proudly presents...

Euphoria Programming Language
version 2.3

Reference Manual

A Preface

This work isn't original at all. Before me, better ones like Travis Beaty and Euman did something like, i.e. an Euphoria docs compilation. Why? Well, maybe like me they also like things all together in place.

I also like things running fast (don't we all?) and that's a good reason to use Winhelp format and not HTML Help. As a matter of fact I can do both as easy as but we can never forget that, despite de fact that it is not so fancy, Winhelp covers all Windows brands from ancient 3.1x to the latest 32 bit marvel, it's faster and smaller.

Travis Beaty's **Encyclopædia Euphoria** impressed me very much. I tried to translate it into Portuguese (Brazilian), but is to much work and I'm not sure if I'd have enough public to benefit from it. So I put it on the back burner.

So why do it again? Well, I have some very nice tools and I planned to use them: one, specifically, is a Help file editor/builder that can also build *flat manuals* from the same Help file project! So, this became my major goal: to build a **printable manual** out of Euphoria's documentation, that we all agree is very good. Off course I choose PDF format that can be read/print in all major platforms without any hassles.

Now, I'm proud to present a new **Refman.hlp** and a **Refman.pdf** that you can use either at your computer or printed as a book. You'll find it very useful as you'll be able to look at language references printed on paper, while coding your latest app on your computer.

I also wish to make some acknowledgments: to Travis Beaty, for his inspiration; to David Cuny et all, for the excellent Win32Lib and IDE; to Euman, for his nice tools, tips & hints; and all those who welcomed me into the Euphorium.

Special thanks to Davi Tassinari de Figueiredo who was responsible for my introduction to Euphoria, and to Robert Craig for his fantastic language!

Enjoy!

Euler German
Sete Lagoas, MG, Brazil

If you find any problems or have suggestions, maybe some criticisms (polite, please), whatever may come, feel free to contact me through Euphoria mailing list, or write me at: efgerman.eudoc@mailnull.com.

An important message for all BASIC programmers

13 Reasons Why You Are Going to Write Your Next Program in Euphoria!

- because Euphoria programs run on DOS, Windows and Linux.
- because Euphoria is actually ***simpler*** than BASIC
- because Euphoria is ***10 to 20 times faster*** than Microsoft QBasic (see **demo\bench**)
- because QBasic limits the size of your program and data to 160K bytes
- because Euphoria checks for uninitialized variables - BASIC just quietly sets them to 0
- because Euphoria lets you say precisely what values may be stored in each variable
- because Euphoria has **true** dynamic storage allocation - you do not wipe out your data when you re-dimension an array
- because Euphoria is ***more flexible*** than BASIC - you can declare types for your variables or not; you can store objects of any size into an array (Euphoria sequence); you can have arrays of mixed type of data
- because Euphoria lets you perform operations on entire sequences
- because BASIC is an old "ad-hoc" language that carries 25 years of redundant, excess baggage along with it
- because there is no effective standard for BASIC across different machines and there probably never will be
- because QBasic provides no built-in functions for using a **mouse**
- because QBasic does not support **SVGA** graphics

An important message for all C/C++ programmers

25 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'd like to forget about near pointers and far pointers, and the small, medium, compact, large and huge memory models
- 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 feel confident about using
- 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 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 as great as you might think. 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.**

Core Language Introduction

Euphoria is a new 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 at least ***10 to 30 times*** faster than conventional interpreters such as Microsoft QBasic, Perl and Python.
- lightning-fast pre-compilation. Your program is checked for syntax and converted into an efficient internal form at over ***35,000 lines per second*** on a Pentium-150.
- 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, 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.

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**.

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. On DOS/Windows an automatic install program, **install.bat** is provided for this purpose. For the latest details, please read the instructions in **install.doc** before you run **install.bat**.

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

\euphoria

readme.doc
readme.htm

\bin

ex.exe and exw.exe, or exu (Linux), ed.bat, guru.bat, other utilities

\include

standard include files, e.g. graphics.e

\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-specific demo programs (optional)

\langwar
language war game (pixel-graphics version for DOS, or text version for Linux)

\bench
benchmark programs

\register
information on ordering the Complete Edition

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

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 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, 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-style \n, and not the DOS/Windows-style \r\n, or the Linux shell might get confused.

You can also run **bind.bat** (DOS32), or **bindw.bat** (Win32) or **bindu** (Linux) 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 Distributing a Program.

Either **exu** or **ex.exe** and **exw.exe** are 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**.

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.

Use of a Swap File

If you run a Euphoria program under Linux 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**.

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 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 and the Windows console. See also **David Cuny**'s excellent **ee.ex** editor for DOS and Linux. 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.

Distributing a Program

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

In the first method you simply ship your users the Public Domain **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 Complete Edition 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 .ex, .exw, or .exu file. You can either encrypt/compact your program to conceal it and make it tamper-proof, or you can leave it readable to allow better diagnostics from your users. **Binding** combines your shrouded program with ex.exe, exw.exe, or exu 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. The Translator is a separate download available on the RDS Web site.

Licensing

You have complete royalty-free rights to distribute any Euphoria programs that you develop. You are free to distribute the Public Domain Edition **ex.exe**, **exw.exe** and **exu** files so anyone can run your program. With the Complete Edition, you can **shroud** or **bind** your programs 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 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.

The only Interpreter-related files that you may ***not*** distribute are the **ex.exe**,

exw.exe, **bind.ex**, **bind.bat**, **bindw.bat**, and **shroud.bat** files that come with the Complete Edition Interpreter for Win32 + DOS32, and **exu**, **bind.ex**, **bindu** and **shroud** that come with the Complete Edition Interpreter for Linux. The sample icon file, **euphoria.ico**, that's included with the Win32 + DOS32 Complete Edition, may be distributed with or without your changes.

The Euphoria Interpreter is written in ANSI C, and can be compiled with many different C compilers. The source is available for purchase. Before buying it, please read the **Interpreter Source License**.

For legal restrictions on the **Euphoria To C Translator**, please download that package from our site.

Language Definition

Objects

- Atoms
- Sequences
- Characters Strings
- Individual Characters
- Comments

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.

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.

Characters Strings and Individual Characters

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

```
        "ABCDEFG"
```

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 "ABCDEFG" 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:

\nnewline
\r carriage return
\t tab
\\ backslash
\" double quote
\' single quote

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

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 treated as a comment.

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.

Relational Operators

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

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

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

Logical Operators

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

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

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

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

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

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

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

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

In some cases **short-circuit** evaluation will be used for expressions containing **and** or **or**.

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.

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
```

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.

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 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.

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.

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)].

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.

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}
```

Precedence Chart

The precedence of operators in expressions is as follows:

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

Thus 2+6*3 means 2+(6*3) rather than (2+6)*3. Operators on the same line above have equal precedence and are evaluated left to right.

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

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.

Declarations

- Identifiers
- Scope
- Specifying the Type of a Variable

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 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.

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*.

In Euphoria, every symbol must be declared before it is used. You can read a Euphoria program from beginning to end without encountering any 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 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.

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.

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 small programs, there is little advantage to defining new types, and beginners may wish to stick with the four predefined types, or even declare all variables as **object**.
For larger programs, strict type definitions can greatly 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 much 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.

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.

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.

Procedure call

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

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

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.

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.

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 Scope.

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", {}}
```

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.

Special Top-Level Statements

Euphoria processes your **.ex** file in one pass, starting at the first line and proceeding through to the last line. When a procedure or function definition is encountered, the routine is checked for syntax and converted into an internal form, but no execution takes place. When a statement that is outside of any routine is encountered, it is checked for syntax, converted into an internal form and then immediately executed. A common practice is to immediately initialize a global variable, just after its declaration. If your **.ex** file contains only routine definitions, but no immediate execution statements, then nothing will happen when you try to run it (other than syntax checking). You need to have an immediate statement to call your main routine (see Example Program). It is quite possible to have a **.ex** file with nothing but immediate statements, for example you might want to use Euphoria as a simple calculator, typing in just one print (or ?) statement 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

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.
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 10 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.
Other than possibly defining a new namespace identifier (see below), an

include statement will be quietly ignored if a file with the same name 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.

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 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 special **with** option where a code number appears after **with**. RDS uses this code to make a file exempt from adding to the statement count in the Public Domain Edition of Euphoria.

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.

Debugging and Profiling

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**.

The full trace facility is part of the Euphoria Complete Edition, but it's enabled in the Public Domain Edition for programs up to 300 statements.

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***.

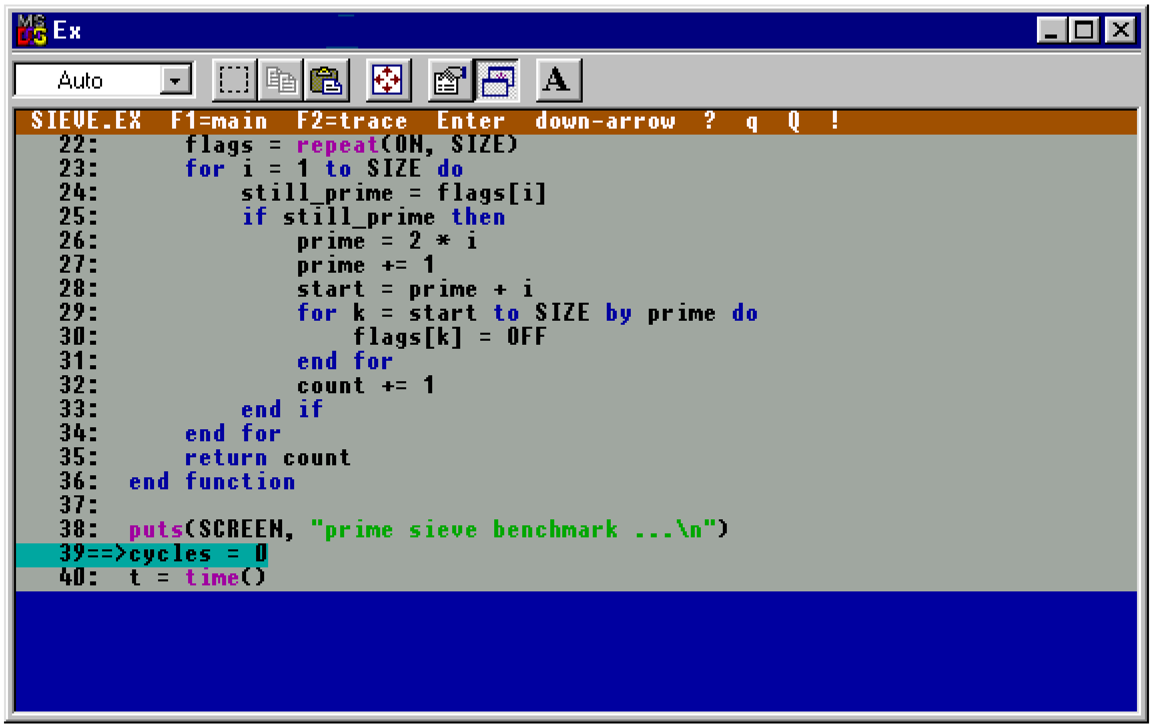
You might want to turn on tracing from within a type. Suppose you run your program and it fails, with the **ex.err** file showing that one of your variables has been set to a strange, although not illegal value, and you wonder how it could have happened. Simply create a type for that variable that executes 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**.

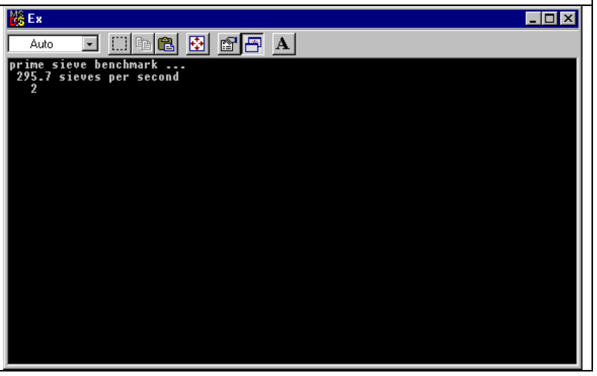
The Trace Screen

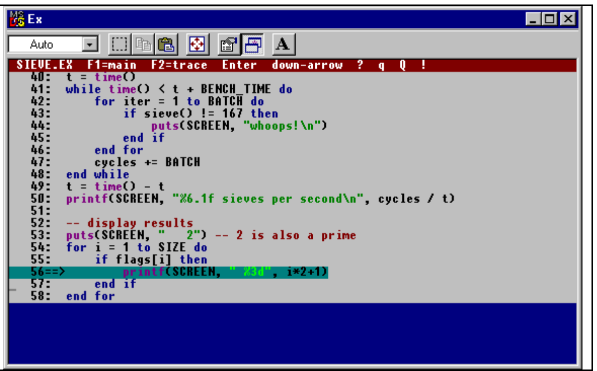
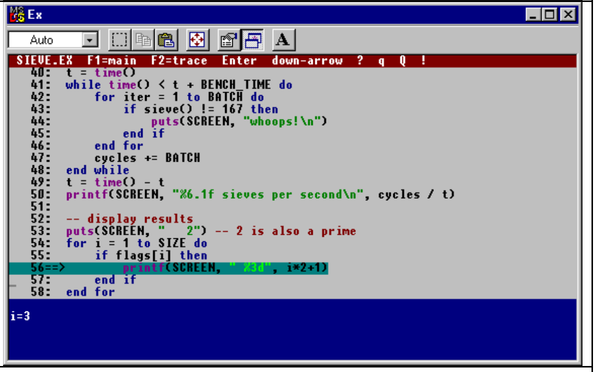
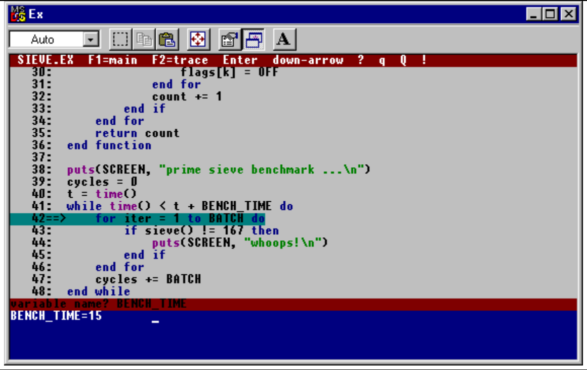
Here is a screen shot of a trace screen. Try clicking on some of its menu items to see other screen shots it may have (same to those underlined items at key listing below:



```
 SIEVE.EX  F1=main  F2=trace  Enter  down-arrow  ?  q  Q  !
   22:        flags = repeat(ON, SIZE)
   23:        for i = 1 to SIZE do
   24:            still_prime = flags[i]
   25:            if still_prime then
   26:                prime = 2 * i
   27:                prime += 1
   28:                start = prime + i
   29:                for k = start to SIZE by prime do
   30:                    flags[k] = OFF
   31:                end for
   32:                count += 1
   33:            end if
   34:        end for
   35:        return count
   36:    end function
   37:
   38:    puts(SCREEN, "prime sieve benchmark ...\n")
   39==>cycles = 0
   40:    t = time()
```

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:

Keys	Action	Screenshot
F1	display main output screen - take a look at your program's output so far	 prime sieve benchmark ... 295.7 sieves per second 2

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. Many variables are displayed automatically as they are assigned a value, but sometimes you will have to explicitly ask for one that is	
	not on display. After hitting **?** you will be prompted for the name of the variable. Variables that are not defined at this point 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	
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**.	

```
SIEVE.EX  F1=main  F2=trace  Enter  down-arrow  ?  q  Q  !
t = time()
while time() < t + BENCH_TIME do
    for iter = 1 to BATCH do
        if sieve() != 167 then
            puts(SCREEN, "whoops!\n")
        end if
    end for
    cycles += BATCH
end while
t = time() - t
printf(SCREEN, "%6.1f sieves per second\n", cycles / t)

-- display results
puts(SCREEN, "   2") -- 2 is also a prime
for i = 1 to SIZE do
    if flags[i] then
==>     printf(SCREEN, " %3d", i*2+1)
    end if
end for
```

```
SIEVE.EX  F1=main  F2=trace  Enter  down-arrow  ?  q  Q  !
t = time()
while time() < t + BENCH_TIME do
    for iter = 1 to BATCH do
        if sieve() != 167 then
            puts(SCREEN, "whoops!\n")
        end if
    end for
    cycles += BATCH
end while
t = time() - t
printf(SCREEN, "%6.1f sieves per second\n", cycles / t)

-- display results
puts(SCREEN, "   2") -- 2 is also a prime
for i = 1 to SIZE do
    if flags[i] then
==>     printf(SCREEN, " %3d", i*2+1)
    end if
end for

i=3
```

```
SIEVE.EX  F1=main  F2=trace  Enter  down-arrow  ?  q  Q  !
                    flags[k] = OFF
                end for
                count += 1
            end if
        end for
        return count
end function

puts(SCREEN, "prime sieve benchmark ...\n")
cycles = 0
t = time()
while time() < t + BENCH_TIME do
==> for iter = 1 to BATCH do
        if sieve() != 167 then
            puts(SCREEN, "whoops!\n")
        end if
    end for
    cycles += BATCH
end while
variable name? BENCH_TIME
BENCH_TIME=15
```

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().

See also: The Trace File

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 Complete Edition of 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**.

Profiling (complete edition only)

If you specify **with profile** (DOS32, Win32 or Linux), 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 **sieve.ex** benchmark program in **demo\bench** under both types of profiling. The results are in **sieve1.pro** (execution-count profiling) and **sieve2.pro** (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.

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.

Library Routines
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:

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

Some routines are only available on one or two of the three platforms. This is noted with "Platform: **DOS32**" or "Platform: **Win32**" or "Platform: **Linux**" in the description of the routine, and with (**DOS32**) or (**Win32**) or (**Linux**) in some other places.

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

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)

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

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

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

Math

These routines can be applied to individual atoms or to sequences of values. See 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...)

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

File and Device I/O

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

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

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

Note:

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

open	- open a file or device
close	- close a file or device
flush	- flush out buffered data to a file or device
lock_file	- lock a file or device
unlock_file	- unlock a file or device
print	- print a Euphoria object with {,,} to show the structure
? x	- shorthand for print(1, x)

sprint	-	return a printed Euphoria object as a string sequence
printf	-	formatted print to a file or device
sprintf	-	formatted print returned as a string sequence
puts	-	output a string sequence to a file or device
getc	-	read the next character from a file or device
gets	-	read the next line from a file or device
get_bytes	-	read the next n bytes from a file or device
prompt_string	-	prompt the user to enter a string
get_key	-	check for key pressed by the user, don't wait
wait_key	-	wait for user to press a key
get	-	read the representation of any Euphoria object from a file
prompt_number	-	prompt the user to enter a number
value	-	read the representation of any Euphoria object from a string
seek	-	move to any byte position within an open file
where	-	report the current byte position in an open file
current_dir	-	return the name of the current directory
chdir	-	change to a new current directory
dir	-	return complete info on all files in a directory
walk_dir	-	recursively walk through all files in a directory
allow_break	-	allow control-c/control-Break to terminate your program or not
check_break	-	check if user has pressed control-c or control-Break

Mouse Support (DOS32)

get_mouse	-	return mouse "events" (clicks, movements)
mouse_events	-	select mouse events to watch for
mouse_pointer	-	display or hide the mouse pointer

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

Special Machine-Dependent Routines

machine_func	- specialized internal operations with a return value
machine_proc	- specialized internal operations with no return value

Debugging Routines

trace	- dynamically turns tracing on or off
profile	- dynamically turns profiling on or off

Graphics & Sound

The following routines let you display information on the screen. 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.

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

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

clear_screen	- clear the screen
position	- set cursor line and column
get_position	- return cursor line and column
graphics_mode	- select a new pixel-graphics or text mode (DOS32)
video_config	- return parameters of current mode
scroll	- scroll text up or down
wrap	- control line wrap at right edge of screen
text_color	- set foreground text color
bk_color	- set background color
palette	- change color for one color number (DOS32)
all_palette	- change color for all color numbers (DOS32)
get_all_palette	- get the palette values for all colors (DOS32)
read_bitmap	- read a bitmap (.bmp) file and return a palette and a 2-d

		sequenceof pixels
save_bitmap	-	create a bitmap (.bmp) file, given a palette and a 2-d sequence of pixels
get_active_page	-	return the page currently being written to (DOS32)
set_active_page	-	change the page currently being written to (DOS32)
get_display_page	-	return the page currently being displayed (DOS32)
set_display_page	-	change the page currently being displayed (DOS32)
sound	-	make a sound on the PC speaker (DOS32)

The following routines work in text modes only:

cursor	-	select cursor shape
text_rows	-	set number of lines on text screen
get_screen_char	-	get one character from the screen (DOS32, Linux)
put_screen_char	-	put one or more characters on the screen (DOS32, Linux)
save_text_image	-	save a rectangular region from a text screen (DOS32, Linux)
display_text_image	-	display an image on the text screen (DOS32, Linux)

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

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

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 complete instructions on how to use it. When using a package such as **Win32Lib** that does not use Euphoria's allocate() function, you can only make limited use of **safe.e**.

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\dos32\callmach.ex** for an example.

peek	-	read one or more bytes from memory
peek4s	-	read 4-byte signed values from memory
peek4u	-	read 4-byte unsigned values from memory
poke	-	write one or more bytes to memory
poke4	-	write 4-byte values into memory
mem_copy	-	copy a block of memory
mem_set	-	set a block of memory to a value
call	-	call a machine language routine
dos_interrupt	-	call a DOS software interrupt routine (DOS32)
allocate	-	allocate a block of memory
free	-	deallocate a block of memory
allocate_low	-	allocate a block of low memory (address less than 1Mb) (DOS32)
free_low	-	free a block allocated with allocate_low (DOS32)
allocate_string	-	allocate a string of characters with 0 terminator
register_block	-	register an externally-allocated block of memory
unregister_block	-	unregister an externally-allocated block of memory
get_vector	-	return address of interrupt handler (DOS32)
set_vector	-	set address of interrupt handler (DOS32)
lock_memory	-	ensure that a region of memory will never be swapped out (DOS32)
int_to_bytes	-	convert an integer to 4 bytes
bytes_to_int	-	convert 4 bytes to an integer
int_to_bits	-	convert an integer to a sequence of bits

bits_to_int	-	convert a sequence of bits to an integer
atom_to_float64	-	convert an atom, to a sequence of 8 bytes in IEEE 64-bit floating-point format
atom_to_float32	-	convert an atom, to a sequence of 4 bytes in IEEE 32-bit floating-point format
float64_to_atom	-	convert a sequence of 8 bytes in IEEE 64-bit floating-point format, to an atom
float32_to_atom	-	convert a sequence of 4 bytes in IEEE 32-bit floating-point format, to an atom
set_rand	-	set the random number generator so it will generate a repeatable series of random numbers
use_vesa	-	force the use of the VESA graphics standard (DOS32)
crash_file	-	specify the file for writing error diagnostics if Euphoria detects an error in your program.
crash_message	-	specify a message to be printed if Euphoria detects an error in your program

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

Calling C Functions

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

Euphoria Database System - 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, very 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 contraints 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 records directly, with no search, if you know the current record number within the table. Record numbers are from 1 to the length (current number of records), of the table.

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 or Linux. The code runs twice as fast on Linux as it does on DOS or Windows. EDS database files can be shared between Linux and DOS/Windows.

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 use these routines

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. That limits the size of a database file to 4 Gb.

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

db_create	- create a new database
db_open	- open an existing database
db_select	- select a database to be the current one
db_close	- close a database
db_create_table	- create a new table within a database
db_select_table	- select a table to be the current one
db_delete_table	- delete a table
db_table_list	- get a list of all the table names in a database
db_table_size	- get the number of records in the current table
db_find_key	- quickly find the record with a certain key value
db_record_key	- get the key portion of a record
db_record_data	- get the data portion of a record
db_insert	- insert a new record into the current table
db_delete_record	- delete a record from the current table

db_replace_data	- replace the data portion of a record
db_compress	- compress a database
db_dump	- print the contents of a database
db_fatal_id	- handle fatal database errors

Listing of Routines

?

Syntax: ? x

Description: This is just a shorthand way of saying: **print(1, x)** - i.e. printing the value of an expression to the standard output.

Example:

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

Comments: ? differs slightly from print() since it will add new-lines to make the output more readable on your screen or wherever you have directed standard output.

See also: 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.

Example:

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

See also: free • allocate_low • peek • poke • call

allocate_low

Platform: DOS32

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

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

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

Example Program: **demo\dos32\dosint.ex**

See also: dos_interrupt • free_low • allocate • peek • poke

allocate_string

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

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

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

Example:
```
atom title

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

Example Program: **demo\win32\window.exw**

See also: allocate • free

allow_break

Syntax: include file.e
allow_break(i)

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

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

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

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

Example:

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

See also: check_break

and_bits

Syntax: x3 = and_bits(x1, x2)

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

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

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

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

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

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

Example 1:

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

Example 2:

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

Example 3:

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

See also: or_bits • xor_bits • not_bits • int_to_bits

append

Syntax: s2 = append(s1, x)

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

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

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

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

```
sequence x

x = {}
for i = 1 to 10 do
    x = append(x, i)
end for
```

```
-- x is now {1,2,3,4,5,6,7,8,9,10}
```

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

```
sequence x, y, z

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

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

See also: prepend • concatenation operator & • sequence-formation operator

arccos

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

Description: Return an angle with cosine equal to x1.

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

A value between 0 and PI radians will be returned.

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

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

Example:

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

See also: cos • arcsin • arctan

arcsin

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

Description: Return an angle with sine equal to x1.

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

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

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

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

Example:

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

See also: sin • arccos • arctan

arctan

Syntax: x2 = arctan(x1)

Description: Return an angle with tangent equal to x1.

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

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

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

Example:

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

See also: tan • arcsin • arccos

atom

Syntax: i = atom(x)

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

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

Example 1:

```
atom a
a = 5.99
```

Example 2:

```
object line

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

See also: sequence • object • integer • atoms and sequences

atom_to_float32

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

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

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

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

Example:

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

See also: atom_to_float64 • float32_to_atom

atom_to_float64

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

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

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

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

Example:

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

See also: atom_to_float32 • float64_to_atom

bits_to_int

Syntax:

```
include machine.e
a = bits_to_int(s)
```

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

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

Example:

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

See also: int_to_bits • operations on sequences

bk_color

Syntax:

```
include graphics.e
bk_color(i)
```

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

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

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

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

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

Example:

```
bk_color(BLACK)
```

See also: text_color • palette

bytes_to_int

Syntax:

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

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

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

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

Example:

```
atom int32

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

See also: int_to_bytes • bits_to_int • peek • peek4s • peek4u • poke

call

Syntax:

```
call(a)
```

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

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

Example Program: **demo\dos32\callmach.ex**

See Also: allocate • free • peek • poke • poke4

call_back

Platform: **Win32, Linux**

Syntax: include dll.e
a = call_back(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 shared library (.so), as a 32-bit "call-back" address for calling your Euphoria routine.

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. 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.

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

See Also: routine_id • **platform.doc**

c_func

Platform: **Win32, Linux**

Syntax: a = c_func(i, s)

Description: Call the C function 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**

Syntax: c_proc(i, s)

Description: Call the C function 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 • 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, a subprocess can change the current directory for itself, but this won't affect the current directory of its parent process.

Example:

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

See Also: current_dir

check_break

Syntax: include file.e
i = check_break()

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

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

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

Example:

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

See Also: allow_break • get_key

clear_screen

Syntax: clear_screen()

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

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

See Also: bk_color • graphics_mode

close

Syntax: close(fn)

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

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

See Also: open • flush

command_line

Syntax: s = command_line()

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

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

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

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

Example 1:

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

cmd = command_line()

-- cmd will be:

     {
     "C:\EUPHORIA\BIN\EX.EXE",
     "myprog",
     "myfile.dat",
     "12345",
     "the end"
     }
```

Example 2:

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

cmd = command_line()

-- cmd will be:

        {
        "C:\MYFILES\MYPROG.EXE",
        "C:\MYFILES\MYPROG.EXE",   -- spacer
        "myfile.dat",
        "12345",
        "the end"
        }

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

See Also: getenv

compare

Syntax: i = compare(x1, x2)

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

Example 1:

```
x = compare({1,2,{3,{4}},5}, {2-1,1+1,{3,{4}},6-1})
-- identical, x is 0
```

Example 2:

```
if compare("ABC", "ABCD") < 0 then        -- -1
        -- will be true: ABC is "less" because it is shorter
end if
```

Example 3:

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

Example 4:

```
x = compare('a', "a")
-- x will be -1 because 'a' is an atom
-- while "a" is a sequence
```

See Also: equal • relational operators • operations on sequences

cos

Syntax: x2 = cos(x1)

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

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

Example:

```
x = cos({.5, .6, .7})
-- x is {0.8775826, 0.8253356, 0.7648422}
```

See Also: sin • tan • log • sqrt

crash_file

Syntax: include machine.e
crash_file(s)

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

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

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

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

Example:

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

See Also: abort • crash_message • debugging and profiling

crash_message

Syntax: include machine.e
crash_message(s)

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

Comments: Normally Euphoria prints a diagnostic message such as "syntax error" or "divide by zero" on the screen, as well as dumping debugging information into **ex.err**. Euphoria's error messages will not be meaningful for your users unless they happen to be Euphoria programmers. By calling crash_message() you can control the message that will appear on the screen. Debugging information will still be stored in **ex.err**. You won't lose any information by doing this.

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

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

Example:

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

See Also: abort • crash_file • debugging and profiling

current_dir

Syntax: include file.e
s = current_dir()

Description: Return the name of the current working directory.

Example:

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

See Also: dir • chdir • getenv

cursor

Syntax:

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

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

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

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

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

Example:

```
cursor(BLOCK_CURSOR)
```

See Also: graphics_mode • text_rows

custom_sort

Syntax:

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

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

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

Example Program: **demo\csort.ex**

See Also: sort • compare • routine_id

date

Syntax: s = date()

Description: Return a sequence with the following information:

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

Example:

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

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

See Also: time

define_c_func

Platform: **Win32, Linux**

Syntax: include dll.e
i1 = define_c_func(a, s1, s2, i2)

Description: Define the characteristics of a C function that you wish to call from your Euphoria program. A small integer, known as a **routine id**, will be returned, or -1 if the function can't be found. a is an address returned by open_dll(). s1 is the name of the function. s2 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**:

```
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: The **routine id**, i1, can be passed to c_func() when you want to call the C function.

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.

In C (on Win32 and Linux), parameter types which use 4 bytes or less are all passed the same way, so it is not necessary to be exact. However the distinction between signed and unsigned may be important when you specify the return type of a function.

Currently, there is no way to pass a C structure by value or get a C structure as a return result. You can only pass a pointer to a structure and get a pointer to a structure as a result.

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

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

See Also: c_func • define_c_proc • c_proc • open_dll • **platform.doc**

define_c_proc

Platform: **Win32, Linux**

Syntax: include dll.e
i1 = define_c_proc(a, s1, s2)

Description: Define the characteristics of a C function that you wish to call as a procedure from your Euphoria program. A small integer, known as a **routine id**, will be returned, or -1 if the function can't be found. a is an address returned by open_dll(). s1 is the name of the function. s2 is a list of the parameter types for the function. A list of C types is contained in **dll.e**, and shown above.

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: The **routine id**, i1, can be passed to c_proc(), when you want to call the C function.

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

In C (on Win32 and Linux), 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 = -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**

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

Description: a2 is the address of a Linux 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**, the only attribute currently available is 'd'.

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

Win32: The file name returned in D_NAME will be a long file name.

Example:

```
d = dir(current_dir())

-- d might have:

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

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

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

See Also: wildcard_file • current_dir • open

display_image

Platform: **DOS32**

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

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

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

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

Example:

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

Example Program: **demo\dos32\bitmap.ex**

See Also: save_image • read_bitmap • display_text_image

display_text_image

Platform: **DOS32, Linux**

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.

Example:

```
sequence registers

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

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

Example Program: **demo\dos32\dosint.ex**

See Also: allocate_low • free_low

draw_line

Platform: **DOS32**

Syntax:

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

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

Example:

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

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

See Also: polygon • ellipse • pixel

ellipse

Platform: **DOS32**

Syntax:

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

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

Example:

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

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

Example Program: **demo\dos32\sb.ex**

See Also: polygon • draw_line

equal

Syntax: i = equal(x1, x2)

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

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

Example 1:

```
if equal(PI, 3.14) then
    puts(1, "give me a better value for PI!\n")
end if
```

Example 2:

```
if equal(name, "George") or equal(name, "GEORGE") then
    puts(1, "name is George\n")
end if
```

See Also: compare • equals operator (=)

find

Syntax: i = find(x, s)

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

Example 1:

```
location = find(11, {5, 8, 11, 2, 3})
-- location is set to 3
```

Example 2:

```
names = {"fred", "rob", "george", "mary", ""}
location = find("mary", names)
-- location is set to 4
```

See Also: match • compare

float32_to_atom

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

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

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

Example:

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

See Also: float64_to_atom • atom_to_float32

float64_to_atom

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

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

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

Example:

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

See Also: float32_to_atom • atom_to_float64

floor

Syntax: x2 = floor(x1)

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

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

Example:

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

See Also: remainder

flush

Syntax: include file.e
flush(fn)

Description: When you write data to a file, Euphoria normally stores the data in a memory buffer until a large enough chunk of data has accumulated. This large chunk can then be written to disk very efficiently. Sometimes you may want to force, or flush, all data out immediately, even if the memory buffer is not full. To do this you must call flush(fn), where fn is the file number of a file open for writing or appending.

Comments: When a file is closed, (see close()), all buffered data is flushed out. When a program terminates, all open files are flushed and closed automatically.

Use flush() when another process may need to see all of the data written so far, but you aren't ready to close the file yet.

Example:

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

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

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

See Also: close • lock_file

free

Syntax: include machine.e
free(a)

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

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

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

Example

Program: demo\dos32\callmach.ex

See Also: allocate • free_low

free_console

Platform: Win32, Linux

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, 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, free_console() will set the terminal parameters back to normal, undoing the effect that curses has on the screen.

In a Linux xterm window, a call to free_console(), without any further printing to the screen or reading from the keyboard, will eliminate the "Press Enter" prompt that Euphoria normally issues at the end of execution.

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

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

There's a Win32 API routine, FreeConsole() that does something similar to free_console(). You should use free_console(), because it lets the interpreter know that there is no longer a console.

See Also: clear_screen • **platform.doc**

free_low

Platform: DOS32

Syntax: include machine.e
free_low(i)

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

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

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

Example Program: **demo\dos32\dosint.ex**

See Also: allocate_low • dos_interrupt • free

get

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

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

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

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

Each call to get() picks up where the previous call left off. For instance, a series of 5 calls to get() would be needed to read in:

```
99 5.2 {1,2,3} "Hello" -1
```

On the sixth and any subsequent call to get() you would see a GET_EOF status. If you had something like:

```
{1, 2, xxx}
```

in the input stream you would see a GET_FAIL error status because xxx is not a Euphoria object.

Multiple "top-level" objects in the input stream must be separated from each other with one or more "whitespace" characters (blank, tab, \r or \n). Whitespace is not necessary ***within*** a top-level object. A call to get() will read one entire top-level object, plus one additional (whitespace) character.

Comments: The combination of print() and get() can be used to save a Euphoria object to disk and later read it back. This technique could be used to implement a database as one or more large Euphoria sequences stored in disk files. The sequences could be read into memory, updated and then written back to disk after each series of transactions is complete. Remember to write out a whitespace character (using puts()) after each call to print().

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

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

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

    {GET_SUCCESS, 77.5}

-- whereas gets(0) would return:

    "77.5\n"
```

Example Program: **demo\mydata.ex**

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

get_active_page

Platform: DOS32

Syntax: include image.e
i = get_active_page()

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

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

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

See Also: set_active_page • get_display_page • video_config

get_all_palette

Platform: DOS32

Syntax: include image.e
s = get_all_palette()

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

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

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

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

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

get_bytes

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

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

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

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

Example:

```
include get.e

integer fn
fn = open("temp", "rb")     -- an existing file

sequence whole_file
whole_file = {}

sequence chunk

while 1 do
    chunk = get_bytes(fn, 100)        -- read 100 bytes at a time
    whole_file &= chunk           -- chunk might be empty, that's ok
    if length(chunk) < 100 then
        exit
    end if
end while
close(fn)
? length(whole_file) -- should match DIR size of "temp"
```

See Also: getc • gets

get_display_page

Platform: **DOS32**

Syntax: include image.e
i = get_display_page()

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

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

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

See Also: set_display_page • get_active_page • video_config

get_key

Syntax: i = get_key()

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

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

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

See Also: wait_key • getc

get_mouse

Platform: **DOS32, Linux**

Syntax: include mouse.e
x1 = get_mouse()

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

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

```
global constant MOVE          =    1,
                LEFT_DOWN     =    2,
                LEFT_UP       =    4,
                RIGHT_DOWN    =    8,
                RIGHT_UP      =   16,
                MIDDLE_DOWN   =   32,
                MIDDLE_UP     =   64
```

x and y are the coordinates of the mouse pointer at the time that the event occurred. get_mouse() returns immediately with either a -1 or a mouse event. It does not wait for an event to occur. You must check it frequently enough to avoid missing an event. When the next event occurs, the current event will be lost, if you haven't read it. In practice it is not hard to catch almost all events. Losing a MOVE event is generally not too serious, as the next MOVE will tell you where the mouse pointer is.

Sometimes multiple events will be reported. For example, if the mouse is moving when the left button is clicked, get_mouse() will report an event value of LEFT_DOWN+MOVE, i.e. 2+1 or 3. For this reason you should test for a particular event using and_bits(). See examples below.

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

In DOS32 text modes you need to scale the x and y coordinates to get line and column positions. In Linux, no scaling is required - x and y correspond to the line and column on the screen, with (1,1) at the top left.

In DOS32, you need a DOS mouse driver to use this routine. In Linux, GPM Server must be running.

In Linux, mouse movement events are not reported in an xterm window, only in the text console.

In Linux, LEFT_UP, RIGHT_UP and MIDDLE_UP are not distinguishable from one another.

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

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

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

The x,y coordinate returned could be that of the very tip of the mouse pointer or might refer to the pixel pointed-to by the mouse pointer. Test this if you are trying to read the pixel color using get_pixel(). You may have to read x-1,y-1 instead.

Example 1: a return value of:

```
{2, 100, 50}
```

would indicate that the left button was pressed down when the mouse pointer was at location x=100, y=50 on the screen.

Example 2: To test for LEFT_DOWN, write something like the following:

```
object event

while 1 do
    event = get_mouse()
    if sequence(event) then
        if and_bits(event[1], LEFT_DOWN) then
            -- left button was pressed
            exit
        end if
    end if
end while
```

See Also: mouse_events • mouse_pointer • and_bits

get_pixel

Platform: **DOS32**

Syntax: x = get_pixel(s)

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

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

Points off the screen have unpredictable color values.

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

Example:

```
object x

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

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

See Also: pixel • graphics_mode • get_position

get_position

Syntax: include graphics.e
s = get_position()

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

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

The coordinate system for displaying text is different from the one for displaying pixels. Pixels are displayed such that the top-left is (x=0,y=0) and the first coordinate controls the horizontal, left-right location. In pixel-graphics modes you can display both text and pixels. get_position() returns the current line and column for the text that you are displaying, not the pixels that you may be plotting. There is no corresponding routine for getting the current pixel position.

See Also: position • get_pixel

get_screen_char

Platform: DOS32, Linux

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 a DOS environment variable. If the variable is undefined return -1.

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

Example:

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

See Also: command_line

gets

Syntax: x = gets(fn)

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

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

After reading a line of text from the keyboard, you should normally output a \n character, e.g. puts(1, '\n'), before printing something. Only on the last line of the screen does the operating system automatically scroll the screen and advance to the next line.

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

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

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

Example 1:

```
sequence buffer
object line
integer fn

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

buffer = {}
while 1 do
    line = gets(fn)
    if atom(line) then
        exit   -- -1 is returned at end of file
    end if
    buffer = append(buffer, line)
end while
```

Example 2:

```
object line

puts(1, "What is your name?\n")
line = gets(0)                  -- read standard input (keyboard)
line = line[1..length(line)-1]      -- get rid of \n character at end
puts(1, '\n')                   -- necessary
puts(1, line & " is a nice name.\n")
```

See Also: getc • puts • open

graphics_mode

Platform: **DOS32**

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

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

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

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

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

On the **Win32** and **Linux** platforms, graphics_mode() will allocate a plain, text mode console if one does not exist yet. It will then return 0, no matter what value is passed as i2.

Example:

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

See Also: text_rows • video_config

instance

Platform: **Win32**

Syntax:

```
include misc.e
i = instance()
```

Description: Return a handle to the current program.

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

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

On **DOS32 and Linux**, 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, 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, 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, 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: Use this to ensure that all code and data required for handling interrupts is kept in memory at all times while your program is running.

Example Program: **demo\dos32\hardint.ex**

See Also: get_vector • set_vector

log

Syntax: x2 = log(x1)

Description: Return the natural logarithm of x1.

Comments: This function may be applied to an atom or to all elements of a sequence. Note that log is only defined for positive numbers. Your program will abort with a message if you try to take the log of a negative number or zero.

Example:

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

See Also: sin • cos • tan • sqrt

lower

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

Description: Convert an atom or sequence to lower case.

Example:

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

See Also: upper

machine_func

Syntax: x1 = machine_func(a, x)

Description: see machine_proc() below.

machine_proc

Syntax: machine_proc(a, x)

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

See Also: machine_func

match

Syntax: i = match(s1, s2)

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

Example:

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

See Also: find • compare • wildcard_match

mem_copy

Syntax: mem_copy(a1, a2, i)

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

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

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

Example:

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

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

mem_set

Syntax: mem_set(a1, i1, i2)

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

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

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

Example:

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

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

message_box

Platform: **Win32**

Syntax: include msgbox.e

i = message_box(s1, s2, x)

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

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

Example:

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

Example Program: **demo\win32\email.exw**

mouse_events

Platform: **DOS32, Linux**

Syntax: include mouse.e
mouse_events(i)

Description: Use this procedure to select the mouse events that you want get_mouse() to report. By default, get_mouse() will report all events. mouse_events() can be called at various stages of the execution of your program, as the need to detect events changes. Under Linux, mouse_events() currently has no effect.

Comments: It is good practice to ignore events that you are not interested in, particularly the very frequent MOVE event, in order to reduce the chance that you will miss a significant event.

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

Example:

```
mouse_events(LEFT_DOWN + LEFT_UP + RIGHT_DOWN)
-- will restrict get_mouse() to reporting the left button
-- being pressed down or released, and the right button
-- being pressed down. All other events will be ignored.
```

See Also: get_mouse • mouse_pointer

mouse_pointer

Platform: **DOS32, Linux**

Syntax: include mouse.e
mouse_pointer(i)

Description: If i is 0 hide the mouse pointer, otherwise turn on the mouse pointer. Multiple calls to hide the pointer will require multiple calls to turn it back on. The first call to either get_mouse() or mouse_events(), will also turn the pointer on (once). Under Linux, mouse_pointer() currently has no effect.

Comments: It may be necessary to hide the mouse pointer temporarily when you update the screen.

After a call to text_rows() you may have to call mouse_pointer(1) to see the mouse pointer again.

See Also: get_mouse • mouse_events

not_bits

Syntax: x2 = not_bits(x1)

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

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

The argument must be representable as a 32-bit number, either signed or unsigned. If you intend to manipulate full 32-bit values, you should declare your variables as **atom**, rather than integer. Euphoria's integer type is limited to 31-bits.

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

Example:

```
a = not_bits(#000000F7)
```

```
-- a is -248 (i.e. FFFFFF08 interpreted as a negative number)
```

See Also: and_bits • or_bits • xor_bits • int_to_bits

object

Syntax: i = object(x)

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

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

Example:

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

See Also: integer • atom • sequence

open

Syntax: fn = open(st1, st2)

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

"r"	-	open text file for reading
"rb"	-	open binary file for reading
"w"	-	create text file for writing
"wb"	-	create binary file for writing
"u"	-	open text file for update (reading and writing)
"ub"	-	open binary file for update
"a"	-	open text file for appending
"ab"	-	open binary file for appending

Files opened for read or update must already exist. Files opened for write or append will be created if necessary. A file opened for write will be set to 0 bytes. Output to a file opened for append will start at the end of file.

Output to **text files** will have carriage-return characters automatically added before linefeed characters. On input, these carriage-return characters are removed. A control-Z character (ASCII 26) will signal an immediate end of file.

I/O to **binary files** is not modified in any way. Any byte values from 0 to 255 can be read or written.

Some typical devices that you can open are:

"CON"	-	the console (screen)
"AUX"	-	the serial auxiliary port
"COM1"	-	serial port 1
"COM2"	-	serial port 2
"PRN"	-	the printer on the parallel port
"NUL"	-	a non-existent device that accepts and discards output.

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

Win32, Linux: Long filenames are fully supported for reading and writing and creating.

Example:

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

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

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

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

See Also: close

open_dll

Platform: **Win32, Linux**

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

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

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

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

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

Example:

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

See Also: define_c_func • define_c_proc • define_c_var • c_func • c_proc • **platform.doc**

or_bits

Syntax: x3 = or_bits(x1, x2)

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

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

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

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

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

Example 1:

```
a = or_bits(#0F0F0000, #12345678)
-- a is #1F3F5678
```

Example 2:

```
a = or_bits(#FF, {#123456, #876543, #2211})
-- a is {#1234FF, #8765FF, #22FF}
```

See Also: and_bits • xor_bits • not_bits • int_to_bits

palette

Platform: **DOS32**

Syntax:

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

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

Example:

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

See Also: all_palette

peek

Syntax:

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

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

Comments: Since addresses are 32-bit numbers, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as **atoms**.

It is faster to read several bytes at once using the second form of peek() than it is to read one byte at a time in a loop.

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

Example: The following are equivalent:

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

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

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

peek4s

Syntax:

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

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

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

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

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

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

Example: The following are equivalent:

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

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

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

peek4u

Syntax:

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

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

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

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

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

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

Example: The following are equivalent:

```
 -- method 1
```

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

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

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

PI

Syntax: include misc.e
PI

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

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

Example:

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

See Also: sin • cos • tan

pixel

Platform: **DOS32**

Syntax: pixel(x1, s)

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

Comments: When x1 is a sequence, a very fast algorithm is used to put the pixels on the screen. It is much faster to call pixel() once, with a sequence of pixel colors, than it is to call it many times, plotting one pixel color at a time. In graphics mode 19, pixel() is highly optimized.

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** or **Linux**.

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.

The include file **misc.e** contains the following constants:

```
    global constant DOS32 = 1,
                    WIN32 = 2,
                    LINUX = 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)
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\dos32\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

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: ? • puts • printf • get

printf

Syntax: printf(fn, st, x)

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

%d - print an atom as a decimal integer
%x - print an atom as a hexadecimal integer
%o - print an atom as an octal integer
%s - print a sequence as a string of characters, or print an atom as a single character
%e - print an atom as a floating point number with exponential notation
%f - print an atom as a floating-point number with a decimal point but no exponent
%g - print an atom as a floating point number using either the **%f** or **%e** format, whichever seems more appropriate
%% - print the '%' character itself

Field widths can be added to the basic formats, e.g. %5d, %8.2f, %10.4s. The number before the decimal point is the minimum field width to be used. The number after the decimal point is the precision to be used.

If the field width is negative, e.g. %-5d then the value will be left-justified within the field. Normally it will be right-justified. If the field width starts with a leading 0, e.g. %08d then leading zeros will be supplied to fill up the field. If the field width starts with a '+' e.g. %+7d then a plus sign will be printed for positive values.

Comments: Watch out for the following common mistake:

```
name="John Smith"
printf(1, "%s", name) -- error!
```

This will print only the first character, J, of name, as each element of name is taken to be a separate value to be formatted. You must say this instead:

```
name="John Smith"
printf(1, "%s", {name})       -- correct!
```

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

Example 1:

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

     The interest rate is:     7.88
```

Example 2:

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

     John Smith,    97
```

Example 3:

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

     ABCD       $ XXX
```

See Also: sprintf • puts • open

profile

Syntax: profile(i)

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

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

Example 1:

```
with profile_time
```

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

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

prompt_number

Syntax:

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

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

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

Example 1:

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

Example 2:

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

See Also: get • prompt_string

prompt_string

Syntax:

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

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

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

Example: name = prompt_string("What is your name? ")

See Also: gets • prompt_number

put_screen_char

Platform: **DOS32, Linux**

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

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

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

It's faster to write several characters to the screen with a single call to put_screen_char() than it is to write one character at a time.

Example:

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

See Also: get_screen_char • display_text_image

puts

Syntax: puts(fn, x)

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

Comments: When you output a sequence of bytes it must not have any (sub)sequences within it. It must be a **sequence of atoms** only.

(Typically a string of ASCII codes).

Avoid outputting 0's to the screen or to standard output. Your output may get truncated.

Example 1: puts(SCREEN, "Enter your first name: ")

Example 2: puts(output, 'A') -- the single byte 65 will be sent to output

See Also: printf • gets • open

rand

Syntax: x2 = rand(x1)

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

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

Example:

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

See Also: set_rand

read_bitmap

Syntax:

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

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

Bitmaps of 2, 4, 16 or 256 colors are supported. If the file is not in a good format, an error code (atom) is returned instead:

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

Comments: You can create your own bitmap picture files using Windows Paintbrush and many other graphics programs. You can then incorporate these pictures into your Euphoria programs.

Example:

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

Example Program: **demo\dos32\bitmap.ex**

See Also: palette • all_palette • display_image • save_bitmap

register_block

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

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

Comments: This routine is only meant to be used for **debugging purposes**. **safe.e** tracks the blocks of memory that your program is allowed to peek(), poke(), mem_copy(), etc. These are normally just the blocks that you have allocated using Euphoria's allocate() or allocate_low() routines, and which you have not yet freed using Euphoria's free() or free_low(). In some cases, you may acquire additional, external, blocks of memory, perhaps as a result of calling a C routine. If you are debugging your program using **safe.e**, you must register these external blocks of memory or **safe.e** will prevent you from accessing them. When you are finished using an external block you can unregister it using unregister_block().

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

Example 1:

```
atom addr
addr = c_func(x, {})
```

```
register_block(addr, 5)
poke(addr, "ABCDE")
unregister_block(addr)
```

See Also: unregister_block • **safe.e**

remainder

Syntax: x3 = remainder(x1, x2)

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

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

Example 1:

```
a = remainder(9, 4)
-- a is 1
```

Example 2:

```
s = remainder({81, -3.5, -9, 5.5}, {8, -1.7, 2, -4})
-- s is {1, -0.1, -1, 1.5}
```

Example 3:

```
s = remainder({17, 12, 34}, 16)
-- s is {1, 12, 2}
```

Example 4:

```
s = remainder(16, {2, 3, 5})
-- s is {0, 1, 1}
```

See Also: floor

repeat

Syntax: s = repeat(x, a)

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

Comments: When you repeat a sequence or a floating-point number the interpreter does not actually make multiple copies in memory. Rather, a single copy is "pointed to" a number of times.

Example 1: `repeat(0, 10)          -- {0,0,0,0,0,0,0,0,0,0}`
Example 2: `repeat("JOHN", 4)  -- {"JOHN", "JOHN", "JOHN", "JOHN"}`

```
-- The interpreter will create only one copy of "JOHN"
-- in memory
```

See Also: append • prepend • sequence-formation operator

reverse

Syntax:

```
include misc.e
s2 = reverse(s1)
```

Description: Reverse the order of elements in a sequence.

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

Example 1:

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

Example 2:

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

Example 3:

```
reverse({99})               -- {99}
```

Example 4:

```
reverse({})                 -- {}
```

See Also: append • prepend • repeat

routine_id

Syntax:

```
i = routine_id(st)
```

Description: Return an integer id number, known as a **routine id**, for a user-defined Euphoria procedure or function. The name of the procedure or function is given by the string sequence st. -1 is returned if the named routine can't be found.

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

The routine named by st must be visible, i.e. callable, at the place where routine_id() is used to get the id number. Indirect calls to the routine can appear earlier in the program than the definition of the routine, but the id number can only be obtained in code that comes *after* the definition of the routine - see example 2 below.

Once obtained, a valid **routine id** can be used at ***any*** place in the program to call a routine indirectly via call_proc()/call_func().

Some typical uses of routine_id() are:

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

Note that C routines, callable by Euphoria, also have routine id's. See define_c_proc() and define_c_func().

Example 1:

```
procedure foo()
    puts(1, "Hello World\n")
end procedure

integer foo_num
foo_num = routine_id("foo")

call_proc(foo_num, {})    -- same as calling foo()
```

Example 2:

```
function apply_to_all(sequence s, integer f)
    -- apply a function to all elements of a sequence
    sequence result
    result = {}
    for i = 1 to length(s) do
        -- we can call add1() here although it comes later in the
program
        result = append(result, call_func(f, {s[i]}))
    end for
    return result
end function

function add1(atom x)
    return x + 1
end function

-- add1() is visible here, so we can ask for its routine id
? apply_to_all({1, 2, 3}, routine_id("add1"))
-- displays {2,3,4}
```

See Also: call_proc • call_func • call_back • define_c_func • define_c_proc •

save_bitmap

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

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

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

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

The result will be one of the following codes:

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

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

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

save_bitmap() produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with read_bitmap(). Windows Paintbrush and some other tools do not support 4-color bitmaps.

Example:

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

See Also: save_image • read_bitmap • save_screen • get_all_palette

save_image

Platform: DOS32

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

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

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

See Also: display_image • save_text_image

save_screen

Platform: DOS32

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

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

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

The result will be one of the following codes:

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

Comments: save_screen() produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with read_bitmap(). Windows Paintbrush and some other tools do not support 4-color bitmaps.

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

Example 1:

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

Example 2:

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

See Also: save_image • read_bitmap • save_bitmap

save_text_image

Platform: **DOS32, Linux**

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

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

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

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

This routine only works in text modes.

You might use this function in a text-mode graphical user interface to save a portion of the screen before displaying a drop-down menu, dialog box, alert box etc. On DOS32, if you are flipping video pages, note that this function reads from the current active page.

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

```
Hello
World
```

And you execute:

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

Then s is something like:

```
{"H-e-l-l-o-", "W-o-r-l-d-"}
```

where '-' indicates the attribute bytes.

See Also: display_text_image • save_image • set_active_page • get_screen_char

scroll

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

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

Comments: You could perform the scrolling operation using a series of calls to puts(), but scroll() is much faster.

The position of the cursor after scrolling is not defined.

Example Program: **bin\ed.ex**

See Also: clear_screen • text_rows

seek

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

Description: Seek (move) to any byte position in the file fn or to the end of file if i2 is -1. For each open file there is a current byte position that is updated as a result of I/O operations on the file. The initial file position is 0 for files opened for read, write or update. The initial position is the end of file for files opened for append. The value returned by seek() is 0 if the seek was successful, and non-zero if it was unsuccessful. It is possible to seek past the end of a file. If you seek past the end of the file, and write some data, undefined bytes will be inserted into the gap between the original end of file and your new data.

Comments: After seeking and reading (writing) a series of bytes, you may need to call seek() explicitly before you switch to writing (reading) bytes, even though the file position should already be what you want.

This function is normally used with files opened in binary mode. In text mode, DOS converts CR LF to LF on input, and LF to CR LF on output, which can cause great confusion when you are trying to count bytes.

Example:

```
include file.e

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

See Also: where • open

sequence

Syntax: i = sequence(x)

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

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

Example 1:

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

Example 2:

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

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

set_active_page

Platform: DOS32

Syntax: include image.e
set_active_page(i)

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

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

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

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

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

Example:

```
include image.e

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

```
set_display_page(0)          -- "This is ..." becomes visible again
set_active_page(0)
```

See Also: get_active_page • set_display_page • video_config

set_display_page

Platform: DOS32

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

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

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

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

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

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

Example: See set_active_page() example.

See Also: get_display_page • set_active_page • video_config

set_rand

Syntax:
```
include machine.e
set_rand(i1)
```

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

Comments: Normally the numbers returned by the rand() function are totally unpredictable, and will be different each time you run your program. Sometimes however you may wish to repeat the same series of numbers, perhaps because you are trying to debug your program, or maybe you want the ability to generate the same output (e.g. a random picture) for your user upon request.

Example:

```
sequence s, t
s = repeat(0, 3)
t = s

set_rand(12345)
s[1] = rand(10)
s[2] = rand(100)
s[3] = rand(1000)

set_rand(12345)     -- same value for set_rand()
t[1] = rand(10)     -- same arguments to rand() as before
t[2] = rand(100)
t[3] = rand(1000)   -- at this point s and t will be identical
```

See Also: rand

set_vector

Platform: **DOS32**

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

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

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

The 16-bit segment can be the code segment used by Euphoria. To get the value of this segment see **demo\dos32\hardint.ex**. The offset can be the 32-bit value returned by allocate(). Euphoria runs in **protected mode** with the code segment and data segment pointing to the same physical memory, but with different access modes.

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

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

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

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

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

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

Example:

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

Example Program: **demo\dos32\hardint.ex**

See Also: get_vector • lock_memory • allocate

sin

Syntax: x2 = sin(x1)

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

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

Example:

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

See Also: cos • tan

sleep

Syntax: include misc.e
sleep(i)

Description: Suspend execution for i seconds.

Comments: On Win32 and Linux, the operating system will suspend your process and schedule other processes. On DOS32, your program will go into a busy loop for i seconds, during which time other processes may run, but they will compete with your process for the CPU.

Example:

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

See Also: lock_file • abort • time

sort

Syntax:

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

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

Example 1:

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

Example 2:

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

Example 3:

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

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

-- after sorting, database is:

     {      {"Clinton", 88.7, 44},
            {"Jones",   77.2, 31},
            {"Smith",   95.0, 29}   }
```

See Also: custom_sort • compare • match • find

sound

Platform: **DOS32**

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

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

Comments: On **Win32** and **Linux** 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 shell.

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

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

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

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

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

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

Example 1:

```
system("copy temp.txt a:\\temp.bak", 2)
-- note use of double backslash in literal string to get
-- single backslash
```

Example 2:

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

Example Program: **bin\install.ex**

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

system_exec

Syntax: i1 = system_exec(st, i2)

Description: Try to run the command given by st. st must be a command to run an executable program, possibly with some command-line arguments. If the program can be run, i1 will be the exit code from the program. If it is not possible to run the program, system_exec() will return -1. i2 is a code that indicates what to do about the graphics mode when system_exec() is finished. These codes are the same as for system():

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

When i2 is 1, a beep sound will be made and the program will wait for the user to press a key before the previous graphics mode is restored.

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

Comments: On DOS32 or Win32, system_exec() will only run **.exe** and **.com** programs. To run **.bat** files, or built-in DOS commands, you need system(). Some commands, such as DEL, are not programs, they are actually built-in to the command interpreter.

system_exec() does not allow the use of command-line redirection in the command string st.

exit codes from DOS or Windows programs are normally in the range 0 to 255, with 0 indicating "success".

You can run a Euphoria program using system_exec(). A Euphoria program can return an exit code using abort().

system_exec() does not start a new DOS shell.

Example 1:

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

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

Example 2:

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

See Also: system • abort

tan

Syntax: x2 = tan(x1)

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

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

Example:

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

See Also: sin • cos • arctan

text_color

Syntax: include graphics.e
text_color(i)

Description: Set the foreground text color. Add 16 to get blinking text in some modes.

See **graphics.e** for a list of possible colors.

Comments: Text that you print ***after*** calling text_color() will have the desired color.

When your program terminates, the last color that you selected and actually printed on the screen will remain in effect. Thus you may have to print something, maybe just '\n', in WHITE to restore white text, especially if you are at the bottom line of the screen, ready to scroll up.

Example: text_color(BRIGHT_BLUE)

See Also: bk_color

text_rows

Platform: **DOS32, Win32**

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

Description: Set the number of lines on a text-mode screen to i1 if possible. i2 will be set to the actual new number of lines.

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

See Also: graphics_mode

tick_rate

Platform: **DOS32**

Syntax: include machine.e
tick_rate(a)

Description: Specify the number of clock-tick interrupts per second. This determines the precision of the time() library routine. It also affects the sampling rate for time profiling.

Comments: **tick_rate()** is ignored on **Win32** and **Linux**. The time resolution on Win32 is always 100 ticks/second.

On a PC the clock-tick interrupt normally occurs at 18.2 interrupts per second. tick_rate() lets you increase that rate, but not decrease it.

tick_rate(0) will restore the rate to the normal 18.2 rate. Euphoria will also restore the rate automatically when it exits, even when it finds an error in your program.

If a program runs in a DOS window with a tick rate other than 18.2, the time() function will not advance unless the window is the active window.

While **ex.exe** is running, the system will maintain the correct time of day. However if **ex.exe** should crash (e.g. you see a "CauseWay..." error) while the tick rate is high, you (or your user) may need to reboot the machine to restore the proper rate. If you don't, the system time may advance too quickly. This problem does not occur on **Windows 95/98/NT**, only on **DOS** or **Windows 3.1**. You will always get back the correct time of day from the battery-operated clock in your system when you boot up again.

Example:

```
tick_rate(100)
-- time() will now advance in steps of .01 seconds
-- instead of the usual .055 seconds
```

See Also: time • time profiling

time

Syntax: a = time()

Description: Return the number of seconds since some fixed point in the past.

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

The resolution with **DOS32** is normally about 0.05 seconds. On **Win32 and Linux** it's about 0.01 seconds.

Under **DOS32** you can improve the resolution by calling tick_rate().

Under **DOS32** the period of time that you can measure is limited to 24 hours. After that, the value returned by time() will reset and start over.

Example:

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

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

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

See Also: date • tick_rate

trace

Syntax: with trace
trace(i)

Description: If i is 1 or 2, turn on full-screen interactive statement tracing/debugging. If i is 3, turn on tracing of statements to a file called **ctrace.out**. If i is 0, turn off tracing. When i is 1 a color display appears. When i is 2 a monochrome trace display appears. Tracing can only occur in routines that were compiled "**with trace**", and trace() has no effect unless it is executed in a "**with trace**" **section** of your program.

See Part I - 3.1 Debugging for a full discussion of tracing / debugging.

Comments: Use trace(2) if the color display is hard to view on your system.

trace(3) is supported by the Complete Edition (only) of the Euphoria To C Translator. Interactive tracing is not supported with the Translator.

All forms of trace() are supported in the Public Domain interpreter, but only for programs up to 300 statements in size. For larger programs you'll need the Complete Edition interpreter. Note that *stamped* include files such as **Win32Lib.ew** and the files in **euphoria\include** do not add anything to your statement count, but you need to place your trace() statement *after* **include win32lib.ew** (or other large include file) to benefit from the stamp.

Example:

```
if x < 0 then
    -- ok, here's the case I want to debug...
    trace(1)
    -- etc.
    ...
end if
```

See Also: profile • debugging and profiling

unlock_file

Syntax: include file.e
unlock_file(fn, s)

Description: Unlock an open file fn, or a portion of file fn. You must have previously locked the file using lock_file(). On DOS32 and Win32 you can unlock a range of bytes within a file by specifying the s parameter as {first_byte, last_byte}. The same range of bytes must have been locked by a previous call to lock_file(). On Linux 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, s must always be {}.

Comments: You should unlock a file as soon as possible so other processes can use it.

Any files that you have locked, will automatically be unlocked when your program terminates.

See lock_file() for further comments and an example.

See Also: lock_file

unregister_block

Syntax: include machine.e (or safe.e)
unregister_block(a)

Description: Remove a block of memory from the list of safe blocks maintained by **safe.e** (the debug version of **machine.e**). The block starts at address a.

Comments: This routine is only meant to be used for **debugging purposes**. Use it to unregister blocks of memory that you have previously registered using register_block(). By unregistering a block, you remove it from the list of safe blocks maintained by **safe.e**. This prevents your program from performing any further reads or writes of memory within the block.

See register_block() for further comments and an example.

See Also: register_block • **safe.e**

upper

Syntax:

```
include wildcard.e
x2 = upper(x1)
```

Description: Convert an atom or sequence to upper case.

Example:

```
s = upper("Euphoria")
-- s is "EUPHORIA"

a = upper('g')
-- a is 'G'

s = upper({"Euphoria", "Programming"})
-- s is {"EUPHORIA", "PROGRAMMING"}
```

See Also: lower

use_vesa

Platform: **DOS32**

Syntax:

```
include machine.e
use_vesa(i)
```

Description: **use_vesa(1)** will force Euphoria to use the VESA graphics standard. This may cause Euphoria programs to work better in SVGA graphics modes with certain video cards. **use_vesa(0)** will restore Euphoria's original method of using the video card.

Comments: Most people can ignore this. However if you experience difficulty in SVGA graphics modes you should try calling use_vesa(1) at the start of your program before any calls to graphics_mode().

Arguments to use_vesa() other than 0 or 1 should not be used.

Example:

```
use_vesa(1)
fail = graphics_mode(261)
```

See Also: graphics_mode

value

Syntax: include get.e
s = value(st)

Description: Read the string representation of a Euphoria object, and compute the value of that object. A 2-element sequence, **{error_status, value}** is actually returned, where error_status can be one of:

```
GET_SUCCESS      -- a valid object representation was found
GET_EOF          -- end of string reached too soon
GET_FAIL         -- syntax is wrong
```

Comments: This works the same as **get()**, but it reads from a string that you supply, rather than from a file or device.

After reading one valid representation of a Euphoria object, value() will stop reading and ignore any additional characters in the string. For example, "36" and "36P" will both give you {GET_SUCCESS, 36}.

Example 1:

```
s = value("12345"}
-- s is {GET_SUCCESS, 12345}
```

Example 2:

```
s = value("{0, 1, -99.9}")
-- s is {GET_SUCCESS, {0, 1, -99.9}}
```

Example 3:

```
s = value("+++")
-- s is {GET_FAIL, 0}
```

See Also: get • sprintf • print

video_config

Platform: **DOS32, Linux**

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**, this "busy waiting" would tend to slow the system down. **wait_key()** lets the operating system do other useful work while your program is waiting for the user to press a key.

You could also use **getc(0)**, assuming file number 0 was input from the keyboard, except that you wouldn't pick up the special codes for function keys, arrow keys etc.

See Also: get_key • getc

walk_dir

Syntax: include file.e
i1 = walk_dir(st, i2, i3)

Description: This routine will "walk" through a directory with path name given by st. i2 is the **routine id** of a routine that you supply. walk_dir() will call your routine once for each file and subdirectory in st. If i3 is non-zero (TRUE), then the subdirectories in st will be walked through recursively.

The routine that you supply should accept the path name and dir() entry for each file and subdirectory. It should return 0 to keep going, or non-zero to stop walk_dir().

Comments: This mechanism allows you to write a simple function that handles one file at a time, while walk_dir() handles the process of walking through all the files and subdirectories.

By default, the files and subdirectories will be visited in alphabetical order. To use a different order, set the global integer **my_dir** to the **routine id** of your own *modified* dir() function that sorts the directory entries differently. See the default dir() function in **file.e**.

Example:

```
function look_at(sequence path_name, sequence entry)
    -- this function accepts two sequences as arguments
    printf(1, "%s\\%s: %d\n",
           {path_name, entry[D_NAME], entry[D_SIZE]})
    return 0    -- keep going
end function

exit_code = walk_dir("C:\\MYFILES", routine_id("look_at"), TRUE)
```

Example Program: **euphoria\bin\search.ex**

See Also: dir • current_dir

where

Syntax: include file.e
i = where(fn)

Description: This function returns the current byte position in the file fn. This position is updated by reads, writes and seeks on the file. It is the place in the file where the next byte will be read from, or written to.

See Also: seek • open

wildcard_file

Syntax: include wildcard.e
i = wildcard_file(st1, st2)

Description: Return 1 (true) if the filename st2 matches the wild card pattern st1. Return 0 (false) otherwise. This is similar to DOS wildcard matching, but better in some cases. * matches any 0 or more characters, ? matches any single character. Character comparisons are not case sensitive.

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
```

Example 2:

```
i = wildcard_file("AB*CD.?", "abcd.ex")
-- i is set to 0, 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

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.

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. 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.

The return codes are: DB_OK - success, DB_OPEN_FAIL - couldn't open the file, and DB_LOCK_FAIL - couldn't lock the file in the manner requested. 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")
        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_delete_table

Syntax: include database.e
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 get a negative value showing where they would be if they existed.

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")
    puts(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 is given by i.

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: 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(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. As a backup, the original, uncompressed file will be renamed with an extension of .t0 (or .t1, .t2 ,..., .t99). If 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

How To Install Euphoria on DOS/Windows

To install Euphoria you need:

- a 386 or higher PC
- at least 640K of memory
- 5 Mb of free hard disk space

The Install Procedure

You can install under **Windows** by double-clicking on **install.bat**
Alternatively, in a DOS window you can:

1. **cd** into the directory containing the Euphoria files

2. type:

   install

If you have downloaded Euphoria and pkunzip'd in (say) **c:\tmp** you would type:

```
cd c:\tmp
install
```

You will be asked which drive to put the **EUPHORIA directory** on. Later you will be asked to approve any changes to your **autoexec.bat** file. Finally, **you need to shut down and restart (soft-reboot) your computer** so the new **autoexec.bat** will take effect.

After installing, see **doc\what2do.doc** (or **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.

Enjoy!

How to Uninstall Euphoria

1. delete the EUPHORIA directory

2. remove the 2 references to "EUPHORIA" from your autoexec.bat file

If you have problems installing ...

- If the install appeared to run ok, did you remember to shut down and restart your computer?
- If a failure happens during the install, shut down any large applications that you have running, and try it again.
- If your system does not use an **autoexec.bat** file (e.g. Windows NT, 2000 etc.) then set the **EUDIR** and **PATH** variables in whatever way your system allows. For example on Windows 2000, select Start Menu - Settings - Control Panel - 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, 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**.
- Disable or cut back programs such as SMARTDRV that may be consuming large amounts of extended memory.
  Try typing:

  ```
  SET CAUSEWAY=LOWMEM:80
  ```

  before you run **install.bat**
- If you have less than 20 megabytes of free disk space, free up a few megabytes, and run **install.bat** again.
- There are several possible reasons why the **install** program will decide not to edit your **autoexec.bat** file. If this happens you should follow the manual procedure described below.

- If you need help installing Euphoria, send e-mail to: rds@RapidEuphoria.com.

How to manually edit autoexec.bat

- In some cases the **install** program will not be able to modify your **autoexec.bat** file for you automatically, and you will have to do the job yourself using a text editor.

- 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.

- 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.

- Reboot (restart) your machine. This will define your new **PATH** and **EUDIR** environment variables.

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. Most demos will try to use **SVGA** modes. You need DOS mouse support to run **mouse.ex** and **ttt.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.

- Try the benchmarks in **demo\bench**. Do you get the same ratios as we did in comparison with QBasic (or Perl or Python)? If you have a C/C++ compiler, how much faster can you get these benchmarks to run? We bet you'll be surprised, especially when you consider that ***Euphoria runs the benchmarks with subscript checking and a host of other run-time checks***.

- 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 BASIC, Fortran, Pascal, C/C++ or any other language that we are aware of.

Platform-Specific Issues for Euphoria

1. Introduction

Euphoria programs can currently run on three 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, NT and later systems.

The third platform is **Linux**. Linux is based on the UNIX operating system. It has recently become very popular on PCs. Linux on the PC is also a 32-bit system.

The Euphoria for DOS32+Win32 .zip 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 have a **.exu** file type.

Many Euphoria programs can be run on two, or all three platforms without change. The file type should indicate the preferred platform for the program. Any Euphoria interpreter will try to run any Euphoria file. You just have to specify the full name of the file, including the type.

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.

2. The DOS32 Platform

If you are new to programming, you should start off with **ex.exe** on the DOS32 platform, or perhaps **exu** on the Linux platform. Windows programming is somewhat more complicated, no matter which language you use.

Programs run in 32-bit (protected) mode and have access to all of the megabytes of memory on the machine. Most 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 32Mb of memory, but your program will run out of memory after using 640K. QBasic is even worse. It limits you to just 160K.

DOS32 programs are typically run with the screen in either text mode or pixel graphics mode, and there is a large library of Euphoria routines that you can call. 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 simple 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. Currently, 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.

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 the package from the Euphoria Web site. Also download Judith's Enhanced IDE for Win32Lib.

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.inprise.com/pub/delphi/techpubs/delphi2/win32.zip

See also the Euphoria Archive Web page - "documentation".

4. The Linux Platform

Euphoria for Linux shares certain features with Euphoria for DOS32, and shares 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 does not have integrated support for pixel graphics like DOS32, but Pete Eberlein has created a Euphoria interface to **svgalib**.

X windows GUI programming is currently possible using Irv Mullin's interface to the **graphapp** package.

When porting code from DOS or Windows to Linux, 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 uses '/' (slash) on file paths. DOS/Windows uses '\\' (backslash).

- Highly specialized things such as dos_interrupt() obviously won't work on Linux.

- Calls to system() and system_exec() that contain DOS commands will obviously have to be changed to the corresponding Linux command. e.g. "DEL" becomes "rm", and "MOVE" becomes "mv".

5. Interfacing with C Code (Win32 and Linux)

On Win32 and Linux it's possible to interface Euphoria code and C code. Your Euphoria program can call C routines and read and write C variables. C routines can even call your Euphoria routines. The C code must reside in a Win32 dynamic link library (.dll file), or a Linux shared library (.so file). By interfacing with .dll's and shared libraries, you can access the full programming interface on both of these systems.

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 hardly ever required in Linux or the Win32 API.

3. Call the C function by calling c_func() or c_proc().

Example: (see next page)

```
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** 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.

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

On Windows and Linux, 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, VEL, 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 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:

(see on next page)

```
/* 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", i.e "standard call". This is the calling convention that's used to call Win32 API routines, and the C pointer to your Euphoria routine must be declared this way too, or you'll get an error when your Euphoria routine tries to return to your .DLL. Note that the *default* calling convention for Windows C compilers is something different, called "__cdecl".

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).

The Euphoria Editor

usage 1: ed filename
usage 2: ed

Summary

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.

If you don't like **ed**, you have many alternatives. David Cuny's **EE editor** is a DOS editor for Euphoria that's 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 and Linux. Search for **editor** in our Archive. In fact, *any* text editor can be used, including DOS Edit or Windows NotePad.

Special Keys

Some keys do not work in a Linux text console, and some keys do not work in an xterm under X windows. Alternate keys have been provided.

Del	Delete the current character above the cursor.
Backspace	Move the cursor to the left and delete a character.
Ctrl-Del	Delete the current line. (**Ctrl-Del** is not available on all systems.)
Ctrl-d	Delete the current line. (same as **Ctrl-Del**)
Insert	Re-insert the preceding series of **Deletes** or **Ctrl-Dels** before the current character or current line.
Ctrl-arrow-left	Move to the start of the previous word. On Linux use **Ctrl-L**
Ctrl-arrow-right	Move to the start of the next word. On Linux use **Ctrl-R**.
Home	Move to the beginning of the current line.
End	Move to the end of the current line.
Ctrl-Home	Move to the beginning of the file. On Linux use **Ctrl-T** (i.e. Top)
Ctrl-End	Move to the end of the file. On Linux use **Ctrl-B**, (i.e. Bottom)
Page Up	Move up one screen. In a Linux xterm use **Ctrl-U**
Page Down	Move down one screen. In a Linux xterm use **Ctrl-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 **Ctrl-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 text console you're stuck with the number of lines available (usually 25). In a Linux 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 **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. When you quit the editor, you will be given a chance to delete **editbuff.tmp**.
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 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 **Ctrl-Del** (or **Ctrl-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 **Ctrl-Del**. 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 **Ctrl-Del** 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 use extended memory. It will edit files that are ***much*** larger than what MS-DOS EDIT or Windows Notepad can handle. With a huge file, inserting or deleting a line near the beginning of the file might take several seconds, due to intense swapping activity. Other operations should be reasonably quick.

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, the **Ctrl-z** character (26) will appear to be the *end of the file*.

Windows 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.

Line Terminator

The end-of-line terminator on Linux 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 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

ed runs best with **ex.exe** or **exu**, but will also run with **exw.exe**.

Shrouding and Binding - (Complete Edition only)

The Shroud Command

Synopsis: shroud [-clear] [-list] [filename]

The ***shroud*** command converts a Euphoria program, typically consisting of a main file plus many include files, into a single, shrouded (i.e. encrypted) file, that's easy to distribute to others. By default, the **shroud** command will perform the following steps:

1. Your main **.ex**, **.exw** or **.exu** file is combined with all of the **.e** files that it directly or indirectly includes. This results in a single Euphoria file with no include statements.
2. After making one pass through your entire program, any unused constants and routines are marked for deletion. This may expose additional constants and routines as being unused. This marking process is repeated until no more constants or routines can be deleted. A second pass is then made, where marked routines and constants are skipped, i.e. not copied to the shrouded file. It's quite common for a program to include many files, but only use a subset of the routines or constants defined in those files.
3. All comments, blank lines, and superfluous blanks and tabs (whitespace) are removed.
4. All keywords and standard built-in routine names are replaced by single-byte codes to save space.
5. All user-defined names are converted to short (typically one or two letter) meaningless names chosen by the shroud program.
6. The very compact file resulting from steps 1 to 5 is further encrypted so it becomes completely unreadable and highly tamper-resistant.

***options* can be:**

clear	Keep the source code in human-readable form. Unused routines and constants will be deleted, and comments and some whitespace will be removed, but the code will be otherwise unchanged. The original variable and routine names will be preserved, except where a naming conflict arises between merged files. Use this option when you want to ship a single source file, and you don't mind if people can see your code. If an error occurs while a user is running your program, the **ex.err** file will be readable. If you shroud your program, the **ex.err** file will contain short, meaningless names, and will be very difficult to understand.
list	Produce a listing in **deleted.txt** of the routines and constants that were deleted, as well as any symbols that had to be renamed.

If you simply type:

shroud

without any options or filename, you'll be prompted for all the information.

shroud only performs very superficial checks on the syntax of your program. You should test your program thoroughly before shrouding or binding it.

You can distribute a **shrouded .e** include file that people can include in their programs without seeing your source code. Symbols declared as **global** in your main **.e** file will not be renamed or deleted, so your users can access routines and variables using meaningful long names.

You can **shroud** or **bind** a program that *includes a* ***shrouded*** *file* (presumably from someone else), however you won't be allowed to use the -CLEAR option, since this would reduce the security of the included shrouded file.

Only RDS has the knowledge required to undo the encryption on a program and we have no tool to do it. Even if someone managed to undo the encryption, they would only recover the version of the source after steps 1 to 5 were applied. The comments, and the meaningful variable and routine names can ***never*** be recovered.

Always keep a copy of your original source files!

The Bind Command

Synopsis: **bind [-clear] [-list] [filename.ex]**
bindu [-clear] [-list] [filename.exu]
bindw [-clear] [-list] [-icon filename.ico] [filename.exw]

bind (**bindw** or **bindu**) does the same thing as **shroud**, and has the same options. It then combines your shrouded (or clear text) file with the Public Domain Edition **ex.exe**, **exw.exe** or **exu** 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.

***options* can be:**

-clear	Same as **shroud** above. The .exe will contain readable code. If an error occurs, the **ex.err** file will be readable.
-list	Same as **shroud** above.
-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 Complete Edition. You can bind it, or distribute it separately, with or without your changes.

If you simply type:

bind (or **bindw** or **bindu**)

without any options or filename, you'll be prompted for all the information.

Only the Public Domain Edition interpreters can be bound. Users of the Euphoria Complete Edition for DOS32 + Win32 will have **ex.exe** (Complete) and **pdex.exe** (Public Domain), as well as **exw.exe** (Complete) and **pdexw.exe** (Public Domain) in **euphoria\bin**. The **bind** (**bindw**) program will use the **pdex.exe** (**pdexw.exe**) files for binding. On Linux, you'll have **exu** (Complete) and **pdexu** (Public Domain), with **pdexu** used for binding.

A one-line Euphoria program will result in an executable file as large as the interpreter 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 18K to the size of the interpreter.** All three interpreters are compressed to reduce their size. **exw.exe** and **exu** are compressed using **UPX** (see http://upx.tsx.org). **ex.exe** is compressed using a tool that comes with the CauseWay DOS extender. **ex.exe** is the largest of the three since it includes a lot of pixel graphics routines, not part of **exw.exe** or **exu**. Note: In some 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. If demo\sanity.ex runs correctly, you can safely ignore these warnings. Otherwise contact RDS.

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.

Interpreter Source License

Rapid Deployment Software (RDS)
Euphoria Interpreter Source Code License
for version 2.3

In exchange for payment, RDS will provide you with C source code that can be used to build a version of the RDS Euphoria interpreter, using any of 6 supported C compilers on 3 platforms. The source code for certain registered features has been removed by RDS. The features removed are: the trace facility, profiling, and code related to binding.

We have tested the source with 6 different C compilers, and we are satisfied that it can be compiled and linked correctly with all 6. However, we can't guarantee that you will be able to compile or link the source, or achieve the same level of speed and reliability. If you have trouble, we will provide you with some assistance during a 3 month technical support period.

The following restrictions will apply.

1. Copying

You are permitted to make backup copies for your own use. You may not distribute copies of this source code, with or without your changes, to anyone other than RDS or other people who you are certain are entitled to this version of our source. If you are in doubt, you must contact RDS to find out if the person is entitled to the source or not. If multiple programmers, either as individuals or in a company, wish to study or work together on the source, they must each pay RDS for a copy.

2. Acceptable Changes to the Source

RDS gives away certain Euphoria features for free, while charging for other (registered) features. You are permitted, and encouraged, to add new features to this source code. However, without written permission from RDS, you are not permitted to distribute any features that are similar enough to a registered feature, so as to significantly reduce peoples' incentive to pay RDS for that feature. The registered features currently include:

- **binding:** You must not provide a way to attach files to a Euphoria interpreter, thus converting a Euphoria program into a single executable file. (*Shrouding* is not restricted, since it can be performed without this source code, and without any knowledge contained in this source code.)

- **trace facility:** You must not provide a source debugging capability, or a log file mechanism for tracing execution of statements.

- **profiling:** You must not provide statement-count or time profiling in any form.

- **Translator to C:** You may not use this source to create and distribute a program that translates Euphoria programs into C/C++ programs.

- **Translator library:** You may not use this source to create and distribute a library for use with the Euphoria To C Translator.

In the future, other features may be added to this list, and some features may be deleted. You are bound by whatever features are listed in the License Agreement at the time you purchase the source.

Note that we are limiting what you are allowed to *distribute* to others, not what you are allowed to construct in the privacy of your own machine.

3. Porting

We encourage you to port this software to new machines and operating systems.

4. Distribution of Executables

Provided your changes are acceptable, you can compile the source, using any compiler, and distribute copies of the resulting executable file (but not the source or the intermediate object files). You can give your executables away for free, or charge any fee you like.

5. Ownership of New Features

You own the source code for any features or changes that you make, and you are not required to make public any source code that you develop. If you do make your source code public, other than to RDS, or those you are certain are entitled to our source code, you must not expose more than a few lines of RDS source code along with it. The fact that you implemented a feature does not preclude RDS or anyone else from implementing the same or similar features.

6. Public Discussion

You can publicly discuss the algorithms used by our source code in open forums, and you can mention the names of C routines, variables and other identifiers in the code, but you must not reveal actual lines of source code.

7. Acknowledgement

You are required to acknowledge the use of our source code, list our Web site, http://www.RapidEuphoria.com, and indicate the general nature of any changes you made to our source code, in any programs that you distribute. The acknowledgement could be displayed by the program, or be in the accompanying documentation, but it must be plainly visible to most users.

8. Damages

RDS is not responsible to you for any damages that arise from your use of this source code, and you must inform your users, in your documentation or in your derived programs, that RDS is not responsible to them for any damages.

9. Evil Intent

You may not use the source to create any viruses, worms, trojans or other software that is intended to cause damage to any computer systems or networks, or that would intentionally harm the reputation of Euphoria or RDS.

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, 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** is especially high (on Windows 95/98 at least). Linux is 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 32 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 (4-byte) 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** 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

You can download the Euphoria To C Translator from the RDS Web site. 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 Resources Available on the Internet

Euphoria Web Page

Check out the Official Euphoria Programming Page on the World Wide Web:

http://www.RapidEuphoria.com

The latest version of Euphoria is always downloadable from this page. You will also find news about Euphoria, a large archive of free downloadable software, and links to Euphoria pages set up by enthusiastic users. You can also view and subscribe to the mailing list from this Web site.

Euphoria Mailing List

Anyone who is interested in discussions about Euphoria can add their e-mail address to the Euphoria mailing list. Euphoria enthusiasts are using the mailing list to exchange ideas and solve problems. All you have to do is send a blank e-mail to:

EUforum-subscribe@topica.com

After subscribing, you can e-mail a message to EUforum@topica.com and your message will automatically be forwarded to everyone on the list.

Everyone benefits from exchanges of ideas and useful Euphoria code. These discussions and code samples are archived so you can retrieve them at any time. To leave the list, send an e-mail message to:

EUforum-unsubscribe@topica.com

Feel free to subscribe again at another time. **You can also interact with the mailing list through the Web interface.** The Euphoria Mailing List page has all the details.

You can search a huge archive of messages dating all the way back to June 1996. Chances are good that your question has already been answered.

Third Party Add-On Libraries and Programs

Enthusiastic users of Euphoria have contributed hundreds of excellent programs and library routines that you can download. This software is available on the Web, either at the Official Euphoria Programming Page, or on a user's own Web page. Most of it is **Public Domain** and includes full source code, however please check the code or README file for any restrictions. At the RDS site you should check the Archive page and the Recent User Contributions page.

The Euphoria Micro-Economy

GET RICH QUICK working in your spare time at home!

Anyone with an e-mail address has heard that one before, but with the system we call the **Euphoria Micro-Economy** you can potentially make a few bucks (very few) working very long hours at home in your spare time. Briefly, the way it works is that each month, registered users of Euphoria are encouraged to send e-mail to RDS donating $3.00 U.S. ***of RDS's money*** to any file on the RDS Archive or Recent User Contributions pages. You can split your 3 dollars across multiple files as you see fit. Gradually, those who have contributed useful programs, documentation, or whatever to the Euphoria community will build up dollars that they can use to reduce or even eliminate the cost of registering or upgrading. There's no limit to what you can make (but don't expect to get rich quick!).

For full details, please go to the Euphoria Web site.

Euphoria Newsgroup

There's an Internet newsgroup, **alt.lang.euphoria**, that is lightly used. Most users prefer the SPAM-free environment of the mailing list. The mailing list operates just like a newsgroup. Feel free to post to the newsgroup or the mailing list, but you will probably get better feedback from the mailing list.

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 Web interface to 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. The mailing list is discussed in **web.doc**.

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: 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: 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: **cls**
   Even when you can't see any keystrokes echoed on the screen this may clear the screen for you.

2. type: **ex**
   The Euphoria interpreter will try to restore a normal text mode screen for you.

3. type: **exit**
   If you are running under Windows, this will terminate the DOS session for you.

4. type: **Ctrl-Alt-Del**
   This will let you kill the current DOS session under Windows, or will soft-reboot your computer if you are running under DOS.

5. 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()** 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: 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**.

About Brainstorm Solutions

This compilation of RDS Euphoria documentation is proudly presented by Brainstorm Solutions — completely free — as most good things built to/by Euphoria! Nevertheless, if you find this compilation useful, please consider spending some of your micro-economy money on us. Your acknowledge, of any kind, will be very welcome.

Brainstorm Solutions is Euler German's brand name for his computer related products and services. Despite the fact that Brainstorm Solutions is quite new to the market, the same doesn't apply to it's experience: we have been in the computer business since 1973 — a lot of history — and we have most of our original personnel and the founder himself!

Our original goal — business software development — isn't on top anymore. Consulting, Security, Internet, Software Translation, Sales and Marketing, are our major activities today.

If you think we can be helpful to you or to your business, please, contact us at:

Brainstorm Solutions
Caixa Postal 232
35701-970 Sete Lagoas, MG
Brazil

eMail: efgerman.bs@mailnull.com

You will be most welcome!

Kind regards,

-- Euler F. German
founder of Brainstorm Solutions