Joyous coding with OpenEuphoria

Our expert

Juliet Kemp always likes a bit of euphoria in the morning.

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

Euphoria was first released back in 1993, when it was shareware. These days, it's still under active development, but it's now an open source project, available for a variety of platforms including Linux. It's fast, can be interpreted or compiled into C, and does a bunch of run-time checking to help you find bugs quickly. It's also fairly easy to read and understand from the get-go, with a minimum of coding punctuation marks and boilerplate.

Euphoria's definitely a bit old-school in style; procedural, with very minimal type checking and minimalist function declarations. If you're an OO fanatic, perhaps best stay away. At times the English-language keyword approach even leaves it feeling close to pseudocode – but it's pseudocode that works. Give it a try; you might find yourself surprisingly taken by the language.

You can download either a Deb package, or a generic **tar.gz** binary package, from the OpenEuphoria website (**http://openeuphoria.org**). At time of writing, the most recent release was 4.0.5. You can also download and compile your own C source code release from the downloads page. Nightly builds are available if you like being bleeding-edge.

If you decide to install the **tar.gz** package, you'll need to unpack it wherever you prefer (probably **/usr/local**). You can then either edit **/etc/profile** as necessary, or create your own **eu.cfg** that looks like this:

```
/path/to/euphoria/include
```

This will tell Euphoria where the various standard libraries are. (You can also set an environment variable **$EUINC** to do the same thing.) At this point, you're ready to go.

Hello World

As is customary, let's start with Hello World. Create a file **helloworld.ex** (by convention, console Euphoria programs end **.ex**, GUI apps end **.exw**, and include files/libraries end **.e**):

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

You run it with **eui helloworld.ex**, and you will see the expected output.

puts() ("put string") may be familiar from other languages. Here, it takes two arguments. The first one tells Euphoria where the output is going to (1 is, as ever, the filehandle associated with standard output); the second one is the string to output. **printf()** is also supported for more complex or formatted output.

To output the same thing to a file, you'd need to open a file first and assign it to an integer filehandle:

```
integer myfile
constant STDERR = 2
constant STDOUT = 1
myfile = open("myfile.txt", "w")
if myfile = -1 then
   puts(STDERR, "couldn't open myfile\n")
else
   puts(STDOUT, "file opened\n")
   puts(myfile, "Hello, World\n")
   close(myfile)
end if
```

One of the four available types in Euphoria is **integer** (*See the bottom of this page for the other three*). Here, we set up an integer to use as a filehandle, then try to open the file. If it works, we'll get a positive integer (3, in this case, unless we've opened another file earlier in the code); if it fails, we'll get the result of -1. We test for success (**if/then** syntax, as here, is straightforward; note the **end if** closing the block) and act accordingly. Note that **puts** takes a filehandle argument; here one statement is output to standard out, and the other to the file, and it's important to remember to close the file afterwards!

Ring the alarm program

Time to get a bit more complicated. We're going to create a program that rings an alarm after a certain number of minutes. In our initial very basic iteration, we'll set it to run for just a few seconds:

```
include std/os.e
global integer Seconds = 5
global object Message = "Time's up!\n"
procedure ring_alarm(object alarm_text)
  puts(1, alarm_text)
end procedure
procedure run_alarm()
  sleep(seconds)
  ring_alarm(message)
end procedure
run_alarm()
```

The first line includes a standard library which allows us to use (among other things) the system **sleep()** function. We then declare two global variables. There are only four variable types in Euphoria:

- **object** (can take on any value at all).
- **sequence** (a sequence of any type of object).
- **atom** (numbers of any sort).
- **integers** (integer numbers between -1073741824 and +1073741823; larger integers can be used but must be declared as atoms).

Numeric calculations are slightly faster on integers than on atoms. You can also define your own variable types if you

need to; check the documentation for more details. Here, integer is fine for our number of seconds variable.

The two building-blocks of an Euphoria program are procedures and functions. A procedure performs some computation, and can take parameters. Functions are like procedures, but return a value; we'll use functions in a later iteration of this program. Both functions and procedures are closed with **end [procedure|function]**. Simply defining a procedure or a function doesn't run it; you have to explicitly call it. As you'll notice, we could do everything here in a single procedure, but it's good practice to break different functions out and makes it easier to extend the program later. Both of these procedures are straightforward. Run the program with **eui pomodoroalarm.ex**, and after five seconds you should see a message displayed on the console.

Coding standards

Of course, five seconds isn't very long. The 'pomodoro' productivity technique uses sections of 25 minutes, so let's change our program to reflect that (only the changes are shown here, the rest of the program stays as-is):

```
global atom Seconds
function set_alarm(atom minutes)
  return minutes * 60
end function
procedure run_alarm()
  Seconds = set_alarm(25)
  -- rest of procedure as before
end procedure
```

Run this, and you should get an alarm after 25 minutes. However, this is a long time to wait for testing, so you might want to swap in a smaller value (try 0.1) for 25 while you're experimenting. Note that if the type of **minutes** in **set_alarm()** were integer rather than atom, you would be limited to whole minutes. We've also changed **Seconds** to atom, for the same reason – you want it to be able to handle random fractions of a minute. Try keeping it as integer, then putting 0.01 in to replace 25; you'll get a type check failure, as the result is 0.06 and seconds cannot be set to 0.06.

A quick note on Euphoria coding standards: the standard for the standard libraries is to use lower case for local variables, uppercase for constants, Sentence case for global variables, and underscores within names. I've broadly stuck with this, though you can choose your own coding standards if you prefer. Euphoria is case-sensitive.

Another aspect of the Pomodoro technique is following your 25 minutes of work with a five-minute break. It would be great if we could set our alarm up to manage that automatically for us. For that, we could use a sequence, and loop over it:

```
global object Work_message = "Work time is up!\n"
global object Break_message = "Break time is up!\n"
global atom Work_minutes = 25
global atom Break_minutes = 5
procedure run_alarm()
  sequence pomodoro = {Work_minutes, Break_minutes}
  sequence pomodoro_message = {Work_message, Break_
message}
  for i=1 to length(pomodoro) by 1 do
    sleep(set_alarm(pomodoro[i]))
    ring_alarm(pomodoro_message[i])
  end for
end procedure
```

This is all fairly self-explanatory. We set up two sequences, one to handle the time, and one to handle the message displayed. Then we use the built-in **length()** function to iterate over the time sequence, **sleep**, and display the appropriate message. (I recommend, again, changing the minute values for testing while you're developing.) Note that Euphoria indexes from 1.

This is a bit error-prone, though; what if **pomodoro** and **pomodoroMessage** are different lengths? You'll get an error. For a more neat, and more maintainable, process you could use a sequence of sequences to associate your ***Minutes** and ***Message** variables:

```
procedure run_alarm()
  sequence pomodoro = {{Work_minutes,  Work_message},
                       {Break_minutes, Break_message}}
  for i=1 to length(pomodoro) by 1 do
    sleep(set_alarm(pomodoro[i][1]))
    ring_alarm(pomodoro[i][2])
  end for
end procedure
```

Or you could achieve a similar result by changing the global variables:

```
global sequence Work_settings = {Work_minutes,  Work_
message}
global sequence Break_settings = {Break_minutes, Break_
message}
procedure run_alarm()
  sequence pomodoro = {Work_settings, Break_settings}
  -- for loop as above
end procedure
```

This is probably the clearest and most maintainable version, but as you can see, Euphoria is quite flexible in how you use its variables!

The next thing we can try is to ask the user how long they want the timer to run for. We'll keep our existing default settings, but add a procedure that asks the user for input:

```
include std/io.e
include std/get.e
procedure get_alarm_times()
```

»

Quick tip

To trace the progress of a Euphoria program (ie, to step through it instruction by instruction), add these lines at the top of any program:

```
with trace
trace(1)
```

Editors and Euphoria

There is a Euphoria-specific editor, **ed.ex** that ships with Euphoria, although to use it you may have to add the Euphoria **bin** directory to your path in **.bashrc**:

```
PATH=$PATH:/usr/share/euphoria/bin
```

To use it, type

```
eui ed.ex file.ex
```

Compared with an editor like *Vim* or *Emacs*, *ed* is pretty basic, but it does do syntax highlighting and editing correctly. It also has one big advantage: if you run it with no argument immediately after encountering an error, it will read the **ex.err** file and launch itself with the cursor on the line the error came from.

However, I found that it simply wasn't a good enough editor to make it worth using, even for this. Check out the OE wiki for a list of alternative editors with Euphoria support, some of which also have this error location support. Unfortunately, *Vim* only has syntax and indent support. To access this, install the files provided by the *eu-editor* project to the appropriate system *Vim* directories, and add this line to your global **filetype.vim** file:

```
au BufNewFile,BufRead *.ex      setf euphoria
```

Turn syntax on in your Euphoria file and you should see lovely colours.

Similar support is available through the *eu-editor* project for *MicroEmacs* and *Nano*, among others.

```
  puts(STDOUT, "Please enter work minutes: ")
  sequence result = get(STDIN)
  Work_minutes = result[2]
  puts(STDOUT, "Please enter break minutes: ")
  sequence result = get(STDIN)
  Break_minutes = result[2]
end procedure
procedure run_alarm()
  get_alarm_times()
  -- rest is as before
end procedure
```

The **std/io.e** library enables us to use the library constants **STDOUT**, **STDIN**, and **STDERR** rather than declaring them ourselves. **get()** (from the **std/get.e** library) enables us to read a string of characters from a file and convert them into a numeric value. The other input options (**gets()**, for example) will not automatically turn characters into a numeric value, and we want a numeric value. **get()** can take several parameters:

```
get(integer file, integer offset = 0, integer answer = GET_
SHORT_ANSWER)
```

file gives the file handle from which to read; **offset** the offset to apply to file position before reading; and **answer** allows you to choose between two forms of answer:

```
GET_SHORT_ANSWER : {integer return_status, object
value_read}
GET_LONG_ANSWER  : {integer return_status, object
value_read,
      integer num_characters_read, integer initial_whitespace}
```

The return status can be **GET_SUCCESS**, **GET_EOF**, **GET_FAIL**, or **GET_NOTHING**. Here we're not bothering to check return status, but best practice would be to include an error check. Since we're reading from standard input, and we only want the value read in, we don't need to include **offset** or **answer** when calling **get**. This means we'll just get the short answer. We then pull out the answer's second index to get our atom to put into **Work_minutes** (and then **Break_minutes**).

However, if you run this, you'll discover that it doesn't seem to be doing the right thing. It seems like the values of **Work_minutes** and **Break_minutes** aren't being changed. What's going on?

In fact, if you put in a few logging statements, you'll find that the problem isn't with **Work_minutes** and **Break_minutes**; they are indeed being changed. The problem is that our global variables **Work_settings** and **Break_settings** do not reflect those changes, because they were assigned before we got the user input. There are a couple of options for fixing this problem:

```
-- Option 1
procedure get_alarm_times()
  -- as above
  Work_settings[1] = Work_minutes
  Break_settings[1] = Break_minutes
end procedure
-- Option 2
procedure run_alarm()
  get_alarm_times()
  sequence pomodoro = {{Work_minutes,  workMessage},
                       {Break_minutes, breakMessage}}
  -- as before
end procedure
```

(Comments in Euphoria use either **--** as here for a single line, or **/* ...**

***/** for multi-line.)

Arguably the first option is slightly neater, but either will do the job. Run it again and it should work fine.

Calling out

If you're deep in the throes of work, though, you might not notice a little message on a console somewhere. It would be better if we could make it play something. Unfortunately, while there's built-in support for Windows system sounds in Euphoria, there isn't anything similar for Linux. So the most straightforward option is to use a system call to fire up an external program and play an MP3:

```
global object Play_alarm_command = "mpg123 -q alarm.
mp3"
procedure ring_alarm(object alarm_text)
  puts(1, alarm_text)
  system(Play_alarm_command, 2)
end procedure
```

I chose **mpg123** just because it's a very simple command-line program that can be run with no output (the **-q** switch); you can use whatever program you like; **system()** just calls out to the system to run a particular command. The mode here, 2, means that the graphics mode will not be restored afterwards; it's not necessary to worry about the graphics or clearing the screen because **mpg123** won't do anything to the screen.

If you run this, you'll find that hitting Ctrl+C cancels the alarm and goes back to our program; until you've cancelled the alarm, the next timer won't start. In this case this is probably what you want. If in another situation you wanted to keep the program running, you'd need to look into using **fork()**, or multitasking. (There's a multitasking guide available in the Euphoria docs).

Let's make one final improvement, which will let us have a quick look at **if/then** structure. We'll set the program up to have a default of running a user-defined number of 'pomodoros' (25 minute + 5 minute sequences), with the option of changing the times. The whole program as it now stands is on the **LXFDVD**.

Other neat language features

There is a big stack of common routines available in the OpenEuphoria API, all well documented in the manual. These include extensive maths support; internet functions like HTTP, DNS, and URL handling; locale support; and a bunch of others. Euphoria also supports multitasking, which is particularly handy if you're interested in writing games or other fast-moving programs, or carrying on behind the scenes while waiting for user input.

If you want to go the graphical route, Euphoria has a few GUI options available. The cross-platform graphics available in the **std/graphics.e** library are console based, and **std/image.e** can handle bitmaps. Alternatively, there is a *GTK* graphics library at **http://eugtk.wikispaces.com**, which also has some support for *Glade*. It's possible to interface Euphoria and C code fairly easily; Euphoria can call C routines and use C variables, using **open_dll()** (for both **.dll** and **.so** files), defining the function, and using **c_func()** or **c_proc()** to call it. Check the Euphoria docs for more; but this gives your Euphoria program even more scope for rapid development. Finally, while Euphoria can be hooked in to all the standard databases, if you want something a little more lightweight, the Euphoria Database System is available to enable you to develop database-lite apps easily and in a Euphoria-oriented way.

The addition we've made is the function **get_number_pomodoros()**, and the **if/then** section in **set_timing()**. As shown here, the **if/then** syntax is pretty straightforward, and again uses the **end *** syntax familiar from functions and procedures. If you wanted an **else**, you would add it in the following way:

```
if a = b then
  -- do something here
elseif a = c then
  -- do something else here
else
  -- more code
end if
```

If the user has specified 0 pomodoros, we let them set their own time preference (and we set the number of pomodoros to 1, which is a slight hack to simplify **run_alarm()**). Having set the timings as the user prefers, we use a double **for** loop in **run_alarm()** to iterate over the 'pomodoro' sequence as often as the user requested.

Operations & sequences

The double **for** loop here seems a little untidy. If you look at the Euphoria manual, there are a lot of operations you can do with sequences to concatenate them in various ways – using the **&** operator, **append()** or **repeat()**. So can we generate an alarm sequence that looks like this?

```
{{Work_minutes, Work_message}, {Break_minutes, Break_
message},
 {Work_minutes, Work_message}, {Break_minutes, Break_
message}}
```

In fact, this is a bit complicated, because the concatenation operators all squash the sequences into one another in various ways.

```
append({a, b}, {c, d}) = {a, b, {c, d}}
append({Work_minutes, Work_message}, {Break_minutes,
Break_message})
  = {Work_minutes, Work_message, {Break_minutes, Break_
message}}
{a, b} & {c, d} = {a, b, c, d}
{Work_minutes, Work_message} & {Break_minutes, Break_
message}
  = {Work_minutes, Work_message, Break_minutes, Break_
message}
repeat({a, b, c, d}, 2) =  {{a, b, c, d}, {a, b, c, d}}
repeat({Work_minutes, Work_message, Break_minutes,
Break_message}, 2)
  = {{Work_minutes, Work_message, Break_minutes, Break_
message}
    {Work_minutes, Work_message, Break_minutes, Break_
message}}
repeat({a, b, {c, d}}, 2) = {{a, b, {c, d}}, {a, b, {c, d}}
repeat({Work_minutes, Work_message, {Break_minutes,
Break_message}}, 2)
  = {{Work_minutes, Work_message, {Break_minutes, Break_
message}}
    {Work_minutes, Work_message, {Break_minutes, Break_
message}}}
```

This leaves us with two possible ways of generating our alarm sequence through sequence operations:

```
procedure set_timing()
  Number_pomodoros = get_number_pomodoros()
  if Number_pomodoros = 0 then
    get_alarm_times()
    Number_pomodoros = 1
  end if
  Alarm_sequence = repeat(Work_settings & Break_settings,
Number_pomodoros)
  /* Alarm_sequence now looks like {{W_min, W_mess, B_
min, B_mess},
    {W_min...}, ... } */
end procedure
procedure run_alarm()
  set_timing()
  for i=1 to Number_pomodoros by 1 do
    sleep(set_alarm(Alarm_sequence[i][1]))
    ring_alarm(Alarm_sequence[i][2])
    sleep(set_alarm(Alarm_sequence[i][3]))
    ring_alarm(Alarm_sequence[i][4])
  end for
end procedure
```

As per the comment, this generates a four-part sequence to repeat. The alternative looks like this:

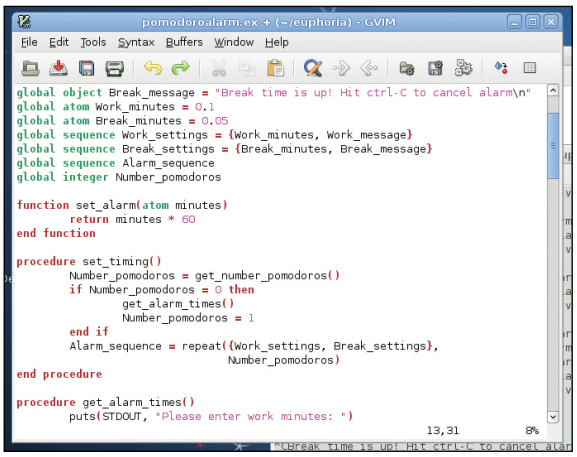
```
procedure set_timing()
  -- as above
  Alarm_sequence = repeat({Work_settings, Break_settings},
Number_pomodoros)
  /* Alarm_sequence now looks like {{{W_min, W_mess}, {B_
min, B_mess}},
                    {{W_min...}, {B_min..}}, ... }*/
end procedure

procedure run_alarm()
  set_timing()
  for i=1 to Number_pomodoros by 1 do
    sleep(set_alarm(Alarm_sequence[i][1][1]))
    ring_alarm(Alarm_sequence[i][1][2])
    -- could also use another for loop here
    sleep(set_alarm(Alarm_sequence[i][2][1]))
    ring_alarm(Alarm_sequence[i][2][2])
  end for
end procedure
```

It's up to you which of these you think is clearest and most maintainable method.

And there's more!

As ever, there are plenty of improvements to be made to this sample code. For example, you could write code to count down minute by minute, or even second by second, how much time there is remaining before your alarm goes off. Or you could set up something graphical to represent your alarm program with nice buttons to push. Overall, we've found Euphoria is quick to develop in and gives relatively clear error messages, so it's easy to dive in and see where your inclination takes you. LXF



You can edit the code with syntax highlighting, and have it running in the background.