under DOS, and is deleted when your program terminates. It starts as a 0-byte file and grows only if actual swapping is needed. It is created in the directory on your hard disk pointed to by the TEMP or TMP environment variable. If neither of these variables have been set, it is created in the directory containing either **ex.exe** or your **bound** Euphoria **.exe** file. You can force it to be created in a particular directory by setting the CAUSEWAY environment variable as follows:

```
SET CAUSEWAY=SWAP:path
```

where **path** is the full path to the directory. You can prevent the creation of a DOS swap file with:

```
SET CAUSEWAY=NOVM
```

When disk swapping activity occurs, your program will run correctly but will slow down. A better approach might be to free up more extended memory by cutting back on SMARTDRV and other programs that reserve large amounts of extended memory for themselves.

When your free disk space is less than the amount of RAM in your machine, no swap file will be created.

### 3. The WIN32 Platform

Euphoria for WIN32 (**exw.exe**) has a lot in common with Euphoria for DOS32. With WIN32 you also have access to all of the memory on your machine. Most library routines work the same way on each platform. Many existing DOS32 **text mode** programs can be run using **exw** without any change. With **exw** you can run programs from the command line, and display text on a standard (typically 25 line x 80 column) DOS window. The DOS window is known as the **console** in Windows terminology. Euphoria makes the transition from DOS32 **text mode** programming, to WIN32 console programming, trivial. **You can add calls to WIN32 C functions and later, if desired, you can create real Windows GUI windows.**

A console window will be created automatically when a WIN32 Euphoria program first outputs something to the screen or reads from the keyboard. You will also see a console window when you read standard input or write to standard output, even when these have been redirected to files. The console will disappear when

your program finishes execution, or via a call to [free\\_console\(\)](#). If there is something on the console that you want your user to read, you should prompt him and wait for his input before terminating. To prevent the console from quickly disappearing you might include a statement such as:

```
if getc(0) then
end if
```

which will wait for the user enter something.

If you want to run Euphoria programs without popping up a new console window use `exwc.exe`. It uses the current console window, just like a DOS program would when using `ex.exe`.

Under WIN32, long filenames are fully supported for reading and writing and creating.

### 3.1 High-Level WIN32 Programming

Thanks to **David Cuny**, **Derek Parnell**, **Judith Evans** and many others, there's a package called **Win32Lib** that you can use to develop Windows GUI applications in Euphoria. It's remarkably easy to learn and use, and comes with good documentation and many small example programs. You can download Win32Lib and Judith's IDE from the Euphoria [Web site](#). Recently, **Andrea Cini** has developed a similar, somewhat smaller package called **EuWinGUI**. It's also available from our site.

### 3.2 Low-Level WIN32 Programming

**To allow access to WIN32 at a lower level, Euphoria provides a mechanism for calling any C function in any WIN32 API .dll file, or indeed in any 32-bit Windows .dll file that you create or someone else creates. There is also a call-back mechanism that lets Windows call your Euphoria routines. Call-backs are necessary when you create a graphical user interface.**

To make full use of the WIN32 platform, you need documentation on 32-bit Windows programming, in particular the WIN32 Application Program Interface (API), including the C structures defined by the API. There is a large WIN32.HLP file (c) Microsoft that is available with many programming tools for Windows. There are numerous books available on the subject of WIN32 programming for C/C++. You can adapt most of what you find in those books to the world of Euphoria programming for WIN32. A good book is:

*Programming Windows*  
by Charles Petzold  
Microsoft Press

A WIN32 API Windows help file (8 Mb) can be downloaded from Borland's Web site:

<ftp://ftp.borland.com/pub/delphi/techpubs/delphi2/win32.zip>

See also the Euphoria [Archive Web page](#) - "[documentation](#)".

## 4. The Linux and FreeBSD Platforms

Euphoria for Linux, and Euphoria for FreeBSD share certain features with Euphoria for DOS32, and share other features with Euphoria for WIN32.

As with WIN32 and DOS32, you can write text on a console, or xterm window, in multiple colors and at any line or column position.

Just as in WIN32, you can call C routines in shared libraries and C code can call back to your Euphoria routines.

Euphoria for Linux and FreeBSD do not have integrated support for pixel graphics like DOS32, but Pete Eberlein has created a Euphoria interface to **svgalib**.

Easy X windows GUI programming is available using either Irv Mullin's EuGTK interface to the GTK GUI library, or wxEuphoria developed by Matt Lewis. wxEuphoria also runs on Windows.

When porting code from DOS or Windows to Linux or FreeBSD, you'll notice the following differences:

- Some of the numbers assigned to the 16 main colors in graphics.e are different. If you use the constants defined in graphics.e you won't have a problem. If you hard-code your color numbers you will see that blue and red have been switched etc.
- The key codes for special keys such as Home, End, arrow keys are different, and there are some additional differences when you run under XTERM.
- The Enter key is code 10 (line-feed) on Linux, where on DOS/Windows it was 13 (carriage-return).
- Linux and FreeBSD use '/' (slash) on file paths. DOS/Windows uses '\\' (backslash).
- Highly specialized things such as dos\_interrupt() obviously won't work on Linux or FreeBSD.
- Calls to system() and system\_exec() that contain DOS commands will obviously have to be changed to the corresponding Linux or FreeBSD command. e.g. "DEL" becomes "rm", and "MOVE" becomes "mv".

## 5. Interfacing with C Code (WIN32, Linux, FreeBSD)

On WIN32, Linux and FreeBSD it's possible to interface Euphoria code with C code. Your Euphoria program can call C routines and read and write C variables. C routines can even call ("callback") your Euphoria routines. The C code must reside in a WIN32 dynamic link library (.dll file), or a Linux or FreeBSD shared library (.so file). By interfacing with .dll's and shared libraries, you can access the full programming interface on these systems.

Using the Euphoria to C Translator, you can translate Euphoria routines to C, and compile them into a .dll or .so file. You can pass Euphoria atoms and sequences to these compiled Euphoria routines, and receive Euphoria data as a result. Translated/compiled routines typically run much faster than interpreted routines. For more information see [the Translator](#).

### 5.1 Calling C Functions

To call a C function in a .dll or .so file you must perform the following steps:

1. Open the .dll or .so file that contains the C function by calling [open\\_dll\(\)](#) contained in

[euphoria\include\dll.e](#).

2. Define the C function, by calling [define\\_c\\_func\(\)](#) or [define\\_c\\_proc\(\)](#) in [dll.e](#). This tells Euphoria the number and type of the arguments as well as the type of value returned.

Euphoria currently supports all C integer and pointer types as arguments and return values. It also supports floating-point arguments and return values (C double type). It is currently not possible to pass C structures by value or receive a structure as a function result, although you can certainly pass a pointer to a structure and get a pointer to a structure as a return value. Passing C structures by value is rarely required for operating system calls.

Euphoria also supports all forms of Euphoria data - atoms and arbitrarily-complex sequences, as arguments to translated/compiled Euphoria routines.

3. Call the C function by calling [c\\_func\(\)](#) or [c\\_proc\(\)](#).

### Example:

```
include dll.e

atom user32
integer LoadIcon, icon

user32 = open_dll("user32.dll")

-- The name of the routine in user32.dll is "LoadIconA".
-- It takes a pointer and an int as arguments,
-- and it returns an int.
LoadIcon = define_c_func(user32, "LoadIconA",
                        {C_POINTER, C_INT}, C_INT)

icon = c_func(LoadIcon, {NULL, IDI_APPLICATION})
```

See [library.doc - Calling C Functions](#) for descriptions of [c\\_func\(\)](#), [c\\_proc\(\)](#), [define\\_c\\_func\(\)](#), [define\\_c\\_proc\(\)](#), [open\\_dll\(\)](#) etc. See [demo\win32](#) or [demo\linux](#) for example programs.

On Windows there is more than one C calling convention. The Windows API routines all use the `__stdcall` convention. Most C compilers however have `__cdecl` as their default. `__cdecl` allows for variable numbers of arguments to be passed. Euphoria assumes `__stdcall`, but if you need to call a C routine that uses `__cdecl`, you can put a '+' sign at the start of the routine name in [define\\_c\\_proc\(\)](#) and [define\\_c\\_func\(\)](#). In the example above, you would have `"+LoadIconA"`, instead of `"LoadIconA"`.

You can examine a .dll file by right-clicking on it, and choosing "QuickView" (if it's on your system). You will see a list of all the C routines that the .dll exports.

To find out which .dll file contains a particular WIN32 C function, run [euphoria\demo\win32\dsearch.exw](#).

## 5.2 Accessing C Variables

You can get the address of a C variable using [define\\_c\\_var\(\)](#). You can then use [poke\(\)](#) and [peek\(\)](#) to access the value of the variable.

## 5.3 Accessing C Structures

Many C routines require that you pass pointers to structures. You can simulate C structures using allocated blocks of memory. The address returned by `allocate()` can be passed as if it were a C pointer.

You can read and write members of C structures using `peek()` and `poke()`, or `peek4u()`, `peek4s()`, and `poke4()`. You can allocate space for structures using `allocate()`. You must calculate the offset of a member of a C structure. This is usually easy, because anything in C that needs 4 bytes will be assigned 4 bytes in the structure. Thus C int's, char's, unsigned int's, pointers to anything, etc. will all take 4 bytes. If the C declaration looks like:

```
// Warning C code ahead!

struct example {
    int a;           // offset 0
    char *b;         // offset 4
    char c;          // offset 8
    long d;          // offset 12
};
```

To allocate space for "struct example" you would need:

```
atom p

p = allocate(16) -- size of "struct example"
```

The address that you get from `allocate()` is always at least 4-byte aligned. This is useful, since WIN32 structures are supposed to start on a 4-byte boundary. Fields within a C structure that are 4-bytes or more in size must start on a 4-byte boundary in memory. 2-byte fields must start on a 2-byte boundary. To achieve this you may have to leave small gaps within the structure. In practice it is not hard to align most structures since 90% of the fields are 4-byte pointers or 4-byte integers.

You can set the fields using something like:

```
poke4(p + 0, a)
poke4(p + 4, b)
poke4(p + 8, c)
poke4(p +12, d)
```

You can read a field with something like:

```
d = peek4(p+12)
```

### Tip:

For readability, make up Euphoria constants for the field offsets. See Example below.

### Example:

```
constant RECT_LEFT = 0,
         RECT_TOP   = 4,
         RECT_RIGHT = 8,
         RECT_BOTTOM = 12
```

```
atom rect
rect = allocate(16)

poke4(rect + RECT_LEFT, 10)
```

```
poke4(rect + RECT_TOP, 20)
poke4(rect + RECT_RIGHT, 90)
poke4(rect + RECT_BOTTOM, 100)

-- pass rect as a pointer to a C structure
-- hWnd is a "handle" to the window
if not c_func(InvalidateRect, {hWnd, rect, 1}) then
    puts(2, "InvalidateRect failed\n")
end if
```

The Euphoria code that accesses C routines and data structures may look a bit ugly, but it will typically form just a small part of your program, especially if you use Win32Lib, EuWinGUI, or Irv Mullin's X Windows library. Most of your program will be written in pure Euphoria, which will give you a big advantage over those forced to code in C.

## 5.4 Call-backs to your Euphoria routines

When you create a window, the Windows operating system will need to call your Euphoria routine. This is a strange concept for DOS programmers who are used to calling operating system routines, but are not used to having the operating system call *their* routine. To set this up, you must get a 32-bit "call-back" address for your routine and give it to Windows. For example (taken from [demo\win32\window.exw](#)):

```
integer id
atom WndProcAddress

id = routine_id("WndProc")

WndProcAddress = call_back(id)
```

[routine\\_id\(\)](#) uniquely identifies a Euphoria procedure or function by returning a small integer value. This value can be used later to call the routine. You can also use it as an argument to the [call\\_back\(\)](#) function.

In the example above, The 32-bit *call-back address*, WndProcAddress, can be stored in a C structure and passed to Windows via the RegisterClass() C API function. **This gives Windows the ability to call the Euphoria routine, WndProc(), whenever the user performs an action on a certain class of window.** Actions include clicking the mouse, typing a key, resizing the window etc. See the [window.exw](#) demo program for the whole story.

### Note:

It is possible to get a *call-back address* for *any* Euphoria routine that meets the following conditions:

- the routine must be a function, not a procedure
- it must have from 0 to 9 parameters
- the parameters should all be of type atom (or integer etc.), not sequence
- the return value should be an integer value up to 32-bits in size

You can create as many call-back addresses as you like, but you should not call [call\\_back\(\)](#) for the same Euphoria routine multiple times - each call-back address that you create requires a small block of memory.

The values that are passed to your Euphoria routine can be any 32-bit **unsigned** atoms, i.e. non-negative. Your routine could choose to interpret large positive numbers as negative if that is desirable. For instance, if a

C routine tried to pass you -1, it would appear as hex FFFFFFFF. If a value is passed that does not fit the type you have chosen for a given parameter, a Euphoria type-check error may occur (depending on with/without type\_check etc.) No error will occur if you declare all parameters as **atom**.

Normally, as in the case of WndProc() above, Windows initiates these call-backs to your routines. **It is also possible for a C routine in any .dll to call one of your Euphoria routines.** You just have to declare the C routine properly, and pass it the call-back address.

Here's an example of a WATCOM C routine that takes your call-back address as its only parameter, and then calls your 3-parameter Euphoria routine:

```
/* 1-parameter C routine that you call from Euphoria */
unsigned EXPORT APIENTRY test1(
    LRESULT CALLBACK (*eu_callback)(unsigned a,
                                    unsigned b,
                                    unsigned c))
{
    /* Your 3-parameter Euphoria routine is called here
       via eu_callback pointer */
    return (*eu_callback)(111, 222, 333);
}
```

The C declaration above declares test1 as an externally-callable C routine that takes a single parameter. The single parameter is a pointer to a routine that takes 3 unsigned parameters - i.e. your Euphoria routine.

In WATCOM C, "CALLBACK" is the same as "\_\_stdcall". This is the calling convention that's used to call WIN32 API routines, and the C pointer to your Euphoria routine should be declared this way too, or you'll get an error when your Euphoria routine tries to return to your .DLL.

If you need your Euphoria routine to be called using the \_\_cdecl convention, you must code the call to call\_back() as:

```
myroutineaddr = call_back({'+', id})
```

The plus sign and braces indicate the \_\_cdecl convention. The simple case, with no braces, is \_\_stdcall.

In the example above, your Euphoria routine will be passed the three values 111, 222 and 333 as arguments. Your routine will return a value to test1. That value will then be immediately returned to the caller of test1 (which could be at some other place in your Euphoria program).

A call-back address can be passed to the Linux or FreeBSD signal() function to specify a Euphoria routine to handle various signals (e.g. SIGTERM). It can also be passed to C routines such as qsort(), to specify a Euphoria comparison function.



# The Euphoria Editor

## Introduction

The Euphoria download package includes a handy, text-mode editor, **ed**, that's written completely in Euphoria. Many people find **ed** convenient for editing Euphoria programs and other files, but there is no requirement that you use it.

If you don't like **ed**, you have many alternatives. David Cuny's **EE editor** is a DOS-based editor for Euphoria that's also written in Euphoria. It has a friendly mouse-based user interface with drop down menus etc. It's available from the RDS Web site. There are several other Euphoria-oriented editors that run on DOS, Windows, Linux and FreeBSD. Check the **Editors** section of our Archive. In fact, *any* text editor can be used to edit a Euphoria program, including DOS Edit or Windows NotePad.

## Summary

**usage 1: ed filename**

**usage 2: ed**

After any error, just type **ed**, and you'll be placed in the editor, at the line and column where the error was detected. The error message will be at the top of your screen.

Euphoria-related files are displayed in color. Other text files are in mono. You'll know that you have misspelled something when the color does not change as you expect. Keywords are blue. Names of routines that are built in to the interpreter appear in magenta. Strings are green, comments are red, most other text is black. Balanced brackets (on the same line) have the same color. **You can change these colors as well as several other parameters of ed.** See "user-modifiable parameters" near the top of **ed.ex**.

The arrow keys move the cursor left, right, up or down. Most other characters are immediately inserted into the file.

In Windows, you can "associate" various types of files with **ed.bat**. You will then be put into **ed** when you **double-click** on these types of files - e.g. **.e**, **.pro**, **.doc** etc. Main Euphoria files ending in **.ex** (**.exw**) might better be associated with **ex.exe** (**exw.exe**).

**ed** is a **multi-file/multi-window** DOS editor. **Esc c** will split your screen so you can view and edit up to 10 files simultaneously, with cutting and pasting between them. You can also use multiple edit windows to view and edit different parts of a single file.

## Special Keys

Some PC keys do not work in a Linux or FreeBSD text console, or in Telnet, and some keys do not work in an xterm under X windows. Alternate keys have been provided. In some cases on Linux/FreeBSD you might have to edit **ed.ex** to map the desired key to the desired function.

- Delete** - Delete the current character above the cursor.
- Backspace** - Move the cursor to the left and delete a character.



**control-Delete** - Delete the current line. (**control-Delete** is not available on all systems.)

**control-d** - Delete the current line. (same as **control-Delete**)

**Insert** - Re-insert the preceding series of **Deletes** or **control-Deletes** before the current character or current line.

**control-arrow-left** - Move to the start of the previous word. On Linux/FreeBSD use **control-L**.

**control-arrow-right** - Move to the start of the next word. On Linux/FreeBSD use **control-R**.

**Home** - Move to the beginning of the current line.

**End** - Move to the end of the current line.

**control-Home** - Move to the beginning of the file. This works with ex.exe only. On Windows/Linux/FreeBSD use **control-T** (i.e. Top)

**control-End** - Move to the end of the file. This works with ex.exe only. On Windows/Linux/FreeBSD use **control-B**, (i.e. Bottom)

**Page Up** - Move up one screen. In a Linux/FreeBSD xterm use **control-U**

**Page Down** - Move down one screen. In a Linux/FreeBSD xterm use **control-P**

**F1 ... F10** - Select a new current window. The windows are numbered from top to bottom, with the top window on the screen being **F1**.

**F12** - This is a special **customizable command**. It is set up to insert a Euphoria comment mark "--" at the start of the current line. You can easily **change it to perform any series of key strokes that you like**, simply by redefining constant CUSTOM\_KEYSTROKES near the top of **ed.ex**.

## Escape Commands

Press and release the **Esc** key, then press one of the following keys:

**h** - Get help text for the editor, or Euphoria. The screen is split so you can view your program and the help text at the same time.

**c** - "Clone" the current window, i.e. make a new edit window that is initially viewing the same file at the same position as the current window. The sizes of all windows are adjusted to make room for the new window. You might want to use **Esc l** to get more lines on the screen. Each window that you create can be scrolled independently and each has its own menu bar. The changes that you make to a file will initially appear only in the current window. When you press an **F-key** to select a new window, any changes will appear there as well. You can use **Esc n** to read a new file into any window.

**q** - Quit (delete) the current window and leave the editor if there are no more windows. You'll be warned if this is the last window used for editing a modified file. Any remaining windows are given more space.

**s** - Save the file being edited in the current window, then quit the current window as **Esc q** above.

**w** - Save the file but do not quit the window.

**e** - Save the file, and then execute it with **ex**, **exw** or **exu**. When the program finishes execution you'll hear a beep. Hit **Enter** to return to the editor. This operation may not work if you are very low on extended memory. You can't supply any **command-line arguments** to the program.

**d** - Run an operating system command. After the beep, hit **Enter** to return to the editor. You could also use this command to edit another file and then return, but **Esc c** is probably more convenient.

**n** - Start editing a new file in the current window. Deleted lines/chars and search strings are available for use in the new file. You must type in the path to the new file. Alternatively, you can drag a file name from a Windows file manager window into the MS-DOS window for **ed**. This will type the full path for you.

**f** - Find the next occurrence of a string in the current window. When you type in a new string there is an option to "match case" or not. Press **y** if you require upper/lower case to match. Keep hitting **Enter** to find

subsequent occurrences. Any other key stops the search. To search from the beginning, press **control-Home** before **Esc f**. The default string to search for, if you don't type anything, is shown in double quotes.

**r** - Globally replace one string by another. Operates like **Esc f** command. Keep hitting **Enter** to continue replacing. Be careful -- *there is no way to skip over a possible replacement*.

**l** - Change the number of lines displayed on the screen. Only certain values are allowed, depending on your video card. Many cards will allow 25, 28, 43 and 50 lines.

In a Linux/FreeBSD text console you're stuck with the number of lines available (usually 25). In a Linux/FreeBSD xterm window, **ed** will use the number of lines initially available when **ed** is started up. Changing the size of the window will have no effect after **ed** is started.

**m** - Show the modifications that you've made so far. The current edit buffer is saved as **editbuff.tmp**, and is compared with the file on disk using the DOS **fc** command, or the Linux/FreeBSD **diff** command. **Esc m** is very useful when you want to quit the editor, but you can't remember what changes you made, or whether it's ok to save them. It's also useful when you make an editing mistake and you want to see what the original text looked like.

**ddd** - Move to line number *ddd*. e.g. **Esc 1023 Enter** would move to line 1023 in the file.

**CR** - **Esc Carriage-Return**, i.e. **Esc Enter**, will tell you the name of the current file, as well as the line and character position you are on, and whether the file has been modified since the last save. If you press **Esc** and then change your mind, it is harmless to just hit **Enter** so you can go back to editing.

## Recalling Previous Strings

The **Esc n**, **Esc d**, **Esc r** and **Esc f** commands prompt you to enter a string. You can recall and edit these strings just as you would at the DOS or Linux/FreeBSD command line. Type up-arrow or down-arrow to cycle through strings that you previously entered for a given command, then use left-arrow, right-arrow and the delete key to edit the strings. Press **Enter** to submit the string.

## Cutting and Pasting

When you **control-Delete** (or **control-D**) a series of consecutive lines, or **Delete** a series of consecutive characters, you create a "kill-buffer" containing what you just deleted. This kill-buffer can be re-inserted by moving the cursor and then pressing **Insert**.

A new kill-buffer is started, and the old buffer is lost, each time you move away and start deleting somewhere else. For example, cut a series of *lines* with **control-Delete**. Then move the cursor to where you want to paste the lines and press **Insert**. If you want to copy the lines, without destroying the original text, first **control-Delete** them, then immediately press **Insert** to re-insert them. Then move somewhere else and press **Insert** to insert them again, as many times as you like. You can also **Delete** a series of individual *characters*, move the cursor, and then paste the deleted characters somewhere else. Immediately press **Insert** after deleting if you want to copy without removing the original characters.

Once you have a kill-buffer, you can type **Esc n** to read in a new file, or you can press an **F-key** to select a new edit window. You can then insert your kill-buffer.

## Use of Tabs

The standard *tab* width is 8 spaces. The editor assumes `tab=8` for most files. However, it is more convenient when editing a program for a tab to equal the amount of space that you like to indent. Therefore you will find that tabs are set to 4 when you edit Euphoria files (or `.c`, or `.h` or `.bas` files). The editor converts from `tab=8` to `tab=4` when reading your *program* file, and converts back to `tab=8` when you save the file. Thus your file remains compatible with the `tab=8` world, e.g. MS-DOS PRINT, EDIT, etc. **If you would like to choose a different number of spaces to indent**, change the line at the top of `ed.ex` that says "constant `PROG_INDENT = 4`".

## Long Lines

Lines that extend beyond the right edge of the screen are marked with an *inverse video* character in the 80th column. This warns you that there is more text "out there" that you can't see. You can move the cursor beyond the 80th column. The screen will scroll left or right so the cursor position is always visible.

## Maximum File Size

Like any Euphoria program, **ed** can access all the memory on your machine. It can edit huge files, and unless disk swapping occurs, most operations will be very fast.

## Non-text Files

**ed** is designed for editing pure text files, although you can use it to view other files. As **ed** reads in a file, it replaces certain non-printable characters (less than ASCII 14) with ASCII 254 - small square. *If you try to save a non-text file you will be warned about this.* (MS-DOS Edit will quietly corrupt a non-text file - do not save!). Since **ed** opens all files as "text" files, a **control-z** character (26) embedded in a file will appear to **ed** to be the *end of the file*.

## Long Filenames

Although **ed** is a DOS editor, you can edit *existing* files that have pathnames with long names in them, and the full file name will be preserved. However in this release **ed** will not create *new* files with long names. The name will be truncated to the standard DOS 8.3 length. (but see Platform below)

## Line Terminator

The end-of-line terminator on Linux/FreeBSD is simply `\n`. On DOS and Windows, text files have lines ending with `\r\n`. If you copy a DOS or Windows file to Linux/FreeBSD and try to modify it, **ed** will give you a choice of either keeping the `\r\n` terminators, or saving the file with `\n` terminators.

## Source Code

The complete source code to this editor is in [bin\ed.ex](#) and [bin\syncolor.e](#). You are welcome to make improvements. There is a section at the top of [ed.ex](#) containing "user-modifiable" configuration parameters that you can adjust. The colors and the cursor size may need adjusting for some operating environments.

## Platform

[euphoria\bin\ed.bat](#) can be set up to run ed.ex using [exwc.exe](#) or [ex.exe](#). You are better off running ed with [ex.exe](#) on Windows 95/98/ME. You'll get much quicker screen updates than with [exwc.exe](#). On Windows XP you'll be a bit better off using [exwc.exe](#). You'll get slightly quicker screen updates, and you'll be able to create files with long names, not just open existing ones. However some special keys won't work with [exwc.exe](#), e.g. you'll have to use control-t and control-b instead of control-Home and control-End. On Linux and FreeBSD there are no problems with long filenames, and the keyboard response is always fast.

# Euphoria Trouble-Shooting Guide

If you get stuck, here are some things you can do:

1. - Type: **guru**  
followed by some keywords associated with your problem.

For example,

**guru declare global include**

2. - Check the list of common problems ([below](#)).
3. - Read the relevant parts of the documentation, i.e. [refman.doc](#) or [library.doc](#).
4. - Try running your program with the statements:

```
with trace
trace(1)
```

at the top of your main **.ex** file so you can see what's going on.

5. - The [Euphoria mailing list](#) has a search facility. You can search the archive of all previous messages. There's a good chance that your question has already been discussed.
6. - Post a message on the mailing list.

Here are some commonly reported problems ( **P:** ) and their solutions ( **S:** ).

**P:** I ran my program with **exw** and the console window disappeared before I could read the output.

**S:** The console window will only appear if required, and will disappear immediately when your program finishes execution. Perhaps you should code something like:

```
puts(1, "\nPress Enter\n")
if getc(0) then
end if
```

at the end of your program.

**P:** At the end of execution of my program, I see "Press Enter" and I have to hit the Enter key. How do I get rid of that?

**S:** Call `free_console()` just before your program terminates.

```
include dll.e

free_console()
```

**P:** I would like to change the properties of the console window.

**S:** Right click on **c:\windows\system\conagent.exe** and select "properties". You can change the font and several other items.

**P:** When I run **ex.exe** in a DOS Window, it makes my small DOS window go to full screen.

**S:** This will only happen the first time you run **ex.exe** after creating a new small DOS window. You can make the window small again by pressing **Alt-Enter**. It will stay small after that. Your Euphoria program can keep the window small by executing:

```
if graphics_mode(-1) then
end if
```

at the start of execution. This may cause some brief screen flicker. Your program can force a text

window to be full-screen by executing:

```
        if graphics_mode(3) then
    end if
```

**P:** My Euphoria CGI program hangs or has no output

**S:** Look for an ex.err file in your cgi-bin directory. Turn on **with trace** / **trace(3)** to see what statements are executed (see ctrace.out in your cgi-bin). Insert **without warning** at the top of your program. On Windows, when there are warnings, Euphoria will issue a prompt before terminating, causing your program to hang. On Windows you should always use **exwc.exe** to run CGI programs, or you may have problems with standard output. With Apache Web Server, you can have a first line in your program of: `#!.\exwc.exe` to run your program using exwc.exe in the current (cgi-bin) directory. On Linux/FreeBSD, be careful that your first line ends in LF, not CR-LF, or the `#!` won't be handled correctly. On Linux you must also set the execute permissions on your program correctly, and ex.err and ctrace.out must be writable by the server process or they won't be updated. See [cgi.htm](#) for more

**P:** How do I read/write ports?

**S:** Get **Jacques Deschenes'** collection of machine-level and DOS system routines from the [Euphoria Web page](#). See his file **ports.e**.

**P:** I'm having trouble running a **DOS32** graphics program. I hit control-Break and now my system seems to be dead.

**S:** Some graphics programs will not run unless you start them from DOS or from a full-screen DOS window under Windows. Sometimes you have to edit the program source to use a lower resolution graphics mode, such as mode 18. Some SVGA graphics modes might not work for you under a DOS window, but will work when you restart your machine in MS-DOS mode. A better driver for your video card might fix this.

You should stop a program using the method that the program documentation recommends. If you abort a program with control-c or control-Break you may find that your screen is left in a funny graphics mode using funny colors. When you type something, it may be difficult or even impossible to read what you are typing. *The system may even appear dead, when in fact it is **not**.*

Try the following DOS commands, in the following order, until you clear things up:

1. type: Alt-Enter to get a normal (non-full-screen) window again.

2. type: **cls**

Even when you can't see any keystrokes echoed on the screen this may clear the screen for you.

3. type: **ex**

The Euphoria interpreter will try to restore a normal **text mode** screen for you.

4. type: **exit**

If you are running under Windows, this will terminate the DOS session for you.

5. type: **Control-Alt-Delete**

This will let you kill the current DOS session under Windows, or will soft-reboot your computer if you are running under DOS.

6. If all else fails, reset or power your computer off and back on. You should immediately run **scandisk** when the system comes back up.

**P:** When I run Euphoria programs in SVGA, the output on the screen is crammed into the top part of the screen.

**S:** Try: `use_vesa(1)` in `machine.e`.

Try downloading the latest driver for your video card from the Internet. ATI's site is:

<http://www.atitech.com>

Others have had their video problems clear up after installing the latest version of DirectX.

**P:** My program leaves the DOS window in a messy state. I want it to leave me with a normal text window.

**S:** When your program is finished, it should call the function `graphics_mode(-1)` to set the DOS window back to normal. e.g.

```
        if graphics_mode(-1) then
end if
```

**P:** When I run my program from the editor and I hit control-c, the program dies with an operating system error.

**S:** This is a known problem. Run your program from the command-line, outside the editor, if you might have to hit control-c or control-Break.

**P:** When I run my program there are no errors but nothing happens.

**S:** You probably forgot to call your main procedure. You need a top-level statement that comes after your main procedure to call the main procedure and start execution.

**P:** I'm trying to call a routine documented in `library.doc`, but it keeps saying the routine has not been declared.

**S:** Did you remember to include the necessary `.e` file from the `euphoria\include` directory? If the syntax of the routine says for example, "include graphics.e", then your program must have "**include graphics.e**" (without the quotes) before the place where you first call the routine.

**P:** I have an include file with a routine in it that I want to call, but when I try to call the routine it says the routine has not been declared. But it *has* been declared.

**S:** Did you remember to define the routine with "**global**" in front of it in the include file? Without "global" the routine is not visible *outside* of its own file.

**P:** How do I input a line of text from the user?

**S:** See `gets()` in `library.doc`. `gets(0)` will read a line from **standard input**, which is normally the **keyboard**. The line will always have `\n` on the end. To remove the trailing `\n` character do:

```
line = line[1..length(line)-1]
```

Also see `prompt_string()` in `get.e`.

**P:** After inputting a string from the user with `gets()`, the next line that comes out on the screen does not start at the left margin.



**S:** Your program should output a *new-line* character e.g. `puts(SCREEN, '\n')` after you do a `gets()`. It does not happen automatically.

**P:** Why aren't my floating-point calculations coming out exact?

**S:** Intel CPU's, and most other CPU's, use binary numbers to represent fractions. Decimal fractions such as 0.1, 0.01 and similar numbers can't be represented precisely. For example, 0.1 might be stored internally as 0.0999999999999999. That means that  $10 * 0.1$  might come out as 0.999999999999999, and `floor(10 * 0.1)` might be 0, not 1 as you'd expect. This can be a nuisance when you are dealing with money calculations, but it's not a Euphoria problem. It's a general problem that you must face in most programming languages. Always remember: floating-point numbers are just an approximation to the "real" numbers in mathematics. Assume that any floating-point calculation might have a tiny bit of error in the result. Sometimes you can solve the problem by rounding, e.g.  $x = \text{floor}(x + 0.5)$  would round  $x$  off to the nearest integer. Storing money values as an integer number of pennies, rather than a fractional number of dollars (or similar currency) will help, but some calculations could still cause problems.

**P:** How do I convert a number to a string?

**S:** Use `sprintf()` in [library.doc](#). e.g.

```
string = sprintf("%d", number)
```

Besides `%d`, you can also try other formats, such as `%x` (Hex) or `%f` (floating-point).

**P:** How do I convert a string to a number?

**S:** Use `value()` in [library.doc](#), or, if you are reading from a file or the keyboard, use `get()`.

**P:** It says I'm attempting to redefine my for-loop variable.

**S:** For-loop variables are declared automatically. Apparently you already have a declaration with the same name earlier in your routine or your program. Remove that earlier declaration or change the name of your loop variable.

**P:** I get the message "unknown escape character" on a line where I am trying to specify a file name.

**S:** *Do not* say "C:\TMP\MYFILE". You need to say "C:\\TMP\\MYFILE". Backslash is used for escape characters such as `\n` or `\t`. To specify a single backslash in a string you need to type `\\`.

**P:** I'm trying to use mouse input in a SVGA graphics mode but it just doesn't work.

**S:** **DOS** does not support mouse input in modes beyond graphics mode 18 (640x480 16 color). **DOS 7.0 (part of Windows 95/98)** does seem to let you at least read the x-y coordinate and the buttons in high resolution modes, but you may have to draw your own mouse pointer in the high-res modes. **Graeme Burke, Peter Blue** and others have good solutions to this problem. See their files on the [Euphoria Archive Web page](#).

**P:** I'm trying to print a string using `printf()` but only the first character comes out.

**S:** See the `printf()` description in [library.doc](#). You may need to put braces around your string so it is seen as a single value to be printed, e.g. you wrote:

```
printf(1, "Hello %s", mystring)
```

where you should have said:

```
printf(1, "Hello %s", {mystring})
```

**P:** When I print numbers using `print()` or `?`, only 10 significant digits are displayed.



**S:** Euphoria normally only shows about 10 digits. Internally, all calculations are performed using at least 15 significant digits. To see more digits you have to use `printf()`. For example,

```
printf(1, "%.15f", 1/3)
```

This will display 15 digits.

**P:** It complains about my routine declaration, saying "a type is expected here".

**S:** When declaring subroutine parameters, Euphoria requires you to provide an explicit type for each individual parameter. e.g.

```
procedure foo(integer x, y)           -- WRONG
procedure foo(integer x, integer y) -- RIGHT
```

In all other contexts it is ok to make a list:

```
atom a, b, c, d, e
```

**P:** I'm declaring some variables in the middle of a routine and it gives me a syntax error.

**S:** All of your *private* variable declarations must come at the beginning of your **subroutine**, before any executable statements. (At the top-level of a program, **outside of any routine**, it is ok to declare variables anywhere.)

**P:** It says:

Syntax Error - expected to see possibly 'xxx', not 'yyy'

**S:** At this point in your program you have typed a variable, keyword, number or punctuation symbol, yyy, that does not fit syntactically with what has come before it. The compiler is offering you one example, xxx, of something that would be accepted at this point in place of yyy. Note that there may be many other legal (and much better) possibilities at this point than xxx, but xxx might at least give you a clue as to what the compiler is "thinking".

**P:** I'm having problems running Euphoria with DR-DOS.

**S:** Your **config.sys** should have just HIMEM.SYS but not EMM386.EXE in it.

**P:** I try to run a program with **exw** and it says "this is a Windows NT character-mode executable".

**S:** **exw.exe** is a **32-bit Windows program**. It must be run under Windows or in a DOS Window. It will not work under plain DOS in an old system. **ex.exe** *will* work under **plain DOS**.

# Binding and Shrouding

## The Shroud Command

### Synopsis:

```
shroud [-full_debug] [-list] [-quiet] [-out shrouded_file] filename.ex[w/u]
```

The *shroud* command converts a Euphoria program, typically consisting of a main file plus many include files, into a single, compact file. A single file is not only convenient, it allows you to give people your program to use, without giving them 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 interpreter back-end: backend.exe (DOS), backendw.exe (Windows) or backendu (Linux/FreeBSD). This backend is freely available, and you can give it to any of your users who need it. It's stored in euphoria\bin in the Euphoria interpreter package. On DOS you can run your .il file with:

```
backend myprog.il
```

On Windows use:

```
backendw myprog.il
```

On Linux or FreeBSD use:

```
backendu myprog.il
```

Although it does not contain any source statements, a .il file will generate a useful ex.err dump in case of a run-time error.

The shrouder will remove any routines and variables that your program doesn't use. This will give you a smaller .il file. There are often a great number of unused routines and unused variables. For example your program might include several 3rd party include files, plus some standard files from euphoria\include, but only use a few items from each file. The unused items will be deleted.

### options can be:

**-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 Linux/FreeBSD, the shrouder will make your shrouded file executable, and will add a **#!** line at the top, that will run backendu. You can override this **#!** line by specifying your own **#!** line at the top of your main Euphoria file.

Always keep a copy of your original source. There's no way to recover it from a shrouded file.

## The Bind Command

### Synopsis:

```
bind [-full_debug] [-list] [-quiet] [-out executable_file] [filename.ex]
bindu [-full_debug] [-list] [-quiet] [-out executable_file] [filename.exu]
bindw [-full_debug] [-list] [-quiet] [-out executable_file] [-con] [-icon
filename.ico] [filename.exw]
```

**bind** (**bindw** or **bindu**) does the same thing as **shroud**, and includes the same options. It then combines your shrouded .il file with the interpreter backend (backend.exe, backendw.exe or backendu) to make a **single, stand-alone executable** file 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.

### options can be:

**-full\_debug** - Same as **shroud** above. If Euphoria detects an error, your executable will generate either a partial, or a full, ex.err dump, according to this option.

**-list** - Same as **shroud** above.

**-quiet** - Same as **shroud** above.

**-out executable\_file** - This option lets you choose the name of the executable file created by the binder. Without this option, bind will choose a name based on the name of the main Euphoria source file.

**-con - (bindw only)** This option will create a Windows console program instead of a Windows 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.

**-icon filename[.ico]** - **(bindw only)** When you bind a program, you can patch in your own customized icon, overwriting the one in **exw.exe**. **exw.exe** contains a 32x32 icon using 256 colors. It resembles an **E** shape. Windows will display this shape beside exw.exe, and beside your bound program, in file listings. You can also load this icon as a resource, using the name "exw" (see euphoria\demo\win32>window.exw 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 (Windows will create a smaller or larger image as necessary). The default **E** icon file, euphoria.ico, is included in the euphoria\bin directory.

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. **When bound, the entire Euphoria editor, ed.ex, adds only 27K to the size of the back-end.** **backendw.exe** and **backendu** (Linux) are compressed using **UPX** (see <http://upx.sourceforge.net>). **backend.exe** is compressed using a tool that comes with the CauseWay DOS extender. **backend.exe** is the largest of the three since it includes a lot of pixel graphics routines, not part of **backendw.exe** or **backendu**. Note: In some very rare cases, a compressed executable may trigger a warning message from a virus scanner. This is simply because the executable file looks abnormal to the virus scanner.

The first two arguments returned by the **command\_line()** library routine will be slightly different when your program is bound. See [library.doc](#) for the details.

A **bound executable** file *can* handle standard input and output redirection. e.g.

```
myprog.exe < file.in > file.out
```

If you were to write a small DOS **.bat** file **myprog.bat** that contained the line "**ex myprog.ex**" you would *not* be able to redirect input and output in the following manner:

```
myprog.bat < file.in > file.out      (doesn't work in DOS!)
```

You *could* however use redirection on individual lines *within* the **.bat** file.

# Euphoria Performance Tips

## General Tips

- If your program is fast enough, forget about speeding it up. Just make it simple and readable.
- If your program is way too slow, the tips below will probably not solve your problem. You should find a better overall algorithm.
- The easiest way to gain a bit of speed is to turn off run-time type-checking. Insert the line:

- ```
without type_check
```

 at the top of your main **.ex** file, ahead of any include statements. You'll typically gain between 0 and 20 percent depending on the types you have defined, and the files that you are including. Most of the standard include files do some user-defined type-checking. A program that is completely without user-defined type-checking might still be speeded up slightly.

Also, be **sure** to remove, or comment-out, any

```
with trace
with profile
with profile_time
```

statements. **with trace** (even without any calls to `trace()`), and **with profile** can easily slow you down by 10% or more. **with profile\_time** might slow you down by 1%. Each of these options will consume extra memory as well.

- Calculations using integer values are faster than calculations using floating-point numbers
- Declare variables as integer rather than atom where possible, and as sequence rather than object where possible. This usually gains you a few percent in speed.
- In an expression involving floating-point calculations it's usually faster to write constant numbers in floating point form, e.g. when `x` has a floating-point value, say, `x = 9.9`

change:

- ```
x = x * 5
```

 to:  

```
x = x * 5.0
```

This saves the interpreter from having to convert integer 5 to floating-point 5.0 each time.

- Euphoria does **short-circuit** evaluation of **if**, **elsif**, and **while** conditions involving **and** and **or**. Euphoria will stop evaluating any condition once it determines if the condition is true or not. For instance in the **if-statement**:

- ```
if x > 20 and y = 0 then
```
- ```
...
```
- ```
end if
```

The "`y = 0`" test will only be made when "`x > 20`" is true.

For maximum speed, you can order your tests. Do "`x > 20`" first if it is more likely to be false than "`y = 0`".

In general, with a condition "A and B", Euphoria will not evaluate the expression B, when A is false (zero). Similarly, with a condition like "A or B", B will not be evaluated when A is true (non-zero).

Simple if-statements are highly optimized. With the current version of the interpreter, nested simple if's that compare integers are usually a bit faster than a single short-circuit if-statement e.g.:

```
if x > 20 then
    if y = 0 then
        ...
    end if
end if
```

- The speed of access to private variables, local variables and global variables is the same.
- There is no performance penalty for defining constants versus plugging in hard-coded literal numbers. The speed of:

- ```
y = x * MAX
```

 is exactly the same as:  

```
y = x * 1000
```

 where you've previously defined:  

```
constant MAX = 1000
```

- There is no performance penalty for having lots of comments in your program. Comments are completely ignored. They are not executed in any way. It might take a few milliseconds longer for the initial load of your program, but that's a very small price to pay for future maintainability, and when you **bind** your program, or **translate** your program to C, all comments are stripped out, so the cost becomes absolute zero.

## Measuring Performance

In any programming language, and especially in Euphoria, **you really have to make measurements before drawing conclusions about performance.**

Euphoria provides both **execution-count profiling**, as well as **time profiling (DOS32 only)**. See [refman.doc](#). You will often be surprised by the results of these profiles. Concentrate your efforts on the places in your program that are using a high percentage of the total time (or at least are executed a large number of times.) There's no point to rewriting a section of code that uses 0.01% of the total time. Usually there will be one place, or just a few places where code tweaking will make a significant difference.

You can also measure the speed of code by using the **time()** function. e.g.

```
atom t
t = time()
for i = 1 to 10000 do
    -- small chunk of code here
end for
? time() - t
```

You might rewrite the small chunk of code in different ways to see which way is faster.

## How to Speed-Up Loops

**Profiling** will show you the *hot spots* in your program. These are usually inside loops. Look at each calculation inside the loop and ask yourself if it really needs to happen every time through the loop, or could it be done just once, prior to the loop.

## Converting Multiplies to Adds in a Loop

Addition is faster than multiplication. Sometimes you can replace a multiplication by the loop variable, with an addition. Something like:

```
for i = 0 to 199 do
  poke(screen_memory+i*320, 0)
end for
```

becomes:

```
x = screen_memory
for i = 0 to 199 do
  poke(x, 0)
  x = x + 320
end for
```

## Saving Results in Variables

- It's faster to save the result of a calculation in a variable, than it is to recalculate it later. Even something as simple as a subscript operation, or adding 1 to a variable is worth saving.

- When you have a sequence with multiple levels of subscripting, it is faster to change code like:

```
• for i = 1 to 1000 do
•   y[a][i] = y[a][i]+1
• end for
```

to:

```
ya = y[a]
for i = 1 to 1000 do
  ya[i] = ya[i] + 1
end for
y[a] = ya
```

So you are doing 2 subscript operations per iteration of the loop, rather than 4. The operations,  $ya = y[a]$  and  $y[a] = ya$  are very cheap. **They just copy a pointer.** They don't copy a whole sequence.

- There is a slight cost when you create a new sequence using **{a,b,c}**. If possible, move this operation out of a critical loop by storing it in a variable before the loop, and referencing the variable inside the loop.

## In-lining of Routine Calls

If you have a routine that is rather small and fast, but is called a huge number of times, you will save time by doing the operation *in-line*, rather than calling the routine. Your code may become less readable, so it might be better to in-line only at places that generate a lot of calls to the routine.

## Operations on Sequences

Euphoria lets you operate on a large sequence of data using a single statement. This saves you from writing a loop where you process one element at-a-time. e.g.

```
x = {1,3,5,7,9}
y = {2,4,6,8,10}
```

```
z = x + y
```

versus:

```
z = repeat(0, 5)  -- if necessary
for i = 1 to 5 do
    z[i] = x[i] + y[i]
end for
```

In most interpreted languages, it is much faster to process a whole sequence (array) in one statement, than it is to perform scalar operations in a loop. This is because the interpreter has a large amount of overhead for each statement it executes. Euphoria is different. Euphoria is very lean, with little interpretive overhead, so operations on sequences don't always win. The only solution is to time it both ways. The per-element cost is usually lower when you process a sequence in one statement, but there are overheads associated with allocation and deallocation of sequences that may tip the scale the other way.

## Some Special Case Optimizations

Euphoria automatically optimizes certain special cases. x and y below could be variables or arbitrary expressions.

```
x + 1      -- faster than general x + y
1 + x      -- faster than general y + x
x * 2      -- faster than general x * y
2 * x      -- faster than general y * x
x / 2      -- faster than general x / y
floor(x/y) -- where x and y are integers, is faster than x/y
floor(x/2) -- faster than floor(x/y)
```

x below is a simple variable, y is any variable or expression:

```
x = append(x, y)  -- faster than general z = append(x, y)
x = prepend(x, y) -- faster than general z = prepend(x, y)

x = x & y          -- where x is much larger than y,
                  -- is faster than general z = x & y
```

When you write a loop that "grows" a sequence, by appending or concatenating data onto it, the time will, in general, grow in proportion to the **square** of the number (N) of elements you are adding. However, if you can use one of the special optimized forms of append(), prepend() or concatenation listed above, the time will grow in proportion to just N (roughly). This could save you a **huge** amount of time when creating an extremely long sequence. (You could also use repeat() to establish the maximum size of the sequence, and then fill in the elements in a loop, as discussed below.)

## Assignment with Operators

For greater speed, convert:

```
left-hand-side = left-hand-side op expression
```



to:

**left-hand-side op= expression**

whenever left-hand-side contains at least 2 subscripts, or at least one subscript and a slice. In all simpler cases the two forms run at the same speed (or very close to the same).

## Pixel-Graphics Tips

- Mode 19 is the fastest mode for **animated graphics** and **games**.
- The video memory (in mode 19) is not cached by the CPU. It usually takes longer to read or write data to the screen than to a general area of memory that you allocate. This adds to the efficiency of **virtual screens**, where you do all of your image updating in a **block of memory** that you get from **allocate()**, and then you periodically **mem\_copy()** the resulting image to the real **screen memory**. In this way you never have to read the (slow) screen memory.
- When plotting pixels, you may find that modes 257 and higher are fast near the top of the screen, but slow near the bottom.

## Text-Mode Tips

Writing text to the screen using **puts()** or **printf()** is rather slow. If necessary, in **DOS32**, you can do it much faster by poking into the **video memory**, or by using **display\_text\_image()**. There is a very large overhead on each **puts()** to the screen, and a relatively small incremental cost per character. The overhead with **exw** (WIN32) is especially high (on Windows 95/98/ME at least). Linux and FreeBSD are somewhere between DOS32 and WIN32 in text output speed. It therefore makes sense to build up a long string before calling **puts()**, rather than calling it for each character. There is no advantage to building up a string longer than one line however.

The slowness of text output is mainly due to operating system overhead.

## Library Routines

Some common routines are extremely fast. You probably couldn't do the job faster any other way, even if you used C or assembly language. Some of these are:

- **mem\_copy()**
- **mem\_set()**
- **repeat()**

Other routines are reasonably fast, but you might be able to do the job faster in some cases if speed was crucial.

```
x = repeat(0,100)
for i = 1 to 100 do
    x[i] = i
end for
```

is somewhat faster than:

```
x = {}  
for i = 1 to 100 do  
    x = append(x, i)  
end for
```

because `append()` has to allocate and reallocate space as `x` grows in size. With `repeat()`, the space for `x` is allocated once at the beginning. (`append()` is smart enough not to allocate space with *every* append to `x`. It will allocate somewhat more than it needs, to reduce the number of reallocations.)

You can replace:

```
remainder(x, p)
```

with:

```
and_bits(x, p-1)
```

for greater speed when `p` is a positive power of 2. `x` must be a non-negative integer that fits in 32-bits.

`arctan()` is faster than `arccos()` or `arcsin()`.

## Searching

Euphoria's `find()` is the fastest way to search for a value in a sequence up to about 50 elements. Beyond that, you might consider a *hash table* ([demo\hash.ex](#)) or a *binary tree* ([demo\tree.ex](#)).

## Sorting

In most cases you can just use the *shell sort* routine in [include\sort.e](#).

If you have a huge amount of data to sort, you might try one of the sorts in [demo\allsorts.e](#) (e.g. *great sort*). If your data is too big to fit in memory, don't rely on Euphoria's automatic memory swapping capability. Instead, sort a few thousand records at a time, and write them out to a series of temporary files. Then merge all the sorted temporary files into one big sorted file.

If your data consists of integers only, and they are all in a fairly narrow range, try the *bucket sort* in [demo\allsorts.e](#).

## Taking Advantage of Cache Memory

As CPU speeds increase, the gap between the speed of the on-chip cache memory and the speed of the main memory or DRAM (dynamic random access memory) becomes ever greater. You might have 256 Mb of DRAM on your computer, but the on-chip cache is likely to be only 8K (data) plus 8K (instructions) on a Pentium, or 16K (data) plus 16K (instructions) on a Pentium with MMX or a Pentium II/III. Most machines will also have a "level-2" cache of 256K or 512K.

An algorithm that steps through a long sequence of a couple of thousand elements or more, many times, from beginning to end, performing one small operation on each element, will not make good use of the on-chip data cache. It might be better to go through once, applying several operations to each element, before moving on to the next element. The same argument holds when your program starts swapping, and the least-recently-used data is moved out to disk.

These cache effects aren't as noticeable in Euphoria as they are in lower-level compiled languages, but they are measurable.

## Using Machine Code and C

Euphoria lets you call routines written in 32-bit Intel machine code. On **WIN32** and **Linux** and **FreeBSD** you can call C routines in .dll or .so files, and these C routines can call your Euphoria routines. You might need to call C or machine code because of something that can't be done directly in Euphoria, or you might do it for improved speed.

To boost speed, the machine code or C routine needs to do a significant amount of work on each call, otherwise the overhead of setting up the arguments and making the call will dominate the time, and it might not gain you much.

Many programs have some inner core operation that consumes most of the CPU time. If you can code this in C or machine code, while leaving the bulk of the program in Euphoria, you might achieve a speed comparable to C, without sacrificing Euphoria's safety and flexibility.

## Using The Euphoria To C Translator

In version 3.0, the full Euphoria To C Translator is included in the installation package. It will translate any Euphoria program into a set of C source files that you can compile using a C compiler.

The executable file that you get using the Translator should run the same, but faster than when you use the interpreter. The speed-up can be anywhere from a few percent to a factor of 5 or more.

# Euphoria Release Notes

## Version 3.0.1 November 3, 2006:

This release cleans up a number of bugs and documentation errors in 3.0.0.

### Improved Documentation

- A separate new document on [multitasking](#) was added.
- Obsolete pre-open-source concepts in the 3.0.0 documentation were deleted or corrected.

### Bug Fixes

- Euphoria-coded .dll / .so files were not working at all, due to a multitasking-related glitch. Thanks to Andrea Cini.

Note: execution of multitasking built-in routines is currently not supported inside a Euphoria-coded .dll / .so. You'll get a compile-time error message from the Translator if you use the -dll option and your library source code contains multitasking operations. Other than that, use of Euphoria-coded .dll / .so files by main programs that do multitasking is fine.

- A text console bug was fixed. It involved output to the screen involving tabs and/or wrap-around (at 80 columns). Thanks to Pete Lomax, Juergen Luethje, Matt Lewis.
- bind.bat, bindw.bat and shroud.bat were changed to use %EUDIR% rather than \EUPHORIA. Thanks to Ray Smith.
- syncolor.e now includes keywords.e (relative to euphoria\bin) rather than \euphoria\bin\keywords.e. Thanks to Rick Bettis.

### C Source Code Changes

- All of the #ifdef ENCURES code was stripped out. We don't plan to use ncurses again, and this change makes parts of the source code much more readable.
- A thread5() machine language macro was added to be\_execute.c This helps the interpreter to compile correctly with Open Watcom, avoiding a C optimizer bug. Thanks to Matt Lewis.

## Version 3.0.0 October 17, 2006:

With this release, Euphoria has become a totally **free** and totally **open source** product! RDS will continue to develop Euphoria with the aid of many additional clever programmers. The free download package now includes the Interpreter (with either a C or a Euphoria-coded back-end), Binder/shrouder, fully-enabled Translator, and the full source code for all of these. Thousands of people can now examine the full source code for bugs, performance improvements, and potential new features.

Rather than having alpha, beta and official releases, we will now simply have numbers, 3.0.0, 3.0.1 ... We expect to have releases more frequently, though each release will likely be a smaller change compared to the

previous release.

## Enhanced Features

- Cooperative Multitasking.** Rather than having just a single thread of execution, you can now create multiple tasks that run independently of one another. Each task has its own currently-executing statement, subroutine call-stack, and private variables for all routines on its call-stack. Tasks share global and local variables but not private variables. At any point during its execution, a task can call `task_yield()` to transfer control to the Euphoria scheduler which will choose the next task to run. When control returns to the original task, execution will continue from the statement after `task_yield()`.

New Run-time routines: `task_create()`, `task_schedule()`, `task_yield()`, `task_suspend()`, `task_self()`, `task_list()`, `task_status()`, `task_clock_start()`, `task_clock_stop()`

- Use of the **ncurses** library has been eliminated for Linux and FreeBSD. `ncurses` routines sometimes caused problems on some Linux/FreeBSD systems. The Euphoria backend now uses ANSI escape chars to get 2-d positioning and colors for a plain text-mode console. Also, the use of `libgpm` (console mouse support) has been dropped on Linux. (It was never supported by Euphoria on FreeBSD).

- Include files** with the same file name but a different path as an earlier include, will no longer be ignored. To be ignored, an include must refer to the exact same file as an earlier include. (a new name space can still be defined, even though the include file is not actually included again.) On Linux/FreeBSD a case-sensitive file-name comparison is now used. There is possible (but unlikely) breakage of old code:

You might start including files that you did not intend to include, if you have a previously-ignored include statement for them in your code. Solution: Delete the undesired include statement.

If an include is incorrect (file is not on the include search path), this error may have been hidden under the old system, if a correct include with the same file name (but different path) came earlier. Solution: Specify the correct path to the include file.

- There is no longer a limit on the number of **warnings** that can be displayed. Instead, you will be given the chance to scroll through all the warnings, 20 at a time. As before, if a `ex.err` file is created, then all warnings will also be stored at the end of the `ex.err` file. Thanks to Judith Evans.
- Translator:** You can set the run-time stack size for your program using the `-stack nnnn` option. The default stack size has been increased for most of the supported C compilers, especially when your program contains a call to `task_create()`.
- If a run-time error occurs and `ex.err` can't be opened, a check will be made to see if it's because you have **too many open files**. An appropriate message will then be issued. Several people ran into this situation.
- execute.e:** Additional run-time error checks were added to several run-time routines to allow the pure-Euphoria source interpreter to catch more errors itself, rather than letting the C-coded run-time routines used by the "real" interpreter or translator catch them.

- **pretty\_print()**, option 3, now includes `\t` `\r` and `\n` as valid "ASCII range" characters. This increases the likelihood of strings being displayed. Thanks to Juergen Luethje.
- The **ASCII 127** char will not be graphically displayed on Linux in trace or in default **pretty\_print()**. It displays as a backspace when ANSI codes are used.
- **Binder** options are checked more strictly. e.g. `-xxxoutxxx` was dangerous. The binder now looks for `match(x,y) = 1`, rather than `match(x,y) != 0`. The binder also avoids overwriting a source file when `-out` is used. Thanks to Mike Sabal, Greg Haberek
- We've switched to using transparent Euphoria icons. Thanks to Vincent Howell.
- Pure Euphoria interpreter (`eu.ex`): The "not initialized" message has been improved to: "xyz has not been initialized"
- Many small improvements were made to the **documentation**. e.g. the use of a single atom value by multiple `printf()` formats was documented for the first time. Thanks to Pete Lomax.

## Bug Fixes

- **bug fixed: Translator**: a Euphoria file named "main.e" or "init.e" could possibly lead to a file naming conflict. The chance of a conflict is now greatly reduced, and if a conflict occurs, a meaningful error message will be issued. Thanks to Vincent Howell.
- **bug fixed: Translator**: using "interrupt" as a Euphoria variable name caused a naming conflict when compiling with Watcom. Thanks to Louis Bryant.
- **bug fixed: Translator**: A backslash at the end of a Euphoria comment could be considered a line continuation character by the C compiler. This could result in incorrect code being generated. Thanks to Mark Honnor.
- **bug fixed: Translator**: A call to a Euphoria-coded routine in a .dll, i.e. via `c_func()`, could cause a bug if the return type was sequence or object. Thanks to Thomas Jansen.
- **bug fixed**: If you pass file number -1 to **printf()** you will now get a run-time error report. In all previous releases, -1 would simply cause `printf()` to produce no output. Thanks to Daniel Kluss.
- **bug fixed**: `scanner.e`: Source lines longer than 10000 characters caused a problem. Also we were not checking for `allocate()` returning 0 in `pack_source()` (i.e. out of memory condition). Thanks to Antonio Alessi.
- **bug fixed: execute.e**: (pure Euphoria interpreter) In some cases slice indexes were not being bounds checked before being used. `eu.ex` would crash rather than report the error in the user's program.
- **bug fixed**: Assignments of the form:  
`x[a][b..c] += expr`  
were likely to trigger an erroneous subscript error in **execute.e**, the pure Euphoria interpreter. Thanks to Vincent Howell.
- **bug fixed: execute.e** did not handle fractional subscripts correctly in some situations. Thanks to C

## Version 2.5 Official Release March 8, 2005:

### Enhanced Features

- The Source Code Product now includes the C code for interactive trace/debug and profiling. This was not provided in any previous release.
- Translator**: A new **-fastfp** option has been added for WATCOM/DOS (registered version only). This option can double the speed of a floating-point intensive, translated/compiled, program. The resulting .exe file requires a Pentium class CPU, or a 386/486 with floating-point hardware.
- eu.ex**: execute.e now uses call-back machine code supplied by Matthew Lewis. As a result, there is no longer a limit on the number of call-backs a program can use.
- Translator**: There is now a **-keep** option that prevents the deletion of any C files or object files created during translation. Thanks to Matthew Lewis.
- Translator**: In order to handle huge programs without exceeding the limits of the C compiler, the translator will split init\_c into several pieces. Thanks to Matthew Lewis.
- Translator**: Better checking is provided for incorrect command-line options.
- Translator**: The translator avoids creating any files until it confirms that a valid source file has been provided on the command line, or via the interactive prompt.
- ed.ex**: **ed** will only call free\_console() at the end of execution on Linux or FreeBSD. It was causing a slight screen flicker on DOS/Windows, and was not necessary.

### Bug Fixes

- bug fixed**: The error traceback from a bound or shrouded program had garbage showing for the file names and line numbers. This bug was introduced in the beta release.
- bug fixed**: Bound or shrouded programs were not starting off with a fresh random seed value. Normal interpreted or translated programs were OK. Thanks to Michael Bolin.
- bug fixed**: The error traceback could be wrong if you (illegally) tried to pass a sequence as the routine id for call\_proc/func. Thanks to Mario Steele.
- bug fixed**: When displaying a large variable value during interactive trace, the first few lines might be printed in white, while the rest was printed in bright white. Thanks to Al Getz.
- bug fixed**: routine\_id() of the form: routine\_id("namespace:name") was not working correctly in the interpreter or in the PD source interpreter. The Translator was ok. This worked fine in 2.4 and earlier. Thanks to Verne Tice.
- bug fixed**: command\_line() now shows the extra file name when the user types the file name interactively at the ex/exw/exu prompt. This is now compatible with 2.4 and earlier. Thanks to Bob

Elia.

- **bug fixed:** Warnings were not getting reported when a program ended with a call to `abort()`, or it ended with a fatal run-time error. Thanks to Juergen Luethje.
- **bug fixed:** The `platform()` function is now evaluated in `backend/backendw/backendu` for `.il` files (rather than in the front-end of the shrouder). This allows a `.il` file to be portable to multiple platforms. In other situations, the interpreter and translator will continue to evaluate `platform()` in the front-end for maximum efficiency. Thanks to Ken Rhodes

## Documentation

- Many small corrections and improvements were made to the documentation. Some important ones were provided by Juergen Luethje and Wolfgang Fritz.
- A paragraph was added to the Reference Manual explaining what happens when you try to modify a variable by function side-effects at the same time that you try to modify it via subscripted assignment. The 2.5 alpha release notes were updated to point out an incompatibility with 2.4.

## Version 2.5 Beta Release January 14, 2005:

### Improved Performance

- The initial parse time (start-up time) of interpreted programs has been reduced, typically by 25% on newer machines, and 40% on older machines, when compared to 2.5 alpha. In the case of old machines running very large programs, the start-up time can be 65% to 90% less, especially when **with trace** or **with profile** are in effect.

Most of the optimizations were made to the Euphoria front-end source that's shared with the Public Domain source interpreter, so **eu.ex** also has reduced parse times.

- A major speed improvement was made in `opASSIGN_SUBS()` in the PD source interpreter. This operation handles assignments to subscripted sequences, and is executed a lot. Thanks to Greg Haberek for showing the slowness.
- **Translator:** In many more cases, the **Translator** now knows the type of all elements of a sequence. This is especially useful when the elements are known to be integers. Smaller and faster code is generated for accessing the elements, and freeing the sequence.
- In most cases, the **Interpreter** and the **Translator** now evaluate sequence-formation `{a, b, c, ...}` expressions once at compile-time, instead of each time at run-time, when the elements are all constant integers (either hard-coded numbers or symbols declared as constant). This saves time, and also reduces code size for translated programs.
- **database.e:** `db_select_table()` returns quickly if you happen to select the table that's already selected. This can save a lot of time in situations where it's not convenient for the application to keep track of which table is currently selected. Thanks to Derek Parnell.

### Enhanced Features



- The **shrouder** will now supply a default **#!** (shebang) line at the top of the **.il** file, if none was specified. Thanks to Kenneth Rhodes.
- On **Linux**, the **shrouder** will give the **.il** file execute permission. Thanks to Kenneth Rhodes.
- On **Linux/FreeBSD** the **Binder** no longer creates bound executable programs with a ".exe" extension. Now, no extension is used. This is compatible with the **Translator**, and more in keeping with the standard on **Linux/FreeBSD** systems. Thanks to Jerry Story.
- database.e**: Extra checks were added to some routines to give a more meaningful error message when no table has been selected yet. Thanks to Derek Parnell.
- Several small improvements in the Source Code product and its documentation make it easier to compile using Open Watcom and other compilers. Thanks to Jean-Marc Duro.
- The main source file is now closed right after the front-end finishes. This allows a program to edit or otherwise modify its own source.
- The initial random seed value is now chosen more randomly on **Windows**. Thanks to Akusaya.

### DOS/Windows Installation Program

- The install program will set up ed.bat to run ed.ex with ex.exe on ME/98/95, and exwc.exe on NT/2000/XP.
- The **install** program will not change any existing file associations for Euphoria file types. (If you install Euphoria in a new directory, you might have to change these yourself.) Thanks to Juergen Luethje and others.
- The Start Menu links created by the **install** program were wrong: exw was incorrectly pointing to **ecw.exe**, and ex was incorrectly pointing to **ec.exe**. Thanks to Akusaya.
- It's ok now to have a blank in the path of the installation directory: e.g. "C:\Program Files". Thanks to Rich.

### Bug Fixes

- bug fixed**: When building a Linux/FreeBSD .so file from Euphoria code that contained **call\_proc()** or **call\_func()**, an undefined "\_\_stdcall" symbol appeared in the C file by mistake. Thanks to Matt Lewis.
- bug fixed**: The opening message displayed by the Binder on Linux/FreeBSD and on Windows was wrong. Thanks to Kenneth Rhodes and Igor Kachan.
- bug fixed**: **routine\_id()** was checking for symbols coming later in the code. This happened in the Interpreter only, and only at the top-level, not inside a routine.

This fix requires .il files shrouded with 2.5 alpha to be reshrouded with 2.5 beta or later. Bound files are not a problem.

- bug fixed**: A complex expression on the left hand side of an assignment, involving nested use of \$,

could crash. Thanks to Andy Serpa.

- bug fixed - Translator:** A call to `routine_id()` would sometimes search past the end of a list and start reading garbage in memory. In most instances this did not cause a problem, but in some cases it could cause a crash. Thanks to Kenneth Rhodes.
- bug fixed - Translator:** In a rare situation, the 2.5 alpha Translator could crash while attempting to optimize some code.
- bug fixed - Translator:** In a rare situation, at the end of execution of a particular routine, the translated code would fail to free some space used for a temporary value. Thanks to Wolfgang Fritz.
- bug fixed:** After a compile error in `exw` you would have to hit Enter, then a message would say: "Press Enter...". (Things worked properly for run-time errors). Thanks to Tone Skoda.
- bug fixed:** An incompatibility existed between the way that `abort(1)` was handled in the Translator vs the Interpreter. Thanks to Juergen Luethje.
- bug fixed:** `save_text_image()`, when run by `exw` or `exwc`, occasionally returned some garbage in the upper bits of the character attributes. Thanks to Brian Broker.
- bug fixed:** The CAPS lock key generated a key code when using `ed` with `exwc`. The key had the desired effect, but it caused `ed` to display a message at the top of the screen. `ed` will no longer display the message. Thanks Alex Caracatsanis.
- bug fixed:** `routine_id("")` would crash the Binder and Translator. Thanks to Antonio Alessi.
- bug fixed:** A line longer than 200 chars crashed the `trace` display. Thanks to Thomas Betterly.
- bug fixed:** Symbols declared as constant were not defined in the `trace` window. Thanks to Matthew Lewis.
- bug fixed:** `printf()` format strings were wrong in two error messages in `execute.e`. Thanks to Derek Parnell.
- bug fixed:** The caret (arrow) symbol, displayed in compile-time error messages, was shifted one position to the right, when compared with version 2.4. This has been corrected. Thanks to Juergen Luethje.
- bug fixed:** For compatibility with the official Interpreter, routine id's now start at 0 instead of 1 in the PD source interpreter.

## Documentation

- A paragraph was added to "Scope" discussing how you can override predefined routines with your own variables and routines. Thanks to Mike.
- A warning was added to `open()` about the use of CON and CON.\*, and other file names that are reserved by DOS. Thanks to Igor Kachan.
- The fact that programs are now fully parsed before being executed was explained better in the

## Reference Manual.

- Obsolete uses of the phrase "Complete Edition" were removed from a few places in the documentation. Thanks to Juergen Luethje.
- Various documents were fixed where miscellaneous 2.4 information has now become now obsolete.
- The Reference Manual said that the limit on nested include files was 10 deep. It should have said 30 deep. Thanks to Darkvincentdude.
- In the PD source interpreter, **eu.exe**, there was an error in the comments at the top. Thanks to Akusaya.
- The fact that **crash\_message()** is now only useful for run-time errors was made clear. Thanks to Juergen Luethje.
- A bad HTML link to the WIN32 API .hlp file was fixed. Thanks to "Bro George".
- Many other minor clarifications were made.

## Version 2.5 Alpha Release November 15, 2004:

### New Features

- **Source code for a complete Euphoria interpreter**, 100% compatible with the official RDS **Interpreter** on all platforms, is now provided. See **euphoria\source**. It's written 100% in Euphoria and has the same front-end as the official RDS **Interpreter**. Since it's Public Domain, it may be used and modified for any purpose, including commercial or closed-source applications. It can be bound into a single executable file using the **Binder**, or translated with the **Euphoria to C Translator** to get a much faster executable.
- The official **Euphoria Interpreter** was re-written as 30% Euphoria (front-end), and 70% hand-coded C (back-end). It was formerly 100% C, and the boundary between front-end and back-end was fuzzy.
- The **Euphoria Translator** has been re-written as 100% Euphoria. It was formerly 100% C. It uses the same front-end as the **Interpreter** and the **Binder**. This reduces future maintenance costs, and helps catch subtle bugs (subscript out of bounds, uninitialized variable etc.)
- The free Public Domain **Interpreter** (**ex.exe**, **exw.exe**, **exu**), now includes full support for **trace()** beyond 300 statements, and for **profiling**. Thus there is no longer a registered "Complete Edition" interpreter. However, registration is still required for the **Binder** / **shrouder**.
- A new **Binder** was developed (100% in Euphoria). Since it shares the same front-end with the **Interpreter** and **Translator**, possible incompatibilities are avoided, and the future maintenance cost has been greatly reduced.
- A bound program can now provide a complete **ex.err** dump, without including any lines of source code in the **.exe** file.

- Bound programs now start up immediately with no parsing required.
- The **Binder** for **Windows** has a **-con** option, so you can make a console application.
- A separate **Interpreter back-end executable** was developed for use with the **Binder**. It executes intermediate code (**.il** files), and is smaller than the **Interpreter**, since it has no front-end (parser). This reduces the size of bound programs.
- A new **\$** symbol represents the length of the current sequence. Instead of saying `s[length(s)]`, you can say `s[$]`. Instead of saying `s[1..length(s)-1]`, you can say `s[1..$-1]`. More complicated expressions like `s[i][$-5..$-1]` are also possible.

**Note:** In order to implement this feature properly, a small incompatibility with version 2.4 was introduced. A change was made to the order of evaluation of expressions in a multiply-subscripted assignment statement:

```
lhs_var[lhs_expr1][lhs_expr2]... = rhs_expr
```

This change will cause a problem only if you are trying to modify `lhs_var` via a function call, during the same statement that is assigning to `lhs_var`. You should change your code to perform any such function calls before you perform the subscripted assignment, e.g.

```
temp = rhs_expr
lhs_var[lhs_expr1][lhs_expr2] = temp
```

The same would apply if one of the left-hand-side expressions contained a function call that tried to modify `lhs_var`.

This situation is very rare (and pretty weird), but a few examples have been found.

- Programs now have the ability to get control after a crash (a Euphoria-detected or machine-detected run-time error). Using **crash\_routine()** you can specify the routine id's of one or more Euphoria routines to be invoked when something goes wrong. You can save critical data to disk, inform the user, inform the programmer, etc.
- The library routines: **put\_screen\_char()**, **get\_screen\_char()**, **save\_text\_image()** and **display\_text\_image()** have been implemented for **Windows text-mode** consoles. These functions now work across all platforms. Suggested by C.K. Lester.
- Translator:** All the files you need for translating Euphoria programs to C are now included in the main download packages for **DOS/Windows** and **Linux/FreeBSD**. You just need one or more of the supported free C compilers.
- Translator:** The free Public Domain **Translator** (**ec.exe**, **ecw.exe**, **ecu**) now includes Euphoria statements as comments in the C source. It also supports **trace(3)** for debugging. When you register, you will eliminate the initial message and delay on compiled programs.
- Translator:** Some new optimizations will speed things up, and also reduce the size of the C code. For example the code block:
  - `if expression then`

- `...`
- `many statements of Euphoria code`
- `...`
- `end if`

will be completely deleted if the **Translator** can tell that *expression* will always be false. This can lead to further optimizations, such as a whole routine being deleted because there are no more calls to it. An example where this is useful is when there is different code for different platforms, e.g.

```
if platform() = DOS then ...
```

**while** *expression* and **elsif** *expression* can also be optimized in the same way. When the expression is always true, the **if/while/elsif** test and jump are eliminated.

- **Translator**: A couple of additional small improvements were made in converting the source from C to Euphoria. They allow for slightly better optimization.
- **Translator**: A minor adjustment lets the Translator produce slightly fewer C files, while reducing the slight risk of creating a huge C file that's too big for the C compiler to handle. Also, some empty .c files that were sometimes created, have now been eliminated.
- **Translator**: The Translator for Windows has a new **-con** option for making a console application. It's supported for each of the three Windows C compilers.
- **Translator**: After successfully building a .exe (or .dll) file, **emake.bat** will delete all the .c and .h files that were created by the Translator. This reduces clutter in the directory.
- The **install** program for **DOS/Windows** automatically creates **exwc.exe**, as well as **exw.exe**.
- **ed**: **ed.bat** has been changed. It now runs ed.ex using **exwc.exe**. Instead of using control-Home/control-End to move to the top/bottom of a file, use control-t/control-b. On **Windows** XP, this change gives **ed** faster keyboard response, and full support for long filenames.
- **ed**: A call to **free\_console()** is made at the very end, so you won't see "Press Enter", as occurs on some **Linux** systems.
- The **Interpreter** **exw** no longer brings up a blank console window while performing **trace(3)**.
- **pretty\_print()** has a new option to limit the number of lines of output.
- Both types of slashes (backslash, forward slash) are now fully supported in pathnames on **DOS** and **Windows**. Backslash is the standard, but since Windows also supports forward slash in most cases, Euphoria also now supports either slash in all situations, not just a few situations as before. On **Linux** and **FreeBSD** only forward slash is supported.
- The **-m486** option was removed from the **gnuexu** batch file used in building the Euphoria interpreter on **Linux**. It causes warnings and is not necessary.

## Bug Fixes

- **bug fixed**: **with profile** and similar **with/without** statements caused an error when followed immediately by a comment, with no whitespace before the start of the comment. Thanks to Daniel Kluss.

- **bug fixed:** **ex.err** would sometimes report the wrong value for a for-loop variable, when the for loop was used during a recursive call. Thanks to Pete Lomax.
- **bug fixed:** On Windows, using either the Interpreter, or translated code compiled by WATCOM, **dir()** caused an exception when given an argument that ended with "\*.". This was due to a bug in the WATCOM routine used to implement dir(). Thanks to Daniel Kluss.
- **bug fixed:** In 2.4, on ME/98/95 the argument to **dir()** could no longer be a long file (or directory) name. This used to work in 2.3. (Note: On XP it can't be long). Thanks to Rangi and Juergen Luethje.
- **bug fixed - Translator:** On Windows when you create a Euphoria .dll, the global routines need to be called using the `__stdcall` convention. However indirect calls to those routines via **call\_func()** and **call\_proc()** from within the same .dll, were using the `__cdecl` convention. Thanks to Andy Serpa.
- **bug fixed:** An error message referred to "printf()", when it should have referred to "sprintf()". Thanks to Al Getz.
- **bug fixed - safe.e:** Poking an empty sequence into a 1-byte block of allocated memory caused a false-alarm error message. e.g. `allocate_string("")` Thanks to Bernie Ryan and Josef Jindra.
- **bug fixed:** If you called **repeat()** with a repetition count that was extremely close to the maximum size of a Euphoria integer (1073741823), you would get a machine exception. You will now get an "Out of memory" error from Euphoria. Thanks to Bob Elia.
- **bug fixed - Translator:** In some cases, when using the Translator to build a dll, WATCOM C would give the error message "statement required after label" near the end of `EuInit()`. Lcc also complained. Borland C did not complain, and things worked fine. Thanks to Matthew Lewis.
- **bug fixed - Interpreter:** In programs larger than a couple of thousand statements, an assignment operation with subscripting on the left might cause a machine exception. The chance of this happening was about 1 in 1000. Any change to the program that shifted the offending statement's position in memory would cause the bug to go away. For example,
  - `x[i] -= y`  
This bug has been lurking since January 1999 (version 2.1 alpha). The Translator was not affected. Thanks to Pete Lomax.
- **bug fixed:** On Windows, Linux and FreeBSD, when reading standard input, and an input line was longer than 1040 characters, **gets()** would set the 1041st character to 0. DOS was OK, and files other than standard input were ok.
- **bug fixed - EDS:** When **db\_open()** opened a database file but failed to secure an exclusive or shared lock on the file, it was neglecting to close the file before returning a failure code.
- **bug fixed - ttt.ex:** A moderately-long human name would damage the board grid. Thanks to Igor Kachan.
- **bug fixed:** The line number reported for run-time errors occurring in a short-circuited if-statement could be off slightly. Thanks to Kat.

- bug fixed - Translator:** The range of return values assumed for `getc()` was [0,255]. It should have been [-1,255].
- bug fixed - Translator:** The generated C code failed to free the storage for the for-loop variable when a return statement was executed inside a for-loop, and the loop variable had a non-integer value. A similar problem could occur when there was an **else**, **exit** or **end while** immediately after the for-loop, and the loop variable value was not an integer.

## Documentation

- A document on [CGI Applications in Euphoria](#) has been added.
- Thanks to Kenneth Rhodes, `db_rename_table()` is now documented. This had been accidentally left out of the **EDS** documentation.
- It's now mentioned that format strings in `pretty_print()` could have additional text along with the format. Thanks to Juergen Luethje.
- The Reference manual now says that hex literals, such as #FFFFFFFF, are never considered to be negative numbers. Thanks to Juergen Luethje.
- `sprintf()/printf()`: It's documented that -1 prints as FFFFFFFF in %x format. Thanks to Juergen Luethje.

## Version 2.4 Official Release July 3, 2003:

- `define_c_proc()` and `define_c_func()` have been extended, so in addition to defining C routines in external .dll's and shared libraries, you can now define the parameters and return value for a machine-code routine that your program pokes into its own memory. You can call the machine-code routine using `c_proc()` or `c_func()`. Thanks to Daniel Kluss.
- Performance Improvement: `get4()` and `put4()` in `database.e` have been speeded up slightly. They are very important to the overall speed of **EDS**. Thanks to Derek Parnell.
- Performance Improvement: `get_bytes()` is now much faster when the number of bytes requested far exceeds the number of bytes remaining in the file. Thanks to Pier Feddema.
- Translator:** When translating a huge Euphoria routine (many hundreds of Euphoria statements), the Translator will now output calls to a dereference routine, rather than using in-lined C statements. This reduces the chance of exceeding a size limit imposed by the C compiler (especially Watcom C). It also reduces the .exe size. Since the dereference routine is more likely than the in-lined statements to be in cache, the speed difference is not that great.
- Interpreter Source Code:** In `watexw.bat`, `runtime windows=4.0` was added to the link command for building `exw.exe`.
- There are now some checks for invalid argument type and invalid return type in calls to `define_c_func()` and `define_c_proc()`.



- Some syntax error messages are now more descriptive when namespace identifiers are involved.
- The filesort.ex tutorial program was altered to make it more usable under Linux and FreeBSD.
- By default, **safe.e** now does a less-strict, "edges-only" check for memory corruption, when the platform is **WIN32**. Windows programs often access memory that was not allocated using Euphoria's **allocate()**.
- bug fixed**: When a literal floating-point constant in the Euphoria program was larger than about 1e308, the Translator would output "inf" in **init.c**. This caused the C compiler to issue an undefined symbol error. Thanks to Juergen Luethje.
- bug fixed**: In a rare case, the Translator was failing to emit C code to make a copy of a sequence with multiple references to it, before overwriting an element of that sequence. Thanks to Juergen Luethje.
- bug fixed**: In certain cases, when a Euphoria program exchanged Euphoria data with a **.dll** written in Euphoria, the data might not be freed (until the program terminated). Thanks to Wayne Overman (Euman).
- bug fixed**: If you used "asm" in your program, as a private variable or parameter name, the Translator would use "\_asm" in the C code. This was not acceptable for some C compilers. The Translator will now avoid using "\_asm", as well as "\_try", "\_Seg16", "\_stdcall" and several other single-underscore names that are reserved by various C compilers. Thanks to George Papadopoulos and Matt Lewis.
- bug fixed**: If the HOT\_KEYS parameter in **ed.ex** was set to FALSE, then **Esc h Enter** would not bring up the help prompt. Thanks to J. Brown.

## Version 2.4 Beta-test Release April 10, 2003:

This release updates the **Euphoria Interpreter**, the **Euphoria To C Translator**, and the **Interpreter Source Code** products, for **Windows**, **DOS**, **Linux** and **FreeBSD**.

### New Features

- bind** and **shroud** now have an option **-out** for specifying the output file, so you won't be prompted for it. Thanks to Jonas Temple, Rusty Davis and others.
- bind** and **shroud** now have an option **-quiet** that suppresses normal messages and statistics, and therefore eliminates the window that normally pops up. Only errors are reported. Thanks to Jonas Temple.
- The namespace error message that's issued when you refer to a global symbol that's defined in two or more files, now gives you a list of all the files where that symbol has been defined. Thanks to Derek Parnell and Irv Mullins.
- Translator**: In many cases, the C code generated for **remainder()**, integer multiplication, and **compare()** is smaller and faster.
- exw**, **ecw -wat**: Deallocation of space for huge numbers of small objects (atoms or small sequences) is



much faster than in 2.4 alpha. Thanks to Andy Serpa. (Note that *allocation* of huge numbers of small objects in **exw**, or **ecw -wat**, became much faster in 2.4 alpha, and remains much faster.)

- When an **ex.err** file is created, any warnings issued against the program will be listed at the end of the **ex.err** file. Thanks to Al Getz.
- The file name is now included in the warning message that you get for some of the common warnings (variable not used, variable not assigned to). Thanks to Al Getz.
- **New Icon:** On **Windows**, Euphoria include files are now labelled with a gray-scale version of the Euphoria E icon. This lets you easily distinguish the executable Euphoria files from the include files. Thanks to Wolfgang Fritz.
- Out-of-bounds floating-point subscript values were being reported after rounding down to an integer. Now the value before rounding is reported.
- **Euphoria Database System (EDS):** **db\_rename\_table()** now checks to see if the target table name already exists, before it renames a table. Thanks to Mike Nelson.
- The first value returned by **rand()** (in the absence of **set\_rand()**) is now more "random". Thanks to Aku.

## Bug Fixes

- **bug fixed:** Due to a change made in 2.4 alpha, the **dir()** routine for Borland and Lcc was concatenating the file attributes characters (if any) to the file name field. Thanks to Dr. Juan R. Herguijuela.
- **bug fixed:** No error message was issued when ',' was followed immediately by ')' in a routine's parameter declaration list. Thanks to Brage Moris.
- **bug fixed:** Huge positive out-of-bounds subscripts (over 2 billion) were reported as huge negative values.
- **bug fixed:** **repeat(0, size)**, where size was a huge positive floating-point number, incorrectly reported: "repetition count must not be negative". Now it reports: "repetition count is too large". Thanks to Martin Stachon.
- **bug fixed:** **machine\_proc(x, 5)**, where x was a huge floating-point number, incorrectly reported: "an integer was expected, not a sequence". Now it reports: "The first argument to machine\_proc/func must be a small positive integer". Thanks to Martin Stachon.

## Version 2.4 Alpha-test Release February 21, 2003:

### New Features

- Most machine-level exceptions (peek/poke to bad addresses etc.) in both the main program and in .dlls, are now caught by **exw** and **exu**, and reported in the usual way, with a full traceback and **ex.err** variable dump. This is a great improvement over the cryptic machine-level messages you used to get (and still get when using compiled languages, and most interpreted languages) about "segmentation

violation", "illegal instruction" etc.). Thanks to Martin Stachon.

- In addition to **\_\_stdcall**, the C **\_\_cdecl** calling convention is now supported for calls to C routines in .dll's and also call-backs to Euphoria routines from C code.
- Euphoria's support for DOS long filenames has been extended to Windows XP.
- The **trace** screen shows you large sequences in pretty-print display on a separate screen. You can scroll through the whole sequence.
- **pretty\_print()** was added to **misc.e**. It lets you display Euphoria objects with a nice, readable structured display, and many formatting options.
- Pretty-printing of sequences is now done in **ex.err**, the **?** command, and **db\_dump()**.
- **Euphoria Database System (EDS)**: **db\_rename\_table(name, new\_name)** was added to **database.e**. This routine was submitted by Jordah Ferguson, and included with only trivial changes.
- **Linux/FreeBSD**: **system()** no longer initializes curses when there is no console window yet. Thanks to Daniel Johnson.
- The number of levels of nested include files has been raised to 30 (from 10). Thanks to Tone Skoda.
- Include statement path names can have double-quotes around them, so paths containing blanks can be handled correctly. This was actually implemented for 2.3 but never documented.
- **exw.exe**, and any executables produced by the **Translator** with Watcom, now have the subsystem set to 4.0 instead of 3.1. This improves the appearance of GUI's in some cases. The utility **make31.exw** will create a version of **exw.exe** that supports Windows GUI 3.1 as before, in the unlikely case that there are compatibility problems with Euphoria 2.3. Thanks to H. W. Overman, Brian Broker and others for recommending this change.
- **makecon.exw** - will create a version of **exw.exe** that operates as a console application - no console window is popped up, and **stdin/stdout** can be redirected
- **trace(1 2 and 3)** are now allowed with **bind -clear** (still not allowed with **shrouded bind** for security reasons). Thanks to Jonas Temple.
- **Translator**: You can now make a Euphoria .dll using Lcc and interface it with interpreted programs running under **exw**, and translated programs using Borland and Watcom. Previously, the main program had to also be compiled with Lcc.
- The **Translator** no longer uses the **-m486** or **-mpentium** options available with GCC and DJGPP. These options were causing warnings, and the C compiler apparently sets the machine model correctly by itself. Thanks to Kenneth Rhodes.
- The **Translator** will now perform automatic calls to user-defined types, in the unusual case where the type routine has side-effects (it sets global variables, performs I/O etc.). Thanks to Andy Serpa.
- **euphoria\demo\bench** compares the **Euphoria Interpreter** and the **Euphoria To C Translator** against more than 20 other interpreted languages.

## Porting Activity

- We ported the **Interpreter** and **Translator** to **FreeBSD**. The source now has several C #ifdef's for FreeBSD.
- Andy Cranston has ported Euphoria to HP Unix, and he plans to do Sun Unix.

## Optimizations

### - Interpreter -

- Typical large slices are faster. About 30% faster for slices from about 100 to 50000 in length. (Overhead dominates for smaller slices, and lack of caching affects larger slices.) This assumes the slice is mostly integers (usually true), and an actual copy of the data is made (usually true since v1.4b).
- Statements that contain multiple **&** concatenations are much faster.
  - e.g. instead of:
  - **result = a & b & c**
  - being evaluated as:
  - 1. copy a and b into temp
  - 2. copy temp and c into result
  - (a and b are effectively copied twice!)
  - 
  - We now do:
  - 1. copy a and b and c directly into result

So there is less copying of data and fewer temp sequences to create. The more **&** operators in an expression, the greater the speed-up. e.g. with 3 **&** operators some of the data was copied 3 times, etc. Jordah Ferguson pointed out that this was slow.

- The time overhead involved in calling and returning from a Euphoria call-back routine has been reduced by about 10 percent.
- In **exw** and **ecw -wat**, allocation of space for large numbers of objects is faster. It can be tremendously faster when hundreds of thousands or millions of objects are involved.
- Better UPX compression has chopped a few K off **exw.exe** vs. 2.3 (even though new code was added). Thanks to Wolfgang Fritz

### - Euphoria Database System -

- Keys and records are read faster due to a faster **decompress()** routine. Almost twice as fast when the key or record data to be retrieved consists mainly of sequences of characters or small integers. This case is quite common.
- Allocating new space in a database is much faster, up to 4x faster, especially in large databases with a large list of free blocks
- Inserting and deleting records in huge tables is now much faster. Combined with the speeded-up slices in Euphoria 2.4, **database.e** is now about 25% faster for a table with 10,000 records and over 3x

faster for a table with 100,000 records. This really only matters if you are trying to insert/delete hundreds of records per second. In the typical case of a human operator entering data via GUI, you would never notice the insert/delete time for one record (a few milliseconds). Derek Parnell pointed out the slowness.

- db\_select\_table()** is significantly faster.
- get4()** is faster which speeds up everything.

#### - Other Optimizations -

- bytes\_to\_int()** in **machine.e** is now more than twice as fast.
- gets()** is about 5% faster
- sort()** and **custom\_sort()** are a few percent faster. Thanks to Ricardo Forno for tweaking the Shell sort algorithm.
- Several additional optimizations have been added to the **Translator**. It produces executables that are faster and smaller than version 2.3. The 2.4 Translator has been successfully tested on hundreds of thousands of lines of Euphoria code, and there are currently no known code generation bugs. Some Translator benchmark results are in **euphoria\demo\bench**.

#### Bug Fixes

##### - Source Code -

- bug fixed:** The **gnubsd** batch file referred to **syncolor.c** and **syncolor.o**. (**gnuexu** was ok.)
- bug fixed:** Karl Bochert pointed out a necessary C coding change to make **poke()** work with the latest version of Lcc. The change corrected the **Translator** (with recent versions of Lcc) and the **Interpreter Source Code** (compiled with Lcc)

##### - Interpreter -

- bug fixed:** A crash might occur in situations where a call-back routine indirectly called itself recursively. Thanks to Matthew Lewis and George Papadopoulos.
- bug fixed:** In for-loops at the top-level of a program (outside of any routine), that incremented the loop variable by an integer other than the default of +1, **end for** was taking up to 15x longer than necessary due to a bug fix that was made back in November 1999. Only the **end for** itself was slow, not the code contained in the body of the loop. Antoine Tammer detected this.
- bug fixed:** On XP, when you open a new DOS window that has more than 25 lines, the Virtual DOS Machine (VDM) is at first confused about the true number of lines. The first time (only) when you ran a Euphoria program in that window, if you ran it near the very bottom of the screen, the output might disappear, or the VDM or Euphoria might report an error etc. Euphoria (**ex.exe**) now detects the rare cases when VDM is confused and clears the screen, which clears up the confusion. A similar problem existed on NT, and was fixed a few years ago.
- bug fixed:** The interpreter was referring to "call back from Windows" in **ex.err**, even on Linux or

FreeBSD. It now says "call-back from external source" on those systems. Thanks to Pete Eberlein.

- bug fixed:** When an include file couldn't be found, the error message referred to "euphoria\include". It now uses %EUDIR%\include
- bug fixed:** An error message will no longer be generated on any platform for **without profile\_time**. Thanks to Alan Oxley.

### - Translator -

- bug fixed:** When assigning the result of an arithmetic calculation (typically multiply) involving two integers, to a variable declared as atom, where the atom variable had already (in the same basic block) been assigned an integer value, the Translator might not output any code to check for integer overflow (result outside of +/- one billion). This could cause a crash. Thanks to Andy Serpa.
- bug fixed:** DJGPP strip.exe command in **emake.bat** would fail on XP,2000 due to a bug in DJGPP. Now **emake.bat** has: **SET LFN=n** to work around the bug in strip.exe
- bug fixed:** Translated code compiled with Borland C was not producing INF's and NAN's, like Watcom and Lcc. Rather, it was crashing when a floating-point overflow (over 1e308), or an undefined f.p. result was calculated. The **Interpreter Source Code** was also corrected for those who wish to compile **exw.exe** using Borland. Thanks to Andy Serpa.
- bug fixed:** In the first basic block of a Euphoria routine (i.e. before any control-flow statements), **peek4u()**, **peek4s()**, and the "add integer 1" operation, would sometimes neglect to check for possible 31-bit integer overflow when assigning to a private variable declared as atom, unless the variable had been previously initialized. Thanks to Mike Duffy.
- bug fixed:** In some cases, when assigning a sequence element to a variable declared as integer, and known to have an integer value at this point, the case where the element was an integer value stored in C double form was not handled correctly.
- bug fixed:** In rare cases, the translator might output two unary minus operators in a row, which would be parsed by a C compiler as the C decrement operator "--".
- bug fixed:** Euphoria .dll's were not always correctly freeing storage allocated by the main program, and vice versa. Memory could be wasted, and you might get a machine-level crash. Thanks to H. W. Overman.

**Note:** Due to this fix, any Euphoria .dll's created with the Translator version 2.3 or earlier, must be re-translated with 2.4, and re-compiled, in order to interface with the Euphoria 2.4 (or later) interpreter or translated code. New .dll's created with version 2.4 or later, will not work with the interpreter version 2.3 or earlier, except in trivial cases.

- bug fixed:** The **sleep(x)** function was only sleeping for x milliseconds when using the Lcc run-time library. It now sleeps for x seconds, to conform with the Euphoria documentation for **sleep()**. Thanks to Wolfgang Fritz.
- bug fixed:** On some versions of Linux, a translated/compiled Euphoria program would crash if standard output was redirected, e.g. for CGI

- bug fixed:** On some versions of Linux, a translated/compiled Euphoria program would crash if `machine(M_GET_SCREEN_CHAR, {row, col})` were called.
- bug fixed:** In some cases the code was not correct when an integer variable was assigned the unary minus of an atom variable.
- bug fixed:** In a very rare case, an uninitialized value in memory might be used to determine if a literal floating-point value should be treated as an integer or not. Incorrect code could result.

#### - Binder -

- bug fixed:** The binder would crash after seeing a comment with no new-line character, just EOF, on the last line of a file. Some versions of Win32Lib.ew had this. Thanks to Henri Goffin.
- bug fixed:** The usage report of **bind/shroud** still said "-scramble", instead of "-clear" and had other errors for Linux/FreeBSD. Thanks to Chris Bensler.
- bug fixed:** **bind/shroud -clear** might neglect to rename a private variable, when an earlier local variable gets renamed into the same name. Thanks to Pete Lomax.
- bug fixed:** When an include file was missing `\n` on the last line, **bind/shroud -clear** might neglect to leave some whitespace before the next word in the main file. Thanks to Pete Lomax.
- bug fixed:** If you defined a constant that was never used, and it was defined using an expression that contained a binary minus operator, a syntax error could occur in the bound or shrouded file that you create. Thanks to Chris Bensler.

#### - Library Routines -

- bug fixed:** `walk_dir('/',...)` would fail on Linux. Thanks to Ricardo Forno.
- bug fixed:** `db_compress()` of EDS had an error if the database file or it's path contained a blank character. A blank is ok now on all platforms except DOS. Thanks to Virtual B.
- bug fixed:** `wildcard_file()` in `euphoria\include\wildcard.e` is now case sensitive on Linux/FreeBSD (but is still case insensitive on DOS/Windows). Thanks to Jeff Fielding.
- bug fixed:** `dir()` was not always reporting file sizes greater than 1 Gb correctly. It now handles up to 4 Gb. Arithmetic performed on the file size would probably fail. Thanks to Ricardo Forno.
- bug fixed:** `where()` was not always reporting file positions greater than 1 Gb correctly. Arithmetic performed on the position would probably fail. It now handles up to 2 Gb.
- bug fixed:** `ex, exw, ec -wat, ecw -wat:` The `dir()` function for DOS and Windows was not handling wildcards correctly when a backslash appeared at the end of the file or directory path. Thanks to Juergen Luethje.

#### - Namespace Feature -

- bug fixed:** If you tried to declare a new variable, using a namespace qualifier, e.g. **integer y:x** (which is illegal) it wouldn't complain, and would just ignore the "y:" part, as long as y:x was a routine (not a

variable) in an earlier file. This is now caught as an error. Thanks to Martin Stachon.

- bug fixed**: When declaring the type of a parameter, using a global type that was defined in more than one file, you would get an unclear error message stating "a type is expected here". Now you'll get a message pointing out that the type requires a namespace identifier to resolve it. Thanks to Juergen Luethje.
- bug fixed**: The error message that you get if you specify a namespace qualifier and a '.', but you neglect to follow it with a properly-formed identifier, has been improved. Thanks to Carl White.

#### - Demo Programs -

- bug fixed**: In the **window.exw** demo program, **allocate(64)** and **allocate(16)** caused a storage leak. Thanks to Wolfgang Fritz and Jordah Ferguson.

#### - Trace/Debug -

- bug fixed**: When **trace(0)** was executed, followed later by **trace(1)**, with no screen I/O in between, the values of some variables on the trace screen were not updated. Thanks to Ricardo Forno.
- bug fixed**: **with trace** / **with profile** / **trace(3)**, used all together, produced garbage source lines in **ctrace.out**. Thanks to Kat.

### Version 2.3 Official Release February 11, 2002:

This release updates the Euphoria Interpreter, the Euphoria To C Translator, and the Interpreter Source Code product, for all platforms.

- The DJGPP version of the interpreter source code now uses GNU C's dynamic labels, just like the Linux version. This allows it to achieve full speed, without the need for any assembly-level tweaking. Thanks to Bernie Ryan.
- The Interpreter Source Code now includes an overview document describing how the interpreter works.
- In the Complete Edition, bind.bat and bindw.bat now use exw to run the binder/shrouder. This avoids problems with long filenames on some systems. A console window will appear while the binder/shrouder is running. Thanks to "void", Bruce Axtens, and others.
- bug fixed** : Due to a bug in the WATCOM 10.6 C library, the ex and exw interpreters, and code translated to C and compiled by Watcom, might get the wrong result from Euphoria's where() function when the file was opened in append mode, and the file pointer was in the uncommitted (not written to disk yet) portion of the file. The bug has been worked around by having Euphoria flush() the file in this particular case, before calling the Watcom routine. Thanks to Bob Elia.
- bug fixed** : A bug introduced in the binder for 2.3 beta, might cause a function call on the last line of a program to be ignored. Thanks to Wolfgang Fritz.
- bug fixed** : Several Euphoria files in the WIN32+DOS32 distribution had Linux-style line terminators



(\n only). This made them hard to view using NotePad and some other editors. Thanks to Pete Lomax.

- **bug fixed** : If "with type\_check" was turned on, ed.ex would get a type\_check failure when the Esc key was pressed. Thanks to Kenneth Orr.

## Version 2.3 Beta Release January 15, 2002:

This release updates the Euphoria Interpreter, the Euphoria To C Translator, and the Interpreter Source Code product, for all platforms.

- You can now override a Euphoria built-in routine with your own variable or namespace identifier of the same name. Previously this was only allowed for user-defined routines. Besides giving programmers more freedom, it will allow RDS to add new built-in routines in the future without breaking existing code.
- The warning about having multiple global symbols with the same name in different source files has been removed. It was felt to be unnecessary since you'll be asked to supply a namespace identifier if you actually make an ambiguous reference to a global symbol.
- You can now have a trailing \ (or Linux /) on the end of all directory names in **EUINC**, and you can have blanks in the directory name.
- To eliminate confusion, the binder/shrouder will now delete the output file if a fatal error occurs during the bind or shroud.
- Numerous improvements and corrections were made to the documentation. Thanks to Igor Kachan.
- Old, pre-ANSI function definitions in the interpreter source code, have been updated to ANSI-style, which is more compatible with C++.
- **bug fixed** : With DJGPP C, when compiling code produced by the Translator, or compiling the Interpreter source code, there was a memory allocation bug that could waste a bit of time, or, in rare cases, cause a crash.
- **bug fixed** : On Windows, using the Interpreter, or a Translated program, you were sometimes required to hit Enter twice to exit from a console window. Thanks to Tone Skoda.
- **bug fixed**: The Euphoria dir() function, as implemented for Lcc or Borland, did not handle directories properly when they had extra attributes set, such as READ\_ONLY. Thanks to Euman, who encountered the problem, and then showed how to fix the Interpreter source code for dir().
- **bug fixed**: You can now declare a namespace identifier with the same name as a built-in function, without causing a lot of errors. Thanks to Martin Stachon (although he recommended a different solution).
- **bug fixed - Binder**: Support for the new **EUINC** environment variable has now been added to the binder. Thanks to Ross Boyd.



- bug fixed - Binder:** Appending resource files to a bound .exe didn't work correctly when bind -clear was used. Thanks to Wolfgang Fritz.
- bug fixed - Binder:** When using the binder interactively, you would get a "variable not initialized" error when trying to substitute a different Windows icon. Thanks to Tony Steward.
- bug fixed - Binder:** In some cases the keyword "constant" would be dropped from the shrouded output, when the previous line of input had a constant declaration ending in ']'. Thanks to Ross Boyd.
- bug fixed - Binder:** When a general expression, (not just a string in quotes), was used as the argument to routine\_id(), local routines that were potentially the target of that expression, might have their names changed (unless -clear was used), thus causing routine\_id() to return -1 at run-time. Global routines were ok.
- bug fixed - Binder:** The binder/shrouder kept going even though an include file was missing. Thanks to Ross Boyd.
- bug fixed - Linux Binder:** The Linux search path bug for bound executables, supposedly fixed by the 2.3 alpha release, wasn't fixed properly. Fixed now. Thanks to Kenneth Rhodes.
- bug fixed - Linux Binder:** bindu -clear and shroud -clear with a file containing DOS/Windows-style \r\n line terminators gave you errors "illegal character" when your shrouded program was run, and "not bound correctly" when your bound file was run.
- bug fixed - Interpreter Source:** The link command for building the DOS interpreter with WATCOM C, listed a non-existent .obj file.

## Version 2.3 Alpha Release November 30, 2001:

This release updates the Euphoria Interpreter for WIN32, DOS32 and Linux. It also updates the Euphoria To C Translator for all platforms, and it introduces a new product - the Euphoria Interpreter Source Code.

- The registration incentives and prices have changed.
  - The Interpreter product now has just one option: WIN32, DOS32 *and* Linux, formerly \$59, for just \$39.
  - The Interpreter source code (minus a few registered features) is now available for \$49. See the [source license](#) and [register.doc](#) for more details.
  - The Translator continues to cost \$29.
  - When Public Domain users reach 300 statements, they will no longer lose the run-time error diagnostics. They will only lose the trace() facility.

[register.doc](#) has more details, including the requirements for getting a free upgrade.

- The Euphoria interpreter can now be built successfully by 6 different C compilers on 3 platforms.

- New *namespace qualifiers* eliminate naming conflicts between identical global symbols declared in different include files. Also, local symbols will now override global symbols of the same name, rather than causing an error. See [Scope Rules](#).
- The Complete Edition Interpreter product comes with a new 2-pass binder/shrouder that eliminates all unused routines and constants, resulting in smaller executable files. It also has an option to bind clear source, so you can get understandable error messages from your users.
- A new environment variable, EUINC, has been introduced. If present, it specifies a list of additional directories that will be searched for include files. The directory containing the main file is always searched first, then the directories in EUINC, then euphoria\include.
- The Interpreter now supports a new trace mode, trace(3). It logs all executed Euphoria statements to a file, so you'll be able to see the last statement that was executed at the time of any crash, as well as the 499 statements that preceded it. This is particularly helpful in the case of machine-level crashes where Euphoria is unable to write an ex.err file. Thanks to Matthew Lewis.
- The Euphoria interpreter can now pass Euphoria data - atoms and sequences, to .dll files coded in Euphoria and built by the Translator. Use the new E\_ types in dll.e
- On Linux we added RTLD\_GLOBAL on the dlopen() call. This lets you link successfully with more shared libraries.
- On Linux, when using the #! feature on the first line, to make your source file directly executable, you no longer need a .exu extension on your file. Linux programmers often prefer to have no extension on their executable files.
- Windows callback routines written in Euphoria can now take 9 arguments (was 8). Thanks to Matt Lewis.
- In addition to C\_DOUBLE (8-byte floating point), C\_FLOAT (4-byte floating-point) is now supported for arguments and return values from C routines. Thanks to David Guy.
- Windows .dlls opened with open\_dll() are now closed automatically when your program terminates. This avoids a tiny bit of memory leakage. Thanks to Euman.
- safe.e has a new option to check just the edges of registered blocks and not complain if other blocks of memory are used.
- get\_bytes() is 30% faster.
- allocate\_string() was speeded up. Thanks to Aku.
- The mydata.ex demo program now uses an EDS database.
- bug fixed - Translator:** When a global or local variable containing a sequence or floating-point number was assigned the result of a function, and at some point during the function call the global or local variable was overwritten, some corruption would take place, leading to a probable crash later in the program. Thanks to Sergio Gelli.

- bug fixed - Linux Interpreter:** programs of more than a few thousand lines had a chance (maybe 20%) of having one of their statements crash whenever it was executed.
- bug fixed - Interpreter:** `s[i][j]...[k] = s` caused the interpreter to crash. i.e. an assignment of an entire sequence to an element of an element of itself using 2 or more levels of subscripting. Thanks to Henri Goffin.
- bug fixed - Linux binder:** (Kenneth Rhodes) Bound executable programs stored somewhere on the search path would not run properly unless they were in the current directory or unless you specified the full path to the executable.
- bug fixed - Binder:** "with profile\_time" in a bound or shrouded program would cause a crash
- bug fixed - DOS Interpreter:** In `image.e`, `put_screen_char()` has been corrected to have: **if overflow > 0 then ...** instead of: **if overflow then ...**
- bug fixed - Interpreter:** A time optimization for slicing could in rare cases cause a lot of wasted space. The optimization has been adjusted to handle those cases. Thanks to Brian Clausing.
- bug fixed - safe.e:** `free()` and `free_low()` were not actually freeing the block of memory, and on Linux `free()` could cause a segmentation violation. The same routines in **machine.e** were ok.
- bug fixed - Translator for DJGPP:** A couple of minor differences from `ex.exe` in the display of text were corrected.
- bug fixed - define\_c\_var():** works on WIN32 too. The documentation was corrected.

### Euphoria to C Translator Official Release June 5, 2001:

- A couple of minor enhancements and bug fixes.

### Euphoria to C Translator Beta-2 Release May 8, 2001:

- You can now use the translator to make a .dll file.

### Euphoria to C Translator Beta-1 Release February 15, 2001:

- Some bugs were fixed. Smaller, faster C code is produced.

### Euphoria to C Translator Alpha-4 Release December 14, 2000:

- Pixel-graphics and mouse support was added for DJGPP C.

### Euphoria to C Translator Alpha-3 Release November 27, 2000:

- Some bugs were fixed. Some new features were added. Borland C++ is now supported.

### Euphoria to C Translator Alpha-2 Release October 26, 2000:

- Support for the Lcc C compiler for Windows was added. DJGPP C (DOS) is partially supported (no pixel graphics).

## Euphoria to C Translator Alpha Release September 6, 2000:

- The first release of the translator. It supports Euphoria version 2.2. It relies on WATCOM C for DOS and Windows, and GNU C for Linux. Prime sieve runs 3.2x faster than with the interpreter. Shell sort runs 4.5 times faster.

## Version 2.2 Official Release for WIN32+DOS32 January 14, 2000:

- Better error messages are now issued from the interpreter, and from the bind program, for some typical errors that users can make when binding a program.
- The documentation has been improved in a few places.
- The **window.exw** demo program shows how to load the Euphoria icon contained in exw.exe.
- Language War uses Jiri Babor's improved version of **putsxy.e**.

## Version 2.2 Beta Release for WIN32+DOS32 December 23, 1999:

Most of the new library routines developed for Euphoria 2.2 on Linux, have now been ported to WIN32 and DOS32. These are: sleep(), chdir(), flush(), lock\_file(), unlock\_file(), crash\_file(), get\_screen\_char() and put\_screen\_char(). For a description of these routines refer to the Linux release notes below, or [LIBRARY.DOC](#).

Some cross-platform bug fixes and other miscellaneous improvements were made during the Linux port. These bug fixes and improvements have now been ported back to WIN32+DOS32. See the Linux release notes (below).

In addition, the following improvements have been made specifically for WIN32 and DOS32:

- exw.exe** now contains a Euphoria icon that is displayed automatically by Windows. The icon was contributed by Gary Dumer. Registered users may change this icon when they bind a program.
- exw.exe** is now a compressed executable of just 73K (or so). It's compressed using the UPX compression tool for .exe files.
- ex.exe** has been updated with the latest release of the CauseWay DOS extender. A problem where CauseWay would sometimes limit you to 64Mb of memory under some DOS configurations has been eliminated, and a few other minor bugs were fixed.
- bug fixed:** The error traceback could sometimes crash or be misprinted when a type\_check failure occurred. It could only happen when 1 was added to an expression, and the non-integer result of the expression was assigned to a variable declared as integer.
- bug fixed:** If text\_rows() was called as the first routine needing a WIN32 console window, Euphoria would fail to set the new number of lines of text on the console.

## Version 2.2 Official Release for Linux November 22, 1999:

- **All platforms: bug fixed:** If a Euphoria routine called itself recursively from inside a for-loop, and at one level of recursion the for-loop counted *up* to an upper limit, and at another level of recursion the for-loop counted *down* to a lower limit, the for-loop would probably malfunction at one of the levels. Thanks to Delroy Gayle.
- The documentation was improved in many places, especially with respect to the Linux platform.

### Version 2.2 beta-test release for Linux October 22, 1999:

*Most of these features and bug fixes will also be made available in Version 2.2 for **WIN32 + DOS32**.*

- **platform()** has been moved from misc.e into exu to eliminate the function call overhead. platform() now takes zero time to compute. The compiler simply plugs in the appropriate constant value.
- **lock\_file()** and **unlock\_file()** have been added to allow multiple processes to share access to files. This can be important in CGI programming and other areas.
- **flush()** will force the contents of the memory buffer out to a file or device.
- **chdir()** will change to a new current directory and let you know if it was successful.
- **sleep()** will suspend execution of your program for a number of seconds, and let the operating system schedule another process.
- **put\_screen\_char()** will write a character and its attributes (colors etc.) to the screen.
- **get\_screen\_char()** will read a character and its attributes from the screen.
- **save\_text\_image()** now works on Linux (as well as DOS32). It copies a rectangular text image from the screen.
- **display\_text\_image()** now works under Linux (as well as DOS32). It writes a rectangular text image to the screen.
- The "short-circuit" warning now gives the filename and line number of the possibly short-circuited call. Minor clarifications were made in some other error messages as well.
- Minor improvements were made to **ed** and **search**.
- A portability problem in **how2reg.ex** was fixed.
- **exu** is compressed better. It's actually a bit smaller now, although functionality has been added to it.

### Version 2.2 alpha-test release for Linux August 24, 1999:

*Many of these features and bug fixes will also be made available in Version 2.2 for **WIN32 + DOS32***

- The documentation has been brought up-to-date to include **Linux**-specific information for library routines and Euphoria in general.

- There is now a Complete Edition for **Linux**, including **binding** and **shrouding**. See [register/register.doc](#).
- There is now **text mode** mouse support using [get\\_mouse\(\)](#). You must have **GPM server** running. It works in a text console or an **xterm** window.
- **Linux**: [define\\_c\\_var\(name\)](#) will return the address of a global C variable in a shared library.
- It was confirmed that you can call Euphoria routines from **Linux** C routines using exactly the same mechanism as in **WIN32** Euphoria. See [euphoria/demo/linux](#).
- An example of creating your own shared library routines and calling them from Euphoria was added. See [euphoria/demo/linux](#).
- **All platforms**: [crash\\_file\(file\\_name\)](#) will cause diagnostic messages to be written to **file\_name** instead of **ex.err**. You can use [crash\\_file\("/dev/null"\)](#) to get diagnostics on screen but not in a file. [crash\\_file\(""\)](#) means "no diagnostics" (to screen or ex.err).
- **Trace** mode in **xterm** now detects the **F1/F2** keys.
- [time\(\)](#) now reports real "wall-clock" time, not CPU time.
- **search**, **guru** and **cdguru** now place their output in your \$HOME directory instead of the current directory.
- **#!** is now restricted to just the first line of a file.
- **All platforms**: In **ed**, the **Esc n**, **Esc d**, **Esc f** and **Esc r** commands will immediately redisplay your last choice. You can press **up-arrow/down-arrow** to see other choices, or clear the choice. If you start typing without editing, it will clear the choice and take your new input.
- [free\\_console\(\)](#) will set the terminal parameters back to normal. Normally, when running a Euphoria program the parameters are set the way that **curses** wants them and they are set back to normal when the program terminates. If your program needs to terminate in a strange way (other than calling [abort\(\)](#)), [free\\_console\(\)](#) should be called first.
- **bug fixed**: [get\(\)](#) now considers '\r' to be a whitespace character. This is important when reading DOS files.
- **All platforms**: **bug fixed**: It was not immediately issuing a **type\_check** failure when 1 was added to an integer variable that was set to the maximum value for an integer (1.07 billion). Thanks to Jeff Fielding.
- **All platforms**: **bug fixed**: It was not always detecting an improperly-formed exponent on a floating-point number. Thanks to Lionel Wong.
- **All platforms**: The performance of the storage allocator has been improved in certain cases. A bug that could cause the interpreter to crash when you are almost out of memory has been fixed.

**Version 2.2 pre-alpha #4 for Linux July 15, 1999:**

- You can call C routines in **Linux** shared libraries (.so files). See [euphoria/demo/linux/callc.exu](#) for some examples.
- If your program does not output anything to the **xterm** window, **exu** will not issue the "Press Enter" prompt.
- **All platforms: ed** now lets you recall previous top-line command text using **up-arrow** and **down-arrow**, similar to doskey in DOS and the shell history in Linux. This works with any strings that you type for **Esc n** (new file), **Esc d** (Linux command), **Esc f** (find string) or **Esc r** (replace string). In addition, you can now use **arrow keys**, **Home**, **End**, **Delete** etc. to edit strings before you press **Enter**.

### Version 2.2 pre-alpha #3 for Linux July 8, 1999:

- In an **xterm** window, **exu** will prompt you to hit **Enter** before it exits. Without this, **xterm** restores the screen so fast that you don't see any output or error messages.
- An internal coding change was made to Euphoria's **rand()** function. (The algorithm has not changed.) Hopefully this will allow **rand()** to work on all distributions of **Linux**. Please let us know if **rand()** still fails.
- **ed**: The **Esc h** command will display the Euphoria help files. This was broken in pre-alpha#2.
- In an **xterm** window, Euphoria's **video\_config()** now reports the correct number of lines and columns - this helps **ed** to work much better. **ed** will work with the initial size of window in effect when **ed** starts up.
- **ed**: **F1**, **F2**, **F3**, **F4**, **Home**, **End**, and the **Delete** key work now in **xterm** (under Red Hat 5.2 at least). The other F-keys were already working. **PageUp/PageDown** and some other keys still don't work - feel free to add your own alternate keys.
- **exu** is now even smaller - just 82K.

### Version 2.2 pre-alpha #2 for Linux July 6, 1999:

- The **ncurses** library has been statically linked into **exu**.
- **exu** is now a compressed executable (97K).
- **bug fixed: ed** can now edit files with upper case letters in the name.
- The fraction of a second delay when you press the **Esc** key in **ed** has been removed.

### Version 2.2 pre-alpha #1 for Linux July 1, 1999:

- The first version of Euphoria for **Linux** was released.



## Version 2.1 Official Release for WIN32 + DOS32 March 29, 1999:

- Updates to the **trace** screen have been optimized. Unnecessary refreshes of the source code, and the variables on the trace screen, have been eliminated. When a refresh is necessary, it is now slightly faster. This makes a noticeable difference in **exw.exe**, and also in **ex.exe** in **pixel-graphics modes**. For **ex.exe** in **text modes**, it reduces screen flicker slightly.
- The **install** program no longer requires that your PATH be less than 128 characters. It will simply warn you if it isn't. Newer versions of DOS allow for a longer PATH. Thanks to Steve Adams.
- An extra error check was added to **unregister\_block()** in **safe.e**. Thanks to David Guy.

## Version 2.1 beta-test release March 5, 1999:

- The rest of the files in the **euphoria\doc** directory have now been converted to HTML. Every **.doc** file in the **doc** directory now has a corresponding **.htm** file in the **euphoria\html** directory. Many improvements and clarifications were made to the documentation.
- You will now be warned when you have code that comes immediately after an **exit**, **return** or **abort()** statement. This code can never be executed. Suggested by Gabriel Boehme.
- **safe.e** no longer includes **graphics.e**. This eliminates possible naming conflicts when **safe.e** is substituted for **machine.e**.
- Using code supplied by David Guy, **safe.e** will now let you add or remove externally-allocated blocks of memory on the "safe address list". **See the new library routines: register\_block() and unregister\_block()**.
- **message\_box()** now uses the **handle of the active window** rather than NULL. This forces the user to reply to your message before he can continue to interact with your program. He won't be prevented from interacting with other programs. Thanks to Austin C.
- **get()** and **value()** have been sped up by a further 5% thanks to Gabriel Boehme.
- **exw.exe** has been made less likely to crash mysteriously when attacked by a virus.
- **sanity.ex** now checks your installation of Euphoria. You'll be warned if your PATH or EUDIR variables are not set, or your **ex.exe**, **exw.exe**, **pdex.exe**, or **pdexw.exe** files have been corrupted or not installed correctly in **euphoria\bin**.
- The security of **bound** and **scrambled** programs has been tightened some more. Thanks to Rusty Davis.
- To save space in **euphor21.zip**, the **install** program now *generates* the HTML and DOC files from a common source, using Junko Miura's **documentation generator**. In the process, the generator is deleted, but you can download it from the RDS site.
- When a **type\_check** failure occurs you'll be warned if the type erroneously returned a *sequence* for it's "true/false" result. Previously, a sequence result was simply reported as a type\_check failure.



Suggested by Ralf Nieuwenhuijsen.

- The code for [demo\win32\winwire.exw](#) was cleaned up considerably.
- The **install** program will warn you to change your **autoexec.bat** file when you install a new release of Euphoria onto a *different* drive.

## Version 2.1 alpha-test release January 15, 1999:

- We've made a number of changes to the packaging, pricing, and registration incentives for the Euphoria product:
  - The Dual-Platform (DOS32+WIN32) package, formerly \$53 has been reduced to \$39 U.S., effective immediately.
  - The Single-Platform (DOS32-only) package, formerly \$32, has been discontinued.
  - The printed manual has been discontinued. Instead, there is now an official HTML version of the manual, included with the Public Domain .zip file.
  - All useful 3rd-party include files, such as **Win32Lib.ew** and many others, will be "stamped" by RDS with a code number that makes them *free*, just like the files in [euphoria\include](#). They will not add to your statement count, provided you do not significantly modify them. This will also allow 3rd-party developers to get better diagnostic information from their users.
  - Binding**, **shrouding** and **profiling** will now be part of the Complete Edition only. These are features that beginners do not require, but serious users might find valuable.
- Short-form assignment operators **+=** **-=** **\*=** **/=** **&=** have been added. For example, instead of saying:
  - ```
count = count + 1
```

You can now say:

```
count += 1
```

Instead of saying:

```
matrix[row][column] = matrix[row][column] * 5.0
```

You can say:

```
matrix[row][column] *= 5.0
```

Instead of saying:

```
test_scores[start..finish] = test_scores[start..finish] / 100
```

You can say:

```
test_scores[start..finish] /= 100
```

See [refman.doc](#) for the details.
  - Euphoria now uses "short-circuit" evaluation of **and** and **or** expressions in **if/elsif/while** conditions. e.g. in an **and** condition:
    - ```
if A and B then ...
```

the interpreter will skip the evaluation of expression B whenever expression A is 0 (false), since it knows that the overall result must be false. In an **or** condition:

```
while A or B do ...
```

the interpreter will skip the evaluation of expression B whenever expression A is non-zero (true), since it knows that the overall result must be true.

Euphoria code written prior to version 2.1 may no longer work correctly if expression B contains a function with *side-effects* such as setting a global variable, doing I/O etc. In practice this kind of code is very rare, but just in case, a warning will now be issued if a function with side-effects might be short-circuited.

By skipping the evaluation of B, short-circuit evaluation is typically faster, and will allow you to write statements such as:

```
if atom(x) or length(x)=1 then ...
```

that would generate an error on older versions of Euphoria whenever x was an **atom**, since `length()` is not defined for atoms.

See [refman.doc](#) for the details.

- Several new routines were added.

#### Built-in to ex.exe/exw.exe:

- `profile()` - turns **profiling on/off** so you can focus your **profile** and **profile\_time** runs on particular events within your program.
- `system_exec()` - gives you the exit code from calling a **.exe** or **.com** file, or another Euphoria program.
- `equal()` - compares any 2 Euphoria objects for equality. equivalent to: `compare(a,b) = 0` but more readable.

#### Added to various include files:

- `walk_dir()` - recursively goes through a directory and subdirectories, calling a routine that you supply.
- `reverse()` - returns a sequence in reverse order.
- `sprint()` - returns the string representation of any Euphoria object.
- `arcsin()` - inverse trig function.
- `arccos()` - inverse trig function.
- `get_bytes()` - returns the next n bytes from a file.
- `prompt_number()` - prompts the user to enter a number.
- `prompt_string()` - prompts the user to enter a string.
- `instance()` - **WIN32:** returns the instance handle of the program.
- PI** - the constant PI - 3.14159... was added to **misc.e**.

See [library.doc](#) for the details.

- The main Euphoria documentation can now be viewed locally with a Web browser. The plain-text files [refman.doc](#) and [library.doc](#) are still available in the **doc** subdirectory, but we now have [refman.htm](#) and [library.htm](#) in the new **html** subdirectory. We have developed a tool (written in Euphoria) that lets us easily maintain both an up-to-date HTML version, and an up-to-date plain-text version of [refman](#) and [library](#).

The documentation has also been clarified and expanded in many places.

- **WIN32:** you can create an **unlimited** number of Euphoria call-back routines, as long as each routine is a function with 0 to 8 parameters. See [platform.doc](#). In version 2.0 you could only have one call-back routine and it had to have exactly 4 parameters.
- The **xor** keyword has been added to complement: **and/or/not** and **xor\_bits()** e.g.
  - ```
if a xor b then...
```
  - **xor** works on sequences too. It's similar to **or**.
- The **dir(path)** library routine now officially supports the use of **wildcards** \* and ? in the path that you supply. This feature was always available, but wasn't documented until now. e.g.
  - ```
info = dir("mydata\\*.d?t")
```
- **optimization:** Subroutine call+return overhead was reduced by an average of 30%. The speed-up occurs for all normal function/procedure/type calls, user-defined type-checks, **call\_proc()/call\_func()** calls using a **routine id**, and Windows call-backs. Only recursive calls cost the same as before. Programs with a reasonably-high frequency of calls can easily be 10% faster overall because of this.
- **optimization:** Branch **straightening** has been implemented. The compiler will optimize branches in the internal code such that a branch from A->B where location B contains a branch to location C, will be optimized to a direct branch from A->C. Even something like A->B->C->D can be straightened to A->D. This often occurs in while-loops that contain if-statements.
- **optimization:** In many cases, variable initialization checks are now replaced by "no-ops" after the first check is performed. Euphoria was already optimizing out many checks at compile-time.
- **optimization:** **get()** and **value()** are now much faster in most cases thanks to Jiri Babor and some further optimizations by RDS. The new v2.1 **ex.exe** with the new v2.1 **get.e** is:
  - 1.45x faster reading a sequence of f.p. numbers from a file and
  - 2.25x faster when reading a sequence of integers from a file.
- **optimization:** **power(x,2)** is converted internally to  $x*x$  which is faster in all cases, especially when x is a large integer or a f.p. number.
- **optimization:** Thanks to Jiri Babor, **int\_to\_bits()** is at least 15% faster in most cases.
- **optimization:** Plotting a long sequence of pixels in 16-color graphics modes is about 3% faster.
- **optimization:** **draw\_line()** has been sped up by a few percent.
- **Language War** has had a major face-lift. It now runs in **pixel-graphics mode 18** (640 x 480 x 16 colors) instead of **text mode**. It also has **fine-grain** parallelism, i.e. virtually anything can happen in parallel with anything else. Multiple torpedos, phasors etc can be drawn on the screen simultaneously, while ships are moving, commands are being entered, things are exploding etc. Even the timing needed for the PC speaker sound effects is handled by the **task scheduler**. There are **no** time-delay "busy" loops executed during the game. The galaxy scan now shows you a scaled picture of the whole galaxy, rather than just a bunch of numbers.
- The default print format for atoms was changed from "%g" to "%.10g". This format is used by **print()**, **?**, the **trace** facility, and **ex.err** dumps. This allows large integers -9,999,999,999 to +9,999,999,999

to be printed as integers, rather than as scientific notation. It also provides about 10 digits of accuracy to be displayed on fractional numbers, rather than just 6. Art Adamson and others made it clear that more digits should be displayed.

- The state of all **with/without** settings is saved upon entering an **included file**, and restored at the end of the **included file**. An included file can change the settings, but they will be restored at the end of the included file. e.g. warnings might be turned off just within the included file (and any files it includes). As a result some programs now display warnings where none were seen before.
- Warnings are now displayed *after* your program finishes execution, so they won't be erased by `clear_screen()`, `graphics_mode()` etc. Some programs now show warnings where none were seen before.
- The security of scrambled code and bound code has been improved thanks to ideas contributed by Rusty Davis. When a bound program starts executing, a quick integrity check will be made to detect any corruption or tampering. It's still ok to add data to the end of a **bound .exe** file, as long as your last line is `abort(x)`.
- The **ed** editor now lets you view and edit beyond column 80.
- ed** has a new command: **Esc m** (modifications). It will show the differences between the original file on disk and the current edit buffer. This can be very useful when you've forgotten what changes you've made, and you are wondering if it's safe to save them.
- The **trace** window now provides an upper case **Q** command which lets the program run to completion, ignoring any `trace(1)` commands. Lower case **q** lets it run to the next `trace(1)`.
- safe.e** (debug version of **machine.e**) has been enhanced. It will now automatically catch additional cases where data is illegally written just *before*, or just *after*, the boundaries of an allocated block of memory. This can be particularly useful in **WIN32** where Windows might overwrite one of your under-sized blocks. Without a tool such as **safe.e**, this type of bug could take hours or even days to track down.
- The **euphoria/tutorial** directory was created to hold several small tutorial programs.
- The limit on the number of open files was raised to 25 from 15. Three of these files are 0,1,2: standard-input, standard-output and standard-error, so you can now have up to 22 of your own files open simultaneously. (As far as we know, no one ever exceeded the old limit, but it seemed wise to raise it.)
- When the user simply types **ex** or **exw** and is prompted for the name of the Euphoria **.ex** or **.exw** file to run, `command_line()` will now be updated to include the filename as the second command-line argument, just as if the user had originally typed: **ex filename**. Thanks to Mathew Hounsell for suggesting this.
- mset.ex** now saves pictures in **.bmp** format. Previously it was using a non-standard, compressed format.
- lines.ex** (**lines.bat**) now reports non-blank/non-comment lines as well. This is *not* the same as the "statement count" used by Euphoria for the diagnostic limit, but it's usually within +/- 10%, assuming

you write one statement per line.

- Numeric literals greater than 1e308 (roughly) are now set to +/- **inf**. They used to cause a compile-time error.
- 

### Version 2.0 Official Release March 25, 1998:

- The install procedure has changed. The Euphoria .zip file now contains a large **bundle.dat** file that contains over 100 files. This makes it easier for people to locate the important files: **readme.doc**, **install.bat**, etc. that they should look at before installing. The .zip file is also 35K smaller as a result.
- **shroud** will warn you to use **bind/bindw** if you try to create a shrouded source file with a name ending in ".exe".

### Version 2.0 (beta) February 26, 1998:

- The WIN32 interpreter, **exw.exe**, is now a **true WIN32 GUI** program. In 2.0 alpha it was a WIN32 **console** program that was always associated with a **console** or DOS-window. A DOS-style console window will now be created only if your program needs one. **exw** will automatically create a new console window the first time your program writes to the screen, reads from the keyboard, or calls any library routine that requires a console to work. The console will disappear automatically when your program finishes execution.
- A new library routine, **free\_console()**, will immediately delete the console window if you currently have one.
- The Complete Edition of Euphoria now provides a **-scramble** option of **bind** and **shroud** to improve the security of programs that you distribute.
- You can now pass Euphoria **atoms** to C routines as 64-bit C double type floating-point arguments, and you can receive a floating-point result back from a C function.
- **exw.exe (beta)** runs 10 to 15% faster than **exw.exe (alpha)** (based on **sieve.ex**, **shell.ex**, etc.). The WATCOM C compiler was doing a bad job of optimizing a critical section of the interpreter when building **exw.exe**, but was producing excellent code when building **ex.exe**. With some trivial changes to the interpreter C code, WATCOM now produces excellent code for both platforms.
- The average program now has 60K more memory available before having to use the swap file.
- The limit on the size of a single procedure, function or type has been eliminated.
- The limit on the size of a single top-level statement has been eliminated.
- The limit on the total number of include files that can make up a program has been increased to 256 from 150.
- Some **optimizations** were added. The following general forms of expression are now faster:
  - $2 * x$

- $x * 2$
- $1 + x$

where x can be any expression, and have type atom or sequence.

- There is a new documentation file, [perform.doc](#) with lots of tips for performance-obsessed programmers.
- If you call a C routine using [c\\_func\(\)](#), but you linked the C routine using [define\\_c\\_proc\(\)](#) you will get an error message. Similarly, if you call it using [c\\_proc\(\)](#), but you linked it using [define\\_c\\_func\(\)](#) you'll get an error message. This restriction was documented, but not actually enforced in 2.0 alpha. Some programs written for the alpha release will have to be corrected.
- You will now see the actual name of the C or Euphoria routine that you were attempting to call, when you get an error message from [call\\_proc\(\)](#), [call\\_func\(\)](#), [c\\_proc\(\)](#), or [c\\_func\(\)](#).
- A new **-clear\_routines** option of [bind](#) and [shroud](#) will leave the names of all routines unshrouded. This is necessary if your program calls [routine\\_id\(\)](#). You'll be warned if you use [routine\\_id\(\)](#) and do not choose this option. (Registered users can use **-scramble** together with **-clear\_routines** to restore a high level of shrouding.)
- If a name conflict arises with a global symbol, the [shrouder](#) will now warn you, and then choose a new name. It used to abort with a message.
- It is no longer possible to [trace](#) or [profile](#) shrouded code.
- A new demo program, [hash.ex](#), was added to [euphoria/demo](#).
- [freq.ex](#) was moved from [euphoria/bin](#) to [euphoria/demo](#) and renamed as [tree.ex](#).
- A new documentation file, [bind.doc](#) describes all the features of [bind.bat](#) and [shroud.bat](#). The previous description in [refman.doc](#) has been shrunk.
- The file [overview.doc](#) gives a quick overview of all documentation files.
- The description of [get\\_mouse\(\)](#) in [library.doc](#) discusses the problem of 320-wide graphics modes (you must divide the x coordinate value by 2).

## Version 2.0 (alpha) November 5, 1997:

- A **new platform** is now supported. [exw.exe](#) will run Euphoria programs using the **WIN32** (Windows 32-bit) operating system. [ex.exe](#) will run programs using **DOS32** (extended DOS). See [platform.doc](#) for further details.
- The following library routines have been introduced.

### For both DOS32 and WIN32:

- [platform\(\)](#) - find out which platform you are executing on. (The PLATFORM constant is available in Euphoria 2.2 and later).
- [routine\\_id\(\)](#) - get a small integer id number for a Euphoria procedure or function.

- call\_proc()** - call a Euphoria procedure by using its id.
- call\_func()** - call a Euphoria function by using its id.
- custom\_sort()** - sort a sequence using a compare function that you specify.
- poke4()** - store a number into 4 bytes of memory. **poke4(address, value)** is at least 10x faster than: **poke(address, int\_to\_bytes(value))**. **poke4()** also works with sequences of values.
- peek4s()** - read 4 bytes of memory as a signed integer (works on sequences too).
- peek4u()** - read 4 bytes of memory as an unsigned integer (works on sequences too). **peek4u(address)** is 10x faster than: **bytes\_to\_int(peek({address, 4}))**.
- allocate\_string()** - allocate and store a 0-terminated string in memory.

#### For WIN32 only:

- open\_dll()** - open a Windows .dll file.
- define\_c\_proc()** - define a C routine that will be called from Euphoria (no value returned).
- define\_c\_func()** - define a C routine that will be called from Euphoria (a value will be returned).
- call\_c\_proc()** - call a C routine from Euphoria (no value is returned).
- call\_c\_func()** - call a C routine from Euphoria (a value is returned).
- call\_back()** - get a *call-back address*, so Windows can call your Euphoria routine when the user interacts with your window.
- message\_box()** - display a simple **Yes/No/Cancel** window.

#### •New Demo programs:

- **csort.ex**
- **email.exw**
- **window.exw**
- **winwire.exw**
- **dsearch.exw**

#### •New include files:

- safe.e** - debug version of **machine.e**
- misc.e** - miscellaneous
- dll.e** - dll access
- msgbox.e** - Windows message box

#### •The following additional improvements have been made to the **DOS32** version:

- On Pentium and higher systems, floating-point calculations are now about 20% faster (and floating-point in **exw** is a further 20% faster than **ex** for 2.0).
- **printf()** to the screen, and **print()** to the screen are both significantly faster in most cases.
- The **trace** screen is updated a bit faster.
- The **time profile** is more accurate regarding **getc()**.



- The **mset.ex** demo runs 30% faster.
- 

## Highlights of Older Releases:

### Version 1.5a June 13, 1997:

- Many operations and library routines were optimized.
  - get\_key()** is 100x faster when there is no key in the buffer.
  - get\_all\_palette()** is over 100x faster and this makes **save\_screen()** much faster.
  - The following routines have now been built directly into **ex.exe**, to avoid the overhead of calling **machine\_proc()** or **machine\_func()**: **pixel()**, **get\_pixel()**, **mem\_set()**, **mem\_copy()**.
  - poke()** of a long sequence into memory, other than video memory, is 50% faster.
  - pixel()** is 4x faster in mode 19.
  - get\_pixel()** is faster in all modes.
  - display\_image()** is about 30% faster in most modes and up to 4x faster in mode 19, because **pixel()** is faster.
  - All arithmetic and bitwise operations applied to sequences of integers are now 29% faster.
  - a **&** b (concatenation) is 15% faster in most cases, and is dramatically faster in the case where you grow a very long sequence by concatenating many small sequences onto it.
  - getc()** is 12% faster.
  - match()** is 8% faster in typical cases.
  - append()/prepend()** are 15% faster in many cases.
  - find()** of an integer within a sequence of integers is 64% faster.
  - Formation of a 2-element sequence **{a,b}** is 11% faster.
  - Internal copying of a shared sequence when it can no longer be shared is 15% faster.

### Version 1.5 March 21, 1997:

- The following library routines were added. They are described fully in **library.doc**.
  - allow\_break()**
  - check\_break()**



- `mem_copy()`
- `mem_set()`
- `atom_to_float32()`
- `atom_to_float64()`
- `float32_to_atom()`
- `float64_to_atom()`
- `get_all_palette()`
- `save_bitmap()`
- `save_screen()`
- `arctan()`
- `and_bits()`
- `or_bits()`
- `xor_bits()`
- `not_bits()`
- `get_vector()`
- `set_vector()`
- `lock_memory()`
- `tick_rate()`

- **with `profile_time`** (time profiling for DOS32) was added.

### Version 1.4b, October 1996:

- **`mset.ex`** has a more visible selector box on it's grid. It has also been sped up.
- **`ed.ex`** now allows special characters greater than ASCII 127 to be entered by pressing the **Alt** key and typing digits on the numeric keypad.

### Version 1.4a, July 1996:

- **`crash_message()`** library routine was added.
- Programs **bound** by registered users will now produce run-time error diagnostics regardless of the size of the program.

- **shroud.bat** has a new option **-full\_keywords**.

### Version 1.4, May 1996:

- You can now convert any Euphoria program into a **stand-alone .exe** file.
- The separate DOS4GW.EXE DOS extender file has been eliminated.
- Windows 95 long filename support.
- Support for DOS software interrupts.
- New utility programs: **key.ex**, **where.ex**, **ascii.ex**, **guru.ex**.
- New demo program: **dosint.ex**.
- New library routines: **set\_rand()**, **use\_vesa()**.
- You can peek or poke an entire sequence of bytes.
- Editor enhancements.
- Reduction in space overhead for sequences.

### Version 1.3, June 1995:

- You can now edit multiple files using multiple edit windows.
- 20 new library routines were added.
- Graphics performance was greatly improved.

### Version 1.2, March 1994:

- A problem that prevented Euphoria from running in a DOS prompt under Windows was eliminated.

### Version 1.1, January 1994:

- Several language features and demo programs were added.

### Version 1.0, July 1993:

- Euphoria was first released after 3 years of research and development and 6 months of Beta testing. Many of the ideas behind Euphoria came from Robert Craig's Master's Thesis in Computer Science at the University of Toronto. That thesis was strongly influenced by the work of John Backus on functional programming (FP) languages.



# Multitasking in Euphoria

## 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.
- Windows, Linux and FreeBSD all 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.

**Time-share tasks** need a share of the CPU but they needn't be rigidly scheduled according to any clock. The 4 sorting tasks in the demo\dos32\tasksort.ex demo, share the CPU, but it isn't important that they be scheduled at a particular time. On the other hand, 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.

It's possible to reschedule a task at any time, and to 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 won't 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 "pre-emptive", 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.

## Built-in Multitasking Routines

All of these routines are built-in to Euphoria, so it's not necessary to include any library file.

**task\_create** - Call this to create a new task. You need to pass the routine id of a Euphoria procedure, as well as a list of initial arguments to pass to the procedure. This is the main procedure for the task. Tasks are always procedures, since it doesn't make sense for a task to return a value (no other task is waiting for it). `task_create()` will return a task id (a small integer). Use this task id to identify the task to the other multitasking routines below. Note that all Euphoria programs start off with one initial task running.

**task\_yield** - A task calls this to yield control, so the Euphoria scheduler can pick a new task to run. A task should call this often enough to avoid hogging the CPU, but should not call it so often that much time is wasted on scheduling. To avoid corruption of data structures, a task should try to complete a logical step before giving up control. It's possible that the scheduler will decide to keep running the same task in which case a quick return from `task_yield()` will occur with no other task getting control.

**task\_schedule** - Schedule a task for execution. After a task is created, it is necessary to schedule it, otherwise it will never run. There are two main ways of scheduling a task. One way specifies a real-time timing interval, min...max. This tells the scheduler that the task must (if possible) be scheduled to run a minimum of min seconds from now, and a maximum of max seconds from now. Subsequent runs of the task will also wait for min/max seconds. So you might say that a task must run every 3.5 to 4.0 seconds. The second way using a time-sharing system where a task can execute `task_yield()` a certain number of times before it must give up the CPU. So one task might be allowed 10 `task_yields`, where another, perhaps lower priority task must give up the CPU on every `task_yield()`, if possible

**task\_list** - Get a list of all tasks

**task\_self** - Return the task id of the current task

**task\_status** - Get the current status (active, suspended, killed) of a task

**task\_suspend** - Suspend a task until further notice.

**task\_clock\_start** - Restart the scheduler's clock. Used in games where the real time clock must be preserved during a stoppage.

**task\_clock\_stop** - stop the scheduler's clock





# CGI (Web) Applications in Euphoria

## Introduction

If you have a Web site, you may be able to set up programs on your server that visitors to your site can run. Most free Web hosting services do not offer this capability, but almost all paid services do. If they advertise features like "cgi-bin", "Perl", "C" etc. then you can most likely run Euphoria programs as well. The hosting company does not have to install Euphoria. You can do it yourself quite easily. As far as the host is concerned, you are just running a program written in C, namely the Euphoria interpreter.

A very simple protocol called CGI (Common Gateway Interface) is supported by almost all paid Web servers, and you can use almost any programming language to develop CGI applications. Perl is currently very popular for CGI, but Euphoria is even better suited than Perl in many ways. Compared to Perl programs, Euphoria programs start up quicker, run faster, and have better error diagnostics. Compared to C, Euphoria programs take less effort to develop, and are easier to change and debug. Euphoria also has much easier string handling, which is often important in CGI programming.

## CGI (Common Gateway Interface)

CGI is a fancy acronym for a very simple protocol. The CGI standard defines how a Web browser passes information to a program running on a Web server, and how that program passes information back to the Web browser.

## Web Hosts

If you have the necessary Web server software, and the approval of your Internet service provider, you can probably run Euphoria CGI programs right on your own (home) machine.

However, most people will have to use a Web hosting service that supports CGI. You can find hosting services that cost anywhere from \$3 U.S. per month up to \$50 or more.

RapidEuphoria.com is on a fairly reliable host (addr.com) that costs only \$10 U.S. per month. For that price we get 3000 Mb of disk space on a FreeBSD machine, and 60 Gb of bandwidth per month. Until recently, for the same price, we were given only 150 Mb of space and 15 Gb of bandwidth. The bandwidth limitation determines the number of bytes that may be downloaded by visitors each month.

EUforum (on our domain ListFilter.com) is hosted on an even cheaper Linux site (host-web-site.com) for \$4/month. It has 200 Mb of space, and 5 GB of bandwidth. Until recently, it cost \$18/month, so hosting prices seem to be headed down.

## Installing Euphoria for CGI

Installing Euphoria on a host is fairly simple. You just have to upload the euphoria directory for the operating system that the host is running. You can put the euphoria directory into the top level of your own personal

space. There is no need to change anything on the system as a whole. Using telnet (or ssh) you can experiment, and make sure that you can execute some of the euphoria\demo programs on the host. You should set up your EUDIR and PATH variables to point to your euphoria directory. This might involve editing a .profile file that's executed each time you log in.

The Euphoria CGI programs that you want people to run, must typically be placed in a special subdirectory of your account, called "cgi-bin". cgi-bin is often located under a directory called "public\_html" which contains all your publicly-visible HTML files. cgi-bin has special permissions that allow visitors to your site to execute the programs it contains.

On Linux and FreeBSD, and also on Windows (if the system uses Apache Web Server), you can use a "shebang" line as the very first line of your main Euphoria file. It will show the system where the Euphoria interpreter (exu or exw.exe) is located. e.g.

```
#!/.exu
```

would say that the interpreter is in the current directory (cgi-bin). If you upload this file to Linux/FreeBSD, be careful that the line ends with just \n, not \r\n.

**Note:** on Windows you should use exwc.exe, not exw.exe. exwc handles standard output better, and that's critical for CGI. You should also say "without warning" at the top of your main file. Warning messages could cause problems in properly terminating your program.

On Linux/FreeBSD you must set execute permission on any CGI program. When a user tries to run your program, a special process on the server actually runs your program and it needs to find and have access to your Euphoria files. You should probably set your files so anyone can read or execute them, but only you can write them. e.g. rwx r-x r-x or **chmod 755 myfile.exu**. Remember also to make exu executable by yourself and others.

On a Linux system, if your /tmp directory is not writable, or for other reasons, you may need to run a non-compressed version of exu. You can get this from the RDS download page, or by decompressing exu using UPX.

To make your life simple, you may wish to copy all files into cgi-bin that your program will need. That includes the standard Euphoria files from euphoria\include, the Euphoria interpreter, and any database files you are going to access. That way you can avoid the sticky issue of the server process not having the same environment variables (PATH, EUDIR) as you do, and not being able to find your Euphoria files.

## HTML Forms

You can make a simple Web link to your CGI program, and run it by clicking the link, but normally you will set up a Web page containing an HTML form. This provides your users with a graphical user interface, where they can supply information to your CGI program.

There are several places on the RapidEuphoria.com Web site where HTML forms are used to communicate with CGI programs written in Euphoria. Perhaps the simplest is the MicroEconomy form at:

<http://www.rapideuphoria.com/eciform.htm>

You can use Internet Explorer's View / Source to see the HTML code for this page.

If you page-down a few times, you'll see the start of a form:

```
<form method="POST" action="cgi-bin/economy.exu">
```

This creates a form for the user to fill out. When the user clicks the submit button for the form, the CGI program, economy.exu, stored in cgi-bin, will be executed on the server.

There are two possible "methods" that you can use. This form uses the "POST" method. The other method is called the "GET" method.

With the POST method, the user's input data is supplied to the Euphoria program as its standard input, and can be read with gets() and getc().

With the GET method, the data is supplied to the Euphoria program as an environment variable called "query\_string".

Which method is best? In most cases it doesn't matter which method you use, as the input data will be the same. Only the method of acquiring that data will be different.

In fact, if you look at the Euphoria program that handles this MicroEconomy form:

<http://www.rapideuphoria.com/economy.txt>

You'll see that procedure read\_input() checks which method was used, and grabs the input data from the appropriate source. Feel free to use read\_input() in your CGI programs.

The form collects data from the user in various ways. One of the simplest ways is with text input tags such as:

```
<input type="text" name="name" size="35">
```

```
<input type="text" name="city" size="35">
```

which let the user type in his name and city into two separate boxes. The values that he types are passed to the Euphoria program, and are labelled as "name" and "city".

So what does the input data look like?

with the GET method the query\_string environment variable (for the first two fields) will look like:

```
"name=John&city=Miami"
```

and you'll be able to see it on the address bar of your Web browser, after a '?' character.

If the POST method is used, the same string will be available via standard input.

You will have to parse this input to break out the two fields. The MicroEconomy program uses parse\_input() to do this. You might want to incorporate parse\_input() into your own programs.

Besides simple text boxes, there are many types of user input available in HTML. You can have single line and multiple line input boxes, password boxes (dots appear), drop down menus, radio buttons, check boxes, etc. Each attaches a name to a piece of data supplied by the user, and your CGI program has to break out

this information using `parse_input()` or some equivalent code. You can even let the user upload large files.

## The Output from Your CGI Program

After examining the query string, your program must generate a reply for the user. This reply must be in the form of a complete HTML page written to standard output (Euphoria file 1). The very first line that your program writes to standard output must be:

Content-type: text/html

That line must then be followed by a bunch of lines that comprise a complete HTML page. For example, the MicroEconomy program starts off by calling procedure `print_HTMLheader()` which does:

```
puts(1, "Content-type: text/html\n\n")
puts(1, "<html><head><title>Thank You</title></head>\n")
puts(1, "<body bgcolor=\"#FFFFFF\">\n")
```

Much later, when it's all finished, it writes out the last line of output which contains `</body></html>` to complete the page. If the program fails to complete the HTML page, the user's browser might not display the page properly, if at all.

## Debugging CGI programs

The first time you run a new CGI program, the chances are quite high that it will fail in some way. You are likely to see an error page displayed that will say very little about what went wrong.

The first thing you should do is check for a Euphoria `ex.err` file in `cgi-bin`. If there's one there, it will tell you why your program died. If there is no `ex.err`, it could be that your program didn't even run, because it's lacking the correct permissions, or the shebang line is wrong; or it may have run but it did not generate a proper HTML page. Be sure that you have the Content-type line at the start, and you have the proper structure, starting with `<html>` and ending with `</html>`.

If you aren't getting an `ex.err`, you can generate an `ex.pro` file, by saying "with profile" at the start of your program. That will give you an idea of what statements your program executed before it quit.

You can also try "with trace" and `trace(3)` to generate a file containing the last 500 statements that were executed before it quit.

Once your program seems to be working, you might want to add a crash routine that will send `ex.err` to you via e-mail. You may discover that your users are trying to do weird things that you never do, and are encountering crashes.

## Speeding up CGI programs

Most CGI programs require little CPU time, and most of that time might be in the start up, i.e. loading the language interpreter and parsing the program. A typical CGI program might read a few parameters that a user types into a form, then access one record in a database, or send one e-mail message, before displaying a simple HTML acknowledgement page for the user.

For CGI applications that do require a lot of time, you should translate your Euphoria program to C, and compile it. The resulting executable file can be used as a CGI program. It will start up faster (no parsing), and run faster. The only problem is that you won't get a full ex.err file when there's a crash, however you can use `crash_routine()` to send yourself an e-mail containing some of the key variables, such as the query string that led to the crash.

## An important message for all C/C++ programmers...

### 24 Reasons Why You Are Going to Write Your Next Program in Euphoria!

- because you are tired of having to re-invent dynamic storage allocation for each program that you write
- because you have spent too many frustrating hours tracking down malloc arena corruption bugs
- because you were once plagued for several days by an on-again/off-again "flaky" bug that eventually was traced to an uninitialized variable
- because no matter how hard you try to eliminate them, there is always one more storage "leak"
- because you are tired of having the machine "lock up", or your program come crashing down in flames with no indication of what the error was
- because you know that **subscript checking** would have saved you from hours of debugging
- because your program should not be allowed to overwrite random areas in memory via "wild" pointers
- because you know it would be bad to overflow your fixed-size stack area but you have no idea of how close you are
- because one time you had this weird bug, where you called a function, that didn't actually return a value, but instead fell off the end and some random garbage was "returned"
- because you wish that library routines would stop you from passing in bad arguments, rather than just setting "errno" or whatever (who looks at errno after every call?)
- because you would like to "recompile the world" in a fraction of a second rather than several minutes -- you can work much faster with a cycle of edit/run rather than edit/compile/link/run.
- because *The C++ Programming Language* 3rd Ed. by Bjarne Stroustrup is 911 very dense pages, (and doesn't even discuss platform-specific programming for DOS, Windows, Linux or any other system).
- because you have been programming in C/C++ for a long time now, but there are still a lot of weird features in the language that you don't fully understand
- because portability is not as easy to achieve as it should be
- because you know the range of legitimate values for each of your variables, but you have no way of enforcing this at runtime
- because you would like to pass variable numbers of arguments, but you are put off by the complicated way of doing it in C
- because you would like a *clean way* of returning multiple values from a function

- because you want an integrated **full-screen source-level debugger** that is so easy to use that you don't have to search through the manual each time, (or give up and recompile with printf statements)
- because you hate it when your program starts working just because you added a debug print statement or compiled with the debug option
- because you would like a reliable, accurate **statement-level profile** to understand the internal dynamics of your program, and to boost performance
- because very few of your programs have to squeeze every cycle of performance out of your machine. The speed difference between Euphoria and C/C++ is not that great, especially when you use the Euphoria to C Translator. Try some benchmark tests. We bet you'll be surprised!
- because you'd rather not clutter up your hard disk with .obj and .exe files
- because you'd rather be running your program, than wading through several hundred pages of documentation to decide what compiler and linker options you need
- because your C/C++ package has 57 different routines for memory allocation, and 67 different routines for manipulating strings and blocks of memory. How many of these routines does Euphoria need?  
**Answer: zero. In Euphoria, memory allocation happens automatically and strings are manipulated just like any other sequences.**

# Euphoria To C Translator

---

1. [Introduction](#)
  2. [C Compilers Supported](#)
  3. [How to Run the Translator](#)
  4. [Dynamic Link Libraries \(Shared Libraries\)](#)
  5. [Executable Size and Compression](#)
  6. [Interpreter vs. Translator](#)
  7. [Legal Restrictions](#)
  9. [Frequently Asked Questions](#)
  10. [Common Problems](#)
- 

## 1. Introduction

**The Euphoria to C Translator will translate any Euphoria program into equivalent C source code.**

There are versions of the translator for **Windows**, **DOS**, **Linux** and **FreeBSD**. 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/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's written 100% in Euphoria.

## 2. C Compilers Supported

The **Translator** currently works with GNU C on **Linux** or **FreeBSD**, with either [Watcom C](#) or [DJGPP C](#) on **DOS**, and with either Watcom C, [Lcc](#) or [Borland 5.5](#) on **Windows**. These are all **free** compilers. GNU C will exist already on your Linux or FreeBSD system. The others can be downloaded from their respective Web sites. For Windows, we strongly recommend Watcom or Borland over Lcc. Lcc is still actively being developed and is getting better, but has some bugs that will make it difficult for you to compile a large Windows program correctly. Watcom and Borland are both rock solid. Watcom usually produces slightly smaller, slightly faster executables, but Borland compiles much faster.

The Watcom DOS32 package includes the CauseWay DOS extender and file compressor. CauseWay is now open source and free. You can find out more about it at: <http://www.devoresoftware.com>

**emake.bat** and **objfiles.lnk** will link in the CauseWay extender automatically. Other DOS extenders, such as DOS4GW, do not work well with the **Translator**.

The **Translator** looks for "WATCOM", "LCC", "BORLAND" or "DJGPP" as either environment variables or directories on your **PATH**. It will generate an **emake.bat** file that invokes the appropriate compiler and linker.

### Notes:

- Unlike Watcom, DJGPP does not map DOS low memory into the same segment as other memory.



Machine code routines written for the Watcom-based **Euphoria Interpreter** or **Translator** will not work with DJGPP, and will likely crash if they try to access low memory, such as video memory. Euphoria's **peek()**, **poke()**, **mem\_copy()**, **mem\_set()** etc. *will* work correctly, as the **Translator** uses a special DJGPP macro to access low memory. You can port these machine code routines to DJGPP, but you'll need to consult the DJGPP docs for the possible ways of accessing low memory.

- DJGPP fully supports long filenames for reading, writing and creation. Watcom doesn't support creation.
- The Translator with DJGPP does not support use of the mouse.
- The Allegro graphics library, that we use with DJGPP, seems to be much faster than the Watcom graphics library in many cases.
- DJGPP supports a few more text modes, e.g. 35-line mode.
- DJGPP lets the user abort a program at any time, by typing control-c.
- The Lcc implementation ignores **lock\_file()** and **unlock\_file()**. They do nothing.
- The **Translator** uses Lcc's -O optimization flag in **emake.bat**. This flag can sometimes add to Lcc's unreliability. If your program doesn't work, try removing this flag from all compile commands in emake.bat.
- Warnings are turned off when compiling with **emake.bat**. 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.
- On Windows, the Watcom linker might issue a warning that it can't open graph.lib. You can ignore this. graph.lib is not used. There doesn't seem to be an easy way to suppress this message. The latest version of Open Watcom seems to have corrected this.
- With Borland and Lcc, Windows console (text-mode) programs cannot read the F-keys or arrow keys, and you may have to hit the Enter key twice. There is no problem with GUI programs.
- The Microsoft C++ compiler for Windows is not yet supported. However you can probably import the C files generated by **ecw.exe**, and the run-time library file for Borland, Lcc or Watcom into a Microsoft project, and compile/link with only minor glitches.

### 3. How to Run the Translator

Running the **Translator** is similar to running the **Interpreter**. On DOS you would type:

```
ec allsorts.ex
or
ec allsorts
```

but instead of running the allsorts.ex program, the **Translator** will create several C source files. Anyone can run the **Translator**. It's included in euphoria\bin along with the interpreter. To compile and link the C files, you need to install one of the supported C compilers. The **Translator** creates a batch file called **emake.bat**

that does all the compiling and linking steps for you, so you don't actually have to know anything about C or C compilers. Just type:

```
emake
```

When the C compiling and linking is finished, you will have a file called: **allsorts.exe** and the C source files will have been removed to avoid clutter.

When you run allsorts.exe, it should run the same as if you had typed: **ex 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, emake removes all the C files that were created. If you want to look at these files, run the translator again and look at the files before running emake.

### Note to Linux and FreeBSD users:

The files will be called **emake** and **shell**, and you type **./emake** to perform the compiles and link, and **./shell** to run the shell sort program.

### Note to Borland and Lcc users:

For Borland and Lcc there is no standard environment variable, so the **Translator** will search your **PATH** variable looking for a likely compiler directory. It looks in standard places such as: ..\LCC, ..\BCC..., ..\Borland.. etc. If you've installed in a nonstandard place you might have to rename your installation directory. To edit your PATH variable on Windows XP, click: Start Menu / Control Panel / Performance and Maintenance / System / Advanced / Environment Variables / User variables for ... Then Select PATH and edit its value. Insert somewhere: C:\BORLAND\BCC55\BIN; or C:\LCC\BIN; Then click OK a couple of times to close the windows. The next DOS window you open should have the new value.

### Command-Line Options

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:

```
-bor  
-lcc  
-wat  
-djg
```

on the command line to **ec** or **ecw**. e.g.

```
ecw -bor pretend.exw
```

Normally, after building your .exe file, the **emake** batch file will delete all C files and object files produced by the Translator. If you want **emake** to keep these files, add the **-keep** option to the Translator command-line. e.g.

```
ec -wat -keep sanity.ex
```

To make a **Windows** .dll file, or **Linux** or **FreeBSD** .so file, just add **-dll** to the command line. e.g.

```
ecw -bor -dll mylib.ew
```

To make a **Windows console** program instead of a Windows GUI program, add **-con** to the command line. e.g.

```
ecw -bor -con myprog.exw
```

To increase or decrease the total amount of stack space reserved for your program, add **-stack nnnn** to the command line. e.g.

```
ec -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.

To make a **DOS** program, compiled by WATCOM, that uses fast hardware floating-point instructions, add **-fastfp** to the command line. e.g.

```
ec -wat -fastfp crunch.ex
```

By default, Euphoria for DOS calls routines to test if hardware floating-point instructions are available. If they are not, then slower software emulation code is used. When **-fastfp** is specified, the compiled code will assume the existence of hardware floating-point. This can cause floating-point intensive programs to run about twice as fast, but they will fail to run at all on old 486's and 386's that are lacking hardware floating-point support. With **-fastfp**, emake.bat chooses faster WATCOM C compiler options, and emake.bat must also link in **ecfastfp.lib** instead of **ec.lib**.

On all other platforms, Euphoria uses fast hardware floating-point instructions, and the operating system handles the case where hardware f.p. is missing.

## 4. Dynamic Link Libraries (Shared Libraries)

Simply by adding **-dll** to the command line, the **Translator** will build a **Windows .dll** (**Linux/FreeBSD .so**) file 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" keyword, will be exported from the .dll (.so).

Any Euphoria program, whether translated/compiled or interpreted, can link with a Euphoria .dll (.so) using the same mechanism that lets you link with a .dll (.so) written in C. The program first calls **open\_dll()** to open the .dll or .so 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()`. See [library.doc](#) for the details.

The routine names exported from a Euphoria .dll will vary depending on which C compiler you use.

GNU C on Linux or FreeBSD 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 symbol 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.

Lcc would export `foo()` as `"__0foo@8"`, where 8 is the number of parameters (2) times 4. You can check the .def file created by the [Translator](#) to see all the exported names.

For Borland the [Translator](#) also creates a .def file, but this .def file renames the exported symbols back into the same names that you used in your Euphoria source, so `foo()` would be exported as `"foo"`.

For Watcom the same renaming as with Borland occurs, but instead of a .def file, an EXPORT command is added to [objfiles.lnk](#) for each exported symbol.

With Borland and Watcom you can edit the .def or [objfiles.lnk](#) file, and rerun [emake.bat](#), to rename the exported symbols, or remove ones that you don't want to export. With Lcc you can remove symbols but you can't rename them.

Having nice exported names is not critical, since the name need only appear once in each Euphoria program that uses the .dll, i.e. in a single `define_c_func()` or `define_c_proc()` statement. The author of a .dll 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 .dll.

When you call `open_dll()`, any top-level Euphoria statements in the .dll or .so 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 .dll (.so) 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 .dll's and .so's.

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 .dll

(.so) open it. Unfortunately there is no simple solution for passing routine id's. This might be fixed in the future.

A Euphoria .dll or .so currently may not execute any multitasking operations. The Translator will give you an error message about this.

Euphoria .dlls (.so's) 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.

## 5. Executable Size and Compression

On DOS32 with Watcom, if the **Translator** finds the CauseWay files, cwc.exe and le23p.exe in **euphoria\bin**, it will add commands to **emake.bat** that will compress your executable file. If you don't want compression, you can edit **emake.bat**, or remove or rename cwc.exe and/or le23p.exe.

On Linux, FreeBSD, Windows, and DOS32 with DJGPP, **emake** does not include a command to compress your executable file. If you want to do this we suggest you try the free UPX compressor. You can get UPX from: <http://upx.sourceforge.net> 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.

## 6. Interpreter vs. 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 emake.bat. 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.

## 7. Legal Restrictions

As far as RDS is concerned, any executable programs or .dll's 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 January 2000, the CauseWay DOS extender was donated to the public domain by Michael Devore. He has surrendered his copyright, and encourages anyone to use it freely, including for commercial use.

In general, if you wish to use Euphoria code written by 3rd parties, you had better honor any restrictions that apply. If in doubt, you should ask for permission.

On Linux, FreeBSD and DJGPP for DOS32, 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 in **emake** does not statically link any FSF libraries, so there should be no problem.

The Allegro graphics library, used by DJGPP, is referred to as "Giftware" in their documentation, and they allow you to redistribute it as part of your program. They ask for, but do not require, some acknowledgement.

### Disclaimer:

This is what we believe to be the case. We are not lawyers. If it's important to you, you should read the GNU Library licence, the legal comments in DJGPP, Lcc and Borland, and Michael Devore's read.me file on his site, to form your own judgement.

## 9. Frequently Asked Questions

**Q - How much of a speed-up should I expect?**

**A** - 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. Watcom, GNU C and DJGPP produce the fastest code. Borland is fairly good. Lcc lags slightly behind the others, even when its -O flag is used.

Borland compiles the fastest. Watcom compiles the slowest.

**Q - *What if I want to change the compile or link options in emake.bat?***

**A -** Feel free to do so, however you should copy **emake.bat** to your own file called (say) mymake.bat, then run mymake.bat after running the Translator. Occasionally the number of .c files produced by the Translator could change.

**Q - *How can I make my program run even faster?***

**A -** 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 and DOS 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 version of **emake.bat** you might get a slight improvement in speed, with a slight risk of buggy code. For DJGPP you might try -O6 instead of -O2.

For DOS we use the Watcom /fpc option which generates calls to run-time routines to perform floating-point operations. If the machine has floating-point hardware it will be used by the routine, otherwise software emulation will be used. This slows things down somewhat, and isn't needed on Pentiums, but it guarantees that your program will run on all 386 and 486 machines, even if they lack floating-point hardware. The DOS run-time library, **ec.lib**, was built this way, so you can't simply remove this option.

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 **emake** might be required.

## 10. 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 work when running the **exw** Interpreter, but will probably fail (crash) when you translate and compile with Borland or Lcc.



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 [emake.bat](#) 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.

Post bug reports on EUforum.

In particular, report any program that does not run the same when compiled as it does when interpreted.