A Beginners Guide To Euphoria

By David Gay

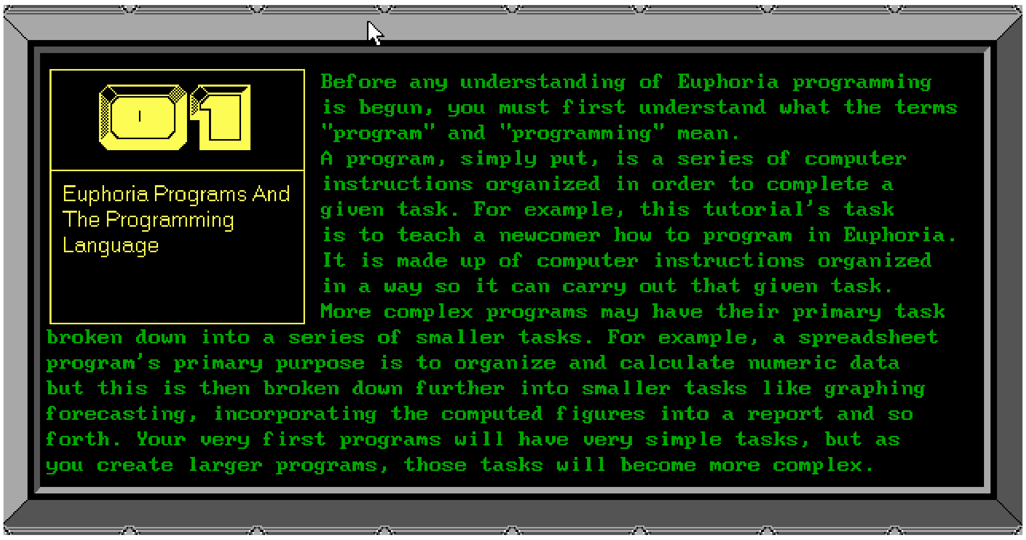
ABGTE **Euphoria**

OpenEuphoria Annotated Version – Tom Ciplijauskas

ABGTE ↻ **Euphoria, Preface**

DOSBox
ABGTE2
A BEGINNER'S GUIDE TO EUPHORIA II
Click The Mouse To Begin Learning Euphoria...

The Opening Screen to ABGTE2.



01

Euphoria Programs And The Programming Language

Before any understanding of Euphoria programming is begun, you must first understand what the terms "program" and "programming" mean.
A program, simply put, is a series of computer instructions organized in order to complete a given task. For example, this tutorial's task is to teach a newcomer how to program in Euphoria. It is made up of computer instructions organized in a way so it can carry out that given task.
More complex programs may have their primary task broken down into a series of smaller tasks. For example, a spreadsheet program's primary purpose is to organize and calculate numeric data but this is then broken down further into smaller tasks like graphing forecasting, incorporating the computed figures into a report and so forth. Your very first programs will have very simple tasks, but as you create larger programs, those tasks will become more complex.

The First Lesson in ABGTE2

ABGTE is synonymous with “Euphoria Tutorial.” David Gay created an ebook reader, code browser, and demonstration programs that operated under DOS and provided an excellent tutorial on the Euphoria programming language. The year was 1997 and Euphoria was at version 1.5.

By version 3.0 Euphoria was multi-platform and open source. With the advent of OpenEuphoria 4.0, the Euphoria language has been changed dramatically. OpenEuphoria no longer supports DOS, now operates under *Windows* and *Unix*, the standard library has been re-written and expanded, and the language has been modernized.

The essence of Euphoria remains the same in OpenEuphoria. That means ABGTE is still a valuable tutorial if the DOS specific content is ignored.

While ABGTE2 is still available, running it under current operating systems is problematic. Jacob extracted the contents of ABGTE2 and produced an html version—making the tutorial accessible once again.

Annotations appear like this.

It was the intent of David Gay that his original work not be edited or altered. This edition preserves his text and artwork. It has been annotated making ABGTE relevant to users of OpenEuphoria.

Tom

HTML Conversion, Original Comments by Jacob

From the Euphoria web site:

Euphoria is a simple, flexible, and easy-to-learn programming language. It lets you quickly and easily develop programs for Windows, DOS, Linux and FreeBSD. Euphoria was first released in 1993. Since then Rapid Deployment Software has been steadily improving it with the help of a growing number of enthusiastic users. Although Euphoria provides subscript checking, uninitialized variable checking and numerous other run-time checks, it is extremely fast. People use it to develop Windows GUI programs, high-speed DOS games, and Linux FreeBSD X Windowsppp programs. It is also very useful for CGI (Web-based) programming.

A description of Euphoria 3

Euphoria is free and open source software. You can download it here.[1]

This is a conversion of David Gay's interactive DOS tutorial, which can still be downloaded from the archive. It overcomes some limitations of the original format, such as not being able to print out any text or demo programs (which are here listed in the body of the text), and of course, the tutorial can now easily be read by Linux / FreeBSD users. The tutorial assumes no prior knowledge of programming in any language, and is a fine complement to the official Euphoria documentation. I suggest you copy and paste each demo program to your favourite editor as you come to it, then add these lines at the top:

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

This will enable you to step through each line of code and see the variables change as the program runs. Note that the content of the tutorial has been faithfully preserved from David's executable, so occasional references to "the console," etc (which may give rise to some puzzlement) are a reflection of the original format. There have been many additions and enhancements to the language during the years since 1997 when this tutorial was originally written. Chapters 21 and 22 should really be re-named 'Euphoria And OS', to reflect the fact that Euphoria is now multi-platform. The download now includes a database system (EDS), and more recently, multi-tasking was built-in to the interpreter. In spite of this, the tutorial still stands as the most comprehensive introduction to the basics. The later chapters were copied directly from a screen, so a few errors may have crept in. Also, although the demo programs should work ok, I haven't tested them all. If you find any errors / bugs, drop me a line at jacobite1@fastmail.fm, and I'll fix them.

[1]www.OpenEuphoria.org

™

ABGTE Euphoria

OpenEuphoria Annotated Version

ABGTE2, Original Comments by David Gay

The Origin Of A Beginner's Guide To Euphoria II

Where to begin? I suppose I should start from my beginning….

It all started when I got my first computer (a TRS-80 Model I) in 1981. I was 17 back then, and like any bright-eyed teen, I was intrigued with the technology of that time. I enjoyed fiddling with the hardware and making it do what I wanted, even though it only had 4K (!!!!!) of RAM and virtually no semblance of an operating system (yep, everything hardcoded into the computer). But what I really liked was the ability to write my own programs. At that time, all I had to work with was a primitive form of Radio Shack BASIC and Z-80 Assembler, but it was worth the effort.

As time went on and the technology became more advanced, I played with more programming languages. I played with GW-BASIC, then QuickBASIC (both the compiler and interpretor versions). I also flirted with Assembler. But like the programming languages on my TRS-80, I couldn't find the language I was really looking for.

One of my reasons for this dissatisfaction was the weak support in graphics. I was certainly not an artist by any standards, plus the amount of graphic commands available did not make it very easy to create images. Even though there were programs out there that created very professional graphics, it was extremely hard to get information in layman's term on how to load them into the programs I wrote. A second beef was the amount of (what I considered personally to be) "junk variable" types that were too complex and too inflexable to be of any use. My third beef was the trend programming languages were heading. I noticed that as programming languages got more powerful, the protective barrier between the program and the operating system became thinner. By the time I went to C programming, I was playing with dangerous commands like pointers and Assembler calls to graphic routines. As a result, I found current programming languages too inflexable for writing software.

This changed when I came across Euphoria. It was version 1.2 at the time, which I downloaded from a BBS. The first thing that caught my eye was the number of variable types. There were only two types: atoms and sequences. I initially found this rather lacking, seeing that C for example had many variable types. What could be done with two? But as I experimented with the code, I understood the reasoning behind Euphoria variables. By only introducing two very elemental types of variables, you have the ability to use them as building blocks to create more complex variable types for your use. The concept of the sequence type variable was extremely radical, yet had the best traits of both singular data values and arrays. With this sort of variable, you can treat it as both a single value and as a list of values. Another thing that I really liked about Euphoria was the emphasis on program stability. Euphoria will not allow you to write programs with uninitialized variables. Euphoria also did away with pointers, yet still allowed programmers to pass a series of values to procedures without any restrictions. C on the other hand needs pointers in order to pass a dynamic list of parameters to procedures. But the best I saved for last: the ability to read in bitmapped graphics (.BMP's) without needing to write the required code. In other words, only one line of code is needed to read in any .BMP made by a professional graphic program. To display that bitmap on the screen, you only needed one line of code. That's two lines of code!

I was won over. I registered at 1.3 and became interested in learning this language in more detail. Actually, I became a fan and started to tell my friends about it. Some of them were a little leery at first. They felt Euphoria's new approach to data organization and the easy way it can handle bitmapped graphics made it more like an intelligent "query" script instead of a programming language. However, once they tried to write sample programs with the shareware release, they realized how powerful Euphoria was.

It was at this point I wanted to kill two birds with one stone, namely to learn the language in more depth and to further advertise Euphoria to other people. This stone turned out to be a web page devoted to teaching novice programmers about Euphoria. I felt I could pull it off because I have tutored people in QuickBASIC before, and also was a computer programmer for two years before the recession of 1990 put a permanent change to my career. Also, while Rapid Deployment's Euphoria manual explained Euphoria's features clearly, it could not teach a user who was totally new to programming how to write a program. I mean, how could they understand what a variable was, or how to perform input and output? I wrote an Email to Rapid Deployment Software asking for permission to write a web page that taught Euphoria programming on a beginner's level. When they gave me the go-ahead, I began to write the first six installments of "A Beginner's Guide To Euphoria."

On April 21st 1996, "A Beginner's Guide To Euphoria" made it's debut on the Internet with the first six installments. As time went on, I wrote more installments and added them to the web page. I received positive comments in Email about how much the web page helped them understand Euphoria. However, one common theme I was reading in the Email was a request to create an "offline" version of the web page. This made sense because it by the time I wrote my last installment, entitled "Euphoria And DOS" on June 28th 1996, "A Beginner's Guide To Euphoria" contained 36 installments. A lot of online time was needed to go through the entire web page.

I didn't want to write just any old text version of the program. I wanted the "offline" version to be entertaining and interesting to view. I also wanted to take the advertising of Euphoria one step beyond by demonstrating what Euphoria can offer as you read the tutorial. So, I decided to write a program version of the web page that people could run on their computer. The program would be based on a futuristic storyline about warring language factions of a spacefaring civilization (Euphoria being one of them) fighting for supremacy. The storyline was based on the very real history of programming language evolution. After all, there are so many programming languages available today. To learn Euphoria, you would sit in front of this training console and read the information on the screen. By using the mouse, you can move anywhere in the tutorial and even run examples of Euphoria's graphic and DOS features.

The first tutorial (as I now like to call it) was released shy of August 1st, 1996. And judging from the number of downloads (nearly 2,500 from August 1996 to July 1997 according to Interlog), and from the Email from users, the tutorial appears to have done it's job very well. However, it really wasn't the tutorial I wanted to release. Due to time constraints, I had to make some concessions on the appearance of the tutorial. Instead of the original design of a pure GUI interface using just a mouse, I created an acceptable yet more awkward interface that mirrored more the keyboard than the mouse. Also, I found that maintaining the text of the tutorial was difficult. So, in February 1997, I decided that a new second tutorial was necessary, and began work on what was originally called "A Beginner's Guide To Euphoria 2.00." The new tutorial would be similar to the first one with the following enhancements:

- the material introduced in the tutorial would follow the style used by the reference manual supplied with the Euphoria software.

- the user would only require a mouse to operate the entire tutorial (with the exception of the keyboard during the execution of any demo programs), using a simple to understand GUI interface.

- the user can view the source of demo programs as well as run them.

- the second tutorial would use graphics and colour more effectively, giving it a more professional look.

- the Euphoria programming language would be covered in a much wider scale to appease those newcomers to Euphoria who were proficient in other languages like QBasic or C. For example, chapters were added explaining the concepts of bits and bytes and how to use Euphoria library routines to manipulate them. At the same time, the material was made more detailed and broadened to help even the completely computer naive person who wanted to learn programming.

By the time I finished, there was so little left of the original tutorial I changed the name to “A Beginner’s Guide To Euphoria II,” Version 1.00.

RDS was asked, as with the first tutorial, to test the tutorial and review the text to make sure everything was accurate. As a result, this tutorial is RDS-approved as a supplement to understanding Euphoria. In addition, several “test drive” versions were released between April 1997 and July 1997 for users to try in order to find any bugs or inaccuracies in the text.

It is my intention to use the easier to maintain structure of this tutorial to add references to any new features of Euphoria that come out, rather than create yet another tutorial software. I do not expect to write a “A Beginner’s Guide To Euphoria III” for at least two years or more.

In the meantime, I am in the process of exporting the text from inside the new tutorial to files stored in directory DATA This is needed in order to reduce the size of the executable down to a managable level. I also want to add a “print” feature and to expand the remote to give it more features, like a “subject index.”

David Gay October 1st, 1997

Thanks

The name Euphoria and the programming language is copyright 1997 Rapid Deployment Software. My warmest thanks to Robert Craig and Junko C. Miura of Rapid Deployment Software for giving me permission to write this program tutorial, and for their incredible efforts in bughunting and proofreading the tutorial. Without them, it just wouldn’t have been possible. Thanks also to RDS for creating the Euphoria programming language!!!! :)

Thanks to Digital Liquid (www.digital-liquid.com) for their (temporary) hosting and maintenance of the “A Beginner’s Guide To Euphoria” web site. While our alliance was short-lived and unsuccessful, it managed to give me precious time that I used to get this tutorial back on track. Thanks Todd!

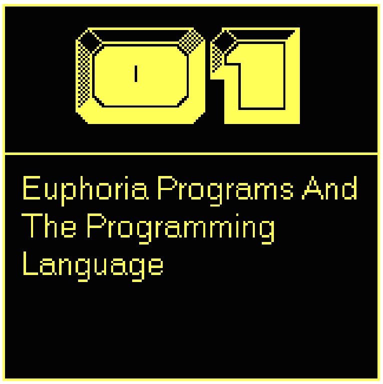
Thanks also to my close friend Jennifer Hanson, pouty web author genius. In addition to her flood of suggestions about the tutorial graphics, her repeating of the following words motivated me enough to finish a difficult project:

"Haven't You Finished That Tutorial Yet?!?!"

Finally, thanks to everyone who tried the first tutorial, and for telling me that it helped them understand Euphoria. May this second one help a new batch of programmers!

David Gay
July 27th, 1997

1. Euphoria Programs And The Programming Language



Before any understanding of Euphoria programming is begun, you must first understand what the terms "program" and "programming" mean. A program, simply put, is a series of computer instructions organized in order to complete a given task. For example, this tutorial's task is to teach a newcomer how to program in Euphoria. It is made up of computer instructions organized in a way so it can carry out that given task. More complex programs may have their primary task broken down into a series of smaller tasks. For example, a spreadsheet program's primary purpose is to organize and calculate numeric data but this is then broken down further into smaller tasks like graphing forecasting, incorporating the computed figures into a report and so forth. Your very first programs will have very simple tasks, but as you create larger programs, those tasks will become more complex.

The original text of ABGTE2 has not been altered. But, for current users, color has been added to show where oE (Open Euphoria) differs from Eu (original Euphoria).

Works on oE, v4 and up

Works on Eu, v3 and down

oE users need to make changes (often just small changes.)

DOS specific

Not for oE

Most of the demonstration programs will run (cut and paste from this document) as presented. Because oE retains the original include library, they execute without problems. However, include is emphasized when the new standard library is the preferred choice.

To create a program on your computer is referred to as "programming." You may have also heard of the phrase "coding," which is the same as "programming." "Coding" comes from the fact that the instructions that make up a program are sometimes referred to as "program code." It doesn't really matter what terms you use, just as long as you understand what they mean. But exactly what do the individual instructions that make up a program look like? And how do we enter and group program instructions to create a program? Just as all words of our vocabulary make up the English language, a set of all related computer instructions make up a programming language.

The one programming language a computer understands is called "binary." It's called binary because the program instructions are made up of two numbers, 1 and 0, like 100101 for example. Binary language is also referred to as "machine language."

Unfortunately, to write a program in this very difficult language is beyond the ability of most people, save for a very chosen few. The very first computers of the 1950's and 1960's could only be maintained by scientists who had a good understanding of machine language.

As time went on, more understandable and easy to use programming languages were created. The program instructions of these languages were more English-like in appearance. Perhaps you may have seen or heard of some of them, such as BASIC, COBOL, RPG, ADA, FORTRAN, PASCAL, C and others.

OpenEuphoria, free and open source.

Euphoria is the newest form of computer programming language. For your personal interest, Euphoria stands for **E**nd **U**ser **P**rogramming with **H**ierarchical **O**bjects for **R**obust **I**nterpreted **A**pplications. Quite a mouthful, but as you will soon discover, the name is the only difficult part of the Euphoria programming language to understand.

Any plain text editor will work. See the oE Wiki for some editor suggestions and syntax files.

To create a Euphoria program, you first start up an editor program that lets you enter Euphoria programming instructions, called “statements.” Just as if you were writing a letter, you start typing at the top of the editor window, and then proceed downwards. Exactly what you will be typing will be revealed soon. Understand for now you will require an editor program to do this. MS-DOS’s EDIT program is satisfactory, or you can use ED, which comes with the Euphoria software. Once you have finished typing your Euphoria program, you can save it to the hard drive or floppy drive as a file with .EX as the extension.

But you’re not finished just yet. Remember that your computer only understands machine language. It cannot understand Euphoria programming statements. So, it must be converted to machine language.

There are two types of programs that can translate Euphoria to machine language. Both do the same thing but they differ in their methods.

The oE compiler is euc

One is called a compiler. It takes a Euphoria program and creates a machine language program. This machine language replica has the same name as your Euphoria program, but has a different file extension of .EXE.

The oE interpreter is eui

The other type of translator program is called an interpreter. It translates each Euphoria program statement to a machine language instruction, which is then run. This differs from a compiler, which creates the entire program to be run. Euphoria both comes with a compiler (BIND.BAT) and an interpreter (EX.EXE).

To run a program using EX.EXE, you type:

For oE interpreter type:

```
eui filename
```

```
EX filename.EX
```

For oE compiler (translator) type:

```
euc filename
```

To create an executable using BIND.BAT, you type:

```
BIND filename.EX
```

When the executable is created, type:

Under Unix run as
```
./filename
```

```
filename
```

Binding combines an interpreter with source-code, producing "stand-alone" executables.

The translator, `euc`, makes actual compiled executables.

The Euphoria program that BIND.BAT will translate into a machine language program is called a "source file." The machine language program is called an "object file." EX.EXE generates "object code."

Sometimes it may not be possible to successfully translate your Euphoria program to machine language. While entering Euphoria statements, it is possible you may misspell a word. Just as we have spelling and grammar rules in the written language, Euphoria also demands we follow set rules when typing in these statements. When a spelling or grammar type error is made during the typing in of these statements, it's referred to as a "syntax error." When your compiler encounters this error in your program while trying to translate it to machine language, it will stop. You will then see a message explaining what the error is, and where it is in your Euphoria program. You then start up your editor, correct the statement in error and try to run the compiler or interpreter again.

All programs, no matter what purpose each serves, perform one function. They all process data. This processing of data is broken down into the following three stages listed below:

1. A program will accept data. The data will consist of numeric figures from either an external source (a keyboard, mouse, digital camera,or voice card) or from somewhere inside the computer (such as a file on a CD-ROM, floppy diskette, or your hard drive).

2. A program will analyze the accepted data. This step involves temporarily storing the data for both arithmetic calculations and comparison against predefined values, and then making a decision based on the result of the calculation or comparison.

3. A program will present data in a meaningful form. This means either displaying figures on the screen or printer, storing information on the hard or floppy drive for another program to use later, or creating graphics and sound from the computer that has a meaning to the person running the program.

No doubt all of this has you eager to start learning how to write your own Euphoria programs. Well, let's get started on learning the actual concepts and instruction statements of the Euphoria programming language!

2. Variables And Data Objects

We mentioned previously that a program will store data for analysis. Your computer's memory (RAM) best resembles a huge group of mail boxes, each of which is uniquely addressed by a number. Each of these storage locations can hold a value between 0 and 255. Each of these values is referred to as a "byte." If values over 255 need to be stored in RAM, it is split up between various locations. While data is always stored as numbers in RAM, it can be both numeric or characters like "A."

To write a program that accesses RAM locations by address number would be tedious, especially when dealing with large data values. Thankfully, Euphoria offers a way to access stored data in RAM not by the actual RAM address number, but by a label like "salary" or "points." This symbolically referenced memory location is called a "variable."

Variables are invaluable for two reasons. First of all, it's much easier to know where all your data is located in your program if you use meaningful names. For example, if you are writing a space combat game and want to store data representing the amount of fuel you have left in your ship, storing it in a variable called "fuel" makes it so much easier to find than some obscure RAM memory address like 32767.

In addition, because a variable holds a single stored value regardless of its size, there's no complex handling of multiple RAM locations when dealing with very large values. Euphoria does this for you behind the scenes. When it comes time to compile or interpret your program to be run, variable names are converted automatically to actual RAM memory addresses. But that is something a Euphoria programmer does not need to be concerned about.

Variable names can be any length in size, and can both be real words or made up ones that border on nonsense, as long as the name itself is meaningful to the programmer. However, Euphoria does place some limits on what you can use for a variable name. First of all, variable names must start with a letter and then can be followed by any combination of letters, numbers and the underscore (" _ "). Second, case is significant. This means the variable name "tax" is not the same as "TAX." Finally, words used in the Euphoria language cannot be used as a variable name. They include, but are not limited to, words like "and," "global," "function," "while," and "exit." These words are called "reserved words." A complete list can be found in the Euphoria Reference Manual.

Now that we have completed our understanding of variables, let's move on to learn about the type of data we can store inside a variable.

In Euphoria, all data is referred to as "data objects." The reason this term is used to describe data is because data isn't something you work out in arithmetic calculations. Instead, data in Euphoria is viewed as tangible items you can merge together, break apart, twist, or alter at the slightest whim. As we go further into understanding the Euphoria language, you will soon see this to be true.

Atoms can now be written with underscore spacers: 1_000_000 or 1_23 or 0.0_3

Data objects come in two types. The first type is the atom, which is a single numeric value. Below are examples of atoms:

```
2001    12.4    -5    3.14e3
```

The first three examples are very familiar to all of us, but the fourth is an example of Standard Notation. The "e" means "times 10 to the power of," with the number following. This means 3.14e3 is really 3.14 times 10 to the power of 3, which works out to 3140.

Standard notation is best used to represent atom values in a compact form. Atoms can either be a floating point (with a decimal point) or integer value (no decimal point) value, and can be either positive or negative. Atoms can have a value range of approximately between -1e300 and +1e300 (that's -1 followed by 300 zeros to 1 followd by 300 zeros, inclusive). While chances are good you will never design a program that handles such huge numbers, it is nice to have that wide a margin to work with.

The second type of data object is called a sequence, and is a little more complex in structure. Sequences are a list of data objects joined in the same manner as links on a chain. Each linked data object is referred to as an "element." Sequences can be made up of either atoms, smaller sequences, or any mixing of both. You can have sequences inside of sequences, which in turn are part of bigger sequences, and so on, to any level of dimension. Computer memory is the only limiting factor.

Sequences always start with a " { ", have commas or "," separating each individual data object, and a closing " } ". Here are some examples of sequences:

A sequence made up of atoms:

```
{2,4,6,8,10}                                                  .
```

A sequence that is made up of three smaller sequences:

```
{{31,32,33,34,35},{41,42,43},{51,52,53,54,55,56}}
```

namely

```
{31,32,33,34,35}, {41,42,43}
```

and

```
{51,52,53,54,55,56}                                           .
```

A sequence that is made up of both an atom and two sequences:

```
{{100,101,102},200,{301,302,{-401,-402},303}}
```

namely

```
{100,101,102},200
```

and

```
{301,302,{-401,-402},303}                                     .
```

Notice that the third sequence example has a sequence within a sequence as the third element.

Sequences can be represented in the form of a character string, like the text you are reading now. Character strings begin with a quotation mark followed by any numbers, letters or special characters, then ended with a second quotation mark. The character string is translated by Euphoria to the sequence's real form automatically.

For example, the following two values:

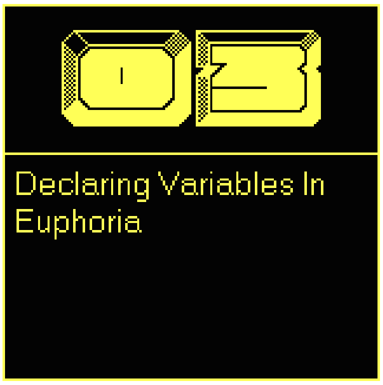
```
"David Alan Gay"

{68,97,118,105,100,32,65,108,97,110,32,71,97,121}
```

are identical in value. They only differ in the way they are presented. The numbers in the second value are the ASCII codes of each character. ASCII is a convention that assigns each character, displayable or not, a numeric code. This convention was created to ensure that all computers, no matter who made them, will display data the same way. ASCII stands for American Standard Code for Information Interchange.

Character strings are best used to define sequences that are to be used for display on a screen or printer, such as names, addresses, etc.

If you are feeling a little overwhelmed by all these terms, like “variables,” “data objects,” “atoms,” and “sequences,” don’t be alarmed. It’s because you haven’t had the chance to see how they work in Euphoria. That will now change. You will now be introduced to your first Euphoria programming statements, by learning how to create variables for your program to use later.

3. Declaring Variables In Euphoria



Before using variables in Euphoria, you first must "declare" them. Declaring a variable is similar to declaring items in front of a customs officer. When you declare items, you tell the officer what they are, what type of items they are and so forth. In Euphoria, declaring variables in a program involves two things: stating what they are going to be named, and defining the type of data they are supposed to hold.

oE is more flexible—can declare and assign in one statement.

Eu style (declare, then assign) still works.

To declare a variable in Euphoria, you use the following syntax:

```
variable type variable name
```

variable type means the type of data object the variable will hold.

variable name means the name of the variable, of course.

The first part of the variable declaration, the variable type, comes in only four accepted words: "sequence," "atom," "object," and "integer." "sequence' means the variable can only hold data objects that are sequences. You cannot place atom data objects in this type of variable. "atom" means the variable can only hold data objects that are atoms. Sequences are not allowed. A variable type of "object" means the variable can hold both atom and sequence data objects. One must wonder why we bother having variables of type atom and type sequence when a type object variable can hold both. Type object variables are needed to hold the result of program data processing where the data type is unknown. "integer" means the variable can hold atoms, but only integer atoms. An integer atom can have a value between -1073741824 and 1073741823. If you want to use even larger integer in your programs, you need to use the type atom variables to hold them.

Declarations do not have to be at the top of the program. It can be more clear if declarations are made just before a variable is used.

Variable declarations usually appear at the top of the program, so they are one of the first things typed in by the programmer. However, there are exceptions where they may appear elsewhere in the program. A variable declaration is entered only once for every variable in the program. Once entered, you cannot enter a second variable declaration using the same variable name, even if the variable type is different.

Let's start entering our very first Euphoria statements. To declare a variable named "address" that holds sequence type data objects, we enter:

```
sequence address
```

To declare a variable named "age" that holds atom data objects we enter:

```
atom age
```

To declare a variable named "grab_bag" that can hold both atoms and sequences, we enter:

```
object grab_bag
```

To declare a variable named "whole_numbers" that can hold integer atoms between -1073741824 and 1073741823, we enter:

```
integer whole_numbers
```

When dealing many variables of the same type, you can declare them all with one variable type followed by a series of variable names.

For example:

```
sequence name, address, city, country
```

will just as easily declare these four variables as if you used a single declaration line for each variable. If you wanted to get really fancy, this is also a legal way to declare multiple variables:

```
atom hours_worked,
     hourly_rate,
     deductions,
     net_income
```

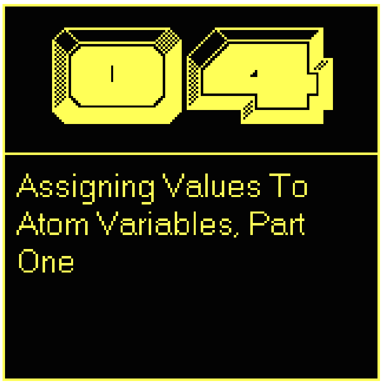
Euphoria offers the ability to split variable declarations (and other types of statements) into several lines as opposed to one line. Just remember the commas, and only split the line where there is a space.

Having a variable type as part of a variable declaration is a kind of safety system. It prevents a programmer from entering the wrong type of value into a variable. It also ensures that certain features of the Euphoria programming language that are meant to work on one data type do not get slipped with a variable value that is of a different data object type.

Which brings us next to the topic of variable values. Is there an initial value placed inside a variable after it is declared? The answer is no. Don't get the impression, however, that there is nothing inside a variable. Most likely it is left-over data from a previous program run, and probably so garbled that it is unusable. For this reason, Euphoria rules dictate that a programmer must place a value inside a variable for the first time before it can be used.

The next few chapters will discuss the process of initializing variables with values, and even changing those values. This is an important part of Euphoria that must be clearly understood, because it involves the primary purpose of all programs, which is to process data.

4. Assigning Values To Atom Variables, Part One



This chapter begins the topic of placing values in atom variables. But first, an important note. Even though this chapter uses atom variables in the examples, what you learn here also pertains to object type variables. You can also use integer variables, but remember that integer variables only hold positive and negative whole numbers. To place values in atom (and in other types of) variables, you use an "assignment statement."

oE lets you declare a variable and assign a value in one statement.

You can still write write programs as described here.

The concept of assignment statements is really quite easy to follow. Here is the syntax of the assignment statement listed below:

```
variable = expression
```

variable is of course, a variable. The equal sign means "is given the value of." **expression** is either a constant value, another variable, or a complex formula. An expression is evaluated to a single value, which is then stored in the variable to the left of the equal sign.

Let's begin with a simple example of assigning a variable with a value:

```
atom year, copy_of_year
year = 1997
copy_of_year = year
```

(Author's note: first, you could say the above is a simple example of a program, and second, from this time forth, any examples involving Euphoria will always show the appropriate variables declared.)

In the example on the previous page, the atom variable "year" is being assigned a value of 1997. The value 1997 is the simplest example of an expression, as it is already a value of 1997. The variable "copy_of_year" is given a value of what was just placed in "year." Euphoria checks to see what is inside "year," and then places that value in variable "copy_of_year," which is 1997.

```
atom letter_of_the_alphabet
letter_of_the_alphabet = 'C'
```

In the example above, the variable, “letter_of_the_alphabet” is being assigned a numeric value, but the numeric value is represented here as a character between single quotes or ’. This character representation is evaluated into the numeric ASCII value of the letter C. This means “letter_of_the_alphabet” ends up containing a value of 67.

Here’s another way to assign an atom variable with a numeric value:

```
atom double_quotation_mark
double_quotation_mark = '\ " '
```

This example program places the ASCII value of a double quotation mark in variable “double_quotation_mark,” which is 34. It works just like the previous example that used a character representation of a numeric value. This form of expression is commonly used for special characters that cannot be entered by keyboard, or to use Euphoria symbols like the for output. Other examples of “\” –prefixed special characters include “\n” for new line, “\t” for tab, and “\\” for reverse slash. A complete list of these special characters is explained in the Euphoria Reference Manual.

But expressions can also be more than just a special representation of a single value. They can be the result of very complex arithmetic calculations. In this case, expressions are a combination of numbers, other variables, and special arithmetic operators. In Euphoria, the following symbols listed below are arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division

Let’s put these operators to use in the program example on the following page. Here is a program example that works out the amount of tax paid and the total cost of a pair of jeans.

```
atom jeans, tax, tax_paid, total_cost
jeans = 22.00
tax = .07
tax_paid = jeans * tax
total_cost = jeans + tax_paid
```

Variable “jeans” represents the sale price of a pair of jeans, so we assign it a value of 22.00 (with decimal points to look like dollars). Variable “tax” is then assigned a value of 7%, which is 0.07. Next, variable “tax_paid” is computed by multiplying the values of variables “jeans” and “tax” together. Finally, variable “total_cost” is assigned the sum of the values stored in “tax_paid ” and‘ ‘jeans.”

The previous program examples have all shown variables being initialized for the first time, and only once. What happens if a variable already contained a value, but was assigned a new value? Well, the old value of the variable would be replaced with the new value. A variable may only hold one value at any time. But this shouldn't be considered a problem, as variable value modification is just as important as initialization. Note the following example:

```
atom counter
counter = 0
counter = counter + 1
counter = counter + 1
counter = counter + 1
```

What is the value of variable "counter" after this program finishes?

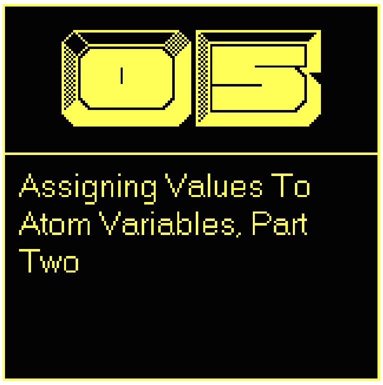
Let's look at the program again, this time examining it line by line:

```
atom counter
counter = 0           -- "counter" is initialized to 0
counter = counter + 1 -- "counter"'s value is now 1, by adding 1 to 0
counter = counter + 1 -- "counter"'s value is now 2, by adding 1 to 1
counter = counter + 1 -- "counter"'s value is now 3, by adding 1 to 2
```

After variable "counter" is set to zero, its old value is replaced with a new one in the next 3 lines, by taking the original value and adding 1 to it to produce a new value. This is done three times in the program example.

An introduction to relational and logical expressions is our next stop!

5. Assigning Values To Atom Variables, Part Two



Continuing our discussion of assigning atom variables with values, we now introduce relational and logical expressions.

There is no 'boolean' type. Euphoria works fine without it. But you may define one if you really need it.

Relational means based on comparison. For example, you can be taller than one person, yet shorter than another. Your parents are older than you, yet at the same time, you are older than your children or younger siblings. It works the same way in Euphoria. Is one variable value larger or smaller than an expected value? Is this value equal or unequal in a comparison with another value? Relational expressions are evaluated to one of two values: 1 for true, or 0 for false.

The same syntax as in most languages.

Here are a list of valid Euphoria relational operators:

```
<    less than
>    greater than
<=   less than or equal to
>=   greater than or equal to
=    equal to
!=   not equal to
```

Here is a program example that uses several of these operators in relational expressions. See if you can guess the value of each variable first before reading the explanation of each line:

```
atom t1, t2, t3, t4

t1 = 5 > 4               --1 (TRUE) in  " t1 "
                         : 5 is GREATER than 4


t2 = 7 != 7              --0 (FALSE) in  " t2 "
                         : 7 is EQUAL to 7, not UNEQUAL

t3 = 13 <= 13            --1 (TRUE) in  " t3 "
                         : 13 is LESS THAN OR EQUAL to 13

t4 = 12 = 2 * 6          --1 (TRUE) in  " t4 "
                         : 12 is EQUAL to 2 * 6, which is 12
```

Logical expressions focus more on whether a given situation is true or false. For example, if one says, “The bedroom light is switched on.” the statement is either true (the light is indeed switched on), or false (the light is not switched on).

The analysis of the truth of a situation can be more complex than the bedroom light example. For instance, the handling of delinquent accounts at a loans company requires more than one condition to be met. First of all, the account must be in a state of not being paid. Second, it must be in that state for a given set of time before the customer is called. Both of these conditions must be true before an account is considered delinquent (the given situation is true). If any one of the conditions are not met, then the account cannot be delinquent (the given situation is false).

Euphoria has logical operators that work using a system called BOOLEAN LOGIC. This system dictates that the outcome of paired conditions, either single values or relational expressions, are based on the rules below:

```
 Condition 1               Condition 2     Result
=============              ===========     =========
1 (true)        and        1 (true)        1 (true)
0 (false)       and        1 (true)        0 (false)
1 (true)        and        0 (false)       0 (false)
0 (false)       and        0 (false)       0 (false)
1 (true)        or         1 (true)        1 (true)
0 (false)       or         1 (true)        1 (true)
1 (true)        or         0 (false)       1 (true)
0 (false)       or         0 (false)       0 (false)
}
```

An “and” logical expression only evaluates to a single true value if both conditions are themselves evaluated to be true, and an “or” logical expression is only false when both conditions are false.

So let’s put this knowledge to use in an example Euphoria program:

```
atom value1,value2,test1,test2,test3,test4

value1 = 50
value2 = 25
test1 = value1 < 10 and value2 = 25
test2 = value1 > 5 and value2 = 12.5 * 2
test3 = value1 != 100 or value2 < 90
test4 = value1 = 5 or value2 > 27
```

Try to guess the value of each variable before going to the next page.

```
value1 = 50
value2 = 25
test1 = value1 < 10 and value2 = 25
```

Because variable “value1” is not smaller than 10, the first relational expression is false. Variable “value2” is equal to 25, so the second part of the “and” expression is true. But because one of the relational expressions were not true, “test1” is assigned a value of 0.

```
test2 = value1 > 5 and value2 = 12.5 * 2
```

Because variable “value1” is larger than 5, the first relational expression is true. Variable “value2” is equal to 12.5 * 2, so the second relational expression of the “and” logical expression is true. Because both expressions are true, “test2” is assigned a value of 1.

```
test3 = value1 != 100 or value2 > 2000
```

The first relational expression of this “or” logical expression works out to be true because 50 is not equal to 100. The second relational expression works out to be false because 25 is not larger than 2000. As a result, the “or” logical expression works out to be true because at least one of the expressions was true, so variable “test3” is assigned a value of 1.

```
test4 = value1 = 5 or value2 < 49
```

Variable “value1” is certainly not equal to 5, so the first relational expression is false. Variable “value2” is not smaller than 49, so the second relational expression is also false. Because neither relational expression worked out to a true value, the “or” logical expression is false. This means “test4” is assigned a value of 0.

Euphoria also has a “not” logical operator that gives you the opposite value of a relational expression. Here’s a program example that best demonstrates this:

```
atom opposite, result

result = 15 < 20
opposite = not result
```

First, “15 < 20” is evaluated, which has 1 (true) stored in variable “result,” Next, “not” reverses the value in “result” to 0 (false), so the variable “opposite” is assigned a value of 0. In short, if the expression beside a “not” works out to be false, “not” reverses it to true. If the expression works out to be true, “not” reverses it to false. “not” gives you the opposite value of the outcome of an expression.

You can link as many relational and logical expressions together to make larger complex expressions. You can even link arithmetic, relational and logical expressions together to formulate a value to be stored in the appropriate variable. However, you should know a few important things.

In a situation where you are using the equal sign (=) in a relational expression, Euphoria uses a rule to define when the equal sign is an assignment operator and when it is a relational operator. The leftmost equal sign on the line is assumed to be an assignment operator. All other equal signs that follow are assumed to be relational operators.

The outcome of arithmetic expressions can be tested with logical operators. This means you can do expressions like the following:

```
complex_result = value1 * 5 and value2 - 8
```

If arithmetic expressions are used with logical operators, non-zero results like 5.2 or -4 are assumed true. Zero results still mean false.

Euphoria also uses a system called precedence that defines which parts of larger complex expressions are done first by default. For example:

```
atom rent1, rent2, rent3, average_rent

rent1 = 500
rent2 = 600
rent3 = 400
average_rent = rent1 + rent2 + rent3 / 3
```

This example is supposed to work out the average of three rents. But precedence dictates that division is done first before addition. This means this program will work out the average of the rents incorrectly!

We can fix this by using parentheses to force different parts of the expression to be worked out in a manner different from precedence of operators:

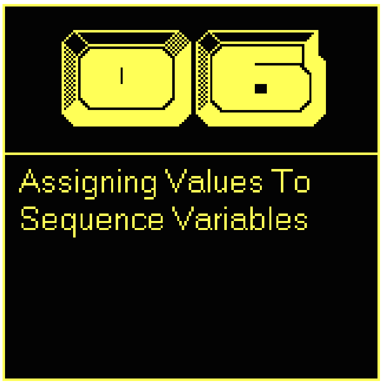
```
average_rent = (rent1 + rent2 + rent3) / 3
```

If in doubt, just add () braces to make things clear.

This forces the three rents to be added together before division takes place. A complete list of precedence operators is in the Euphoria Reference Manual.

This completes your introduction to assigning atom (and object) type variables. If you clearly understand variable assignment, arithmetic, logical and relational expressions, and precedence, move on to sequence variable assignment in the next chapter. If not, please go over the two chapters again. You must understand these concepts before moving on to sequence variable assignment.

6. Assigning Values To Sequence Variables



Assigning values to sequence variables is just like assigning values to atom variables. Sequence variables can be given an actual value, or the result of an expression. And as with atom variables, the expressions can be arithmetic, relational or logical. However, because sequences consist of linked data objects instead of a single one, there are a few twists to learn about. It's important to note that the examples used in this chapter can also be applied to object type variables.

Sequences are what make Euphoria flexible and powerful.

It is important to fully understand how sequences work.

Here's a simple example of a sequence variable being assigned a value:

```
sequence list_of_values

list_of_values = {1,2,3,4,5,6,7,8,9,0}
```

It is also legal to assign a sequence variable the following value:

```
sequence Mister_No_Elements

Mister_No_Elements = {}
```

This is an example of a sequence value with no elements. You would use this approach to assign a sequence variable an "empty" value before using it in the program. It's similar to setting an atom variable with a zero value.

If you wanted to store a sequence value that was a person's name, address or city of residence, you could use the previously mentioned character string method to represent a sequence value.

```
sequence my_name

my_name =  "David Alan Gay"
```

In this program example, we're storing "David Alan Gay" into a sequence variable meant to be my name. But understand that Euphoria converts this character string into the actual sequence value before it is stored in variable "my_name." If you could look inside variable "my_name," this is what you would see:

```
{68,97,118,105,100,32,65,108,97,110,32,71,97,121}
```

It's important to note that a character string of `"D"` for example is NOT the same as `'D'`. The former is a character string representation of a sequence value that is one element long, while the latter is a character representation of an atom value.

Important: an atom is not a sequence.

But, sometimes, x and { x } 'behave' as if they were the same. For example we say the length of x and { x } are the same, because it is convenient for some operations.

Having said this, you shouldn't assume also that you can assign a one element long sequence value into both an atom variable and a sequence variable. An atom value and a one element long sequence value are NOT identical!

The previous sequence variable assignment examples used values that are considered one-dimensional (where the elements are all atom values). Remember that sequences can also be composed of smaller sequences or a mix of atoms and smaller sequences. This program example assigns more complex sequence values to a series of sequence variables:

```
sequence seq1, seq2

seq1 = {{1,2,3}, {4,5,6}, {7,8,9}}
seq2 = {{5,5,5,5,5,5,5}, -98, {100,-50,20.3}}
```

You could also represent the previous program example to help clarify the individual elements of each sequence using split lines:

```
sequence seq1, seq2

seq1 = {{1,2,3},
       {4,5,6},
       {7,8,9}}

seq2 = {{5,5,5,5,5,5,5},
       -98,
       {100,-50,20.3}}
```

Instead of using constant elements in a sequence value, you are also allowed to use variable names and expressions to define the elements in a sequence value to be stored in a sequence variable:

```
sequence mixed_bunch
atom some_atom_element

some_atom_element = 502
mixed_bunch = { "Euphoria" , some_atom_element,
                some_atom_element/2}
```

Before this sequence value is stored in variable "mixed_bunch," each individual element must be worked out. In element 1, the character string "Euphoria" is converted to a true sequence value. In element 2, the value of "some_atom_element" is used. In element 3, the expression of the value of "some_atom_element" being divided by 2 is worked out. Once these steps are completed, the actual value is stored in "mixed_bunch."

You can also use arithmetic, relational, and logical expressions to assign values to sequence variables, using the same operators introduced in the previous chapters on atom variable value assignment. This program example below works out a rent increase for five different apartments:

```
sequence old_rents, new_rents
atom rent_increase

old_rents = {413,500,435,619,372}
rent_increase = 1.05
new_rents = old_rents * rent_increase
```

Variable "old_rents" is given a 5-element sequence value that is the old rent amounts for five apartments. "rent_increase" is assigned the percentage to raise the rents by. "new_rents" is the new rents of the five apartments.

Let's look at the statement that works out the new rents more closely:

```
new_rents = old_rents * rent_increase
```

When using both atom and sequence variables or values in a binary expression (something that is made up of two parts), the atom part is replaced with a temporary sequence value that is as long as the other sequence. This is done before the expression is evaluated. As a result, variable "rent_increase"'s value becomes

`{1.05,1.05,1.05,1.05,1.05}`.

So, the above expression is then worked out in the following manner:

```
element 1 of "old_rents" (413) x element 1 of temporary sequence (1.05)
element 2 of "old_rents" (500) x element 2 of temporary sequence (1.05)
element 3 of "old_rents" (435) x element 3 of temporary sequence (1.05)
element 4 of "old_rents" (619) x element 4 of temporary sequence (1.05)
element 5 of "old_rents" (372) x element 5 of temporary sequence (1.05)
```

The result of this element by element multiplication stores the value of:

```
{433.65, 525, 456.75, 649.95, 390}
```

in variable "new_rents."

This also works for logical and relational expressions as well. Here is a program example that uses sequence and atom in both expression types:

```
sequence test1,test2

test1 = {1,0,0} and 0
test2 = {20,30,40} <= 30
```

Try to determine the value of "test1" and "test2" before going on to the next page. It's just like the rent increase program example!

```
element 1 of {1,0,0} (1) and element 1 of {0,0,0} (0) = 0
element 2 of {1,0,0} (0) and element 2 of {0,0,0} (0) = 0
element 3 of {1,0,0} (0) and element 3 of {0,0,0} (0) = 0
```

Thus, variable "test1" is assigned a value of `{0,0,0}`

```
element 1 of {20,30,40} (20)  <=  element 1 of {30,30,30} (30),
                                  giving a value of 1.

element 2 of {20,30,40} (30)  <=  element 2 of {30,30,30} (30),
                                  giving a value of 1.

element 3 of {20,30,40} (4d0) <=  element 3 of {30,30,30} (30),
                                  giving a value of 0.
```

Thus, variable "test2" is assigned a value of `{1,1,0}`

If two sequence variables or values are involved in a binary expression, both sequence lengths must be the same. For example, you cannot have an expression that tries to add a five element sequence and a six element sequence together to produce a result. You can, however, use sequences that are the same length but with different element types. This means you can add a sequence composed of atoms to a sequence composed of sequences. This program example helps clarify this point:

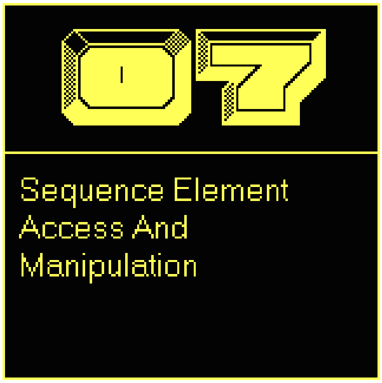
```
sequence seq1, seq2
seq1 = {1,0,1,0,1} or {0,1,0,1,0}
seq2 = {2,{2,2},2} + {{2,2},2,{2,2}}
```

Variable "seq1" is assigned a value of `{1,1,1,1,1}`, while "seq2" is assigned a value of `{{4,4},{4,4},{4,4}}`. The rule of atoms and sequences in a binary expression was used in working out "seq2" 's value.

This chapter not only taught you how to assign values to sequence type variables, it also showed you how powerful sequences can be in data manipulation. With a single Euphoria statement, you can change a series of linked values in a flash. Try doing that with a handful of atom variables!

But all of this is at the top level of the sequence. How do we access and change individual elements of a sequence, or create new sequence values by joining existing atom and sequence values in our programs? Well, the next chapter will focus on these topics as we journey further in the adventure that is the Euphoria language!

7. Sequence Element Access And Manipulation



Even though we devoted an entire chapter of this tutorial to the assignment of sequence variables with entire sequence values, it is actually the handling of elements that will take up most of your time when programming with sequences. This makes sense, as sequences allow the programmer to handle many individual data objects very quickly and all at once. This chapter will show you how to extract elements out of sequences and also to link them together to make new sequences.

To start off, you need to know the Euphoria syntax required to reference individual elements of a sequence. It is shown on the next page, as it requires a considerable amount of explanation.

```
variable  =  sequence variable[element number(s)]
```

Accessing an element involves the use of an assignment statement. **sequence variable** contains the value that we want to reference the elements in **element number(s)** means one or more element numbers that are being referenced. The square braces [and] are used to tell Euphoria that this is an element number and are always on either side of the number. The first element number in a sequence always starts at 1, and increases by one with the second and following elements after. Accessed elements of a sequence are stored in a receiving variable, defined here to the left of the equal sign as **variable**. The actual type of the variable to receive the element depends on the type of data object the element is. If the element is a sequence, the receiving variable must be able to hold sequences. If the element is an atom, the receiving variable must be able to hold atoms.

```
sequence list_of_days, list_of_months, month_name
atom days_in_month

list_of_months =
{ "January", "February", "March", "April",
  "May", "June", "July", "August",
  "September", "October", "November", "December"}

list_of_days = {31,28,31,30,31,30,31,31,30,31,30,31}

days_in_month = list_of_days[3]
month_name = list_of_months[3]
```

This program example obtains the name and number of days for the month March from two sequence variables ("list_of_months" and "list_of_days") that were previously assigned data. "month_name" is given the value of the third element of "list_of_months," and "days_of_month" is given the third element of "list_of_days." Notice both receiving variables have the correct variable type needed to accept the values. This is important!

When accessing an element of a sequence that is itself a sequence, it is possible to access the elements inside that sequence as well. What is required is another element index to access a second level. To access elements within a two-dimensional index, you use the syntax below:

```
variable  =  sequence variable [element number(s)][element number(s)]
```

If we were dealing with sequence made up of sequences, the first **element number(s)** serves as the index number needed to access any of the sequences making up the main sequence. The second **element number(s)** serves to access the elements within the sequence referenced by the first **element number(s)**.

A program example on the next page will help clear up any confusion in understanding how to use two indexes to reference elements.

```
sequence days_of_months
atom no_leap_february, leap_february

days_of_months = {31,{28,29},31,30,31,30,31,31,30,31,30,31}
no_leap_february = days_of_months[2][1]
leap_february = days_of_months[2][2]
```

This program example extracts the number of days in February in both a leap year and non-leap year situation. Look at the second element of the sequence value stored in variable “days_of_months.” It is itself a sequence two elements long. The atom variable “no_leap_february” is assigned the value of 28, the first element that makes up the sequence that is the second element of “days_of_months” ([2][1]), while atom variable “leap_february” is assigned 29, the second element that makes up the second element of “days_of_months” ([2][2]). Notice the indexes are in order of descending level, giving it a reversed appearance.

Here are the “ dos ” and “ don’ts ” of using element indexing in any sequence. First of all, you can replace a constant number with a variable name, like “[element_id]” for example. You can also have more than two levels of element access if the sequence is built for it. However, if an element is an atom, you cannot use another level of element indexing to go deeper into that element: only elements that are sequences can be accessed in this manner. Also, you cannot use an element number of zero, or an element number that is larger than the length of the sequence (for example, trying to access element 4 in a sequence that is only 3 elements long).

It’s possible to reference more than one element of a sequence at a time. The syntax below is a variation of what you were introduced to before:

```
seq. variable  =  seq. variable[starting element..ending element]
```

Euphoria uses double periods (..) to indicate that this is not a single element we are accessing but an inclusive (meaning including the starting and ending points) range of elements.

The program example below shows how to access a range of elements:

```
sequence nine_numbers, first_four, last_five

nine_numbers = {1,2,3,4,5,6,7,8,9}
first_four = nine_numbers[1..4]
last_five = nine_numbers[5..9]
```

The symbol $ now means ‘the last element‘.

In this example nine_numbers[5..9] is the same as nine_numbers[5..$]

```
"first_four" contains {1,2,3,4} }
"last_five"  contains {5,6,7,8,9}
```

A receiving variable is always of type sequence, as using a range returns a sequence value, even if the starting and ending elements are the same.

Accessing ranges follow the same rules as accessing individual elements. In addition, the starting and ending element numbers must be within the length of the accessed sequence. If the starting element number is equal to the ending element number + 1, a null sequence ({ }) is returned. You cannot, however, have a situation where the starting element number is greater than the ending element number by more than 1. You can use ranges when dealing with multi-dimensional sequences, but there are some limitations as listed below in this example:

```
sequence bigseq, seq1, seq2
bigseq = {{1,1,1},{2,2,2},{3,3,3}}
seq1 = bigseq[1][1..2]
seq2 = bigseq[1..2][1]
```

While you can access a range in a sequence element, you cannot reverse it to access an element out of a range of sequence elements, as in “seq2.”

If we changed the syntax around to place the referenced sequence element on the left side of the equal sign, we would have the ability to change the value of a specific element:

```
sequence variable [element number(s)] = expression
```

Here is a program example that demonstrates how to change both a single element and a range of elements:

```
sequence bunch
bunch = { "cat",5,{1,9,8,4},{0,0,0}}
bunch[1][1] = ’b’
bunch[2] = {7,7,7} + 1
bunch[3][3..4] = {9,7}
bunch[4][1..3] = -20
```

Let’s walk through the program example to understand what is going on:

```
bunch = { "cat",5,{1,9,8,4},{0,0,0}}
-- assign variable "bunch" with the value of
-- {{99,97,116},5,{1,9,8,4},{0,0,0}}

bunch[1][1] = 'b'
-- access first element in element 1 of "bunch"
-- and change it from 99 ('c') to (98) ('b'),
-- "bunch[1]" is now {98,97,116}


bunch[2] = {7,7,7} + 1
-- access second element of "bunch"
-- and change it  to the value  of expression {7,7,7} + 1,
-- "bunch[2]" is now {8,8,8}

bunch[3][3..4] = {9,7}
-- access third and fourth elements in element 3 of "bunch",
-- and replace with {9,7}, "bunch[3]" is now {1,9,9,7}

bunch[4][1..3] = -20
-- access all three elements in element 4 of "bunch"
-- and replace with {-20,-20,-20},
-- "bunch[4]" is now {-20,-20,-20}
```

When a single atom value is being assigned to a range of elements, Euphoria converts this to a temporary sequence value equal to the length needed to cover the starting and ending element numbers in the range. The sequence value is made up of the original atom value repeated as many times as needed. That is why "bunch[4]" is assigned a value of `{-20,-20,-20}`.

When the program example completes, the value of variable "bunch" is:

```
{{98,97,116},{8,8,8},{1,9,9,7},{-20,-20,-20}}
```

You can also create new sequence values in variables by adding together existing atom or even sequences. To do this, you use the syntax below:

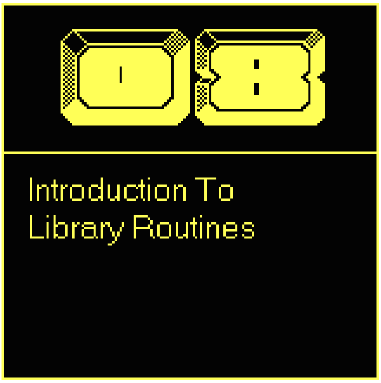
```
sequence variable = expression & expression
```

The & operator joins together expressions that evaluate into atom or sequence values to form new sequences. The result is a sequence that is as long as (in elements) the total sum of the lengths of the atoms and sequences used in the joining. For example, joining a four element long sequence with an atom gives a five element long sequence. This program example demonstrates the results of several joining attempts:

```
sequence s1, s2, s3

s1 = 5 & 4
     -- s1 is assigned {5,4}

s2 = 90 & {30,60}
     -- s2 is assigned {90,30,60}

s3 = {{1,1},{2,2,2}} & {3,3,3} + 1
     -- s3 is assigned {{1,1},{2,2,2},4,4,4}
```

Congratulations! You now understand the basic concepts of Euphoria! Feel free to review the previous chapters again. A full understanding of the core concepts is needed to learn the topics in the next chapters ahead.

8. Introduction To Library Routines



The chapters you have read so far have helped teach you how to accomplish the primary purpose of a program: the processing of data. You know the two types of data, you know how to declare variables to hold data, and you know how to use assignment statements in order to initialize and change values in variables. However, this is only the core. The declaration and assignment statements alone are not enough to make a program that actually does something useful.

oE Many of the libraries described here are still available, but are intended for backwards compatibility. Use the standard library instead. DOS specific routines (graphics, mouse, sound) are limited Eu3

Euphoria has something called "library routines" that allow you to do things beyond the power of simple assignment statements. When requested by the programs that you will write, they allow you to access specific features of your computer. This allows you to write software like games, office and home applications, or system utilities.

oE library is larger, and organized differently

Euphoria (Version 1.5) has grouped all library routines into the following categories:

- Predefined Types - library routines that test the type of a data
- Sequence Manipulation - library routines that offer advanced sequence handling features
- Searching and Sorting - library routines that compare, look for, and sort data objects
- Pattern Matching - library routines that can convert alphabet case and allow pattern matching in a string of characters
- Math - library routines that handle advanced mathematic formulas far beyond the power of the math operators +,-,/, and *

- Bitwise Logical Operators - library routines that handle binary bits
- File And Device I/O - library routines that let programs use the hard and floppy drives, screen, keyboard, and other computer hardware.
- Mouse Support - library routines that let your programs use the features of the mouse, like clicking buttons and pointer movement.
- Operating System - library routines that handle your program's relationship with the operating system on your computer.
- Special Machine Dependent Routines - library routines that allow direct access to computer resources normally accessed by Euphoria.
- Debugging - library routines that lets you do diagnostics on the program while it runs.
- Graphics And Sound - library routines that lets your programs perform multimedia features like graphics and sound.
- Machine Level Interface - library routines that allow access to low level (machine language) features of the computer

We will cover most of the library routines in this tutorial, except those that involve machine language work. These are for the advanced programmer who has machine language experience. The library routines will be introduced based on the subject in each of the next chapters ahead.

Where are the library routines stored? Well, they are defined in one of two places. Some are written in machine language and are defined in the interpreter EX.EXE. The rest are written in the Euphoria programming language. Library routines written in Euphoria require a special Euphoria programming statement called an "include" statement. The syntax is listed below:

Legacy Eu include files are in /include. Current oE include files are in /include/std

```
include (file name).e
```

Routines within are interpreter itself are called built-in routines—no include statement is needed for them to work.

`include` is no longer restricted to the top of your program. But, can not be nested

Compile using euc

The '.e' is common but not mandatory, 'any' or 'no' extension is permitted

To use library routines written in Euphoria, an include statement MUST be at the top of your program or they will not work. When you use EX.EXE to interpret your program, or BIND.BAT to create a machine language replica, (file name) is referenced to get the programming statements of **(file name).e**, the library routine(s) stored in there. Include files always end in an .e extension.

The name of the include file will depend on the Euphoria-coded library routine you are using. Euphoria (Version 1.5) has 8 include files, listed below with a brief description of each:

These are legacy library files. Use the standard library instead

- GRAPHICS.E —Graphics And Sound
- SORT.E —Sorting Routine
- GET.E —Input And Conversion Routines
- MOUSE.E —Mouse Routines
- FILE.E —Random Access File Operations And Directory Functions
- MACHINE.E —Machine Level Programming For 386's And Higher
- WILDCARD.E —Wildcard String Matching And Conversion
- IMAGE.E —Graphical Image Routines

Remember to look up the new replacements in the standard library

When a new Euphoria-coded routine is introduced in the tutorial, the proper include file will be shown, so don't worry about remembering all these include file names.

The value returned by a function may now be ignored.

In this sense, functions can be treated as procedures.

There are two kinds of library routines: procedures and functions. The only difference between a procedure and a function is that a function will return a value after it finishes running, while a procedure does not. To call a library routine that is a procedure, here is the following syntax below:

```
procedure name (parameters)
```

The 'parameter' is properly called an 'argument'

The **procedure name** is followed by a pair of brackets that contain a list of values to be sent to the procedure for processing called **parameters**. Parameters can be data object values, variable names, or expressions that work out to a value. Procedures can have no parameters (if they are not required), or an endless list of parameters. If more than one parameter is passed to a procedure, each parameter is separated by commas. If no parameters are passed, then the brackets are placed back-to-back with no spaces in-between, like "()".

The following are actual Euphoria procedures that you will be learning, about later. They are being presented here for you to see how each accepts parameters. Don't worry about what they do just yet.

```
print(1, "Hey,now!" )

clear_screen()
```

```
pixel({BRIGHT_BLUE,BRIGHT_RED,BRIGHT_GREEN},{100,150})
```

Functions are identical in appearance to procedures, so the syntax of a function isn't hard to learn if you already understand the syntax of procedures. However, a function requires a slight addition to its syntax in order to return a value, as previously mentioned. The syntax of a function is on the next page.

```
receiving variable = function name (parameters)
```

Because a function returns a value after processing is complete, it must be used in an assignment statement with a variable on the left side of the equal sign to receive the returned value. You could say a function is very similiar to an expression because, like an expression, it returns a single value. Functions can return a value of any data object type (an atom or sequence). Some Euphoria functions can return both data object types. For this reason, it is advisable to have a receiving variable of type object to handle both types of return values. You may remember, in our discussion on declaring variables, we mentioned object type variables are used when the type of data being returned from a program process is unknown. Now you know why object type variables are important when dealing with functions!

This is how a function is used most often.

Because a function's syntax requires the use of an assignment statement, and because it returns a single value, it can be part of an expression that is itself evaluated to a single value. Here are some examples of actual Euphoria functions, the last of which is being used as part of an expression. Again, don't try to understand the meaning of each function.

```
index_bitmap = read_bitmap( "index.bmp" )

pressed_key = get_key()

computed_result = sqrt(25) * (30 + units)
```

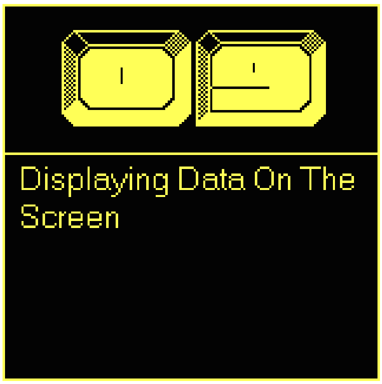
While a function can modify the original value of a receiving variable, both procedures and functions do not modify the parameter values that are passed to them. Also, procedure and function names follow the same rules that variable names must adhere to.

The following abbreviations will be used every time a new library routine is introduced to you. Take a moment to look them over:

- a —either an atom data object or a variable of type atom
- i —either an integer data object or a variable of type integer
- o —either an object data object or a variable of type object
- s —either a sequence data object or a variable of type sequence
- ra,ri,ro,rs —receiving variable, the second letter means variable type.
- If any of these abbreviations are used more than once, a number will follow the letter (i.e. s1, s2, s3 or o1, o2) to separate the parameters apart.

The next chapter will begin your learning of Euphoria library routines!

9. Displaying Data On The Screen



Now that you have reached this current level of understanding about the Euphoria programming language, the tutorial's style of teaching is going to change. Instead of reading program examples on the screen, you will now be able to execute actual Euphoria programs and view their source code. This approach is necessary, as these advanced features of Euphoria need more than reading text in the tutorial in order to be clearly understood.

The program examples you have studied through the past chapter have one thing in common. All the data they process is stored in the body of the program, in the form of assignment statements. In real life, this is not the case. Today's programs do not keep data as part of the source code, but instead obtain it from elsewhere.

The idea of retrieving and storing data outside the program makes good sense, for three reasons. First of all, the data can be edited without the program itself being changed. Second, if more than one program uses the same kind of data (like an employee list), there is no data duplication. The programs share the same data instead of each program having its own copy of the data. Finally, having the ability to send data outside the program means the data can be made presentable to human eyes. Up to now, you have taken the word of this tutorial and its author that what is stored in the variables of example programs was the case. In real life, people need more than that in the form of printed reports and spreadsheet figures on the screen.

Data that is accepted by a program is called "input." Data that is produced by a program is called "output."

Programs send data to and receive data from components of your computer called "devices." These device can either be "input devices" that send data to a program, or "output devices" that receive data from a program. There is a third type of device called an "I-O device," which is both an input and an output device, but this type will be covered in a later chapter. The keyboard and mouse are examples of input devices. The computer screen and printer are examples of output devices.

When a Euphoria program starts running, some devices are automatically allocated for use. These three devices are listed below:

- 0 or standard input, the keyboard by default
- 1 or standard output, the screen by default
- 2 or standard error, the screen by default

Devices 1 and 2 are actually the same, but are defined separate in case there is a reason one type of screen output should be made distinct from the rest. Notice the phrase "by default." This implies we can make the numbers mean something else other than keyboard and screen. In a later chapter, you will be shown exactly how to do this. To keep it simple for now, understand that 0 means "keyboard," and 1 means "screen." This chapter will introduce screen output, while a future chapter will focus on keyboard input.

Now that you are familiar with Euphoria's device numbers, let's introduce you to your very first Euphoria library routine, called `print()`:

```
print(1,o)
```

print() displays a Euphoria data object on the screen. The object is displayed at the current cursor position.

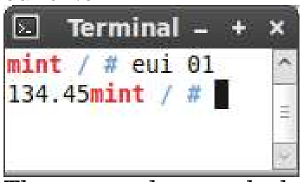
`print()` displays data on the screen "as is," meaning what is displayed on the screen is the actual value of the data object. With sequences, you will be able to see the braces and commas, even if you used a character string to represent the value. If you click the demo button on the remote, you will be able to run and view the source of demo programs that use `print()`.

You can cut and paste these demo programs into an editor,

save with extension .ex,

and execute them in a terminal.

eui 01.ex



```
mint / # eui 01
134.45mint / #
```

The expected output looks like:

```
134.45
```

program 1

```
atom some_atom_value
some_atom_value = 134.45
print(1,some_atom_value)
```

```
134.45
```

program 2

```
sequence some_sequence_value
some_sequence_value = {1,2,3,4,3,2,1}
print(1,some_sequence_value)
```

```
{1,2,3,4,3,2,1}
```

program 3

```
print(1,36/2)
```

```
18
```

program 4

```
sequence my_name
my_name = "David Gay"
print(1,my_name)
```

```
{68,97,118,105,100,32,71,97,121}
```

One minor drawback with `print()` is that it displays the actual values of atom and sequence data objects. As a result, any data object values meant to be shown on the screen as ASCII characters cannot be displayed using `print()`. However, Euphoria has another library routine that can display screen output in human-readable form:

```
puts(1,o)
```

Because `puts()` displays data as text characters instead of actual values, cursor control codes such as line feed (10), carriage return (13), and tab control (9) can be utilized. You can have text strings separated by lines or formatted in different tab columns. This makes `puts()` useful in print control as a result.

Two notes about puts(). First of all, this library routine can only print one-dimensional sequences (those composed of atoms as elements). This makes sense, as text character strings are really a representation of one dimensional sequences. Also, any attempt to print an atom value larger than 255 will result in an incorrect character value being displayed. Again, this makes sense because the ASCII code set goes from 0 to 255.

Some demo programs are available from this screen to show how `puts()` works:

program 5

```
atom a_character
a_character = 'A'
puts(1,a_character)
```

```
A
```

program 6

```
sequence a_string
a_string = "Utter Nonsense"
puts(1,a_string)
```

```
Utter Nonsense
```

program 7

```
puts(1,"Column 1\tColumn 2\tColumn 3\n\n")
puts(1,"********\t********\t********\n")
puts(1,"********\t********\t********\n")
puts(1,"********\t********\t********\n")
puts(1,"********\t********\t********\n")
puts(1,"********\t********\t********\n")
puts(1,"********\t********\t********\n")
puts(1,"********\t********\t********\n")
puts(1,"********\t********\t********\n")
puts(1,"********\t********\t********\n")
puts(1,"********\t********\t********\n")
```

```
Column 1    Column 2    Column 3

********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
```

It is sometimes necessary to display additional screen output to help clarify information. This process is called "print formatting." Examples of print formatting are adding commas and dollar signs to financial figures or to limit the number of decimal places shown in scientific data. We can use the library routine `printf()` to display formatted screen output:

```
printf(1,s,o)
```

printf() prints data (shown as o) as a character string on the screen just like puts(). Before it does this, however, it formats the data using a format string (shown as s). The format string must contain special format codes that will edit the way the data will appear on the screen, and optionally, some extra text. The data to be displayed can either be a single atom value, or a sequence that contains a list of values to be edited using the format string.

The number of format codes used will depend on the number of values that are to be edited and then printed. Let's look more closely at the format codes available for the programmer to use:

`%d` — this will edit any atom values to appear as a decimal integer. Decimal integers are the numbers we are all familiar with, using a numbering system of 10 digits from 0 to 9.

`%x` — this will edit any atom values to appear as a hexadecimal integer. Hexadecimal integers are numbers that are made up of digits from 0 to 9, A, B, C, D, E, and F. It is also known as a "base-16" numbering system (our decimal system is a "base-10" numbering system).

`%o` — this will edit any atom values to appear as an octal integer. Octal integers are numbers that are made up of digits from 0 to 7. It is also known as a “base-8” numbering system.

`%s` — this will edit any sequence values to appear as character strings. This works just like `puts()` but with extra formatting options.

`%e` — this will edit any atom values to appear as a floating point number using exponential (or standard) notation.

`%f` — this will edit any atom values to appear as a floating point number, but not using exponential notation.

`%g` — this will edit any atom values to appear as a floating point number, using either the format offered by %e or %f, whichever works best for the display of the value.

`%%` — this will display the percentage character.

Format codes can have a field width number added to control the number of displayed characters. For example, %6d means a minimum display size of 6 digits. If the number ends up smaller than 6 digits in this example, it will be right-justified with spaces. Adding a - in front of the number, like %-6d, will make it left-justified. If a zero is placed in front of the number, like %06d, zeroes instead of blanks will be used to fill up any leftmost field positions not used by the value. Placing a plus sign (+) in front of the number, like %+6d, will make positive values appear with a leading plus sign. Normally, positive values are displayed without a sign.

The addition of a field size is not limited to decimal integers alone. You can even add them to other format codes like %s or %f (though using to fill any unused leftmost positions of a string with zeroes is unusual, even if it is considered valid).

With floating point format codes (%f), the field size can be a decimal number. The number to the LEFT of the decimal point is the field size, while the number to the RIGHT of the decimal point is the precision. Precision means how many digits of the decimal fraction you want to have shown. For example, %7.3f will show the floating point value 13.1705 as 13.170 with a leading blank (13.170). The decimal fractions are not cut off, however: they are rounded up. This means %6.2f will display the value 899.987 as 899.99 with no leading blanks.

Remember when we stated the data to be displayed in `printf()` is either a single atom value or a sequence representing a list of values to be displayed? For this reason, if you are attempting to display a SINGLE sequence value as a string, you must make it the only element of a sequence.

This step is necessary, otherwise the sequence itself will be treated as a list of values and only the first character will be printed:

```
printf(1, "%s\n", "Euphoria")   -- Only "E" is printed

printf(1, "%s\n",{"Euphoria"}) --"Euphoria" is printed
```

A series of demo programs have been assigned to this page to help clarify `printf()`.

program 8

```
atom     atom_value
integer  integer_value

atom_value = 1233.14
integer_value = 255

print(1,atom_value)
printf(1,"\n  is shown as %d when using %%d", atom_value)
printf(1,"\n  is shown as %e when using %%e", atom_value)
printf(1,"\n  is shown as %f when using %%f\n\n", atom_value)

print(1,integer_value)
printf(1,"\n  is shown as %x when using %%x", integer_value)
printf(1,"\n  is shown as %o when using %%o\n\n", integer_value)

atom_value = 'A'
print(1,atom_value)
printf(1,"\n  is shown as %s when using %%s\n", atom_value)
```

```
1233.14
  is shown as 1233 when using %d
  is shown as 1.233140e+03 when using %e
  is shown as 1233.140000 when using %f

255
  is shown as FF when using %x
  is shown as 377 when using %o

65
  is shown as A when using %s
```

program 9

```
puts(1,"Field Width Using %9s\n")
puts(1,".........\n")
printf(1,"%9s", {"CAT"})
puts(1,"\n\n")

puts(1,"Field Width Using %-9s\n")
puts(1,".........\n")
printf(1,"%-9s", {"CAT"})
puts(1,"\n\n")

puts(1,"Field Width Using %4d\n")
puts(1,"....\n")
printf(1,"%4d", 123)
puts(1,"\n\n")

puts(1,"Field Width Using %04d\n")
puts(1,"....\n")
```

```
printf(1,"%04d", 123)
puts(1,"\n\n")

puts(1,"Field Width Using %+4d\n")
puts(1,"....\n")
printf(1,"%+4d", 123)
puts(1,"\n\n")
```

```
Field Width Using %9s
.........
      CAT

Field Width Using %-9s
.........
CAT

Field Width Using %4d
....
 123

Field Width Using %04d
....
0123

Field Width Using %+4d
....
+123
```

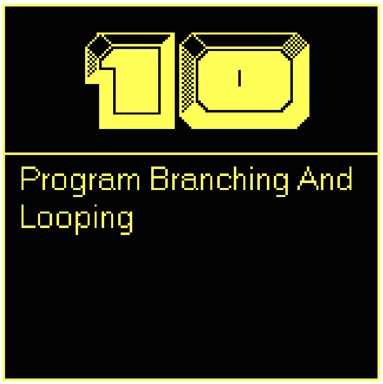
program 10

```
puts(1, "How Decimal Precision Affects Floating Point Values\n\n")
printf(1,"%f  <- Using Straight %%f\n", 123.7651)
puts(1,".......\n")
printf(1,"%7.3f  <- %%7.3f\n", 123.7651)
printf(1,"%7.2f  <- %%7.2f\n", 123.7651)
printf(1,"%7.1f  <- %%7.1f\n", 123.7651)
printf(1,"%7.0f  <- %%7.0f\n", 123.7651)
```

```
How Decimal Precision Affects Floating Point Values

123.765100  <- Using Straight %f
.......
123.765  <- %7.3f
 123.77  <- %7.2f
  123.8  <- %7.1f
    124  <- %7.0f
```

We would normally explore the other side of screen output by introducing library routines that accept data from the keyboard. However, because some keyboard library routines require a program to execute a block of statements based on a condition or for a set number of times in order to work properly, a detour is needed. The next chapter will show you how to change the way your program executes.

10. Program Branching And Looping



The demonstration programs in this tutorial, while helpful in understanding Euphoria, are not good models of how the majority of programs work today. Program execution is not a straight line from program start to program end. A program may execute a group of statements meant to handle an anticipated condition, or repeat those statements until a condition is met. In short, program execution is a series of backtracks and detours.

oE added 'end loop', 'switch' flow control statements.

The basics, described here, are all you need for flow control.

A group of statements that runs only when a specific condition is met is called a program branch. A group of statements that is meant to repeat while a condition is being met is called a program loop.

Let's begin our learning by introducing the "if" statement, which is the workhorse of program branching:

```
if expression then
one or more Euphoria statements
end if
```

The "if" statement will execute *one or more Euphoria statements* only if *expression* evaluates to be true. If the expression evaluates to to be false, program control will skip the conditioned statements and then resume with the first Euphoria statement following the `end if` line.

The expression of an "if" statement can be any arithmetic, logical, or relational expression, or even a constant value. Most of the time, however, an expression is either a simple logical expression or a larger relational expression linking logical expressions together.

A demo program is available to demonstrate a series of "if" statement examples. But before you run the demo, view the source to see if you can guess which of the `puts()` lines will be printed.

program 11

```
atom first, second, sum
first = 36
second = 1.5
sum = first * second

if second >= 1.4 and second <= 1.6 then
```

```
     puts(1,"Condition 1 is true\n")
end if

if sum = 54 then
     puts(1,"Condition 2 is true\n")
     first = 63
end if

if first = 36 then
     puts(1,"Condition 3 is true\n")
end if
```

```
Condition 1 is true
Condition 2 is true
```

The “if” statement can be used to execute a group of statements for both false and true outcomes of a condition. One way to do this is to write a second “if” statement that checks for the opposite of the first “if” statement.

```
if speed > 65 then
     puts(1,"You are driving too fast!\n")
end if

if speed <= 65 then
  puts(1,"Thank you for driving within the speed limit!\n")
end if
```

The use of two “if” statements works, but it is inefficient because both “if” statements are checked even though only one of two outcomes will happen. You can’t have a condition that is both true and false. It would be nice to have the “if” statement work like a toggle switch, executing the right block of statements with only one condition check.

Fortunately, the “if” statement can be modified to handle two outcomes without using two “if” statements:

```
if expression then
    one or more statements
    that will only run if expression is true
else
    one or more statements
    that will only run if expression is false
end if
```

When the “if” statement has an “else” option added, only one of two things will happen. If the expression portion of the “if” statement works out to be true, the statements immediately following below are executed, and the statements under the “else” line are ignored. If the expression works out to be false, it’s the reverse that happens. The statements under the “if” line are ignored, but the statements under the “else” line are executed. Whatever the outcome of the expression, the first statement following “end if” is always executed once the “if” statement completes. A demo is available on this screen to show the use of “else.”

program 12

```
atom character

character = 'a'

if character = 'a' then
     puts(1,"********************************\n")
     puts(1,"* This line should be printed! *\n")
     puts(1,"********************************\n")
else
     puts(1,"This line should not be printed!\n")
end if

character = 'b'

if character = 'a' then
     puts(1,"This line should not be printed!\n")
else
     puts(1,"********************************\n")
     puts(1,"* This line should be printed! *\n")
     puts(1,"********************************\n")
end if
```

```
********************************
* This line should be printed! *
********************************
********************************
* This line should be printed! *
********************************
```

Using an “if-else” combination can handle situations that result in only one of two outcomes. For situations that can result in more than two outcomes, we replace “else” with “elsif.”

```
if  expression 1 then
     one or more statements
     that execute if expression 1 is true
elsif expression 2 then
     one or more statements
     that execute if expression 2 is true
elsif expression 3 then
     one or more statements
     that execute if expression 3 is true
elsif expression 4 then
     one or more statements
     that execute if expression 4 is true
end if
```

Using "elsif" with "if" creates a decision chain, where each of the expressions are checked one after another, starting with the first one, *expression 1*. Once an expression is found to be true, the statements conditioned to that expression are run. After this, all other expressions that follow in the "if-elsif" chain are skipped, and the first statement following "end if" is executed.

Using "if-elsif" allows you to check for a specific set of outcomes to a situation, rather then using "either this or the other" kind of logic. It also lets you analyze a problem from a step-by-step approach, eliminating the need for designing complex expressions and typing out many "if" statements.

A demo program is available that shows one use of "if-elsif" chains:

program 13

```
atom character

character = 'M'

if character >= 'A' and character <= 'J' then
     puts(1, "This line should not be printed\n")
elsif character >= 'K' and character <= 'T' then
     puts(1, "********************************\n")
     puts(1, "* This line should be printed! *\n")
     puts(1, "********************************\n")
elsif character >= 'U' and character <= 'Z' then
     puts(1, "This line should not be printed\n")
end if
```

```
********************************
* This line should be printed! *
********************************
```

You can have what is called nested "if" statements, where one "if" statement leads to a second "if" statement when its expression is true:

```
if expression 1 then
    if expression 2 then
          one or more statements
          that run only if expression 2 is true
    end if
end if
```

In the case of the example on the previous page, it is the same as:

```
if expression 1 and expression 2 then
     one or more expressions
     to be run only if both expressions are true
end if
```

However, using nested "if" statements allow you more flexibility, particularly when you want to mix "if" statements, library routine calls, and assignment statements in a block of conditioned statements to be run. It also gives you the option of avoiding the use of "if" statements that have complex relational expressions made up of smaller expressions.

When using nested "if" statements, make sure each "if" has a matching "end if" for each "if" level, just like the example you saw earlier. Make sure that any optional "else" and "elsif" used are on the correct level as the "if" statement they are associated with.

You've probably noticed, in all of the "if" statement examples we have shown, we used expressions that involved atom variables and values. The reason for this is because "if" statements cannot handle direct comparisons using sequence data objects. However, there is a library routine that can help you get around this problem:

```
ri  = compare(o1,o2)
```

`compare()` takes two data objects, **o1** and **o2**, and compares them, returning an integer value that represents the result of the comparison. If **o1** is equal to **o2**, 0 is returned. If **o2** is smaller than **o1**, -1 is returned. If **o2** is larger than **o1**, 1 is returned. `compare()` can compare two atoms, two sequences, or a sequence and an atom. Atoms are always assumed to be smaller than sequences in value. Where sequence comparison is concerned, it's an element by element comparison until a difference is found, either in the length of the sequences or a different element value.

So, to compare one sequence with another, you simply follow the format shown in this example below:

```
sequence my_name

my_name = "David Gay"
if compare(my_name, "David Gay") = 0 then
     puts(1,"Hi, David!\n")
else
     puts(1,"Who the heck are you?\n")
end if
```

A program demo is available to help clarify `compare()` if needed:

program 14

```
sequence animal1, animal2

animal1 = "cat"
animal2 = animal1

if compare(animal1,animal2) = 0 then
     puts(1, "These strings are the same\n")
else
     puts(1, "These strings are unequal\n")
end if

animal2 = "CAT"

if compare(animal1,animal2) = 0 then
     puts(1, "These strings are the same\n")
else
     puts(1, "These strings are unequal\n")
end if
```

```
These strings are the same
These strings are unequal
```

Program branching is only one part of changing the way your program runs. You can also make a group of statements repeat for as long as an expression remains true. This is done by using the “while” statement:

```
while expression do
    one or more statements
    that are executed while condition is true
end while
```

When the “while” statement first starts, it checks *expression* to see if it is true. If it is not, the next programming statement following **end while** is executed. If the expression is true, the “while” statement will continue to repeat the statements between “while” and “end while” until the expression portion of the “while” statement becomes false. This tells you that the repeating statements must be able to make the expression change to false, or the program will be stuck in an endless “while” loop.

A demo program using a “while” statement is available.

program 15

```
atom count
puts(1,"Hello, I’m Sparky I, The Counting Euphoria Program\n")
puts(1,"Watch me count to 10!\n\n")
count = 1
while count < 11 do
     printf(1,"%d\n",count)
     count = count + 1
end while
```

```
Hello, I’m Sparky I, The Counting Euphoria Program
Watch me count to 10!

1
2
3
4
5
6
7
8
9
10
```

If you want to repeat a group of statements for a specific number of times, Euphoria offers the “for” statement:

```
for ra = start value to end value by increment do
     one or more statements to repeat
     until ra > end value
end for
```

When the “for” statement starts, it declares a temporary index variable, *ra*, assigning it a *start value*. It then processes the statements down to the *end for* line. Once it reaches that point, it changes ra’s value by adding the *increment*. The program then loops back up to check if ra is larger than the *end value*. If not, the statements are processed again, and ra is changed by adding the value *increment*. This loop will keep repeating until ra is larger than *end value*. When this occurs, *ra* is “undeclared,” and the program continues with the next statement after the “end for” line.

The increment portion of the “for” statement is an optional feature. If left out, the default increment is 1. You can have a negative increment value, so the “for” statement counts down instead of up, but set the end value of the “for” statement to be smaller than the start value.

Unlike the “while” statement, the “for” statement cannot be locked into an endless loop, because the expression change is handled automatically. Also, the index variable cannot be changed by any statements within the loop, but it can be referenced for use (such as an element index in a sequence). If the value in the index variable is required for later use, you should save it in a variable before the loop ends, because both the index variable and its value will vanish when the “for” statement ends. This temporary index variable has an attribute called “scope” , meaning it is accessible only for a specific duration in the program. This topic will be revisited later in this tutorial. In the meantime, run a program demo now to demonstrate how the “for” statement can be used.

program 16

```
puts(1,"Hello, I’m Sparky II, The Counting Euphoria Program\n")
puts(1,"Watch me count to 5, then back again to 1!\n\n")
for count = 1 to 5 do
     printf(1,"%d\n",count)
end for
puts(1,"\n")
for count = 5 to 1 by -1 do
     printf(1,"%d\n",count)
end for
```

```
Hello, I’m Sparky II, The Counting Euphoria Program
Watch me count to 5, then back again to 1!

1
2
3
4
5

5
4
3
2
1
```

Both the “for” and “while” statements can be nested, meaning that you can have smaller loops within larger ones. However, the nested “while” and “for” statement levels should have correctly matching “end while” and “end for” statements.

You can force both a “while” and a “for” statement to end early using the " exit " statement. The loop ends at the moment the “exit” statement is executed. The program will then resume at the next statement following “end for” or “end while”. The “exit” statement is best used as either an “emergancy brake” when something unexpected comes up, or for creating a loop that needs to end for a set of reasons uniquely separate from each other. A program demo shows the “exit” statement in action.

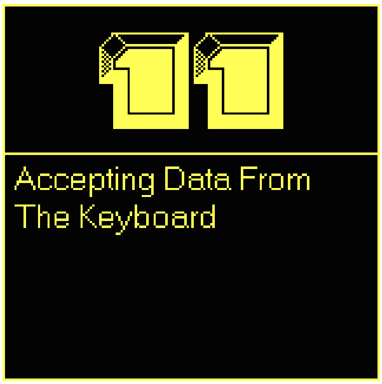
You have now learned everything you need to know to control the course of program execution. These tools will be valuable in the next chapter, when you learn how to accept data from the keyboard.

program 17

```
atom pet_index
sequence pets
puts(1,"Examples Of Animals You Can Keep As Pets\n")
puts(1,"========================================\n\n")
pets = {"Cat",
        "Dog",
        "Hamster",
        "Rat",
        "Snake",
        "Parrot",
        "Budgie",
        "Unelected Politician",
        "*END*"}
pet_index = 1

while 1 do
     if compare(pets[pet_index],"*END*") = 0 then
          exit
     else
          puts(1,pets[pet_index])
          puts(1,"\n")
     end if
     pet_index = pet_index + 1
end while
puts(1,"\nList Completed!\n")
```

```
Examples Of Animals You Can Keep As Pets
========================================

Cat
Dog
Hamster
Rat
Snake
Parrot
Budgie
Unelected Politician

List Completed!
```

11. Advanced Data Handling Part One



Displaying data on the screen is only one side of program-human interaction. While it is great that we can now write programs that present text on the screen, our programs can be even more useful if they can accept data for processing later. A program could be written to accept keyed-in monthly expenses and compute either a surplus or a deficit in your household budget. Euphoria has a set of library routines to handle keyboard input. Here's the first one listed below:

for oE use include std/console.e

```
include get.e
```

```
ri = wait_key()
```

`wait_key()` causes the program to pause until a key is pressed, then stores that value in an integer variable to the left of the equal sign.

You will notice that `wait_key()` is an example of a library routine that is externally defined in an include file, called get.e. get.e must be present at the top of your program in order to use this library routine.

oE uses include std/console.e

If a key generating a displayable character (alphabet, numbers, symbols, and punctuation) or cursor control (tab, linefeed, backspace, etc) is pressed, the value stored in the receiving integer value is the ASCII code for that value. This means you can use `puts()` or `print()` to display the value on the screen as a text character, or move the cursor. Keys like the function keys from F1 through F12, arrow keys, insert and delete keys, and any key pressed while the ALT key is pressed down generate a value higher than 255. These keys are meant to be defined for the use of the programmer and do not generate a displayable screen character.

A demo program showing how `wait_key()` is used is available on this page:

program 18

use

include std/console.e

```
include get.e
integer keystroke

puts(1,"Please press a key on the keyboard\n")
keystroke = wait_key()
puts(1,"\nThank you!\n")
printf(1,"You pressed the %s key!\n",keystroke)
```

It is sometimes necessary to read in an entire character string instead of a single character. Euphoria has a library routine that performs this:

```
ro = gets(0)
```

Like `wait_key()`, `gets()` pauses the program run for keyboard data. Unlike `wait_key()`, it is not a single value but a string of data that is stored in a variable at the left of the equal sign. However, notice that the receiving variable is an object type variable. This is necessary because a value of -1 (meaning that no keyboard data string was retrieved) may also be returned. It's also important to note that the Enter key that you pressed to end the string to send to `gets()` is a part of the string, at the very end as value 10. The receiving object variable will contain a sequence composed of atoms that represent ASCII codes. This means `puts()` and `printf()` can be used to display the data received by `gets()`. A demo program is available to show one use of `gets()` from this screen:

program 19

```
object name, city
puts(1, "Hello, what is your name?\n")
name = gets(0)
printf(1, "\nHello, %sWhat city are you from?\n",{name})
city = gets(0)
printf(1, "\nI've always wanted to go to %s", {city})
```

```
Hello, what is your name?
Fred

Hello, Fred
What city are you from?
RacoonCity
I've always wanted to go to RacoonCity
```

All the library routines introduced so far have one thing in common, and that is the accepted keyboard data is treated like a character string to be redisplayed as text later. This means if we entered a value of 16.13 using `gets()`, it will not be stored as an atom value of 16.13, but a 5 element long character string "16.13". This makes the library routines we have mentioned unsuitable for handling numeric data. However, Euphoria does have a more sophisticated routine that can accept both numeric and character data:

use

include std/get.e

```
include get.e
```

```
rs = get(0)
```

get() accepts a Euphoria data object and stores it into a receiving sequence variable. This means you can enter any atom value, character string representing a sequence, or even a very complex multi-dimensional sequence value.

get() converts keyboard input to an actual Euphoria data object and then stores it as the second element of a two sequence value. The receiving sequence variable is assigned this sequence value.

The first element of this sequence value serves as an error code on whether or not the string of characters you typed is a valid Euphoria data object. The error code is an atom value, and can be any one of the values listed below:

- 0 — Accepted value is a valid Euphoria data object.
- -1 — End of data hit before any data objects were read.
- 1 — Accepted value is not a valid Euphoria data object.

It's important to monitor the error code, as it tells you if the data that was keyed in is valid. Using the "if" statement to check the error code can help you ensure you do not receive any garbage input.

While actual atom and sequence values can be keyed in as is (like 45.1 or {123,{6,7,-1},17}), character strings must be enclosed with the double quotation mark or " .

get() has many advantages over gets() in a number of ways. First of all, you can enter both numeric and character data in any format. Second, you can choose to send a list of data objects on a single line separated by spaces or tabs, or enter them one at a time using the Enter key. However, get() only reads these values one at a time. As a result, whether you place four values on one line and press Enter, or enter one on each line and press Enter, you need four get()'s in your program to read them. Finally, get() does not include the Enter key, space, or tab values as part of the keyboard data.

use

A demo is available on this page to demonstrate how get() accepts both character and numeric data:

include std/console.e

program 20

```
include get.e

object input
atom value1, value2, value3, average, error_code
sequence name

error_code = 999
```

```
puts(1,"Hello! Enter your first name in quotes below, like \"John\":\n")

while error_code != 0 do
     input = get(0)
     error_code = input[1]
end while

name = input[2]

puts(1, "\nThank you! Now please enter any three numeric values below,\n")
puts(1, "and you will see both your name and the average of the three\n")
puts(1, "numbers you entered. Separate the numbers with the space bar:\n")

error_code = 999
while error_code != 0 do
     input = get(0)
     error_code = input[1]
end while
puts(1, "\nGot first number!\n")
value1 = input[2]

error_code = 999
while error_code != 0 do
     input = get(0)
     error_code = input[1]
end while
puts(1, "\nGot second number!\n")
value2 = input[2]

error_code = 999
while error_code != 0 do
     input = get(0)
     error_code = input[1]
end while
puts(1, "\nGot third number!\n")
value3 = input[2]

average = (value1+value2+value3)/3
printf(1,"\nHello, %s, your computed average is %4.2f\n",{name, average})
```

```
Hello! Enter your first name in quotes below, like"John":
"Fred"

Thank you! Now please enter any three numeric values below,
and you will see both your name and the average of the three
numbers you entered. Separate the numbers with the space bar:
3

Got first number!
10

Got second number!
77

Got third number!

Hello, Fred, your computed average is 30.00
```

You may have played some games that are able to accept data from the keyboard without pausing. Actually, they do pause, but only for a tiny fraction of time to check the keyboard buffer for a pressed keystroke. Because computers can operate at such high speeds, this delay goes unnoticed. The next library routine would be of interest to game designers:

```
ri = get_key()
```

`get_key()` stores each keystroke waiting in the keyboard buffer into a receiving varible. Multiple `get_key()`'s are required to read more than one waiting keystroke. If there are no keystrokes waiting, a -1 is returned instead. Unlike `wait_key()`, `get_key()` does not pause the program.

`get_key()` will also accept keystrokes from special keys like the function keys and arrow keys. A demo is ready to show how `get_key()` can interrupt a countdown from 100000 to 1. When you view the demo program's source code, see how `get_key()` is located in a program loop, like the "while" statement. This makes sense, as the keyboard buffer should be repeatedly checked for any waiting keystrokes.

program 21

```
integer keystroke, counter

keystroke = -1
counter = 10000
while keystroke = -1 and counter > 0 do
     keystroke = get_key()
     printf(1,"%5d\n",counter)
     counter = counter - 1
end while
if counter = 0 then
     puts(1, "\nHey, why didn't you press any of the keys?\n")
end if
```

```
   12
   11
   10
    9
    8
    7
    6
    5
    4
    3
    2
    1

Hey, why didn't you press any of the keys?
```

The one keystroke that cannot be accepted by a Euphoria program is Ctrl-C, generated by either pressing the c or break key while holding down the Ctrl key at the same time. If you press these combination of keys, the operating system will abruptly stop the program. In the case of larger, more complex programs, this can result in an unexpected loss of data. It would be preferable for the program to know that Ctrl-C has been pressed and perform an orderly shutdown.

First of all, we need to disable Ctrl-C so it cannot kill the program outright. This is done by using the `allow_break()` library routine:

use

include std/console.e

```
include file.e
```

```
allow_break(i)
```

Setting the parameter i to 1 will allow Ctrl-C to abruptly halt the program, while 0 would prevent usage of Ctrl-C to halt the program.

The most obvious benefits of `allow_break()` are in applications that offer menu options to users based on their security level. If a hacker could disable a menu program using Ctrl-C, s/he then could access the program of interest directly, bypassing security. However, having a program issue `allow_break(0)` at the start of its run would prevent this.

But remember that a program cannot directly accept a value of Ctrl-C, `allow_break()` or not. How can it know when to perform an orderly shutdown if it can't accept that value? Well, while it cannot accept Ctrl-C, it can DETECT it using this library routine:

use

include std/console.e

```
include file.e
```

```
ri = check_break()
```

`check_break()` returns the number of times Ctrl-C has been pressed since the start of the program, or since the last time `check_break()` was issued. This does not return the actual Ctrl-C code, it just returns a value that indicates the number of times Ctrl-C was detected by the program.

You must make sure you are using `get_key()` to read the keyboard buffer, in order for `check_break()` to work properly, even though it cannot read in the Ctrl-C code. A demo program has been made available from this page to show how `allow_break()` and `check_break()` are used to prevent Ctrl-C from stopping the program. You need to hit Ctrl-C or Ctrl-break three times to stop the demo from running.

include std/console.e

program 22

```
include file.e
```

```
integer control_break_counter, pressed_key
allow_break(0)

puts(1, "************************************\n")
puts(1, "* The Program That Would Not Die! *\n")
puts(1, "************************************\n")
puts(1,"\n")

control_break_counter = 0
while control_break_counter < 3 do
     pressed_key = get_key()
     if check_break() then
          control_break_counter = control_break_counter + 1
          if control_break_counter <= 2 then
               puts(1,"Neener-neener, can't kill meeeeeeee!\n")
          end if
     end if
end while

puts(1, "\n")
puts(1, "************************************\n")
puts(1, "* Okay, I'll be nice and stop.... *\n")
puts(1, "************************************\n")
```

```
************************************
* The Program That Would Not Die! *
************************************

Neener-neener, can't kill meeeeeeee!

[1]+  Stopped                 eui 22
```

Now that you have learned how to display data on the screen, and how to accept data from the keyboard, let's learn some powerful library routines that show you what you can really do with this data.

12. Advanced Data Object Handling, Part One

If you want to get the most out of the Euphoria programming language, getting a handle on how to work with data objects should be at the top of your list. There are a lot of library routines that are meant for handling data objects. You can check for the type of a data object. You can also change the way a data object is presented, such as going from 2.34 to “2.34”, and can even go beyond the simple operators of Euphoria to create your own unique data structures.

This is the secret to Euphoria flexibility.

Because some of the library routines in Euphoria can return either an atom or a sequence, it is very important to test for the type of a data object. The library routines involved are named the same as the variable type portion of a variable declaration statement.

To begin, `atom()` returns a 1 if the data object is an atom value:

```
ri = atom(o)
```

`integer()` returns a 1 if the data object is an integer:

```
ri = integer(o)
```

`sequence()` returns a 1 if the data object is a sequence:

```
ri = sequence(o)
```

also object()

oE object() returns i a s information or value unassigned yet

All of these library routines will return 0 if data object o is not the expected type. Run a demo from this screen now to demonstrate how to test for the type of a keyed-in data value. For obvious reasons, there is no such thing as an `object()` library routine.

program 23

use
include std/console.e

```
include get.e

atom atom_received, integer_received, sequence_received
sequence inputted_string

atom_received = 'n'
```

```
integer_received = 'n'
sequence_received = 'n'

puts(1,"Enter any numeric value (such as 5.004 or -7)\n")

while atom_received = 'n' and integer_received = 'n' do
     inputted_string = get(0)
     if inputted_string[1] = 0 then
          if integer(inputted_string[2]) then
               integer_received = 'y'
               puts(1,"\nThank you! This value is an integer!\n\n")
          else
               if atom(inputted_string[2]) then
                    atom_received = 'y'
                    puts(1,"\nThank you! This value is an atom!\n\n")
               end if
          end if
     end if
end while

puts(1,"Enter a sequence value (such as {54,-8,2.3} or \"Hi There!\")\n")

while sequence_received = 'n' do
     inputted_string = get(0)
     if inputted_string[1] = 0 then
          if sequence(inputted_string[2]) then
               sequence_received = 'y'
          end if
     end if
end while

puts(1,"\nThank you! Program Finished!\n")
```

```
Enter any numeric value (such as 5.004 or -7)
8.002

Thank you! This value is an atom!

Enter a sequence value (such as {54,-8,2.3} or"Hi There!")
{3, -1, 0.3}
Thank you! Program Finished!
```

To convert a character string to either an atom or a sequence value, you use the `value()` library routine:

use
include std/get.e

```
include get.e
```

```
rs = value(s)
```

This works the same way as `get()`, as it returns a two element sequence value, the first element being the error code, and the second element, if successful, being the actual atom or sequence value equivalent of the character string. The only difference is that the source is a sequence variable (s) and the destination is a receiving sequence variable (rs).

A demo is available showing how three character strings are converted to atom and sequence values:

program 24

```
include get.e

sequence character_strings, value_string
object actual_value

character_strings = {"45.99","{1,2,3,{4,4},5}","\"Euphoria\""}

for ix = 1 to 3 do
     printf(1,"Character String: %s\n",{character_strings[ix]})
     value_string = value(character_strings[ix])
     actual_value = value_string[2]
     puts(1,"Euphoria Data Object After value(): ")
     print(1,actual_value)
     puts(1,"\n\n")
end for
```

```
Character String: 45.99
Euphoria Data Object After value(): 45.99

Character String: {1,2,3,{4,4},5}
Euphoria Data Object After value(): {1,2,3,{4,4},5}

Character String: "Euphoria"
Euphoria Data Object After value(): {69,117,112,104,111,114,105,97}
```

You can also convert a Euphoria data object into a character string, or as part of a formatted character string by using `sprintf()`:

```
rs = sprintf(s,o)
```

`sprintf()` is similar to printf(), as it takes either an atom value or a sequence representing a list of values (o), which it then edits by using a format string (s). The format codes introduced in our discussion on `printf()` are used in `sprintf()` as well. The only differences between `printf()` and `sprintf()` is that `sprintf()` sends the formatted string to a receiving sequence variable (rs), and that sprintf() only requires two parameters in comparison to `printf()`'s three. A demo is available showing one possible use of `sprintf()`.

program 25

```
sequence print_line

for line = 1 to 5 do
    print_line = sprintf("This is line %03d\n", line)
    puts(1,print_line)
end for
```

```
This is line 001
This is line 002
This is line 003
This is line 004
This is line 005
```

Another way of changing the state of a data object is by altering the case of alphabetic characters. Euphoria has two library routines that can do this for you. To change any characters between 'A' and 'Z' in both atom and sequence data objects to lowercase, you use the lower() library routine:

use include std/text.e

```
include wildcard.e
ro = lower(o)
```

With lower(), a character string of "Euphoria" and "EUPHORIA" passed to this library routine as o would produce a string of "euphoria" that is then stored in the receiving variable ro. With single atom values like 'J' or 74, what is stored in the receiving variable is 'j' or 106.

The opposite of lower() is upper(), which coverts any characters in both atoms and sequences that are between 'a' and 'z' to uppercase:

use include std/text.e

```
include wildcard.e
ro = upper(o)
```

A character string of "Euphoria" and "euphoria" passed to upper(), as o would produce a string of "EUPHORIA," which is then stored in the receiving variable ro. With single atom values like 'z' or 122, upper() will store 'Z' or 90 in the receiving variable.

A demo program is available to show how upper() and lower() works.

program 26

use include std/text.e

```
include wildcard.e
sequence test_string, uppered, lowered

test_string = "This is a fun way to learn Euphoria!"
uppered = upper(test_string)
lowered = lower(test_string)
printf(1,"Original String     : %s\n",{test_string})
printf(1,"     After lower()  : %s\n",{lowered})
printf(1,"     After upper()  : %s\n",{uppered})
```

```
Original String     : This is a fun way to learn Euphoria!
     After lower()  : this is a fun way to learn euphoria!
     After upper()  : THIS IS A FUN WAY TO LEARN EUPHORIA!
```

Earlier in the tutorial, we introduced the & operator, which allows you to join atom or sequence data objects together to create new sequences. Euphoria offers other ways of linking data objects together, the first being append():

```
rs = append(s,o)
```

`append()` creates a new sequence, which is stored in variable rs, by adding o to the end of s as an entire element. So how does this differ from the & operator? Do we really need `append()` anyways?

When & joins any two or more data objects, it creates a sequence that is as long as the total number of elements made up by the participating data objects. However, `append()` will always create a new sequence that is one element longer than the length of s. The reason for this is because `append()` will make o the new last element of s.

When only atoms are involved in the joining, & and `append()`'s efforts produce the same result. The difference is only apparent when sequences are involved. A demo program is available to help you clarify the usage of `append()`, with mention being made to show how it differs from the & operator.

program 27

```
atom     atom1, atom2
sequence seq1, seq2, union

atom1 = 83
atom2 = 4
seq1 = {1,1,1,1}
seq2 = {2,2}

puts(1,"& And append()\n")
puts(1,"==============\n\n")

printf(1,"Atom Values: %d, %d\n",{atom1,atom2})
puts(1,"   Using &: ")

union = atom1 & atom2
print(1,union)
puts(1,"\n")

puts(1,"   Using append(): ")
union = {}
union = append(union,atom1)
union = append(union,atom2)
print(1,union)
puts(1,"\n\n")

puts(1,"Sequence Values: ")
print(1,seq1)
puts(1," ")
print(1,seq2)
puts(1,"\n")
puts(1,"   Using &: ")

union = seq1 & seq2
print(1,union)
puts(1,"\n")

puts(1,"   Using append(): ")
union = {}
union = append(seq1,seq2)
print(1,union)
puts(1,"\n\n")
```

```
& And append()
==============

Atom Values: 83, 4
   Using &: {83,4}
   Using append(): {83,4}

Sequence Values: {1,1,1,1} {2,2}
   Using &: {1,1,1,1,2,2}
   Using append(): {1,1,1,1,{2,2}
```

A related library routine to append() is prepend():

```
rs = prepend(s,o)
```

prepend() creates a new sequence, which is stored in variable rs, by adding o to the beginning of s as a single element. This means o becomes the new first element of s, or s[1] in Euphoria terms. Like append(), prepend() creates a new sequence that is one element longer than s. A demo program is ready to demonstrate prepend().

program 28

```
atom     atom1, atom2
sequence seq1, seq2, union

atom1 = 83
atom2 = 4
seq1 = {1,1,1,1}
seq2 = {2,2}

puts(1,"& And prepend()\n")
puts(1,"==============\n\n")

printf(1,"Atom Values: %d, %d\n",{atom1,atom2})
puts(1,"   Using &: ")

union = atom1 & atom2
print(1,union)
puts(1,"\n")

puts(1,"   Using prepend(): ")
union = {}
union = prepend(union,atom1)
union = prepend(union,atom2)
print(1,union)
puts(1,"\n\n")

puts(1,"Sequence Values: ")
print(1,seq1)
puts(1," ")
print(1,seq2)
puts(1,"\n")
puts(1,"   Using &: ")

union = seq1 & seq2
print(1,union)
puts(1,"\n")

puts(1,"   Using prepend(): ")
union = {}
union = prepend(seq1,seq2)
print(1,union)
```

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

```
& And prepend()
==============

Atom Values: 83, 4
   Using &: {83,4}
   Using prepend(): {4,83}

Sequence Values: {1,1,1,1} {2,2}
   Using &: {1,1,1,1,2,2}
   Using prepend(): {{2,2},1,1,1,1}
```

Both `append()` and `prepend()` are handy when you want to stack atoms and sequences in a queue, like candies in a "PEZ" dispenser. When `append()` or `prepend()` is used to attach a data object to a larger sequence, that data object retains its separate identity.

Sequences can also be created "on the fly" by repeating a value so many times. The `repeat()` library routine was created to perform just that:

```
rs = repeat(o,a)
```

`repeat()` creates a sequence value a elements long, where each element has the value of o. `repeat()` can create sequences that are made up of atom or sequence elements. These created sequences can be as long as you need them to be. A demo is available showing some sequences made by `repeat()`.

program 29

```
puts(1,"Some Sequences Created By repeat()\n\n")
puts(1,"repeat(20,10):\n   ")
print(1,repeat(20,10))
puts(1,"\n\n")
puts(1,"repeat({1,1,1,1,1},5):\n   ")
print(1,repeat({1,1,1,1,1},5))
puts(1,"\n\n")
puts(1,"repeat(\"Tim\",4):\n   ")
print(1,repeat("Tim",4))
puts(1,"\n")
```

```
Some Sequences Created By repeat()

repeat(20,10):
   {20,20,20,20,20,20,20,20,20,20}

repeat({1,1,1,1,1},5):
   {{1,1,1,1,1},{1,1,1,1,1},{1,1,1,1,1},{1,1,1,1,1},{1,1,1,1,1}}

repeat("Tim",4):
   {{84,105,109},{84,105,109},{84,105,109},{84,105,109}}
```

Once you've created your sequences, it's a good idea to know how large they are if you plan to examine one or many of the elements that make the sequence up. The `length()` library routine returns the length of a sequence in number of elements:

```
ri = length(s)
```

the length of an atom is 1 , new since oE

The length of the sequence, shown here as s, is placed in the receiving variable ri. Note that the returned value is how long sequence s is in elements, not atoms. This means the sequences `{{1,1},{1,1}}` and `{1,1}` are the same length in elements, even though the former contains more atoms. A sequence value of `{}`, which is a null sequence, returns a value of 0.

A demo program is available to show how `length()` can be used in a "for" statement to create a loop that adjusts to the size of a sequence.

program 30

```
sequence seq
seq = {{0,0,0},{65,{7,7,7},23.1},"Timmy"}
for element1 = 1 to 3 do
    puts(1,"Elements For Sequence ")
    print(1,seq[element1])
    puts(1,":\n")
    for element2 = 1 to length(seq[element1]) do
         printf(1,"   Element %d: ",element2)
         print(1,seq[element1][element2])
         puts(1,"\n")
    end for
    puts(1,"\n")
end for
```

```
Elements For Sequence {0,0,0}:
   Element 1: 0
   Element 2: 0
   Element 3: 0

Elements For Sequence {65,{7,7,7},23.1}:
   Element 1: 65
   Element 2: {7,7,7}
   Element 3: 23.1

Elements For Sequence {84,105,109,109,121}:
   Element 1: 84
   Element 2: 105
   Element 3: 109
   Element 4: 109
   Element 5: 121
```

The next chapter will introduce library routines that search sequences!

13. Advanced Data Object Handling, Part Two

As you build larger, more complex sequence data objects, you are going to need tools that can search them quickly. One way is to write some code using “if,” “while,” and “for” statements that compare each element against a specific value. A much easier way is to take advantage of Euphoria’s sequence search library routines. These library routines allow the programmer to find elements either using a specific value or a partial search string.

searching in oE is greatly expanded with new routines

```
ri = find(o,s)
```

`find()` searches sequence s for element o. If found, the element number of o is stored in receiving variable ri. If not, 0 is stored in ri.

`find()` starts searching a sequence from the first element onward. If there is more than one element with the search value present in the sequence, `find()` only returns the element number of the first one. If you want to continue searching after the first match, you will need to search a segment of the sequence, starting with the element following the first match.

A demo program demonstrates the use of `find()` and also how to match more than one element using `find()`:

program 31

```
atom found, more_finds, offset
sequence search_string

search_string = {34,5,106,72,65,5,90,17,5,13}
puts(1,"Searching Sequence ")
print(1,search_string)
puts(1," For 5\n\n")

offset = 0
more_finds = 'y'

while more_finds = 'y' do
     found = find(5,search_string)
     if found then
          offset = offset + found
          printf(1,"5 Found As Element %d\n",offset)
          search_string = search_string[found+1..length(search_string)]
     else
          more_finds = 'n'
     end if
end while

puts(1,"\nProgram completed\n")
```

```
Searching Sequence {34,5,106,72,65,5,90,17,5,13} For 5

5 Found As Element 2
5 Found As Element 6
5 Found As Element 9

Program completed
```

While `find()` finds a single element in a sequence, `match()` allows you to search a sequence for a specfic group of elements:

```
ri = match(s1,s2)
```

`match()` searches sequence s2 in order to find a sequence of elements, shown here as s1. If successful, receiving variable ri is assigned the element number in s2 where the first element of s1 is located.

What would each of the receiving variables contain after each `match()`?

```
atom element_id1, element_id2, element_id3

element_id1 = match( "Al" ,  "David Alan Gay" )
element_id2 = match({3,4,5}, {1,2,3,4,5,6,7,8,9,0})
element_id3 = match({ {50,23,4},{-1,-2} },
                    { {15,89},{50,23,4},{-1,-2} })
```

Here are the results:

```
 "David Alan Gay"
     -- element_id1 is assigned the value of 7

{1,2,3,4,5,6,7,8,9,0}
     -- element_id2 is assigned the value of 3

{ {15,89}, {50,23,4}, {-1,-2} }
    -- element_id3 is assigned the value of 2
```

If `match()` cannot find what it is looking for in the searched sequence, then 0 is returned. A demo program is available to further clarify `match()` if needed.

program 32

```
sequence nursery_rhyme
atom found
nurs-
ery_rhyme = "Jack and Jill went up the hill to fetch a pail of water"
```

```
puts(1,nursery_rhyme & "\n\n")
printf(1,"Searching String For \"%s\"\n",{"hill"})
found = match("hill",nursery_rhyme)
printf(1,"     Found \"%s\" beginning at element %d\n\n",{"hill",found})
printf(1,"Searching String For \"%s\"\n",{"l of water"})
found = match("l of water",nursery_rhyme)
printf(1,"     Found \"%s\" beginning at element %d\n\n",{"l of water",
                                                         found})
puts(1,  "Searching String For ")
print(1, {97,110,100})
puts(1,"\n")
found = match({97,110,100},nursery_rhyme)
puts(1,"     Found ")
print(1, {97,110,100})
printf(1," beginning at element %d\n\n",found)
```

```
Jack and Jill went up the hill to fetch a pail of water

Searching String For "hill"
     Found "hill" beginning at element 27

Searching String For "l of water"
     Found "l of water" beginning at element 46

Searching String For {97,110,100}
     Found {97,110,100} beginning at element 6
```

now has std/search.e with many more routines. There is also std/regex.e for regular expression searching

The previous library routines we introduced in this chapter used an actual value to match one or a specific group of elements in a sequence. The next library routine uses “wildcards” to search for elements in sequences. Before we introduce this library routine, we need to discuss what wildcards are.

Wildcards are single substitution characters used in conjunction with other characters to make a generic search string. You may have seen wildcards at work when you use the DIR command in DOS as follows:

* and ? are multiplatform file wildcards

`dir *.com` — lists all files ending with an extension of .COM

Euphoria has two wildcard characters that closely follow DOS’ version:

`*` — matches any 0 or more characters.

For example, a generic search string of “A*” will match values like “Apple”,“Acorn”, and “A”. A generic search string of “*e” would match values like “value”, “cue ”, and “e”.

`?` — matches any single character.

A generic search string of “g???” for example would match values like “game”, “gone”, and “goat”. A generic search string of “??t” would match values like “Cat”, “dot”, and “Hit”.

With wildcards now explained, we can introduce the `wildcard_match()` library routine.

use
include

```
include wildcard.e
```

```
ri = wildcard_match(s1,s2)
```

`wildcard_match()` checks if sequence s2 matches the wildcard pattern defined in sequence s1. A 1 is returned if s2 matches s1, otherwise a 0 is returned. The return value is stored in the receiving variable ri.

The use of matching sequence values with wildcard patterns is a little tricky. This is because `wildcard_match()` takes into consideration both alphabetic case and the placement of the wildcard characters in the pattern string. Look at the examples on the next page to see some common mistakes when using `wildcard_match()`.

```
atom match_1, match_2, match_3

match_1 = wildcard_match("a*", "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
match_2 = wildcard_match("?Z", "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
match_3 = wildcard_match("PQR*","ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

All three variables above will be assigned a value of 0. The first `wildcard_match()` line won't match because the pattern has a lowercase letter while the searched string is all in uppercase. The second `wildcard_match()` will not match because ? was used instead of *, even though "Z" is indeed the last letter of the searched string. "?Z" means any two character string, the second character being "Z". The last `wildcard_match()` line will not match because even though "PQR" does exist in the searched string, the pattern "PQR*" implies a match only if the searched string begins with "PQR".

If you want to search for a substring of characters anywhere in the target sequence, place a * wildcard on both sides of the substring. This will produce a match pattern of "*PQR*" for example. You can also mix wildcards in a match pattern, such as "A?c*1". This means `wildcard_match()` will only get a match on this pattern if the searched string's first character is "A", its third character is "c", and its last character is "1".

A demo program is available that allows you to experiment with pattern strings, to help you get a better understanding of `wildcard_match()`:

program 33

use
include
std/wildcard.e

```
include wildcard.e

sequence pattern_string, search_string
atom halt_program, matched

halt_program = 'n'

search_string = "wildcard_match() is a powerful feature of Euphoria."

while halt_program = 'n' do
     puts(1,search_string & "\n")
     puts(1,"Enter a wildcard pattern string or \"STOP\": ")
     pattern_string = gets(0)
     pattern_string = pattern_string[1..(length(pattern_string)-1)]
     if compare(pattern_string,"STOP") = 0 then
          halt_program = 'y'
     else
          matched = wildcard_match(pattern_string, search_string)
          puts(1,"\n")
          if matched then
               puts(1,pattern_string & " matches above string.\n\n")
          else
               puts(1,pattern_string & " does not match above string.\n\n")
          end if
     end if
end while
```

```
wildcard_match() is a powerful feature of Euphoria.
Enter a wildcard pattern string or "STOP": ?

? does not match above string.

wildcard_match() is a powerful feature of Euphoria.
Enter a wildcard pattern string or "STOP": !

! does not match above string.

wildcard_match() is a powerful feature of Euphoria.
Enter a wildcard pattern string or "STOP": *

* matches above string.

wildcard_match() is a powerful feature of Euphoria.
Enter a wildcard pattern string or "STOP": STOP
```

If you want to write your own sequence search programs, one important factor is the speed on finding the element you are searching for. To optimize your search, it is best to sort the sequence and then look up each element until you either get a match, or compare a value that is larger than the element you are looking for (which means not found).

To sort a sequence to be searched, you use the `sort` library routine.

use
include std/sort.e

```
include sort.e
rs = sort(s)
```

`sort()` sorts sequence s, but does not change s. The sorted sequence is instead stored in receiving variable rs. The sequence to be sorted can be made up of any combination of atoms and sequences. The result of the sort will be a sequence with its elements sorted in ascending order, with any atom values appearing first before sequence elements. Elements that are sequences are sorted based on an element by element comparison, starting with the first element onward. The comparison is based on the value of each element.

Run a demo program now that demonstrates how `sort()` works with sample data.

program 34

use
include std/sort.e

```
include sort.e
sequence sorted, unsorted

unsorted = {"world","the","Euphoria","rules"}
sorted = sort(unsorted)

puts(1,"Unsorted: ")
for words = 1 to length(unsorted) do
     puts(1,unsorted[words] & " ")
end for
puts(1,"\n")
puts(1,"Sorted: ")
for words = 1 to length(sorted) do
     puts(1,sorted[words] & " ")
end for
puts(1,"\n\n")

unsorted = {{1,1,8,2},5,{1,2,3},{1,1,9},-45,{1,2,1}}
sorted = sort(unsorted)
puts(1,"Unsorted: ")
print(1,unsorted)
puts(1,"\n")
puts(1,"Sorted: ")
print(1,sorted)
puts(1,"\n")
```

```
Unsorted: world the Euphoria rules
Sorted: Euphoria rules the world

Unsorted: {{1,1,8,2},5,{1,2,3},{1,1,9},-45,{1,2,1}}
Sorted: {-45,5,{1,1,8,2},{1,1,9},{1,2,1},{1,2,3}}
```

This concludes your introduction to advanced data object handling library routines. The next chapter will introduce library routines that can enhance any arithmetic computations you may have in your programs.

14. Advanced Arithmetic Library Routines

Euphoria offers a set of library routines that save the programmer from coding complex programs to handle advanced math formulas. These library routines can handle some algebra and trigonometry formulas, rounding down of numbers to the smallest whole number, and even the generation of random numbers. Note that some of these library routines require a strong background in certain areas of mathematics before you attempt to use them.

see std/math.e for an expanded math library

The first library routine we will introduce is `sqrt()`:

```
ro = sqrt(o)
```

`sqrt()` returns the square root of o, which is then stored in variable ro.

To refresh your memory, a square root of a number is any number that is multiplied by itself to produce a number. For example, the square root of 25 is 5, because 5 times 5 gives 25.

This library routine can accept both atom and sequence data objects. For sequence data objects passed to `sqrt()`, a sequence with the same length is generated, with each atom element a sequare root of the elements in the original sequence. For example, `{16,0.25}` passed to `sqrt()` returns a value of `{4,0.5}`. The one word of caution is not to pass a negative atom value or a sequence with any negative atom elements to `sqrt()`, or your program will encounter a runtime error message. A demo program is available showing how atoms and sequences are squared using `sqrt()`.

program 35

```
atom value1
sequence value2

value1 = 25
value2 = {81,{9,4},100}

puts(1, "The square root of ")
print(1,value1)
puts(1, " is ")
print(1,sqrt(value1))
puts(1,"\n")
puts(1, "The square root of ")
```

```
print(1,value2)
puts(1, " is ")
print(1,sqrt(value2))
puts(1,"\n")
```

```
The square root of 25 is 5
The square root of {81,{9,4},100} is {9,{3,2},10}
```

The opposite of square-rooting a number is to raise a number by the power of 2. Euphoria has a library routine that can raise a number to the power 2, or any power for that matter.

Here is the library routine used to raise a data object value by a given power:

```
ro = power(o1,o2)
```

`power()` raises o1 to the power of o2, the result being stored in receiving variable ro.

Because `power()` can use raise both atoms and sequence data objects to a power that in itself can also be either an atom or a sequence data object, conversion of values must first occur. If o1 is an atom and o2 is a sequence, Euphoria converts o1 to a sequence value that is the same length as o2, composed of elements that equal the value of o1. If o1 is a sequence and o2 is an atom, Euphoria converts o2 to a sequence that is the same length as o1, composed of elements equalling the value of o2. This concept was introduced in"Assigning Values To Sequence Variables."

Only after any needed conversion of atoms to sequence data objects is completed does `power()` execute. A demo is available to show how `power()` works with both atoms and sequences data objects.

program 36

```
atom base, result
sequence base2, result2

base = 2

for exponent = 1 to 4 do
     result = power(base,exponent)
     printf(1, "%d to the power of %d is %d\n",{base,exponent,result})
end for

puts(1,"\n")

for exponent = 1 to 4 do
    base2 = repeat(10,5)
    result2 = power(base2,exponent)
    print(1, base2)
    puts(1, " to the power of ")
    print(1,exponent)
    puts(1, " is ")
    print(1,result2)
    puts(1,"\n")
end for
```

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

base = 3
result2 = power(base,{2,3,4})
print(1, base)
puts(1, " to the power of ")
print(1,{2,3,4})
puts(1, " is ")
print(1,result2)
puts(1,"\n")
```

```
2 to the power of 1 is 2
2 to the power of 2 is 4
2 to the power of 3 is 8
2 to the power of 4 is 16

{10,10,10,10,10} to the power of 1 is {10,10,10,10,10}
{10,10,10,10,10} to the power of 2 is {100,100,100,100,100}
{10,10,10,10,10} to the power of 3 is {1000,1000,1000,1000,1000}
{10,10,10,10,10} to the power of 4 is {10000,10000,10000,10000,10000}

3 to the power of {2,3,4} is {9,27,81}
```

Another library routine related to `sqrt()` is `log()`:

```
ro = log(o)
```

`log()` returns the natural logarithm of a number, o. A logarithm is the exponent that raises a number, called the base, to a specific power. A natural logarithm is the exponent that raises a base number of 2.71828 (approximately) to a specific power. For example, log(27) will return a value of 3.29584. If you were to raise 2.71828 to the power of 3.29584 by using `power()`, you would get back a value of 26.9999. `log()` can handle both atoms and sequences in the same fashion `sqrt()` can, returning a value that is stored in a receiving variable, named ro here.

One note: do not pass a negative number or a zero to `log()`, or your program will encounter a runtime error. Natural logarithms are used in calculations like statistics and trigonometry, so it is unlikely you will use this library routine much, unless you are working in areas of mathematics that require natural logarithms. A demo program is available if you want to see some examples of how `log()` is used.

program 37

```
atom log1, base, result1
sequence log2, result2

base = 2.71828

log1 = log(63)
log2 = log({100,50,25})

result1= power(base,log1)
result2= power(base,log2)

printf(1,"The natural logarithm of 63 is %f\n",log1)
printf(1,"%.5f to the power of %f is %.0f (rounded up)\n\n",
         {base,log1,result1})

puts(1,"The natural logarithm of ")
print(1,{100,50,25})
puts(1," is ")
print(1,log2)
puts(1,"\n")

printf(1,"%.5f to the power of ",base)
print(1,log2)
puts(1," is ")
printf(1,"{%.0f,%.0f,%.0f} (rounded up)\n\n",result2)
```

```
The natural logarithm of 63 is 4.143135
2.71828 to the power of 4.143135 is 63 (rounded up)

The natural logarithm of {100,50,25} is {4.605170186,3.912023005,3.218875825}
2.71828 to the power of {4.605170186,3.912023005,3.218875825} is {100,50,25} (rounded up)
```

When one number does not evenly divide by another, have you wondered what the decimal part of the result means? Well, the numbers to the right of the decimal is what is left over, shown as a decimal fraction. For example, 3.5 returned from dividing 7 by 2 means “3 with 1 left over.” How did we get 1? Multiply .5 by the divisor 2 and you get 1. Sometimes this remainder of a division is just as important as the result, and usually it is preferable to have that remainder shown as a whole number. You could use the technique shown above to determine the remainder, or you can use this next Euphoria library routine instead.

```
ro = remainder(o1,o2)
```

`remainder()`returns the remainder left over from dividing o1 by o2, and stores it in the receiving variable ro. The value returned by `remainder()`is always less than the value of o2, the divisor.

Because `remainder()`can be either atoms or sequences for both the quotient and the divisor, Euphoria will convert an atom to a sequence that will match the length of the other sequence before `remainder()`is executed. The elements of this new other sequence will have the value of the original atom value. You may want to review `power()` or the chapter “Assigning Values To Sequence Variables” if you need any help in understanding this. A demo program is available that uses various examples of `remainder()`.

program 38

```
sequence format_string
atom leftovers, result
for-
mat_string = "10 divided by %d goes %d time(s) with %d left over.\n"
for divisor = 2 to 9 do
     leftovers = remainder(10,divisor)
     result = 10/divisor
     printf(1,format_string,{divisor,result,leftovers})
end for
```

```
10 divided by 2 goes 5 time(s) with 0 left over.
10 divided by 3 goes 3 time(s) with 1 left over.
10 divided by 4 goes 2 time(s) with 2 left over.
10 divided by 5 goes 2 time(s) with 0 left over.
10 divided by 6 goes 1 time(s) with 4 left over.
10 divided by 7 goes 1 time(s) with 3 left over.
10 divided by 8 goes 1 time(s) with 2 left over.
10 divided by 9 goes 1 time(s) with 1 left over.
```

If you are working with floating point numbers and want to convert that value to an integer, one way to do it is to use the `floor()` library routine:

```
ro = floor(o)
```

`floor()` takes any data object, o,and rounds it down to the nearest integer value. If o is a sequence, each atom element in that sequence is rounded down to the nearest integer value.

Understand that using `floor()` is not the same as chopping the decimal fraction part off a value. While examples like floor(64.55) or floor (32.1, 56.87, 2.044) would result in truncation (64 and 32, 56, 2), floor(-55.3) and floor(-1.3, -0.5) would return -56 and -2,-1, respectively. If `floor()` receives an integer, that same value is returned. A demo program is ready to show examples of `floor()`.

program 39

```
object value1, value2

value1 = 5.3
value2 = -5.3

printf(1,"%.1f floor()'d gives %d\n",{value1,floor(value1)})
printf(1,"%.1f floor()'d gives %d\n",{value2,floor(value2)})
puts(1,"\n")

value1 = {35.3,-46.1,22.9,-.7345}
print(1, value1)
puts(1," floor()'d gives ")
print(1, floor(value1))
puts(1,"\n")
```

```
5.3 floor()'d gives 5
-5.3 floor()}'d gives -6

{35.3,-46.1,22.9,-0.7345} floor()'d gives {35,-47,22,-1}
```

Euphoria has a library routine that may be of interest to those who want to write their own games. It involves the creation of random numbers, a key element in any game, whether it involves randomly generating a number to guess, or creating an unexpected malfunction aboard a spaceship during a critical moment in battle. Here is the syntax of the `rand()` library routine:

```
ro = rand(o)
```

`rand()` will return a randomly generated number. If o is an atom value, the number generated will be from 1 to o. If o is a sequence, the number generated will be from a sequence composed of 1's to o. A randomly generated sequence value will always have the same length as o. The largest integer value (either passed as a single atom or as one or more elements of a sequence) `rand()` will generate is 1,073,741,823. A demo program is available to show `rand()`'s use in a guess the number game.

program 40

use include std/console.e

```
include get.e

atom random_number,guessed_number, end_program
sequence input_data

puts(1,"Guess any number between 1 and 10\n")

end_program = 'n'
random_number = rand(10)

while end_program = 'n' do
     input_data = get(0)
     if input_data[1] = 0 then
          if integer(input_data[2]) then
               guessed_number = input_data[2]
               if guessed_number = random_number then
                    end_program = 'y'
                    puts(1,"\nBingo!\n")
               elsif guessed_number < random_number then
                    puts(1,"\nYou're too low! Try again!\n")
               elsif guessed_number > random_number then
                    puts(1,"\nYou're too high! Try again!\n")
               end if
          end if
     end if
end while
```

```
Guess any number between 1 and 10
4

You're too low! Try again!
8

You're too low! Try again!
9

Bingo!
```

Left on its own, `rand()` will generate a unpredictable random value every time it is called. However, some programs may need to repeat the same random values more than once. This is done by setting the random number generator's "seed" (a source value where all `rand()`-generated values are drawn from) to a certain value. You can do this by using the library routine `set_rand()`:

```
include machine.e
```

use include std/rand.e

```
set_rand(a)
```

If you execute `set_rand()` to set the random number generator seed to a, execute `rand()` three times to generate three different random numbers (let's pretend we generated 51, 67, and 2), then execute `set_rand()` again using the same value of a, the next three executions of `rand()` will generate the same values of 51, 67, and 2.

If you use the same value to set the random number generator seed, the following `rand()` executions will always produce the same random values, no matter how many times you re-run your program. A demo program is available showing how to generate the same set of random numbers based on a value from the keyboard:

program 41

use include std/console.e std/rand.e

```
include get.e
include machine.e

atom seed, end_program
sequence input_data, prompt, line1, line2, line3
line1 = "\nYou entered "
line2 = " as the value to set the random number generator seed to.\n"
line3 = "The next 10 numbers generated by rand(100) will always be:\n"
prompt = "\nEnter any value or 0 to end this program program\n"

end_program = 'n'

while end_program = 'n' do
     puts(1,prompt)
     input_data = get(0)
     if input_data[1] = 0 then
          if integer(input_data[2]) then
               seed = input_data[2]
               if seed != 0 then
                    puts(1,line1)
                    print(1,seed)
                    puts(1,line2)
                    set_rand(seed)
                    puts(1,line3)
                    for i = 1 to 10 do
                         print(1,rand(100))
                         puts(1, " ")
                    end for
                    puts(1,"\n")
               else
                    end_program = 'y'
               end if
          end if
     end if
end while
```

```
Enter any value or 0 to end this program program
7

You entered 7 as the value to set the random number generator seed to.
The next 10 numbers generated by rand(100) will always be:
42 89 53 69 55 13 2 97 58 100

Enter any value or 0 to end this program program
7

You entered 7 as the value to set the random number generator seed to.
The next 10 numbers generated by rand(100) will always be:
42 89 53 69 55 13 2 97 58 100

Enter any value or 0 to end this program program
0
```

To close this chapter, we will briefly introduce a series of library routines devoted to trigonometry:

`ro = sin(o)` —Returns the sine of an angle

`ro = cos(o)` —Returns the cosine of an angle

`ro = tan(o)` —Returns the tangent of an angle

`ro = arctan(o)` —Returns the angle of a tangent (opposite of tan())

The first three library routines accept an atom or a sequence value, o, that is an angle (in radians), and returns the appropriate value in variable o. The last accepts an atom or sequence value that represents a tangent and returns an angle atom or sequence, measured in radians.

These library routines come in handy for work in trigonometry, such as using the law of sines and the law of cosines to determine the length of a side of a triangle, or to generate a sine wave on a plane graph. Because trigonometry is a subject out of the scope of this tutorial, we will focus on them long enough just to let you know about them. If you are not familiar with trigonometry, these library routines will not be of any interest to you. They are not mandatory to know in order to learn how to program in Euphoria.

This concludes our introduction to math library routines in Euphoria. The next chapter will show you how to work with files and devices.

15. Opening Files And Devices In Euphoria

The screen and keyboard are not the only things Euphoria programs can access on your computer. You can write programs that read and change file data on your floppy drive or hard drive, or create new files as a form of output. You can even access devices on your computer such as the printer and modem. However, for the sake of brevity, this chapter will focus more on using files on your computer with a passing reference or two to some devices.

std/io.e which has many routines for reading and writing files

A Euphoria program accesses a file or device for use by requesting the operating system to check if that file or device is free. If so, a buffer that will be used to hold data between the program and the file or device is created, and a number is returned to the program. This number is used by the program to reference the file or device. Every file accessed by a Euphoria program is assigned a unique number, so multiple files and devices can be handled without any confusion. Each file or device is also given its own data buffer. When a program outputs data to a file or device, it really goes to the buffer assigned to that file or device. When it is full, only then does it get sent to the file or device.

Any files or devices being used by a Euphoria program remains allocated until either the program stops running, or the program informs the operating system that it is finished with the device or file. When this occurs, any data that was wrtten to the buffer is sent to the file or device, and the file and device is then free for use by another program.

Let's begin the process of accessing files and devices by first showing how a Euphoria program allocates, or "opens," a file or device for use.

```
ri = open(s1,s2)
```

`open()` requests the operating system to allocate a file or device (s1) to be used in a certain mode (s2). If allowed, the number to reference the file or device by is stored in ri. If the file or device cannot be allocated, -1 is stored in ri instead.

Accessing a file or device in a particular mode (s2) involves two parts. The first part is how you want to open the file or device. If you want to open a file or device only as a source of data for your program, and do not plan to send data back to it, then you want to open this file or device in "read" mode. This means the program can only accept data from a file or device, not send to it. In the case of a file, this means you cannot change the data inside the file. A file or device must exist in order to open it in "read."

If your intention is to treat a file or device as a source of output, and do not intend to accept any data from it, then you want to open this file or device in "write" mode. This means the program can only send data to the file or device, not read from it. In the case of a file, opening in "write" mode will destroy any data stored in the file. If you wish to preserve the existing data in the file when outputting data to it, you can open the file in "append" mode, a variation of the"write" mode. "append" mode allows output data to be tagged at the end of the existing file data. While a device must exist in order to use "write" and "append," a file is created if it does not exist.

If you want to work with devices and files as both a source of data input and a place to send data as output, then you want to open in an "update" mode. This means programs can both send and receive data from files and devices. A file or device must exist in order to open it in "update" mode.

The second part of opening in a certain mode is how the data is to be treated. If you are treating the data as something to be used in text editors, then you want the data to be handled in text mode. Data outputted in text mode has carriage return codes (13 or "`\r`") added before any linefeed codes (10 or "`\n`"). Data inputted in text mode will have carriage return codes automatically removed. The ASCII value of 26 in text mode means the end of data to be read in the file.

If the data you are handling is more along the lines of digital pictures or compressed archive files like .ZIP, then you should handle the data in binary mode. This means the data is not altered in any way, and all ASCII values from 0 to 255 can be read or written unconditionally.

By combining 4 open type and 2 data handling modes together, we produce the following 8 modes on the next page.

	Read Data	Write Data	Append Data	Update Data
Treat data as text	r	w	a	u
Treat data as binary	rb	wb	ab	ub

Euphoria supports Windows 95's long file name format when using `open()` on any existing file in any of the modes listed above. However, if you try to open a new file using modes "wb," "w," "ab," or "a" under Windows 95, the name of the new file will be truncated to DOS's 8.3 format (an eight character filename, then a period, followed by a three character extension) if it is lengthy.

oE works with Windows and Unix filesystems.

The tutorial for the most part will use DOS's 8.3 format in fairness to all users with different operating systems, unless otherwise necessary.

To open an existing file in the current directory you are in for reading, and to handle file data as text, you would type:

```
integer file_id
file_id = open( "text.doc" , "r" )
```

To open an existing file in another directory on drive C: for appending data to the end of the file, and to treat the data as binary, you type:

file_id = open(c:/binary/database.bin , ab)

```
integer file_id
file_id = open( "c:\\binary\\database.bin" , "ab" )
```

Notice you have to use two slashes instead of one when defining the directory path. This is because \ is also a special character prefix.

Modern operating systems use / as the directory path separator. Newer versions of Windows allow you to use either. When / is used, you no longer have to double-up on the separator.

To create a new file in a different directory on drive F: for writing output data, and to handle data as text, you would type:

```
integer file_id
file_id = open( " f:\\output\\write.dta " , " w " )
```

To open an existing file on the C: drive for updating (using long file names) under the Windows95 operating system, and handling the data as text, you would type:

```
integer file_id
file_id = open("c:\\EuphoriaFiles\\tutorialfile.textfile","u")
```

You can use the `open()` library routine to open devices for use. The six allowed device names are listed below:

- CON —Console (screen)
- AUX —Auxiliary serial port
- COM1 —Serial port 1
- COM2 —Serial port 2
- PRN —Parallel port printer
- NUL —Non-existent device that discards accepted output

We will not place too much emphasis on working with devices in this tutorial, but here is how to open the printer, a device you'll use often:

```
integer file_id

file_id = open( "PRN" , "w" )
```

Notice that accessing the printer uses a mode of write text data. This makes sense, as printers cannot send data, and all printer line output (in text mode) is terminated with carriage and line-feed codes.

You must know the specific printer control codes of your printer. Otherwise you get just garbage.

After going through so much reading just to learn how to open a file or device for your program to use, you probably assume that the process of passing data between your program and the devices and files on your system is just as complex. That assumption couldn't be further from the truth. As a matter of fact, you have already learned about the library routines that handle data transfer between the program and any files or devices it works with. The next chapter will explain this in greater detail.

16. File And Device Data Handling

You'll recall earlier that Euphoria has a set of library routines that allow a program to accept data from the keyboard and send data to the screen. If the screen and keyboard can be used to access external data, this means they are just like files and other devices. If this is the case, then it would be better to have the library routines for screen and keyboard able to work with other devices and files, instead of designing a new set of library routines that does the same thing.

It makes sense. If Euphoria automatically assigns the number 0 for keyboard, and 1 and 2 for the screen, then replacing these numbers with a value returned by the `open()` library routine would have the input and output library routines handle data from other sources. This means you already know how to pass data between the program and other files and devices.

Let's expand this a little further, to help those that do not understand.

If i is a value returned by `open()`, then the following library routines can be used to send data to a file or device opened for data output:

- `print(i,o)` — sends a Euphoria value to a file or device
- `puts(i,o)` — sends an atom or a sequence as a character string to a file or device
- `printf(i,s,o)` — sends an atom or a sequence as part of an edited string to a file or device

In short, all that was done was to replace 1 (screen) with the value returned by `open()`. The library routine still works the same way.

The same can also be done for library routines that handle input. If i is a value returned by `open()`, then the following library routines can be used to receive data from a file or device opened for data input:

use include std/console.e

```
include get.e
```

`rs = get(i)` — retrieve a Euphoria data object from a file or device, and store as a two element sequence

`ro = gets(o)` — retrieve a character string from a file or device up to and including the `'\n'` code, or -1 if no data is available

`wait_key()` and `get_key()` only works with the keyboard. However, Euphoria has a counterpart for `get_key()`, called `getc()`:

`ri = getc(i)`

`getc()` retrieves a single byte from a file or device, defined as i, and stores the byte value into receiving variable ri. i is generated by the `open()` library routine when it successfully opens a file or device for input. If there is no data available to read, a value of -1 is returned.

While screen and keyboard come from distinctly separate sources (meaning screen and keyboard data are not sharable as a single source), it is possible, in the case of files, to retrieve information previously stored by your program or by another Euphoria program. For example, data stored in a file by a previous `print()` can be retrieved later by using `get()`. Data sent to a file by `puts()` or `printf()` can be retrieved as a string in one read (`gets()`) or character by character (`getc()`).

When a Euphoria program is finished using a file or device it can release (or close) it, so another program can use it, without having to stop running in order to do this.

`close(i)`

`close()` closes a file or device, defined as i by `open()`. This will send any data still in the buffer to the file or device being closed.

use include std/console.e

To help you put this all together, a demo program will show how to open a file or device, send and receive data between files and devices, and close a file when not needed any more.

program 42

```
include get.e
sequence input_data
integer file_id, byte

puts(1,"File And Device I/O Demo Program\n")
puts(1,"=================================\n\n")
puts(1,"Creating a new file called demo.fle on your system......\n\n")
file_id = open("demo.fle","w")
if file_id != -1 then
     puts(1,"Successfully created file demo.fle on your system!\n\n")
     puts(1,"Writing a character string of \"Euphoria\" in demo.fle\n")
     puts(1,"using puts().....\n\n")
     puts(file_id,"Euphoria")
     puts(1,"Done....closing file demo.fle\n\n")
     close(file_id)
end if
puts(1,"Press any key to continue......\n\n")

while get_key() = -1 do
end while

puts(1,"-------------------------------------------------------------\n")
puts(1,"Opening demo.fle on your system for reading......\n\n")
file_id = open("demo.fle","r")
if file_id != -1 then
     puts(1,"Successfully opened file demo.fle on your system!\n\n")
     puts(1,"Read character string from demo.fle\n")
     puts(1,"in one shot using gets().....\n\n")
     input_data = gets(file_id)
     if sequence(input_data) then
          printf(1,"The string read in is: %s\n\n", {input_data})
     else
          puts(1,"Error reading data from file!\n")
     end if
     puts(1,"Done....closing file demo.fle\n\n")
     close(file_id)
end if
puts(1,"Press any key to continue......\n\n")

while get_key() = -1 do
end while

puts(1,"-------------------------------------------------------------\n")
puts(1,"Opening demo.fle on your system for reading......\n\n")
file_id = open("demo.fle","rb")
if file_id != -1 then
     puts(1,"Successfully opened file demo.fle on your system!\n\n")
     puts(1,"Read character string from demo.fle\n")
     puts(1,"one character at a time using getc().....\n\n")
     byte = getc(file_id)
     puts(1, "The character string read in is: ")
     while byte != -1 do
          puts(1,byte)
          byte = getc(file_id)
     end while
     puts(1,"\n\nDone....closing file demo.fle\n\n")
     close(file_id)
end if
puts(1,"Press any key to continue......\n\n")

while get_key() = -1 do
end while

puts(1,"-------------------------------------------------------------\n")
puts(1,"Opening demo.fle on your system to clear data......\n\n")
file_id = open("demo.fle","w")
if file_id != -1 then
     puts(1,"Successfully cleared file demo.fle!\n\n")
     puts(1,"Writing a Euphoria data object of ")
     print(1,-36.5)
     puts(1," using print().....\n\n")
     print(file_id,-36.5)
     puts(1,"Done....closing file demo.fle\n\n")
     close(file_id)
end if
puts(1,"Press any key to continue......\n\n")

while get_key() = -1 do
end while

puts(1,"------------------------------------------------------------\n")
puts(1,"Opening demo.fle on your system for reading......\n\n")
file_id = open("demo.fle","r")
if file_id != -1 then
     puts(1,"Successfully opened file demo.fle on your system!\n\n")
     puts(1,"Read a Euphoria data object from demo.fle\n")
     puts(1,"using get().....\n\n")
     input_data = get(file_id)
     if input_data[1] = 0 then
          printf(1,"The value read in is: %.1f\n\n", {input_data[2]})
     else
          puts(1,"Error reading data from file!\n")
     end if
     puts(1,"Done....closing file demo.fle\n\n")
     close(file_id)
end if
puts(1,"Press any key to continue......\n\n")

while get_key() = -1 do
end while

puts(1,"------------------------------------------------------------\n")
puts(1,"Opening printer PRN......\n\n")
file_id = open("LPT1","w")
if file_id != -1 then
     puts(1,"Successfully opened printer for use!\n\n")
     puts(1,"Printing a line on your printer using puts()\n")
     puts(1,"Press ’y’ to print or ’n’ to skip\n\n")
     byte = get_key()
     while byte != ’n’ do
          if byte = ’y’ then
               puts(file_id,"*************************\n")
               puts(file_id,"* Euphoria in hardcopy! *\n")
               puts(file_id,"*************************\n")
          end if
          byte = get_key()
     end while
     puts(1,"Done....closing PRN\n\n")
     close(file_id)
end if
```

```
File And Device I/O Demo Program
================================

Creating a new file called demo.fle on your system......

Successfully created file demo.fle on your system!

Writing a character string of "Euphoria" in demo.fle
using puts().....

Done....closing file demo.fle

Press any key to continue......
```

```
-----------------------------------------------------------------
Opening demo.fle on your system for reading......

Successfully opened file demo.fle on your system!

Read character string from demo.fle
in one shot using gets().....

The string read in is: Euphoria

Done....closing file demo.fle

Press any key to continue......
-----------------------------------------------------------------
Opening demo.fle on your system for reading......

Successfully opened file demo.fle on your system!

Read character string from demo.fle
one character at a time using getc().....

The character string read in is: Euphoria

Done....closing file demo.fle

Press any key to continue......
-----------------------------------------------------------------
Opening demo.fle on your system to clear data......

Successfully cleared file demo.fle!

Writing a Euphoria data object of -36.5 using print().....

Done....closing file demo.fle

Press any key to continue......
-----------------------------------------------------------------
Opening demo.fle on your system for reading......

Successfully opened file demo.fle on your system!

Read a Euphoria data object from demo.fle
using get().....

The value read in is: -36.5

Done....closing file demo.fle

Press any key to continue......
-----------------------------------------------------------------
Opening printer PRN......

Successfully opened printer for use!

Printing a line on your printer using puts()
Press ’y’ to print or ’n’ to skip

Done....closing PRN
```

Whenever you open a file for use, there is a bookmark that determines where at the byte position the next read or write will occur in the file. When a file is opened for read, write and update (whether in text or binary handling), the byte position is 0, meaning the start of the file. For files open for append, the byte position is the last byte of the file. Any data output will change the byte position value.

It's possible for a Euphoria programmer to control where the next read or write will occur in the file by setting the current byte position to a new location. This is done by using the library routine `seek()`:

```
include file.e
```

```
ri = seek(i1,i2)
```

`seek()` sets the next read or write in file i1 (returned by `open()`) to byte position i2. i2 is the number of bytes from the first byte in the file. For example, `seek(0)` would have the next read or write occur at the first byte. seek(2999) would have the next read or write occur at the 30,000th byte. A value of 0 is returned to receiving variable ri if `seek()` successfully changes the current byte position. If unsuccessful, `seek()` returns a non-zero value.

You can `seek()` to a position past the actual end of the file. If this happens, extra byte values of 0 will be added to the end of the file, making it long enough to match the byte position you are `seek()`-ing to.

A demo program is available to show how `seek()` is used to change specific byte locations in a file:

program 43

```
include file.e
sequence string, seek_positions, vowels
object input_line
atom file_id, status

vowels = "aeiou"

string =
"The beauty of the seek() library routine is that you can control\n" &
"where the next write or read will occur. This will allow you to\n" &
"update any old information in your file with new data. This eliminates\n" &
"the need to maintain different versions of the same data in the\n" &
"file.\n\n"

seek_positions = {}
for element = 1 to length(string) do
     if find(string[element],vowels) then
          seek_positions = append(seek_positions,{element, string[element]})
           string[element] = ' '
     end if
end for
file_id = open("demo.fle","wb")
puts(file_id,string)
close(file_id)
```

```
puts(1,"The paragraph below has been written to a file named demo.fle,\n")
puts(1,"after the vowels were removed and stored in a sequence. This\n")
puts(1,"demo will use seek() to return the vowels back to the paragraph\n")
puts(1,"in the file\n\n")
puts(1,string)
file_id = open("demo.fle","ub")
for element = 1 to length(seek_positions) do
     status = seek(file_id,seek_positions[element][1]-1)
     if status = 0 then
          puts(file_id,seek_positions[element][2])
     end if
end for
close(file_id)
file_id = open("demo.fle","rb")
input_line = gets(file_id)
puts(1,"Adding vowels now.....\n\n")
while compare(input_line,-1) != 0 do
     puts(1,input_line)
     input_line = gets(file_id)
end while
```

```
The paragraph below has been written to a file named demo.fle,
after the vowels were removed and stored in a sequence. This
demo will use seek() to return the vowels back to the paragraph
in the file

Th  b   ty  f th  s  k() l br ry r  t n   s th t y   c n c ntr l
wh r  th  n xt wr t   r r  d w ll  cc r. Th s w ll  ll w y   t
 pd t   ny  ld  nf rm t  n  n y  r f l  w th n w d t . Th s  l m n t s
th  n  d t  m  nt  n d ff r nt v rs  ns  f th  s m  d t   n th
f l .

Adding vowels now.....

The beauty of the seek() library routine is that you can control
where the next write or read will occur. This will allow you to
update any old information in your file with new data. This eliminates
the need to maintain different versions of the same data in the
file.
```

If you want to know the current byte position in the file, you can find out using the `where()` library routine:

```
include file.e
```

```
ri = where(i)
```

`where()` is best used when you want to find out the current byte position where the next read or write will occur in file i. i is a value returned by the `open()` statement. A demo program is available showing `where()` in use, returning the current byte position at the time of opening a file, and after a few writes have been made to the file.

program 44

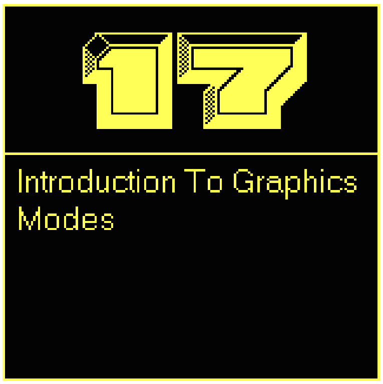
```
include file.e
sequence list_of_words, input_string
integer file_id, current_location

list_of_words = {"Euphoria ","rocks"}
```

```
puts(1,"This demo program will show how the current byte position is\n")
puts(1,"updated with every I/O made to a file, courtesy of the where()\n")
puts(1,"library routine. Remember that the current byte position is\n")
puts(1,"the number of bytes from the first byte in the file!\n\n")

file_id = open("demo.fle","wb")

current_location = where(file_id)
printf(1,"Opening file in write mode, the current byte position is %d\n\n",
          current_location)

for word = 1 to length(list_of_words) do
     printf(1,"Writing \"%s\" to file....\n",{list_of_words[word]})
     puts(file_id,list_of_words[word])
     current_location = where(file_id)
     if word < 2 then
          printf(1,"The next write or read will occur at %d\n\n",
                   current_location)
     else
          puts(1,"Closing file\n\n")
     end if
end for

close(file_id)
file_id = open("demo.fle","ab")

current_location = where(file_id)
printf(1,"Opening file in append mode, the current byte position is %d\n\n",
          current_location)
printf(1,"Writing \"%s\" to file....\n",'!')
     puts(file_id,"!")
puts(1,"Closing file\n\n")

close(file_id)

puts(1,"Opening file in read mode now...\n\n")
file_id = open("demo.fle","rb")
input_string = gets(file_id)
printf(1,"The file contains the following string: %s\n",{input_string})

close(file_id)
```

```
This demo program will show how the current byte position is
updated with every I/O made to a file, courtesy of the where()
library routine. Remember that the current byte position is
the number of bytes from the first byte in the file!

Opening file in write mode, the current byte position is 0

Writing "Euphoria " to file....
The next write or read will occur at 9

Writing "rocks" to file....
Closing file

Opening file in append mode, the current byte position is 14

Writing "!" to file....
Closing file

Opening file in read mode now...

The file contains the following string: Euphoria rocks!
```

The next few chapters will make learning Euphoria more fun, by writing programs that generate and manipulate colourful text and graphics.

17. Introduction To Graphics Modes



Multimedia is the use of graphics, text, and sound by your program to interact with the person running it. It's a popular trend in both the business and home entertainment software industry. Euphoria's language set has a wealth of library routines that can allow you to handle powerful text and graphic output. Do not be intimidated by the idea of handling graphics in your programs. Euphoria remains true to its reputation of being an easy-to-understand language, even with graphics!

Before you are introduced to library routines that add colour and style to your screen output, you need to know how your computer screen works.

When your computer screen shows information, it is presenting the data using a display format, otherwise known as graphics mode.

Text mode is still the same. Use a GUI package for pixel based graphics.

There are two types of graphics modes, a text mode and a pixel-graphics mode. Text mode is the default setting of your computer screen when your system is started up. It allows the display of characters at specific screen locations by using a paired co-ordinate system of row first, and column second. The default number of columns is 80 characters across, and the default number of rows is 25 rows down, though these can be changed to the needs of the program. Combining a row and a column position produces a screen location to start displaying text at. For example, the top left corner of the screen in text mode is 1,1 (row 1, column 1). The bottom left corner of the screen in text mode is 25,80 (row 25, column 80).

Characters in text mode can be displayed with colour as well. The colour the text is displayed in is called the foreground colour. The area around each character that is not being used can also have a colour different from the foreground colour. This is called the background colour.

The default foreground colour for the text being displayed in Euphoria is white, and the default background colour black, in text mode.

Pixel-graphics mode also allows text to be displayed, and in a foreground colour. It also allows displaying what is known as a pixel. Pixels are tiny dots that can be used to assemble a graphics image. This tutorial program for example uses pixels to assemble the graphic images of the remote, the index screen, and the grey tiling around the viewport. This graphics mode also uses a paired co-ordinate system to display a pixel on the screen, but with two differences. First, the address pair to display a pixel is reversed in comparison to text screen addresses. Pixels are displayed using a pixel column first, pixel row last, address pair. Also, pixel row and column locations start at 0, not 1. This means the top left pixel corner is 0,0 (column 0, row 0). Because pixels are very small, the row and column positions can number into the hundreds, or in the case of high-resolution screens, the thousands.

But what is a high resolution? Well, a screen resolution is the number of pixels available on the screen. But how does one increase the number of pixels on a screen when the computer monitor itself remains the same size? This is accomplished by compressing the size of the pixel in each different screen resolution. For example, a high-resolution screen would have the pixels very small in order to fit a lot of them on one monitor. While this results in displaying images with very fine detail, the image itself would appear smaller than expected. On the other hand, a pixel-graphics mode in low resolution would use pixels that are a little larger, seeing that there would be fewer pixels available on the screen. This would make graphic images appear larger in size, but the quality of the picture would be more coarse and grainy. High-definition television screens for example use pixels that are much smaller than in conventional television sets, so the picture quality is much better.

Pixels, like text, can be displayed in a specific colour, so they also have a foreground colour. But the similarities end when discussing background colour in pixel-graphics modes. Any screen area not being occupied by a pixel is considered the background, so setting the background colour affects the entire screen area. As a result, any text shown in pixel graphics mode does not have a background colour. Text in this case is treated more like a group of pixels organized into a graphics shape than actual characters.

Each text and pixel graphics mode comes with a set of colours. this set of colours is called the palette. It works much like a painter's palette used in the creation of art. The painter's palette holds a number of tins of paint, each tin holding a different paint colour. If a new colour not on the palette is needed, one tin is switched with another tin containing the desired colour. The graphic mode palette works the same way, as each colour on the palette is identified by a number.

Each palette colour is created by mixing three elemental colours together. These three colours are red, blue, and green. By varying the intensities of each of the three colours differently, a new colour can be produced. The colours of the graphics mode palette are initially set to default levels of red, green, and blue, but as mentioned they can be changed to make a new colour. Because red, green and blue is used to generate the colours on a palette, each colour is said to have an RGB intensity level.

While the computer monitor can only show one screen display at a time, it is possible to maintain more than one virtual screen, called a screen page, in certain graphics modes. This means you can send screen output (using puts() for example) to one of many screen pages your Euphoria program is juggling. The screen page where all screen output is sent is called the active page. The screen page currently being shown on the computer monitor is called the display page.

The active page and the display page do not have to be the same screen page. For example, you can be sending screen output to one page while displaying a different page on the monitor. The number of screen pages available depends on the amount of memory on your video card in your computer.

Before any text or pixel-graphics images can be shown on the screen, the appropriate graphics mode must be selected. To do this, you use the graphics_mode() library routine:

```
include graphics.e
ri = graphics_mode(i)
```

graphics_mode() sets the screen to the appropriate graphics mode, an integer value of i. If successful, a value of 0 is returned to the receiving variable ri. If unsuccessful, a value of 1 is returned instead.

This is to set a DOS video mode.

Each graphics mode is identified as a unique number, and is either a text mode or a pixel graphics mode. Within each of these two groups are variations on the number of pixels or characters available for each graphics mode. As of version 1.5, here are the accepted mode numbers you can pass to graphics_mode(), with a description for each:

```
   0 = 40x25 text, 16 grey
   1 = 40x25 text, 16/8 color
   2 = 80x25 text, 16 grey
   3 = 80x25 text, 16/8 color
   4 = 320x200 pixels, 4 color
   5 = 320x200 pixels, 4 grey
   6 = 640x200 pixels, BW
   7 = 80x25 text, BW
   11 = 720x350 pixels, BW
   13 = 320x200 pixels, 16 color
   14 = 640x200 pixels, 16 color
   15 = 640x350 pixels, BW
   16 = 640x350 pixels, 4 or 16 color
   17 = 640x480 pixels, BW
   18 = 640x480 pixels, 16 color
   19 = 320x200 pixels, 256 color
   256 = 640x400 pixels, 256 color
   257 = 640x480 pixels, 256 color
   258 = 800x600 pixels, 16 color
   259 = 800x600 pixels, 256 color
   260 = 1024x768 pixels, 16 color
   261 = 1024x768 pixels, 256 color
```

Not all video cards will support all of these modes. The older the video card, the least number of the listed modes that will be available to you.

To switch back to the previous video mode you were using, pass a value of -1 to graphics_mode() as a parameter. You should monitor for the return code generated by graphics_mode() to make sure the graphics mode was changed successfully before attempting any text or graphics mode operations. A demo program is available to list the graphics modes available on your computer. Do not be alarmed at the screen flickering or cracking while it runs.

program 45

```
include graphics.e
sequence screen_modes, good_modes, bad_modes
atom counter
integer screen_set_status
screen_modes = {0,1,2,3,4,5,6,7,11,13,14,15,16,17,18,19,256,257,258,259,260,
261,262,263,-999}
counter = 1
good_modes = {}
bad_modes = {}
while screen_modes[counter] != -999 do
     screen_set_status = graphics_mode(screen_modes[counter])
     if screen_set_status = 0 then
          good_modes = append(good_modes,screen_modes[counter])
     else
          bad_modes = append(bad_modes, screen_modes[counter])
     end if
     counter = counter + 1
end while
screen_set_status = graphics_mode(-1)
puts(1, "The modes that can be used for your video card are:\n")
print(1, good_modes)
puts(1, "\n\nThe modes that cannot be used for your video card are:\n")
```

```
print(1, bad_modes)
```

```
The modes that can be used for your video card are:
{0,1,2,3,4,5,6,7,11,13,14,15,16,17,18,19,256,257,258,259,260,261,262,263}

The modes that cannot be used for your video card are:
{}
```

This example was run an a Unix system.

On a Unix system the answer is that it is not possible to set a video mode.

Whatever graphics mode you are in, it's important to know what you can and cannot do while in this graphics mode, because going over the limits defined in the graphics mode could lead to a program error.

std/graphics.e, video_config() still available

This library routine will give you information about the graphics mode you are in:

```
include graphics.e
```

```
rs = video_config()
```

When executed, video_config() will return a sequence value made up of eight atom elements, which is stored in receiving variable rs. Each element represents an attribute about the graphics mode. The structure is as follows:

```
{colour monitor (1 means yes, 0 means no),
 graphics mode number,
 number of text rows,
 number of text columns,
 number of pixels across,
 number of pixels down,
 number of colours supported,
 number of pages}
```

Pixel count will always be zero—there is no pixel graphics mode in oE

If the value of number of pixels across and down are both zero, you are in a text mode. Only pixel-graphics modes can support the use of pixels. The next demo program will not only tell you which graphics modes your system can support, it will give you the details about each.

program 46

```
include graphics.e
sequence screen_modes, video_settings
atom counter
integer screen_set_status
screen_modes = {0,1,2,3,4,5,6,7,11,13,14,15,16,17,18,19,256,257,258,259,260,
261,262,263,-999}
counter = 1
while screen_modes[counter] != -999 do
     screen_set_status = graphics_mode(screen_modes[counter])
     if screen_set_status = 0 then
          video_settings = video_config()
          printf(1, "Mode %d supports the following attributes:\n\n",
               {screen_modes[counter]})
          if video_settings[1] = 1 then
               puts(1, "     Has Colour\n")
```

```
            else
                puts(1, "     Has No Colour\n")
            end if
            printf(1, "     Has %d text rows and %d text columns\n",
                {video_settings[3], video_settings[4]})
            if video_settings[5] + video_settings[6] > 0 then
               printf(1, "     Has %d pixels across and %d pixels down\n",
                    {video_settings[5], video_settings[6]})
            end if
           printf(1, "     Has %d colours available\n", {video_settings[7]})
            printf(1, "     Has %d display pages accessible\n\n",
                {video_settings[8]})
            puts(1, "Press Any Key To Continue\n")
            while get_key() = -1 do
            end while
        end if
        counter = counter + 1
end while
if graphics_mode(-1) then
end if
```

Tested on a Unix system. All 'video modes' produce the same answer—only a text screen is available.

```
Mode 0 supports the following attributes:

     Has Colour
     Has 24 text rows and 80 text columns
     Has 16 colours available
     Has 1 display pages accessible

Press Any Key To Continue
Mode 1 supports the following attributes:

     Has Colour
     Has 24 text rows and 80 text columns
     Has 16 colours available
     Has 1 display pages accessible

Press Any Key To Continue
```

Whenever you switch to any new graphics mode, the screen automatically clears any data off the screen. However, you may want to clear the screen without having to reset the graphics mode:

```
clear_screen()
```

`clear_screen()` clears the screen using the current background colour. Later in the tutorial, you will learn how to set the background colour in both text and pixel graphics mode. `clear_screen()` works in any graphics mode. Because this is a relatively straightforward routine to use, no demo is required.

Now that you understand how to set the screen graphics mode to your liking, and are able to obtain information about the selected mode, let's move on to putting text mode displays to good use. Once you have finished reading the next chapter, you'll never consider text output as dull again.

18. Colouring And Animating Text

Most of the demos you have seen involved presenting text on the screen. While it is very easy to create a formatted text line using `printf()`, it's entirely another matter to have text, formatted or otherwise, to appear anywhere on the screen. The special characters '\t' and '\n', and generous usage of the spacebar isn't enough to present text in professional form, such as in columns, horizontally centered, or even in colours other than white on black. This chapter will change all that.

oE is text mode — use a GUI library for modern graphics and windows.

`position(i1,i2)`

`position()` moves the screen cursor to any row (i1) and column (i2) location where you want the next screen print (by puts(), print(), or printf(), for example) to appear. For example, issuing position(15,40) will move the cursor to the 15th row and the 40th column from the top-left corner of the screen (row 1, column 1). Any attempt to go off the screen will result in a program error. A demo program is available to show one humourous use of `position()` in displaying text.

program 47

```
clear_screen()
position(3,15)
puts(1, "A Program Example To Demonstrate Text Positioning")
position(4,30)
puts(1, "Written By David Gay")
position(5,20)
puts(1, "Author, \"A Beginner's Guide To Euphoria II\"")
position(8,1)
puts(1, "Top Ten Reasons why you should purchase Euphoria:")
position(9,1)
puts(1, "==============================================")
position(11, 5)
puts(1, "Number 10: It's (thankfully) not a Microsoft product.")
position(12, 5)
puts(1, "Number 9: Because C is like tax laws: too complex to figure out.")
position(13, 5)
puts(1, "Number 8: Euphoria is more fun than this year's prime time TV season.")
position(14, 5)
puts(1, "Number 7: The money spent will go to BASIC's retirement home.")
position(15, 5)
puts(1, "Number 6: Because \"Ernest Learns Euphoria\" hits the theatres soon.")
```

```
position(16, 5)
puts(1, "Number 5: You sound very smart when you say you work with atoms.")
position(17, 5)
puts(1, "Number 4: You can write word games you already know the answers to.")
position(18, 5)
puts(1, "Number 3: You can declare things without cross border shopping.")
position(19, 5)
puts(1, "Number 2: At last! A reason to use the { and } keys on the keyboard!")
position(20, 5)
puts(1, "Number 1: It’s a great product!!!!! :)\n\n")
```

```
               A Program Example To Demonstrate Text Positioning
                              Written By David Gay
                    Author,"A Beginner’s Guide To Euphoria II"


Top Ten Reasons why you should purchase Euphoria:
==============================================

 Number 10: It’s (thankfully) not a Microsoft product.
 Number 9: Because C is like tax laws: too complex to figure out.
 Number 8: Euphoria is more fun than this year’s prime time TV season.
 Number 7: The money spent will go to BASIC’s retirement home.
 Number 6: Because"Ernest Learns Euphoria"hits the theatres soon.
 Number 5: You sound very smart when you say you work with atoms.
 Number 4: You can write word games you already know the answers to.
 Number 3: You can declare things without cross border shopping.
 Number 2: At last! A reason to use the { and } keys on the keyboard!
 Number 1: It’s a great product!!!!! :)
```

If you need to know where the cursor is on the screen, you can receive the cursor location by using the library routine get_position():

oE is textmode only

```
include graphics.e
rs = get_position()
```

get_position() returns a two-element long string, representing the current cursor position, to the receiving variable rs.

The sequence value returned by get_position() is composed of two atoms, the first atom element is the current row the cursor is on, the second, the current column the cursor is on. The format of the sequence value is:

```
{current row position, current column position}
```

The current cursor position is always updated whenever a puts(), print(), or printf() library routine sends any text output to the screen. A demo program is available to show how the current cursor position changes whenever text print is sent to the screen and even when position() is used.

program 48

```
include graphics.e
integer element, keystroke, update
sequence some_text, current_position
```

```
clear_screen()
element = 1
update = 'y'
some_text = "As the screen is continually updated, the cursor position\n" &
            "is updated automatically. get_position() will report the\n" &
             "current position of the cursor.\n\n"
position(1,1)
puts(1,"An example of get_position()")
position(2,1)
puts(1,"============================")
position(25,1)
puts(1,"Press Q to quit, or any other key to display a character")
position(5,1)
while element <= length(some_text) do
     if update = 'y' then
          update = 'n'
          puts(1,some_text[element])
          current_position = get_position()
          position(23,1)
          printf(1,"Cursor at row %2d, column %2d",current_position)
          position(current_position[1],current_position[2])
     end if
     keystroke = get_key()
     if keystroke = 'q' then
          element = length(some_text) + 1
     elsif keystroke != -1 then
          element = element + 1
          update = 'y'
     end if
end while
```

```
An example of get_position()
============================


As the screen is continually updated, the cursor position
is updated automatically. get_position() will report the
current position of the cursor.



Cursor at row  9, column  1
Press Q to quit, or any other key to display a character
```

If text sent to the screen is longer than the number of columns per row, it is wrapped at the right margin of the screen to continue on the next row below. There may be times where you want any text that goes over the right margin to be simply truncated and therefore not seen.

Euphoria has a library routine that allows you to choose whether to wrap long text strings that go past the right margin or to truncate:

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

If parameter i is equal to 1, any printed text that would go past the right margin would be wrapped and appear on the start of the next row below. If parameter i is 0, any printed text that would go past the right margin would be truncated. `wrap()` works in both text and pixel-graphics mode. A demo program shows how `wrap()` works based on the parameter passed to it.

program 49

```
include graphics.e
sequence some_string
some_string = "This is a text string that is longer than " &
              "the width of your screen. See how wrap() handles the " &
              "output of the text string when printed. "
for modes = 1 to 0 by -1 do
     wrap(modes)
     printf(1,"When wrap(%d) is used:\n\n", modes)
     puts(1,some_string & "\n\n")
end for
```

```
When wrap(1) is used:

This is a text string that is longer than the width of your screen. See how wrap()
handles the output of the text string when printed.

When wrap(0) is used:

This is a text string that is longer than the width of your screen. See how wrap
```

The standard 25 rows in most graphics modes is more than enough for displaying full-screen text. But if you need to show a lot of text data on the screen, having more rows per screen would help.

For Unix systems set the terminal window, and then call Euphoria. This is the only time you can set the number of rows and columns.

On Windows systems, the number of rows can be changed.

For text modes only, Euphoria has a library routine that can change the number of rows that can be displayed on the screen:

```
include graphics.e
ri = text_rows(i)
```

i represents the number of text rows you want the screen to show. The accepted values are 25, 28, 43, and 50. If possible, `text_rows()` will change the number of rows on the text mode you are in. You will notice the rows will appear flattened. Regardless of whether or not the number of rows on your screen has been changed successfully, the number of rows on the screen will be returned to the receiving variable ri.

Watch how the demo program assigned to this screen will change the number of rows in a text mode.

On a Unix system if you resize a terminal window (from a menu or with a mouse) *before* starting Euphoria, then Euphoria will recognize and use the available rows and columns.

program 50

```
include graphics.e
integer current_rows
sequence text_length
```

```
puts(1, "The following will demonstrate how to use text_rows() to\n")
puts(1, "increase the number of rows available in this program. Note\n")
puts(1, "how the spacing between the lines narrows. For those\n")
puts(1, "text rows that are not supported, this program will simply skip\n")
puts(1, "them. You will be required to press any key in order for this\n")
puts(1, "program to continue. Press any key to start the demonstration\n")

while get_key() = -1 do
end while

text_length = {25,28,43,50}
for ix = 1 to 4 do
     current_rows = text_rows(text_length[ix])
     if current_rows = text_length[ix] then
          for iy = 1 to text_length[ix] do
               print(1, iy)
               if iy < text_length[ix] then
                    puts(1, "\n")
               end if
          end for
          while get_key() = -1 do
          end while
     end if
end for
current_rows = text_rows(25)
```

On a Unix system, nothing happens, lines can not be changed from within a progrogram.

```
The following will demonstrate how to use text_rows() to
increase the number of rows available in this program. Note
how the spacing between the lines narrows. For those
text rows that are not supported, this program will simply skip
them. You will be required to press any key in order for this
program to continue. Press any key to start the demonstration
```

Another way to handle large amounts of text on the screen is by scrolling. Scrolling involves treating the screen as a movable window to view any part of the text. It's like riding in a glass elevator: as you go up or down, the view from your position will appear to vertically roll. Wordprocessors use this approach to handle documents larger than the size of the screen.

While you could write some program code to perform text scrolling, Euphoria has a powerful library routine that can do this for you:

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

`scroll()` makes a segment of screen text, starting from row i2 to row i3 inclusive, roll by i1 rows. The roll is towards the top (roll up) if i1 is negative, or towards the bottom (roll down) if i1 is positive.

If the text scrolls towards the top of the screen, extra blank rows will appear starting at the bottom row of the scroll area. If the text scrolls towards the bottom of the screen, extra blank rows will appear starting at the top row of the scroll area.

`scroll()` works in both text and pixel graphics modes. A demo program is available showing one example of how `scroll()` works.

program 51

The default code page and font (determining which characters are in output) is different for Windows and Unix systems.

```
include graphics.e

integer keystroke, current_position

clear_screen()

position(5,1)
puts(1, repeat('Í',80))

position(20,1)
puts(1, repeat('Í',80))

position(11,25)
puts(1, "ÉÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ»")

position(12,25)
puts(1, "º Welcome To The SCROLL ZONE º")

position(13,25)
puts(1, "ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼")

position(22,1)
puts(1, "Press 8 for up, 2 for down, or q to quit this program")

keystroke = get_key()
current_position = 11

while keystroke != 'q' do
     if keystroke = '8' and current_position > 6 then
          scroll(1,6,19)
          current_position = current_position - 1
     end if
     if keystroke = '2' and current_position < 17 then
          scroll(-1,6,19)
          current_position = current_position + 1
     end if

     keystroke = get_key()
end while
```

```
integer keystroke, current_position
clear_screen()
position(5,1)
puts(1, repeat('Ã‡',80))
position(20,1)
puts(1, repeat('Ã‡',80))
position(11,25)
puts(1,"Â–Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Â»")
position(12,25)
puts(1,"Âº Welcome To The SCROLL ZONE Âº")
position(13,25)
puts(1,"ÃflÃ‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Å´")
position(22,1)
puts(1,"Press 8 for up, 2 for down, or q to quit this program")
keystroke = get_key()
current_position = 11
while keystroke != 'q' do
     if keystroke = '8' and current_position > 6 then
          scroll(1,6,19)
          current_position = current_position - 1
     end if
     if keystroke = '2' and current_position < 17 then
          scroll(-1,6,19)
          current_position = current_position + 1
     end if
     keystroke = get_key()
end while
```

```
Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡

     Ã–Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Â»
    Ã– Welcome To The SCROLL ZONE Âº
     ÃflÃ‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Å´

Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡Ã‡

Press 8 for up, 2 for down, or q to quit this program
```

Typically a Windows console will output characters that make it easy to draw boxes on the screen.

For a Unix system, you have to explicitly set the font and codepage to one to provide these drawing characters.

The keycodes reported by Windows and Unix are different—same computer and same keyboard. For this reason this demo program will not work on a Unix computer without rewriting the keycode logic.

Perhaps the best way to present text is in colour. This tutorial, for example, uses colour to separate Euphoria statements and specific points apart from the regular text. There is a library routine you can use to present text in a certain colour.

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

`text_color()` causes any following screen text to be printed in the colour i. Any text already on the screen will not have its foreground colour changed. To start printing separate text in a new colour, you have to issue `text_color()` again. i is a number representing a colour in the palette of the text or pixel-graphic mode you are in. While some numbers may mean different colours in different graphic modes, the first 16 colours are as follows:

```
   1 = blue
   2 = green
   3 = cyan
   4 = red
   5 = magenta
   6 = brown
   7 = white
   8 = gray
   9 = bright blue
   10 = bright green
   11 = bright cyan
   12 = bright red
   13 = bright magenta
   14 = yellow
   15 = bright white

   (note: 0 = black)
```

When your program terminates, any screen text after will appear in the colour set by the last `text_color()` issued. As a result your program should issue a `text_color(7)` (white) followed by a puts(" \ n ") before it completes. text_color() works in any text or pixel-graphics modes. A demo program is available to show one use of `text_color()`.

program 52

```
include graphics.e
integer foreground_colour
sequence list_of_colours, already_used

clear_screen()
```

```
list_of_colours = {"blue","green","cyan","red","magenta","brown",
                   "white","gray","bright blue","bright green",
                   "bright cyan","bright red","bright magenta",
                   "yellow","bright white"}

already_used = {}

for ix = 1 to 10 do
     foreground_colour = rand(15)
     while find(foreground_colour,already_used) do
          foreground_colour = rand(15)
     end while
     already_used = already_used & foreground_colour
     text_color(foreground_colour)
     printf(1, "Text in %s\n",{list_of_colours[foreground_colour]})
end for
```

On a Unix terminal, the actual colors displayed can depend on the terminal being used and its color settings.

```
Text in red
Text in bright green
Text in blue
Text in bright blue
Text in green
Text in gray
Text in bright red
Text in bright white
Text in white
Text in yellow
```

To set the background colour of any text printed in a text mode, or the entire background of a screen in pixel graphics mode, you use the following library routine:

```
include graphics.e
```

```
bk_color(i)
```

In text mode, issuing `bk_color()` will cause the next screen text to be printed in the background colour i. Any text already on the screen will not have its background colour changed.

In pixel-graphics mode, the entire screen background will be changed instantaneously to the new colour i. This differs from text mode where only the background of any printed text following bk_color() is affected. In addition, any parts of graphic images on the screen that are in black (0) will show the background colour underneath when it is changed to any non-black colour. A demo program is available showing the differences between text and pixel-graphics modes where bk_colour() is concerned.

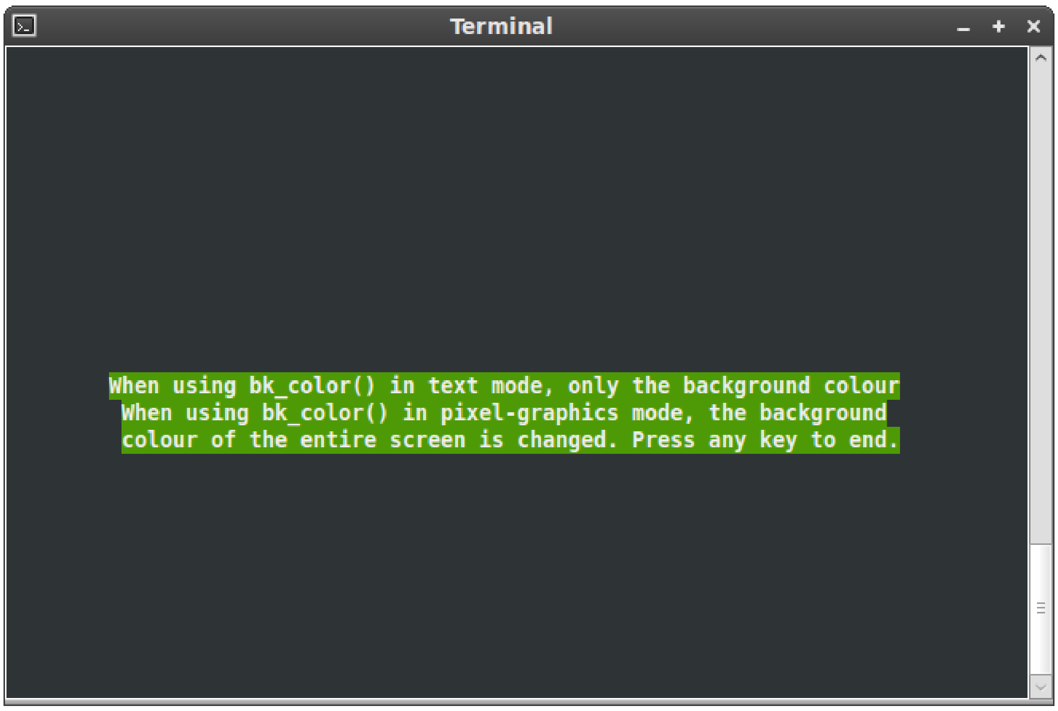
program 53

```
include graphics.e
clear_screen()
text_color(15)
bk_color(2)
position(13,9)
puts(1,"When using bk_color() in text mode, only the background colour")
position(14,10)
puts(1,"of the text itself is affected. Press any key to continue...")
while get_key() = -1 do
```

```
end while
if graphics_mode(18) = 0 then
     text_color(15)
     bk_color(2)
     position(14,10)
     puts(1,"When using bk_color() in pixel-graphics mode, the background")
     position(15,10)
     puts(1,"colour of the entire screen is changed. Press any key to end.")
     while get_key() = -1 do
     end while
     if graphics_mode(-1) then
          puts(1,"Resetting to the previous screen does not work for you!")
     end if
end if
```

```
  When using bk_color() in text mode, only the background colour
  When using bk_color() in pixel-graphics mode, the background
  colour of the entire screen is changed. Press any key to end.
```

oE text mode colors work. But, effects needing pixel graphics are not available.



```
When using bk_color() in text mode, only the background colour
 When using bk_color() in pixel-graphics mode, the background
 colour of the entire screen is changed. Press any key to end.
```

Rather than have text sit motionless on the screen, you can make it more interesting to look at through animation, as in moving groups of text horizontally and vertically around the screen. To perform this trick of movement, you must be able to grab text right off the screen, then re-display it at a new location. Euphoria has two library routines that can do this for you.

```
include image.e
```

```
rs = save_text_image(s1,s2)
```

save_text_image() captures a rectangular area of text on your screen, storing the saved data in receiving variable rs. The rectangular area is created by specifying the top left corner (s1) and the bottom right corner (s2). s1 is a sequence made up of two atom elements, the first element being the top left row of the screen, the second being the top left column of the screen, so the format of s1 is:

```
{top left row, top left column}
```

s2 is also a sequence made up of two atoms, the first being the bottom right row, the second being the bottom right column:

```
{bottom right row, bottom right column}
```

The saved data in rs is a sequence made up of sequence elements. Each sequence element represents a row of text in the rectangular area of the screen that was saved. The sequence elements themselves are made up of atom values. The odd-numbered elements are the actual text characters, and the even-numbered elements are values representing the combined foreground and background colours of each text character. So saving a rectangular area containing the following text:

```
Welcome to
Euphora!
```

would retrieve the value:

```
{
{87,13,101,13,108,13,99,13,111,13,109,13,101,13,32,13,84,13,111,13},
{69,13,117,13,112,13,104,13,111,13,114,13,105,13,97,13,33,13,32,13}
}
```

The foreground/background colour value always FOLLOWS the text character it is associated with. This value, called the attribute byte, is created by multiplying the background number by 16, then adding the foreground colour number to the result. In the example above, the text foreground was 13 or bright magenta with a background of 0 or black, so the attribute byte was 0 times 16, giving 0, then we added 13 to the result, giving 13.

Now that you have saved the image, you want to redisplay it in a new location on the screen. The following library routine below will do this:

```
include image.e
```

```
display_text_image(s1,s2)
```

`display_text_image()` displays s2, a sequence of sequences containing text and colour attributes, on the screen at location s1.

s1 is the top left corner on the screen where you would display the sequence s2, so the format of s1 would be:

```
{top left row, top left column}
```

s2 would very likely be the saved data obtained by the `save_text_image()` library routine. You could, however, create your own sequences to be displayed by `display_text_image()` using the structure shown to you earlier. Both `save_text_image()` and `display_text_image()` only work in text_modes. A demo program is available to show how `save_text_image()` and `display_text_image()` moves a block of text around the screen.

Under Unix, because the keycodes differ, this demonstration is ineffective.

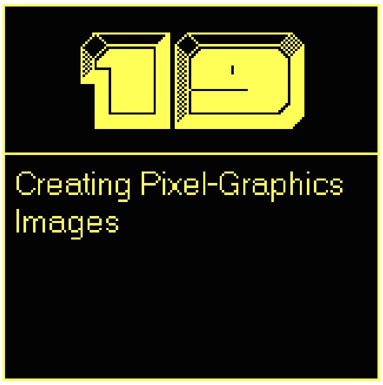
program 54

```
include image.e
sequence stored_image
integer input_key, line, column
line = 1
column = 1
bk_color(2)
clear_screen()
text_color(14)
position(2,2)
puts(1, "******************************")
position(3,2)
puts(1, "* Use the left, right, up,   *")
position(4,2)
puts(1, "* and down arrow keys to     *")
position(5,2)
puts(1, "* move box, press Q to quit! *")
position(6,2)
puts(1, "******************************")
stored_image = save_text_image({1,1},{7,32})
input_key = get_key()
while input_key != 'q'do
     display_text_image({line, column},stored_image)
     if input_key = 328 and line > 1 then
          line = line - 1
     end if
     if input_key = 336 then
          line = line + 1
     end if
     if input_key = 331 and column > 1 then
          column = column - 1
     end if
     if input_key = 333 then
          column = column + 1
     end if
     input_key = get_key()
end while
bk_color(0)
text_color(7)
```

```
  ******************************
  * Use the left, right, up,   *
  * and down arrow keys to     *
  * move box, press Q to quit! *
  ******************************
```

The next chapter promises to be more entertaining, as it shows you how to work with pixels and even graphic shapes in a pixel graphics mode.

19. Creating Pixel Graphics Images



Personal computers have undergone major changes since they were introduced in the early 1970's. The most significant change is the way the person works with the computer. Before the creation of Windows and Mac OS, you had to enter cryptic commands on a prompt. Today, we operate our computers using a graphics-based menu. Graphics are present in any software written these days because people relate to images easier than actual system commands.

Pixel graphics apply to DOS and Eu3 only.

Now you would open a WINDOW in a GUI and draw on a canvas—the ideas are similar to those described here.

The smallest element to work with in pixel-graphics modes is the pixel. Pixels can be placed anywhere on the screen, and can appear in any of the colours supported in the graphics mode you are in. Euphoria has a library routine that can set one or more pixels to any colour.

```
include graphics.e
pixel(o,s)
```

pixel() sets either one or a series of pixels starting at a position on the screen to a colour. s represents a sequence value, made up of two atom elements, that is the screen location of the pixel:

```
{pixel column location, pixel row location}
```

if o is an atom value, this value represents the colour number to set one pixel to at position s. If o is a sequence, this value is a series of colour numbers to set a group of pixels to, beginning at s, and then advancing one pixel to the right onward. Let's look at some examples of pixel on the next page.

```
pixel(6,{200,100})
```

This will set a pixel located at pixel column 200 and pixel row 100 to colour number 6, which is brown.

```
pixel({9,2,14},{63,120})
```

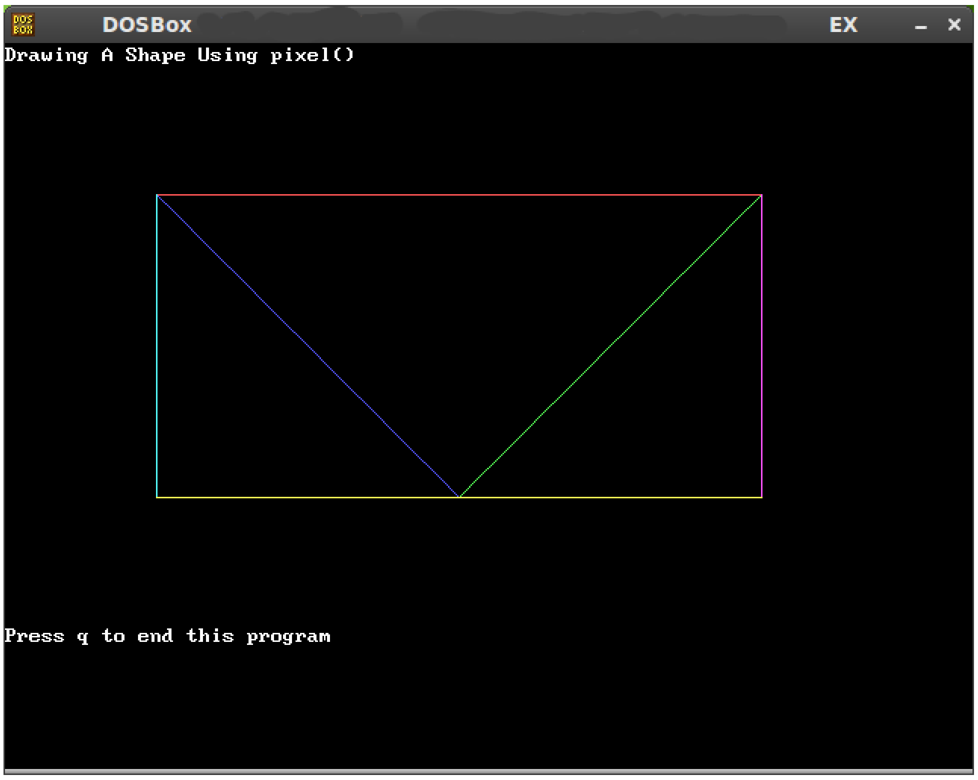
This will:

- set a pixel located at pixel column 63 and pixel row 120 to 9, which is grey
- set a pixel located at pixel column 64 and pixel row 120 to 2, which is green.
- set a pixel located at pixel colomn 65 and pixel row 120 to 14, which is yellow.

If you are working with a series of pixels on the same pixel row,it is faster to issue pixel() once with a series of colours than to set them individually one at a time. pixel() works only in pixel-graphics modes. A demo is available showing how pixel() is used to create a shape on the screen.

program 55.ex

```
include graphics.e
integer x_or_y_work_variable
if graphics_mode(18) then
     puts(1, "640 X 480 mode, 16 colours, not supported!\n")
else
     text_color(15)
     position(1,1)
     puts(1,"Drawing A Shape Using pixel()")
     position(25,1)
     puts(1, "Press q to end this program")
     for ix = 100 to 300 do
          pixel(9, {ix , ix})
     end for
     pixel(repeat(12,400),{100,100})
     x_or_y_work_variable = 300
     for ix = 300 to 500 do
          pixel(10, (ix & x_or_y_work_variable))
          x_or_y_work_variable = x_or_y_work_variable - 1
     end for
     for ix = 100 to 300 do
          pixel(11, {100, ix})
          pixel(13, {500, ix})
     end for
     for ix = 100 to 500 do
          pixel(14, {ix, 300})
     end for
     while get_key() != ’q’ do
     end while
     if graphics_mode(-1) then
          puts(1, "Unable To Reset\n")
     end if
end if
```



```
Drawing A Shape Using pixel()

Press q to end this program
```

You can find out what colour one or a series of pixels on the screen is set to by using the get_pixel() library routine:

```
include graphics.e
ro = get_pixel(s)
```

If s is in the format:

```
{pixel column location, pixel row location}
```

then the colour of the pixel at that location is returned, and is stored in receiving variable ro. If s is in the format:

```
{pixel column location, pixel row location, number of pixels}
```

then a series of colours is returned as a sequence, and stored in receiving variable "ro."

This sequence is all the colours of the pixels starting at that pixel column and pixel row location, and continuing right for the specified number of pixels.

```
one_colour = get_pixel({167,231})
```

This will return the colour of one pixel located at pixel column 167 and pixel row 231. The value is stored in variable "one_colour."

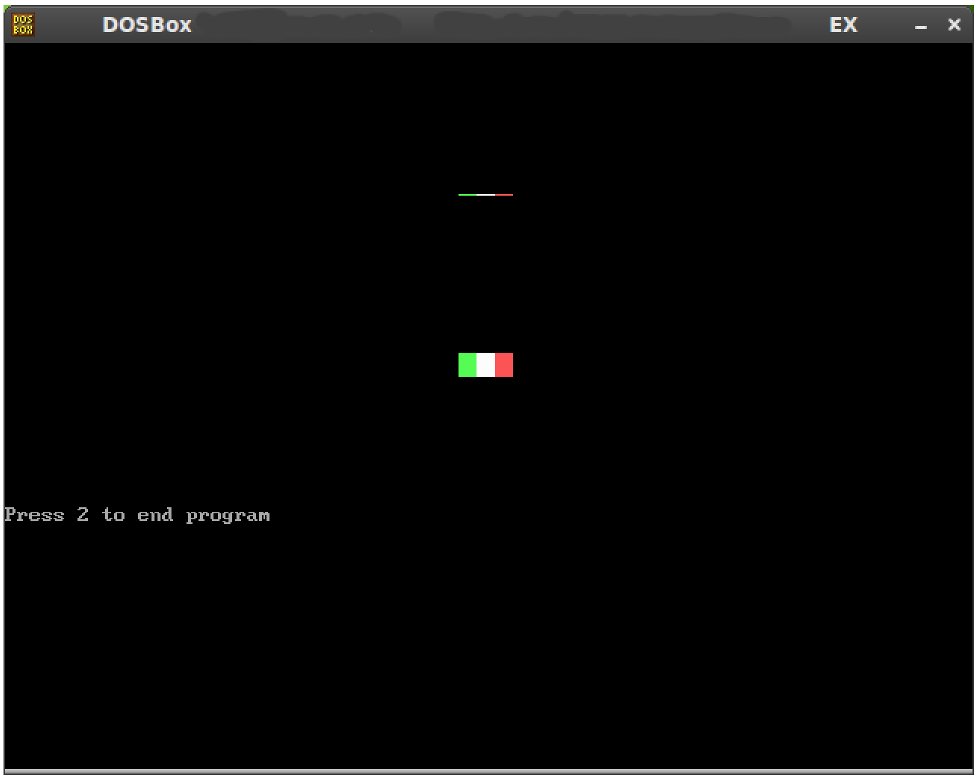
```
series_of_colours = get_pixel({1500,10,15})
```

This will return the colours of 15 pixels, beginning at pixel column 500 and pixel row 10, and stopping at pixel column 514 and pixel row 10. The sequence is stored in variable "series_of_colours."

get_pixel() only works in pixel-graphics modes. Do not attempt to obtain a pixel colour at a location that is off the screen. A demo program is available showing how to grab a series of pixels and then redisplay them several times to make a shape.

program 56

```
include graphics.e
sequence copy_buffer
if graphics_mode(18) then
     puts(1, "Mode 18 not supported\n")
else
     pixel({10,10,10,10,10,10,10,10,10,10,10,10,15,15,15,15,15,15,15,15,
            15,15,15,15,12,12,12,12,12,12,12,12,12,12,12,12},{300,100})
     position(20,1)

     puts(1, "Press 1 to continue")

     while get_key() != '1' do
     end while

     copy_buffer = get_pixel({300,100,36})

     for ix = 205 to 220 do
          pixel(copy_buffer,{300,ix})
     end for

     position(20,1)

     puts(1, "Press 2 to end program")

     while get_key() != '2' do
     end while

     if graphics_mode(-1) then
          puts(1, "Reset failed!\n")
     end if
end if
```



```
Press 2 to end program
```

Remember the demo program that created a shape using pixel()? While this worked well, it was awkward using pixel() to create even this simple shape. Euphoria has a library routine that can draw a line of pixels on the screen using a series of pixel screen locations:

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

i represents the colour number to draw a line of pixels in. s is a sequence made up of two or more sequence elements, where each element is a pixel location on the screen.

Here's the structure of the sequence used by draw_line() to make a line:

```
{{pixel column location, pixel row location},
 {pixel column location, pixel row location},
 {pixel column location, pixel row location},...}
```

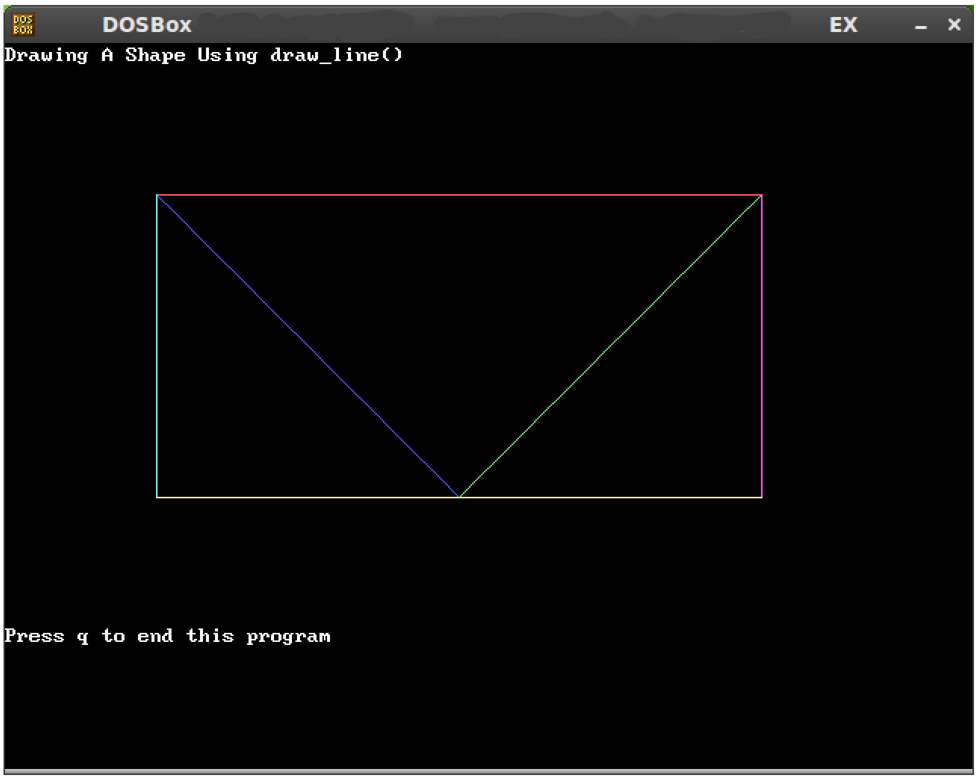
draw_line() draws the line starting with the first two elements. If there is a third pixel location as an element, the line continues to be drawn using the second and third elements. If there is a fourth pixel location as an element, then the line continues to be drawn using the third and fourth elements. This goes on until the entire sequence is processed. Here's how draw_line() draws a line that closes in on itself:

```
draw_line(14,{{50,50},{150,50},{150,150},{50,150},{50,50}})
```

A demo is ready to show how to draw a shape using draw_line():

program 57

```
include graphics.e
if graphics_mode(18) then
     puts(1, "640 X 480 mode, 16 colours, not supported!\n")
else
     text_color(15)
     position(1,1)
     puts(1, "Drawing A Shape Using draw_line()")
     position(25,1)
     puts(1, "Press q to end this program")
     draw_line(9,{{100,100},{300,300}})
     draw_line(12,{{100,100},{500,100}})
     draw_line(10,{{300,300},{500,100}})
     draw_line(11,{{100,100},{100,300}})
     draw_line(13,{{500,100},{500,300}})
     draw_line(14,{{100,300},{500,300}})
     while get_key() != 'q' do
     end while
     if graphics_mode(-1) then
          puts(1, "Unable To Reset\n")
     end if
end if
```



```
Drawing A Shape Using draw_line()

Press q to end this program
```

If you really want to draw a shape, Euphoria has a library routine that draws a polygon. A polygon is any 2-D plane figure with more than 4 sides, though with this library routine you can draw squares and triangles too:

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

Like draw_line(), polygon() uses the same pixel location format, s, to draw a shape on the screen, with the shape being drawn in a colour, i1. polygon() , however, uses a “fill” parameter (i2). Fill simply means to colour in the area bordered by the line that forms the shape. If it is 1, the area will be filled in using the same colour as the line. If it is 0, the area will be left empty. This way, you can control whether you have a wireframe shape or a solid shape.

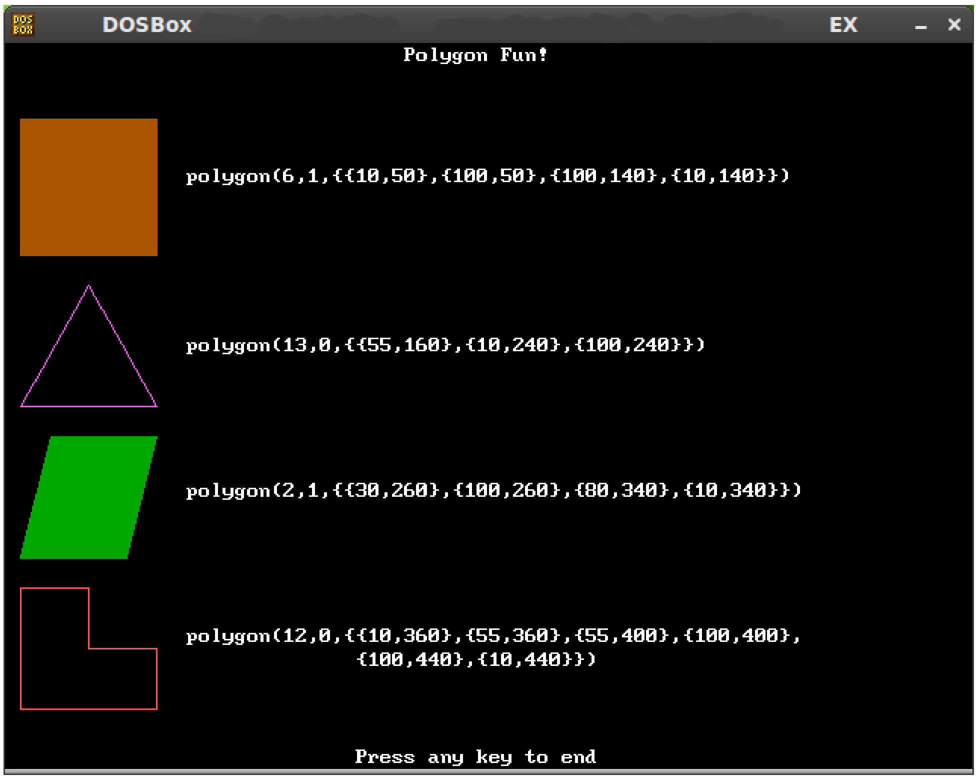
Also, to complete the shape of the polygon, you do not have to specify a pixel location to close the shape. polygon() automatically draws a line using the last element of s to the first element of s as pixel locations.

The order of the elements representing pixel locations in sequence s is important, as it dictates the shape of the polygon. when creating a shape using polygon(), you should order the elements in a circular pattern, unless your intention is to create a bizarrely-formed shape.

polygon() only works in pixel-graphics modes. A demo program is available that demonstrates some examples of shapes drawn by polygon().

program 58

```
include graphics.e
include image.e

if graphics_mode(18) = 0 then
     text_color(15)
     position(1,34)
     puts(1,"Polygon Fun!")
     position(30,30)
     puts(1,"Press any key to end")

     position(6,16)
     puts(1,"polygon(6,1,{{10,50},{100,50},{100,140},{10,140}})")
     polygon(6,1,{{10,50},{100,50},{100,140},{10,140}})

     position(13,16)
     puts(1,"polygon(13,0,{{55,160},{10,240},{100,240}})")
     polygon(13,0,{{55,160},{10,240},{100,240}})

     position(19,16)
     puts(1,"polygon(2,1,{{30,260},{100,260},{80,340},{10,340}})")
     polygon(2,1,{{30,260},{100,260},{80,340},{10,340}})

     position(25,16)
     puts(1,"polygon(12,0,{{10,360},{55,360},{55,400},{100,400},")
     position(26,16)
     puts(1,"                      {100,440},{10,440}})")
     polygon(12,0,{{10,360},{55,360},{55,400},{100,400},
              {100,440},{10,440}})

     while get_key() = -1 do
     end while
     if graphics_mode(-1) then
          puts(1,"Reset Failure")
     end if
end if
```



```
                         Polygon Fun!

          polygon(6,1,{{10,50},{100,50},{100,140},{10,140}})

          polygon(13,0,{{55,160},{10,240},{100,240}})

          polygon(2,1,{{30,260},{100,260},{80,340},{10,340}})

          polygon(12,0,{{10,360},{55,360},{55,400},{100,400},
                    {100,440},{10,440}})

                       Press any key to end
```

Drawing shapes with multiple sides is not too difficult, even without library routines like polygon(). Drawing oval shapes is a different matter entirely. A background in trigonometry was the only way you could draw any kind of circle or ellipse shape.

Fortunately, Euphoria comes to the rescue with a library routine that makes drawing circles and ellipses easy:

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

ellipse() draws an ellipse on the screen in colour i1. You have the option of having the area bordered by the circle to be filled (i2) using colour i1. A value of 1 means fill the area, while 0 means do not fill. The ellipse is drawn by defining a rectangular area on the screen, which will control the size and kind of ellipse that will be drawn. The area requires two pixel locations, an upper left corner (s1) and lower right corner (s2) in order for it to be defined. Both of these sequence values follow the format listed below:

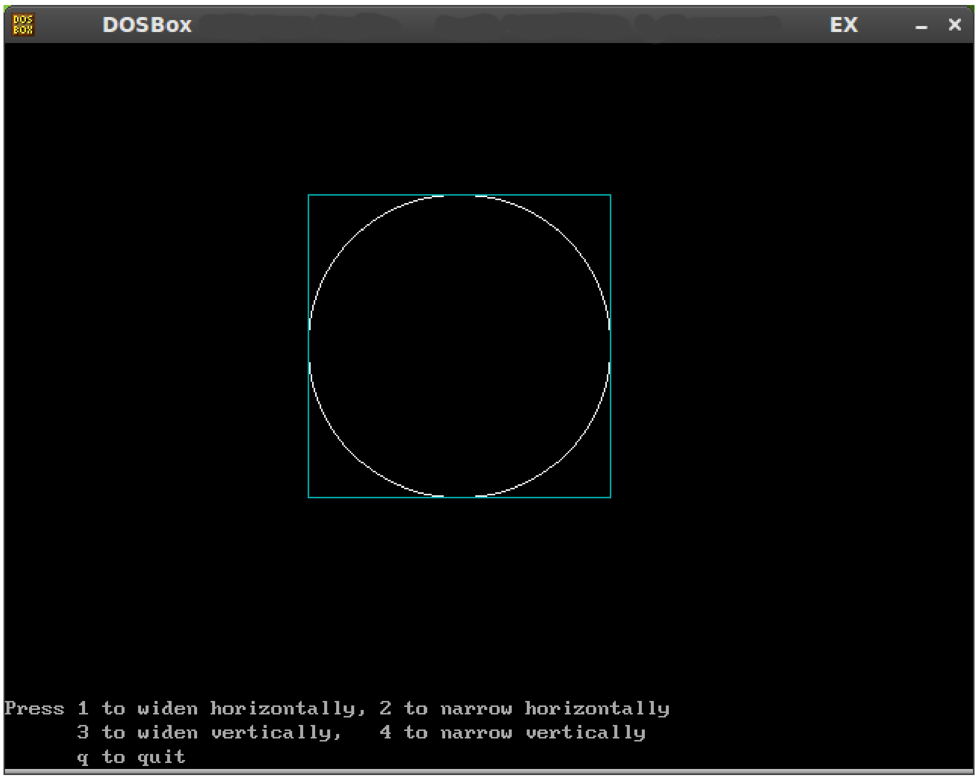
```
{pixel column location, pixel row location}
```

If you specify an area that is a perfect square, the oval drawn will be a circle. ellipse() only works in pixel-graphics modes. A demo program is ready to demonstrate how ellipse()'s defining of a rectangular area will dictate the shape and size of the circle.

program 59

```
include graphics.e
integer input_key, tl1, tl2, br1, br2, update
tl1 = 200
tl2 = 100
br1 = 400
br2 = 300
update = 'y'
if graphics_mode(18) then
    puts(1, "Mode not available")
else
    position (28,1)
    puts(1, "Press 1 to widen horizontally, 2 to narrow horizontally")
    position (29,1)
    puts(1, "      3 to widen vertically,   4 to narrow vertically")
    position (30,1)
    puts(1, "      q to quit")
    input_key = get_key()
    while input_key != 'q' do
         input_key = get_key()
         if update = 'y' then
              update = 'n'
              ellipse(15, 0, {tl1,tl2},{br1,br2})
              polygon(3,0, {{tl1,tl2},{br1,tl2},{br1,br2},{tl1,br2}})
         end if
         if input_key = '1' then
              ellipse(0, 0, {tl1,tl2},{br1,br2})
              polygon(0,0, {{tl1,tl2},{br1,tl2},{br1,br2},{tl1,br2}})
              tl1 = tl1 - 3
              br1 = br1 + 3
              update = 'y'
         end if
         if input_key = '2' then
              ellipse(0, 0, {tl1,tl2},{br1,br2})
              polygon(0,0, {{tl1,tl2},{br1,tl2},{br1,br2},{tl1,br2}})
              tl1 = tl1 + 3
              br1 = br1 - 3
              update = 'y'
         end if
         if input_key = '3' then
              ellipse(0, 0, {tl1,tl2},{br1,br2})
              polygon(0,0, {{tl1,tl2},{br1,tl2},{br1,br2},{tl1,br2}})
              tl2 = tl2 - 3
              br2 = br2 + 3
              update = 'y'
         end if
         if input_key = '4' then
              ellipse(0, 0, {tl1,tl2},{br1,br2})
              polygon(0,0, {{tl1,tl2},{br1,tl2},{br1,br2},{tl1,br2}})
              tl2 = tl2 + 3
              br2 = br2 - 3
              update = 'y'
         end if
    end while
    if graphics_mode(-1) then
```

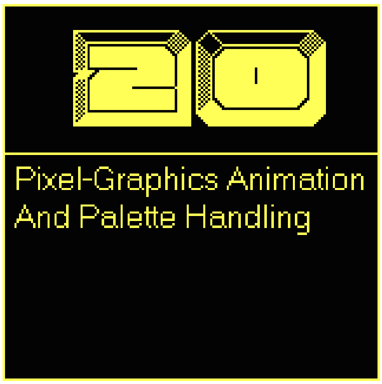
```
            puts(1, "Mode not available")
        end if
end if
```



```
Press 1 to widen horizontally, 2 to narrow horizontally
      3 to widen vertically,   4 to narrow vertically
      q to quit
```

The next chapter will show you how to make graphic images more animated instead of sitting motionless on the screen.

20. Pixel-Graphics Animation And Palette Handling



You remember from the chapters on text mode programming that you can create moving text in Euphoria. This is also possible with pixel graphic images. You can make images created with draw_line(), polygon() and ellipse() appear to move around the screen by redisplaying the images over and over while varying the screen location. Another form of animation is quickly changing one or more image colours without having to redisplay the image itself.

DOS and Eu3 only

Graphics is now done with GUI libraries. Windows, Unix, and multiplatform libraries are available.

Just as save_text_image() and display_text_image() are used to grab and redisplay text in text mode, Euphoria has library routines to grab and redisplay pixel images in pixel-graphics mode.

```
include image.e
rs = save_image(s1,s2)
```

save_image() saves an image within a rectangular area on the screen. The area is stored as a sequence made up of sequence elements that represent rows of pixel colours. The value is stored in receiving variable rs, and consists ofthe following structure:

```
{{pixel colour, pixel colour, pixel colour, pixel colour,...}
 {pixel colour, pixel colour, pixel colour, pixel colour,...}
 {pixel colour, pixel colour, pixel colour, pixel colour,...},
 ...}
```

Both the top left corner (s1) and the bottom right corner (s2) of the defined rectangular area being saved follow the format below:

```
{pixel column location, pixel row location}
```

Once you have captured the image using save_image(), you can then redisplay it using display_image():

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

n

display_image() displays a sequence value, s2, containing pixel colours at location s1 on the screen. Most likely s2 is a sequence variable containing the data retrieved by the save_image() library routine. You could, however, create an image to display with this library routine by following the format on the previous page. s1 is the location on the screen to display the image, and follows the format below:

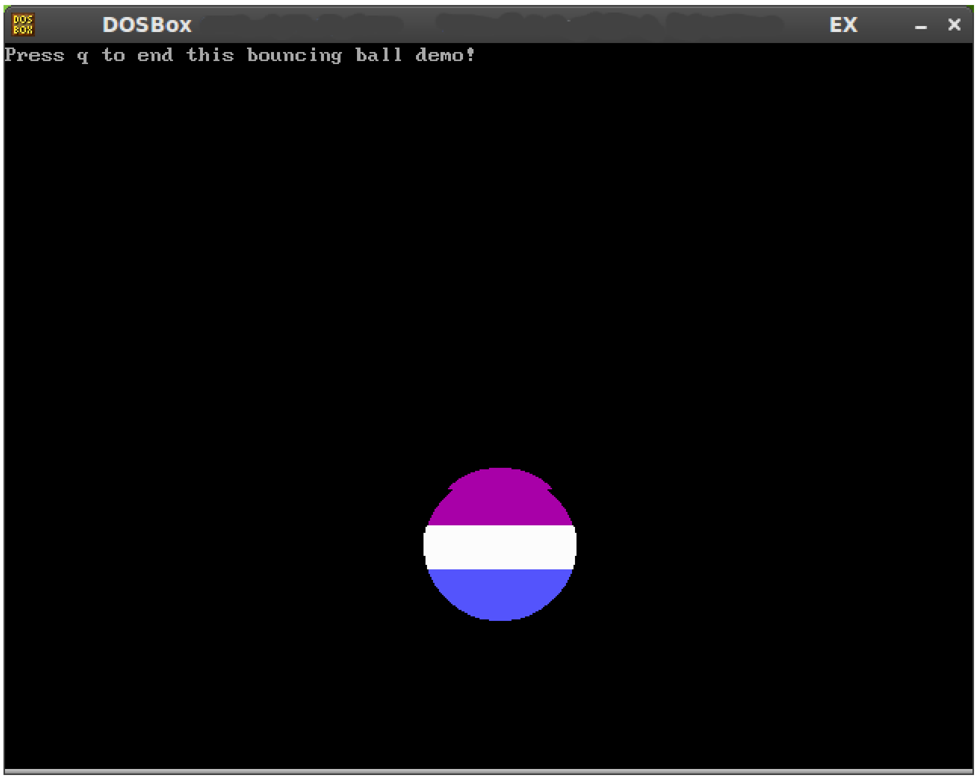
```
{pixel column location, pixel row location}
```

The first pixel colour in s2 will be displayed at screen location s1. Because the data in s2 implies a rectangular shape, s1 is where the top left corner of the image is displayed.

Both save_image() and display_image() only work in pixel-graphics modes. A demo program is available to show how to use both save_image() and display_image() to make a bouncing ball animation.

program 60

```
include image.e
include graphics.e
sequence ball, bounce
bounce =
{200,200.125,200.375,200.75,201.25,201.875,202.625,203.5,204.5,205.625,
206.875,208.25,209.75,211.375,213.125,215,217,219.125,221.375,223.75,226.25,
228.875,231.625,234.5,237.5,240.625,243.875,247.25,250.75,254.375,258.125,
262,266,270.125,274.375,278.75,283.25,287.875,292.625,297.5,302.5,307.625,
312.875,318.25,323.75,329.375,335.125,341,347,353.125}
if graphics_mode(18) then
     puts(1, "Mode Set Failure")
else
     clear_screen()
     ellipse(5, 1, {200,200}, {300,300})
     for iy = 234 to 266 do
          for ix = 200 to 300 do
               if get_pixel({ix, iy}) = 5 then
                    pixel(15,{ix, iy})
               end if
          end for
     end for
     for iy = 267 to 300 do
          for ix = 200 to 300 do
               if get_pixel({ix, iy}) = 5 then
                    pixel(9,{ix, iy})
               end if
          end for
     end for
     ball = save_image({193,193},{308,308})
     clear_screen()
     position(1,1)
     puts(1, "Press q to end this bouncing ball demo!")
     while 1 = 1 do
          if get_key() = 'q' then
               exit
          end if
          for ix = 1 to length(bounce) do
               display_image({270,bounce[ix]},ball)
          end for
          for ix = length(bounce) to 1 by -1 do
               display_image({270,bounce[ix]},ball)
          end for
     end while
     if graphics_mode(-1) then
          puts(1, "Mode Set Failure")
     end if
end if
```



```
Press q to end this bouncing ball demo!
```

Another kind of animation is colour-shifting, where one or more colours that the text or graphic image is in is changed. One way of doing it is to perform a save_image() or save_text_image(), search through the retrieved sequence value for the colour to be changed, and change where a match is made. Once done, you redisplay using either display_image() or display_text_image(). However, that is a lot of work just to change the colour of something displayed on the screen. Can we change the image colour without having the image itself redisplayed?

The answer is yes, by changing the palette of the graphics mode you are in. Changing any of the numbers on the palette to another colour will be instantaneously shown on the screen in any text or image that uses the colour. To do this, you first need to get a copy of the palette:

```
include image.e
rs = get_all_palette()
```

get_all_palette() returns the entire colour set of a palette as a sequence, which is then stored in receiving variable rs. The sequence is composed of sequence elements, where each element is the red, green, and blue intensities that make up a colour in the palette. The first element is the RGB intensity level for colour 0 (black), the second element is the RGB intensity level for colour 1 (blue), and so forth. These elements are made up of 3 atom values, the first being the level of red, the second being the level of green, the third being the level of blue.

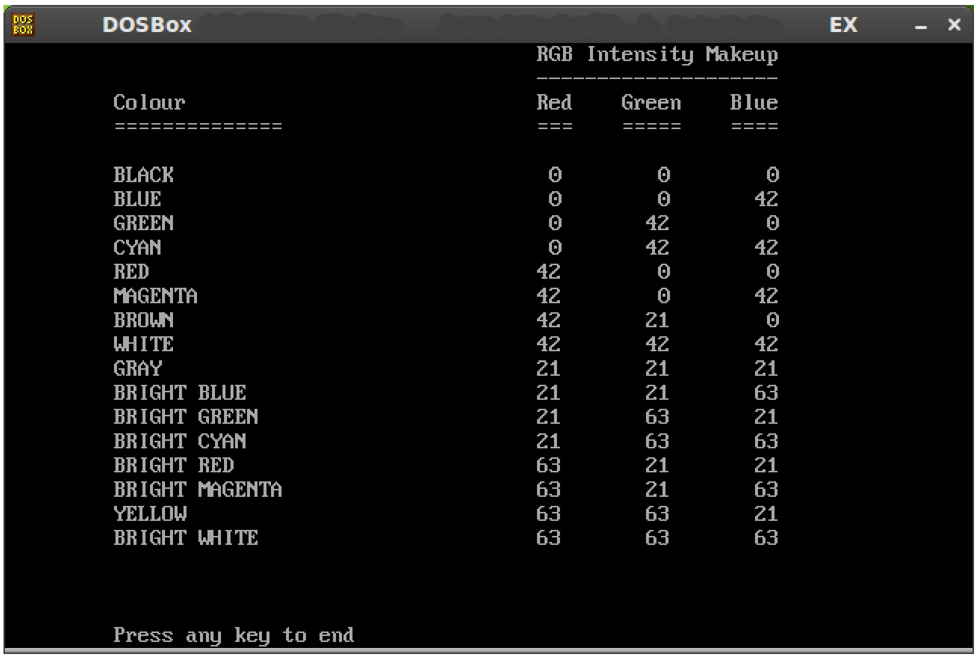
To help clarify, the format of the sequence representing the palette of the current graphics mode you are in is shown below:

```
{{red level, green level, blue level},                <-----colour 0
 {red level, green level, blue level},                <-----colour 1
 {red level, green level, blue level},                <-----colour 2
 {red level, green level, blue level},...}            <-----colour 3
```

Each red, green, and blue level can have a value from 0 (no intensity) to 63 (maximum intensity). get_all_palette() works in any text or pixel graphics mode. A demo program is available from this page. It will show the RGB intensities for each colour in a palette of a given screen mode.

program 61

```
include graphics.e
include image.e

sequence colour_list,rgb_levels

colour_list = {{"BLACK",0},{"BLUE",0},{"GREEN",0},{"CYAN",0},{"RED",0},
               {"MAGENTA",0},{"BROWN",0},{"WHITE",0},{"GRAY",0},
               {"BRIGHT BLUE",0},{"BRIGHT GREEN",0},{"BRIGHT CYAN",0},
               {"BRIGHT RED",0},{"BRIGHT MAGENTA",0},{"YELLOW",0},
               {"BRIGHT WHITE",0}}

if graphics_mode(3) = 0 then
     rgb_levels = get_all_palette()
     if graphics_mode(-1) then
         puts(1,"Reset Failure!")
     end if
end if

if graphics_mode(3) = 0 then
     for rgb = 1 to length(colour_list) do
         colour_list[rgb][2] = rgb_levels[rgb]
     end for
     position(1,10)
     puts(1,"                                RGB Intensity Makeup")
     position(2,10)
     puts(1,"                                --------------------")
     position(3,10)
     puts(1,"Colour                          Red     Green     Blue")
     position(4,10)
     puts(1,"==============                  ===     =====     ====")
     for colours = 1 to length(colour_list) do
          position(5+colours,10)
         printf(1,"%-14s                   %2d       %2d       %2d",
                {colour_list[colours][1],colour_list[colours][2][1],
                 colour_list[colours][2][2],
                 colour_list[colours][2][3]})
     end for
     position(25,10)
     puts(1,"Press any key to end")
     while get_key() = -1 do
     end while
     if graphics_mode(-1) then
         puts(1,"Reset Failure!")
     end if
end if
```



```
                                        RGB Intensity Makeup
                                        --------------------
        Colour                          Red    Green    Blue
        ==============                  ===    =====    ====

        BLACK                             0      0        0
        BLUE                              0      0       42
        GREEN                             0     42        0
        CYAN                              0     42       42
        RED                              42      0        0
        MAGENTA                          42      0       42
        BROWN                            42     21        0
        WHITE                            42     42       42
        GRAY                             21     21       21
        BRIGHT BLUE                      21     21       63
        BRIGHT GREEN                     21     63       21
        BRIGHT CYAN                      21     63       63
        BRIGHT RED                       63     21       21
        BRIGHT MAGENTA                   63     21       63
        YELLOW                           63     63       21
        BRIGHT WHITE                     63     63       63



        Press any key to end
```

Now that you have a copy of the palette, you can start changing colour numbers to mean a different colour. This is done by changing either one or more of the RGB intensity levels assigned to that colour.

Here is the library routine that can change a colour's RGB intensity:

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

palette() changes the colour number i to another colour by assigning it new RGB intensity levels stored in s. If successful, the previous RGB intensity levels will be returned as a sequence, to be stored in receiving variable ro, and any image or text on the screen that uses the colour i will immediately show the change. If unsuccessful, -1 is returned.

s is a sequence composed of 3 atom elements, namely the red, green, and blue intensity levels of what colour i is being set to:

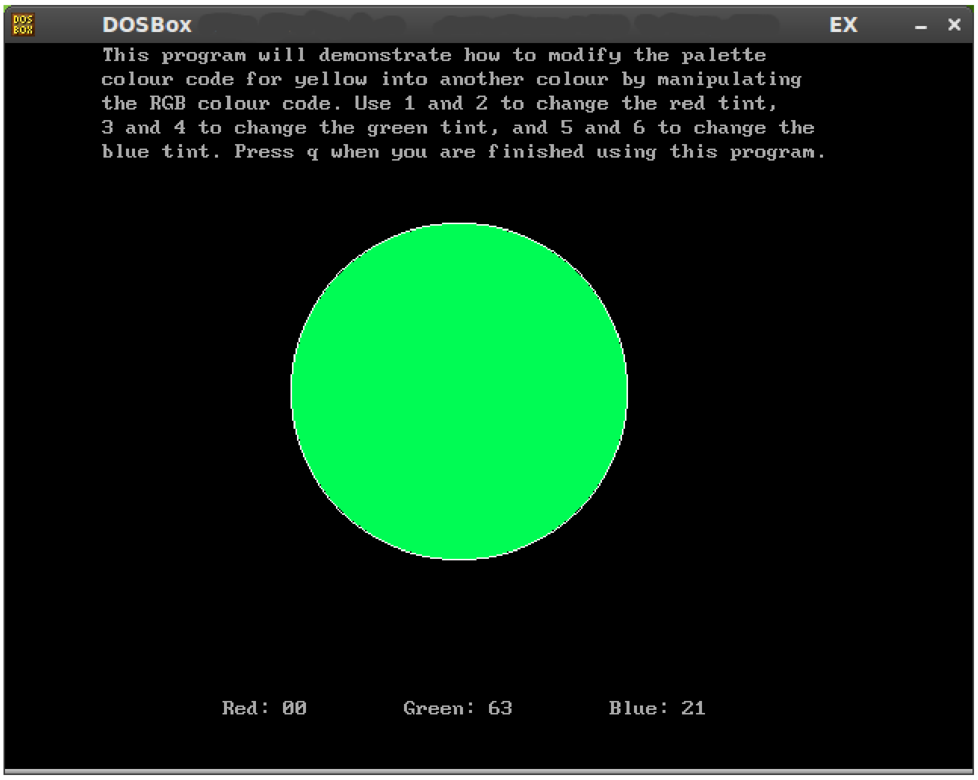
```
{red intensity level, green intensity level, blue intensity level}
```

The returned sequence value representing the old RGB intensity of the colour being changed (if successful) is also made up of threee atom elements, and is in the same format shown on the previous page.

The atom elements in the sequence passed to palette() must be between the values of 0 and 63 inclusive. palette() works in both text and pixel-graphics modes. It's always a good idea to save the returned value from palette(), or even using get_palette() to save an old copy of all the colours of the palette, before attempting to change any colour. A demo program is available from this screen, showing how to change the colour number associated with yellow to a new colour.

program 62

```
include graphics.e
object changed_palette
integer Red_, Green_, Blue_, input
if graphics_mode(18) then
     puts(1, "Mode Set Failure")
else
     for ix = 0 to 60 do
          ellipse(14, 1, {250-ix, 180-ix}, {350+ix, 280+ix})
          ellipse(15, 0, {249-ix, 179-ix}, {351+ix, 281+ix})
     end for
     Red_ = 0
     Green_ = 0
     Blue_ = 0

     changed_palette = palette(14, {Red_, Green_, Blue_})

     Red_ = changed_palette[1]
     Green_ = changed_palette[2]
     Blue_ = changed_palette[3]
     input = get_key()
     position(1,9)
     puts(1,"This program will demonstrate how to modify the palette")
     position(2,9)
     puts(1,"colour code for yellow into another colour by manipulating")
     position(3,9)
     puts(1,"the RGB colour code. Use 1 and 2 to change the red tint,")
     position(4,9)
     puts(1,"3 and 4 to change the green tint, and 5 and 6 to change the")
     position(5,9)
     puts(1,"blue tint. Press q when you are finished using this program.")
     while input != 'q' do
          if input = '1' and Red_ > 0 then
               Red_ = Red_ - 1
          end if
          if input = '2' and Red_ < 63 then
               Red_ = Red_ + 1
          end if
          if input = '3' and Green_ > 0 then
               Green_ = Green_ - 1
          end if
          if input = '4' and Green_ < 63 then
               Green_ = Green_ + 1
          end if
          if input = '5' and Blue_ > 0 then
               Blue_ = Blue_ - 1
          end if
          if input = '6' and Blue_ < 63 then
               Blue_ = Blue_ + 1
          end if
          changed_palette = palette(14, {Red_, Green_, Blue_})
          position(28,19)
          printf(1, "Red: %.2d        Green: %.2d        Blue: %.2d",
                 {Red_,Green_,Blue_})
          input = get_key()
     end while
     if graphics_mode(-1) then
          puts(1, "Mode Set Failure")
     end if
end if
```



```
This program will demonstrate how to modify the palette
colour code for yellow into another colour by manipulating
the RGB colour code. Use 1 and 2 to change the red tint,
3 and 4 to change the green tint, and 5 and 6 to change the
blue tint. Press q when you are finished using this program.

Red: 00          Green: 63          Blue: 21
```

If you want to change all the colours in the palette, you can either issue palette() once for every colour, or use a Euphoria library routine created to change all colours at once.

```
include graphics.e
all_palette(s)
```

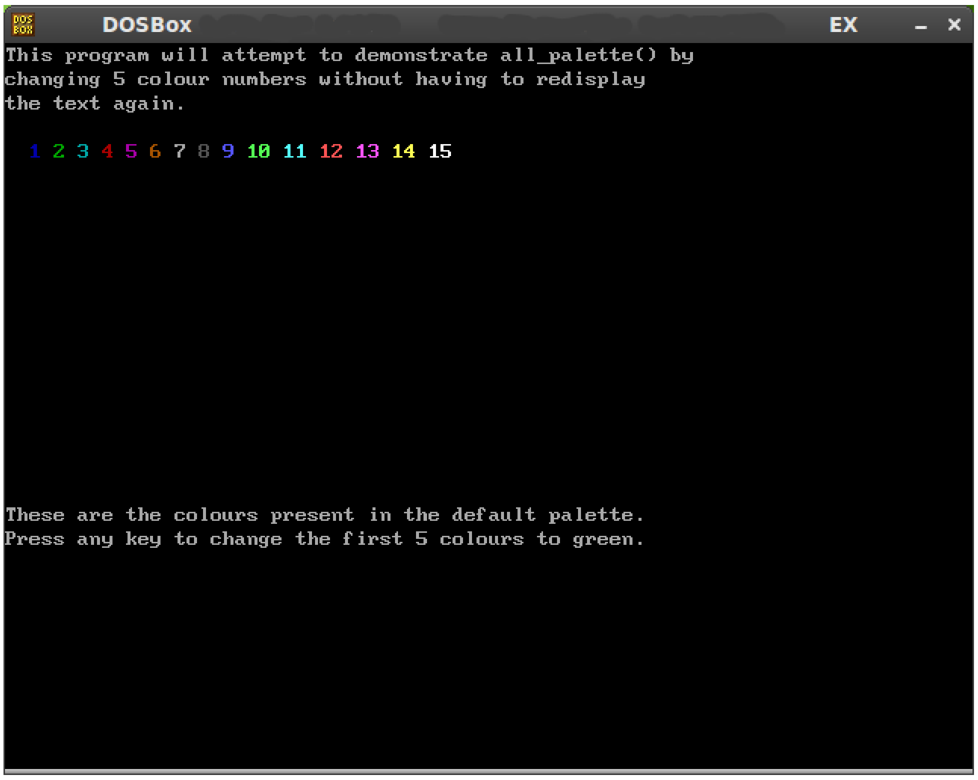
all_palette() is used to change all colours of a palette in the current graphics mode. s is a list of new RGB intensities for the entire palette to be set to, and is a sequence made up of sequence elements. The element position represents a colour, starting with 0 (black) for the first element, 1 (blue) for the second element, 2 (green) for the third element, and so forth. Each element of s is made up of three atom elements, the first being the red intensity level, the second being the green intensity level, and the third being the blue intensity level:

```
{{red level, green level, blue level},         <---- new colour for 0
 {red level, green level, blue level},         <---- new colour for 1
 {red level, green level, blue level},         <---- new colour for 2
 {red level, green level, blue level},...}     <---- new colour for 3
```

As with palette(), each red, green, and blue intensity must be between the values of 0 and 63 inclusive. Any text or image on the screen that is using the colours changed by all_palette() will instantaneously show the changes on the screen once this library routine is executed. This is handy if you are performing a large scale of colour changing involving many images or text on the screen. Using all_palette() to change a series of colours is much faster than using palette() to change each colour individually. all_palette() works in both text and pixel-graphics modes. A demo program is available to show how colours 1 through 5 are changed using all_palette().

program 63

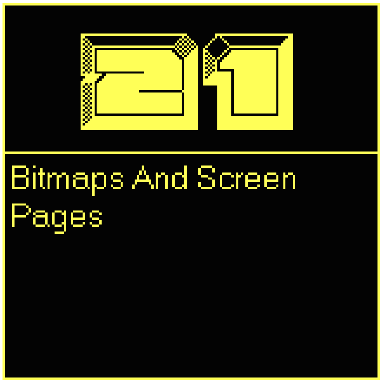
```
include graphics.e
include image.e

sequence colours_one_to_five, original

if graphics_mode(18) = 0 then
     original = get_all_palette()

     clear_screen()

     text_color(7)

     puts(1,"This program will attempt to demonstrate all_palette() by\n")
     puts(1,"changing 5 colour numbers without having to redisplay\n")
     puts(1,"the text again.\n\n")

     colours_one_to_five = original
     colours_one_to_five[2..6] = {{0,63,0},{0,63,0},{0,63,0},{0,63,0},
                                  {0,63,0}}

     for ix = 0 to 15 do
          text_color(ix)
          print(1,ix)
          puts(1," ")
     end for

     for retry = 1 to 2 do
          text_color(7)
          position(20,1)
          if retry = 1 then
               puts(1,
              "These are the colours present in the default palette.\n")
               puts(1,
              "Press any key to change the first 5 colours to green.\n")
          end if
          if retry = 2 then
               all_palette(colours_one_to_five)
               puts(1,
              "Colour numbers from 1 to 5 now have an RGB setting   \n")
               puts(1,
              "of {0,63,0}, or pure green. Press any key to end.    \n")
          end if
          while get_key() = -1 do
          end while
     end for

     all_palette(original)

end if

if graphics_mode(-1) then
     puts(1,"reset failure\n")
end if
```



```
This program will attempt to demonstrate all_palette() by
changing 5 colour numbers without having to redisplay
the text again.

  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15












These are the colours present in the default palette.
Press any key to change the first 5 colours to green.
```

It is important to understand how to change palettes in graphics modes, as you will need to do so when dealing with .BMP files, or Windows bitmaps. The next chapter will introduce you to bitmaps, as well as how to direct screen output (text or pixel graphics images) to multiple screen pages.

21. Bitmaps And Screen Pages



This final chapter on text and pixel-graphics output will focus on bitmaps and screen pages. Euphoria allows the programmer to use bitmapped (.BMP) files created by popular paint programs like Neopaint and Windows Paintbrush, without needing to write any code to read in the data. This saves a lot of time in graphics programming. The programmer also has the option of redirecting screen output to more than one virtual screen page, and deciding which page to show on the screen.

DOS graphics and Eu3 only

Before using any .BMP files in your program, you have to read them in. This is easily done using the read_bitmap() library routine.

```
include image.e
ro = read_bitmap(s)
```

read_bitmap() reads in the data stored in a .BMP file, s,and stores that data in receiving variable ro as a sequence value. The sequence value is composed of two elements. The first element is the palette, a sequence containing the RGB instensity levels for colours the .BMP file used. The second element is a sequence containing the pixels themselves, arranged in a format that display_image() can use to show on the screen:

```
{{{red level, green level, blue level},...},
  {{pixel colour, pixel colour, pixel colour,...},
  {pixel colour, pixel colour, pixel colour,...},
  {pixel colour, pixel colour, pixel colour,...},...}}
```

If the palette of the .BMP file uses the same colour scheme as the palette of the graphics mode you are in, it's no hassle. You just store the second element of the sequence value returned by read_bitmap() in a variable, and then use the library routine display_image() to display the contents of that variable on the screen.

If the .BMP file uses a palette colouring scheme different from the palette of the graphics mode you are in, you have to do some extra steps before displaying the .BMP file on the screen. First you take the first element of the sequence returned by read_bitmap() (the .BMP's palette), and divide it by four. You then pass the adjusted palette to all_palette(), to change the graphic mode palette to match that of the .BMP file. Once this is done, you then take the second element of the sequence returned by read_bitmap(), store it in a variable, then use display_image() to present it on the screen.

Why use all_palette() to have the graphic mode's palette adjusted to match that of the .BMP file's palette? Pretend you are attempting to display a .BMP file of a flower, and petals use colour 2 to show a faded pink. In any of the graphic modes, colour 2 is green. So when you try to display this picture of the flower, the petals will appear…you guessed it…GREEN, not PINK. By adjusting the the palette to match that used by the .BMP file, the picture shows up properly.

Why divide the .BMP file palette by four before using it with all_palette()? The red, green, and blue intensity levels of a .BMP file palette go from 0 to 255. In Euphoria, the graphic mode palette for red, green, and blue intensities go from 0 to 63. This means the .BMP file palette uses an intensity scheme that is four times the size of the graphics mode palette. To properly adjust for this, you must divide by four to bring it in line with what the graphic mode can use.

.BMP files using 2, 4, 16, or 256 colours are supported. If something goes wrong during the reading of the .BMP file data by read_bitmap(), the library routine will return any one of the following error codes:

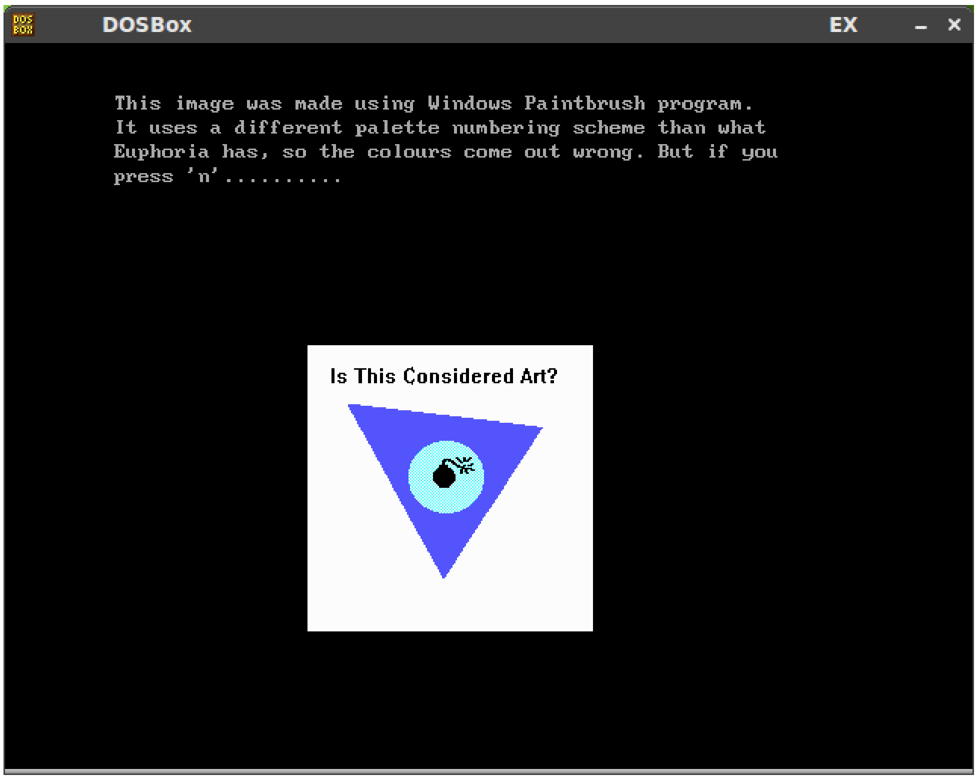
- 1 = open failed (probably spelt the name of the file wrong)
- 2 = unexpected end of file (the end of the file was reached before all the required data was read in
- 3 = unsupported format (Euphoria may not recognize that format even though other paint programs can load it with no problems)

A demo program shows how to load a .BMP file and display it, with a note about adjusting the palette first before displaying it.

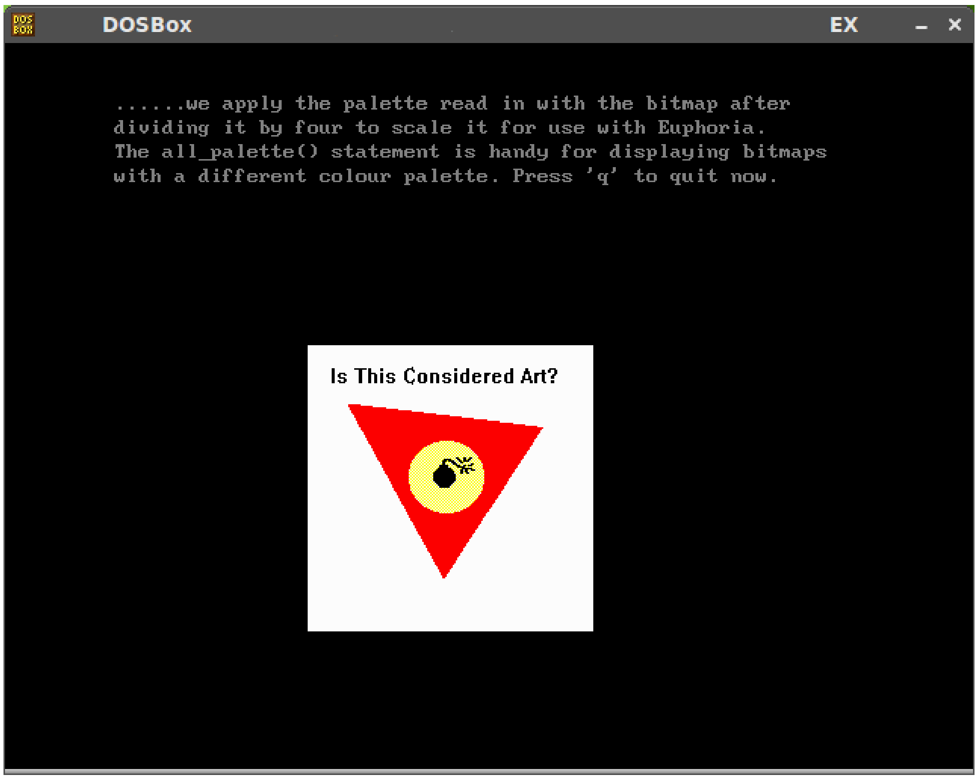
program 64

```
include graphics.e
include image.e
sequence demo_bitmap
if graphics_mode(18) then
     puts(1, "Mode Failure")
else
     demo_bitmap = read_bitmap("d2105a.bmp")
     position(3,10)
```

```
          puts(1, "This image was made using Windows Paintbrush program.")
          position(4,10)
          puts(1, "It uses a different palette numbering scheme than what")
          position(5,10)
          puts(1, "Euphoria has, so the colours come out wrong. But if you")
          position(6,10)
          puts(1, "press 'n'..........")
          display_image({200,200}, demo_bitmap[2])
          while get_key() != 'n' do
          end while
          position(3,10)
          puts(1, "......we apply the palette read in with the bitmap after   ")
          position(4,10)
          puts(1, "dividing it by four to scale it for use with Euphoria.     ")
          position(5,10)
          puts(1, "The all_palette() statement is handy for displaying bitmaps")
          position(6,10)
          puts(1, "with a different colour palette. Press 'q' to quit now.    ")
          all_palette((demo_bitmap[1]/4))
          while get_key() != 'q' do
          end while
          if graphics_mode(-1) then
               puts(1, "Mode Failure")
          end if
     end if
```



```
This image was made using Windows Paintbrush program.
It uses a different palette numbering scheme than what
Euphoria has, so the colours come out wrong. But if you
press 'n'.........
```



```
......we apply the palette read in with the bitmap after
dividing it by four to scale it for use with Euphoria.
The all_palette() statement is handy for displaying bitmaps
with a different colour palette. Press 'q' to quit now.
```

Your Euphoria programs can take screen images and save them into a .BMP file to be viewed and edited by a paint program, or loaded and displayed by another Euphoria program using read_bitmap().

```
include image.e
ri = save_bitmap(s1,s2)
```

save_bitmap() creates a Windows .BMP file, s2, using data stored in s1. s1 is a two element sequence, the first being a sequence representing the palette of the .BMP file being created, and the second being a sequence representing the coloured pixels that make up the .BMP file data. s1 matches the format introduced in read_bitmap().

The palette for the .BMP file you are reading is easily obtained using get_all_palette(). Because .BMP file palettes use red, green, and blue intensities between 0 and 255, you must multiply the palette returned by get_all_palette() by 4.

Getting the .BMP file data is even easier. You use save_image() to grab whatever image on the screen you want saved in the .BMP file.

Once the .BMP file palette and data are obtained, you use append() to join them together to make a new sequence, which save_bitmap() uses to create your new BMP file.

If the .BMP file data sequence was created other than by using save_image(), make sure all sequence elements are of the same length. Note that some paint programs do not support a 4-colour .BMP file, even though save_bitmap() produces 2, 4, 16, and 256 colour .BMP files.

save_bitmap() returns an integer value to report how well things went:

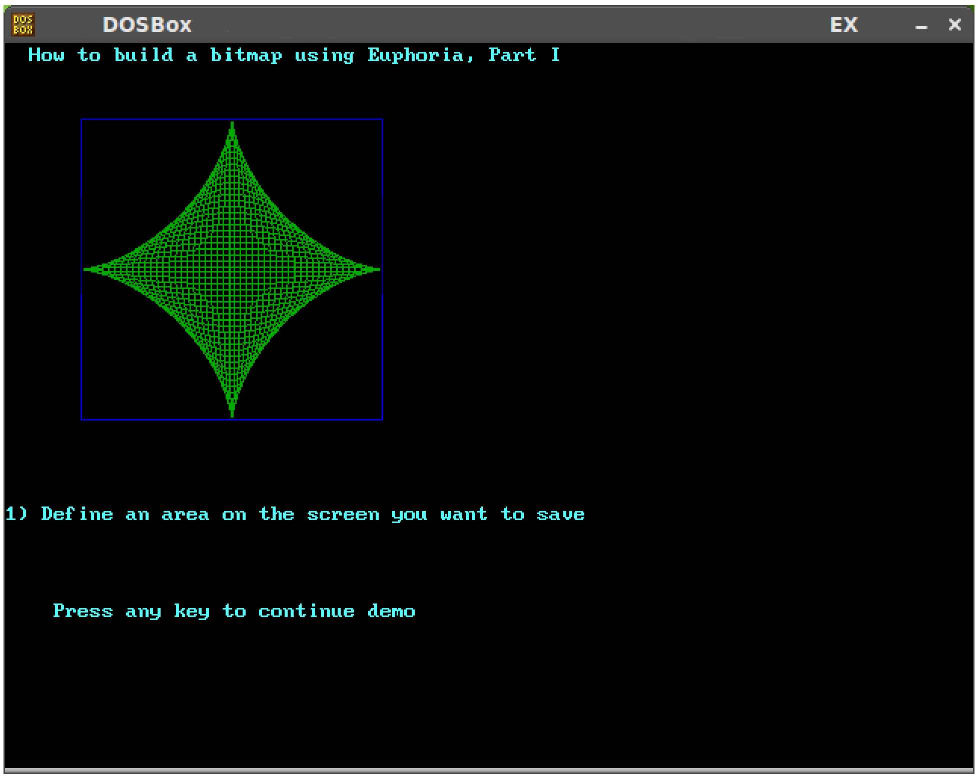
- 0 = .BMP file created successfully (it works!)
- 1 = .BMP file open failed (probably mis-spelled the name)
- 4 = .BMP file invalid mode (invalid graphic mode or parameters received)

A demo is available, showing the steps to create a .BMP file.

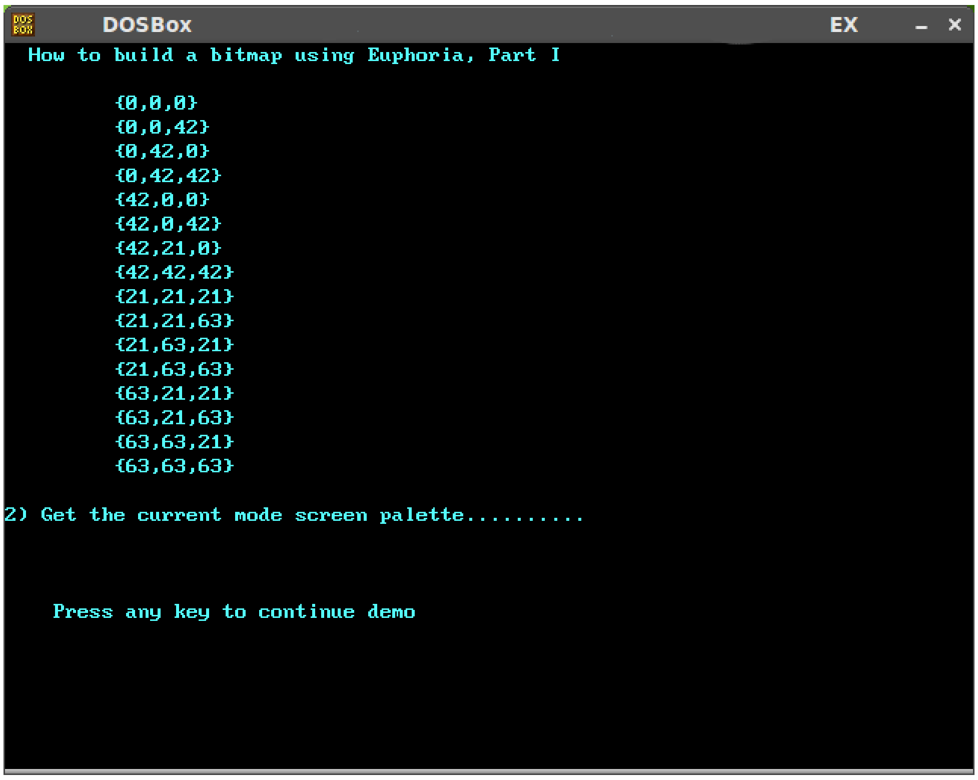
program 65

```
include graphics.e
include image.e
atom blue, increment, status
sequence palette_mode,new_blue, previous_blue, bitmap_data, bitmap_file
blue = 30
increment = 1
if graphics_mode(18) then
     puts(1,"Unable to go into mode 18!")
else
     palette_mode = get_all_palette()
     position(1,3)
     text_color(11)
     puts(1,"How to build a bitmap using Euphoria, Part I")
     position(24,5)
     puts(1,"Press any key to continue demo")
     for ix = 0 to 50 by 4 do
          ellipse(2,0,{100+ix,100-ix},{199-ix,199+ix})
          ellipse(2,0,{100-ix,100+ix},{199+ix,199-ix})
          polygon(1,0,{{50,50},{50,249},{249,249},{249,50}})
     end for
     position(20,1)
     puts(1,"1) Define an area on the screen you want to save")
     while get_key() = -1 do
          if blue = 63 then
               increment = -3
          elsif blue = 30 then
               increment = 3
          end if
          blue = blue + increment
          new_blue = {0,0,0}
          new_blue[3] = blue
          previous_blue = palette(1,new_blue)
     end while
     polygon(0,0,{{50,50},{50,249},{249,249},{249,50}})
     bitmap_data = save_image({50,50},{249,249})
     polygon(0,1,{{50,50},{50,249},{249,249},{249,50}})
     for ix = 3 to 18 do
          position(ix,10)
          print(1,palette_mode[-2+ix])
     end for
     position(20,1)
     puts(1,"2) Get the current mode screen palette..........")
     while get_key() = -1 do
     end while
     palette_mode = palette_mode * 4
     for ix = 3 to 18 do
          position(ix,10)
          print(1,palette_mode[-2+ix])
     end for
     position(20,1)
     puts(1,"3) ....and scale it for use as a bitmap palette. ")
     while get_key() = -1 do
     end while
     clear_screen()
     puts(1,"You then use both the palette and the bitmap data to\n")
     puts(1,"construct the bitmap using save_image..\n")
     bitmap_file = {}
     bitmap_file = append(bitmap_file,palette_mode)
     bitmap_file = append(bitmap_file,bitmap_data)
```

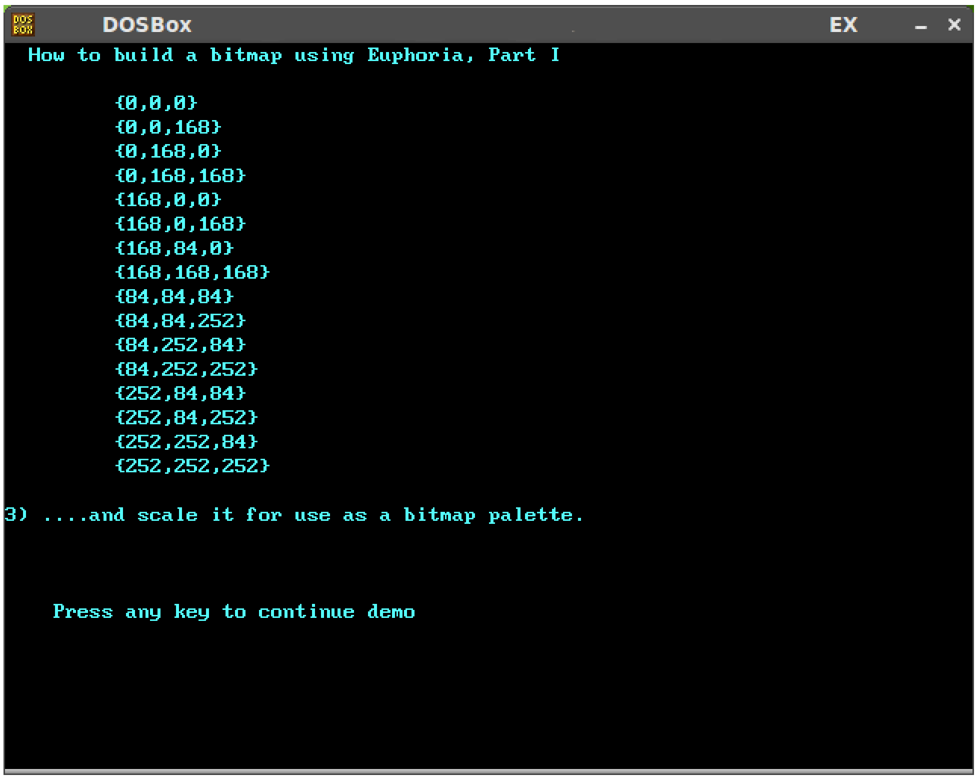
```
          while get_key() = -1 do
          end while
          if graphics_mode(-1) then
               puts(1,"Unable to reset mode!")
          end if
          status = save_bitmap(bitmap_file,"d2107a.BMP")
          if status != 0 then
                puts(1,"Bitmap creation failure!")
          end if
end if
```



```
  How to build a bitmap using Euphoria, Part I

1) Define an area on the screen you want to save

    Press any key to continue demo
```



```
  How to build a bitmap using Euphoria, Part I

         {0,0,0}
         {0,0,42}
         {0,42,0}
         {0,42,42}
         {42,0,0}
         {42,0,42}
         {42,21,0}
         {42,42,42}
         {21,21,21}
         {21,21,63}
         {21,63,21}
         {21,63,63}
         {63,21,21}
         {63,21,63}
         {63,63,21}
         {63,63,63}

2) Get the current mode screen palette..........

    Press any key to continue demo
```



```
 How to build a bitmap using Euphoria, Part I

        {0,0,0}
        {0,0,168}
        {0,168,0}
        {0,168,168}
        {168,0,0}
        {168,0,168}
        {168,84,0}
        {168,168,168}
        {84,84,84}
        {84,84,252}
        {84,252,84}
        {84,252,252}
        {252,84,84}
        {252,84,252}
        {252,252,84}
        {252,252,252}

3) ....and scale it for use as a bitmap palette.


    Press any key to continue demo
```

If you find the steps taken to assemble a sequence used by save_bitmap() to create a .BMP file a little daunting, Euphoria has another approach that is somewhat simpler:

```
include image.e
ri = save_image(o,s)
```

save_screen() saves either the entire screen or a rectangular area of the screen to a Windows .BMP file, named s. If o is equal to 0, everything displayed on the screen is saved to .BMP file s. If o is a sequence value, it defines a rectangular area on the screen that is to be saved to .BMP file s. o follows the structure format listed below:

```
{{top left pixel colmn, top left pixel row},
 {bottom right pixel column, bottom right pixel row}}
```

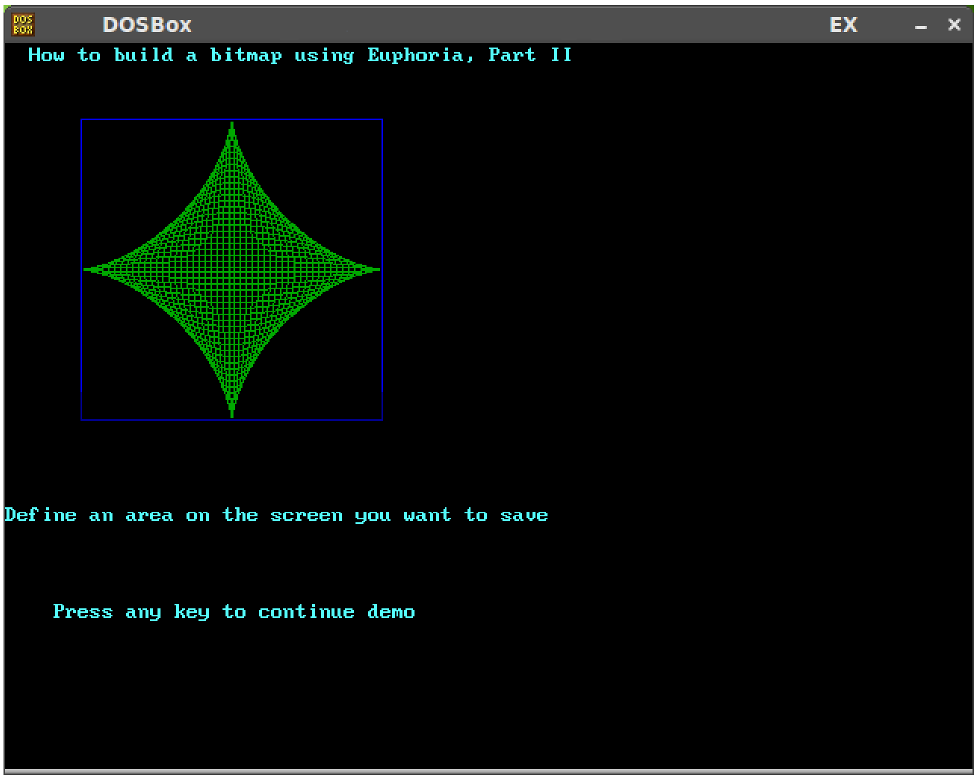
save_screen() uses the palette of the current graphics mode you are in as the .BMP file's palette. The defined rectangular area on the screen is the data that make up the .BMP file image. When o is 0, the rectangular area is from the top left and the bottom right corners of the screen, and is defined automatically for you.

Like save_bitmap(), save_screen() returns and integer value based on the whether or not the .BMP file was created successfully. They are the same as those values returned by save_bitmap(). Also, save_screen() creates .BMP files that have 2, 4, 16, or 256 colours. Some paint programs cannot read 4 colour .BMP files, though any image created by save_bitmap() and save_screen() can be read by read_bitmap().

save_screen() only works in pixel-graphics modes. The demo program is a remake of the save_bitmap() demo program, using save_screen() instead to create a .BMP file.

program 66

```
include graphics.e
include image.e
atom blue, increment, status
sequence palette_mode,new_blue, previous_blue
blue = 30
increment = 1
if graphics_mode(18) then
     puts(1,"Unable to go into mode 18!")
else
     palette_mode = get_all_palette()
     position(1,3)
     text_color(11)
     puts(1,"How to build a bitmap using Euphoria, Part II")
     position(24,5)
     puts(1,"Press any key to continue demo")
     for ix = 0 to 50 by 4 do
         ellipse(2,0,{100+ix,100-ix},{199-ix,199+ix})
         ellipse(2,0,{100-ix,100+ix},{199+ix,199-ix})
         polygon(1,0,{{50,50},{50,249},{249,249},{249,50}})
     end for
     position(20,1)
     puts(1,"Define an area on the screen you want to save")
     while get_key() = -1 do
          if blue = 63 then
               increment = -3
          elsif blue = 30 then
               increment = 3
          end if
          blue = blue + increment
          new_blue = {0,0,0}
          new_blue[3] = blue
          previous_blue = palette(1,new_blue)
     end while
     all_palette(palette_mode)
     polygon(0,0,{{50,50},{50,249},{249,249},{249,50}})
     status = save_screen({{50,50},{249,249}},"d2109a.bmp")
     polygon(0,1,{{50,50},{50,249},{249,249},{249,50}})
     clear_screen()
     puts(1,"save_screen() produces the same result as save_bitmap()\n")
     puts(1,"but without the steps shown in the save_bitmap() demo.\n")
     while get_key() = -1 do
     end while
     if graphics_mode(-1) then
          puts(1,"Unable to reset mode!")
     end if
end if
```



```
 How to build a bitmap using Euphoria, Part II

Define an area on the screen you want to save

    Press any key to continue demo
```

With handling .BMP files now explored, let's move on to handling screen pages.

Each screen page is assigned a number, beginning with screen page 0. The default active page (where screen output is sent) and the default display page (the page that you want displayed on the screen) are both screen page 0.

To change the screen page where all screen output is sent to, you use the following library routine below:

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

i is the screen page number you want screen output to be sent to. The number of pages available depends on the graphics mode you are in.

If the new active page is not the same as the display page, you will see no changes on the screen when you perform any kind of screen output. Only when you change the display page to the new active page will you see what was sent. You cannot change the active page if you are in partial screen window under Windows. Only if you are in DOS or a full screen window under Windows. If you are uncertain on how many screen pages you have in the graphics mode you are in, use video_config().

To select a screen page you want displayed on the screen, you use the following library routine:

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

i is the screen page number you want displayed on the screen. The number of pages you can display depends on the graphics mode you are in.

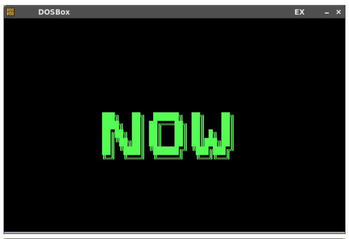
If the new display page is not the same as the active page, you will see no changes on the screen when you perform any kind of screen output. Only when you change the active page to the new display page will you see what was sent. Like set_active_page(), set_display_page() only works in DOS or in a full screen window under Windows.

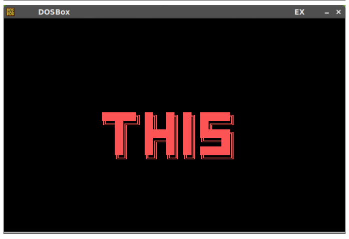
A demo program will use set_active_page() and set_display_page() to first send screen output to each separate screen page, then to display each screen page one at a time.

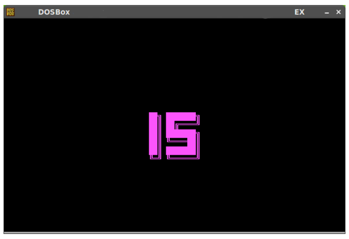
program 67

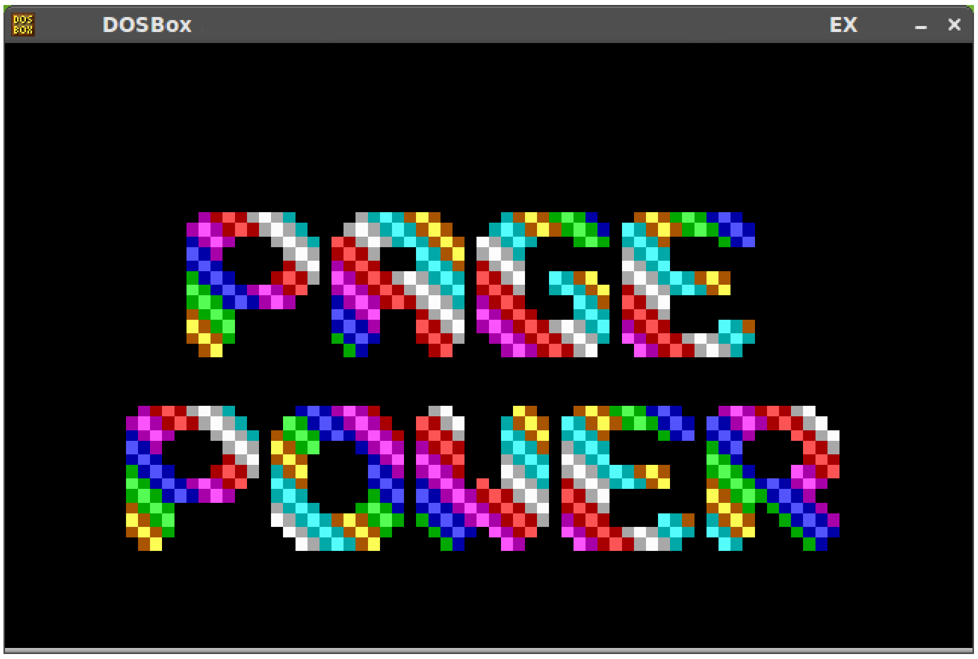
```
include graphics.e
include image.e
include get.e
integer file_id
sequence capture_buffer, video_data

video_data = video_config()
```

```
clear_screen()
printf(1, "%d pages available, 5 pages required\n", {video_data[8]})
if video_data[8] < 5 then
     puts(1, "Sorry, you have insufficient pages on your video card\n")
else
     puts(1, "Stand By, Loading Each Virtual Page\n")

     set_active_page(1)
     file_id = open("now.bin","rb")
     capture_buffer = get(file_id)
     close(file_id)
     display_text_image({1,1}, capture_buffer[2])

     set_active_page(2)
     file_id = open("this.bin","rb")
     capture_buffer = get(file_id)
     close(file_id)
     display_text_image({1,1}, capture_buffer[2])

     set_active_page(3)
     file_id = open("is.bin","rb")
     capture_buffer = get(file_id)
     close(file_id)
     display_text_image({1,1}, capture_buffer[2])

     set_active_page(4)
     file_id = open("ppower.bin","rb")
     capture_buffer = get(file_id)
     close(file_id)
     display_text_image({1,1}, capture_buffer[2])

     set_active_page(0)

     clear_screen()
     puts(1, "Done....to cycle through all the four pages, press 'n'.\n")
     puts(1, "Press any key to start cycling now.\n")
     while get_key() = -1 do
     end while

     for ix = 1 to 4 do
          set_display_page(ix)
          while get_key() != 'n' do
          end while
     end for

     clear_screen()
end if

set_active_page(0)
set_display_page(0)
```









If you want to know what screen page you have made either an active page or a display page, there are two library routines that can help you find out.

To determine which screen page is the current active page, you use:

```
include image.e
ri = get_active_page()
```

get_active_page() returns the screen page number that screen output is being sent to. The screen page number is stored in receiving variable ri.

To determine which screen page is the current display page, you use:

```
include image.e
ri = get_display_page()
```

get_display_page() returns the screen page number that is being displayed. The screen page number is stored in receiving variable ri.

You now know how to create programs that use colourful text and graphic images. But just in case you do run into some difficulty using graphics, particularly in SVGA (Super Video Graphics Adaptor) mode, where you use colours of 256 and higher, you may need to use a video graphics standard called VESA instead of Euphoria's method of working with the video card:

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

If i is 1, Euphoria uses the VESA graphics standard to generate SVGA mode graphics. Otherwise, 0 means Euphoria uses its own methods. You should issue use_vesa() before using the graphics_mode() library routine. However, it's rare that you need to use this library routine.

Euphoria programs on their own can do a lot. But what if you can access the features of your operating system? The next chapter shows you how!

22. Euphoria And DOS, Part One

The operating system is the bridge between the program and the actual hardware of your computer. If a program can directly access operating system features, it would be able to handle tasks that would normally be beyond the scope of the programming language that it is made of. Euphoria has a set of library routines that allow access to the date and time, run DOS programs and commands, accept values from outside the program as parameters, and much more!

How Euphoria interfaces to operating systems has not changed much—various routines apply to many platforms

If you are writing programs for other people to use, the one feature people like to see most often is the date and time on the screen as they work at their computer. Euphoria has a library routine that returns the date and time to your program.

```
rs = date()
```

`date()` returns a sequence value composed of eight atom elements, which is stored in receiving variable rs The sequence is the date and time on the computer, formatted in the following manner:

```
{number of years since 1900,
 month number (where January is 1),
 day of month (starting at one),
 hour (between 0 and 23),
 minute (between 0 and 59),
 second (between 0 and 59),
 day of the week (where Sunday is 1),
 number of days since the start of the year}
```

If the first element of this sequence is greater than or equal to 100, then you are dealing with dates in the 21st century. For example, 101 is actually the year 2001. A demo program is available to show the system date and time in human readable form after `date()` is used to get them.

program 68

```
integer  curr_year, curr_day, curr_day_of_year,
         curr_hour, curr_minute, curr_second
```

```
sequence system_date, word_week, word_month, notation,
         curr_day_of_week, curr_month

word_week =   {"Sunday",
               "Monday",
               "Tuesday",
               "Wednesday",
               "Thursday",
               "Friday",
               "Saturday"}

word_month =  {"January",
               "February",
               "March",
               "April",
               "May",
               "June",
               "July",
               "August",
               "September",
               "October",
               "November",
               "December"}


system_date = date()

curr_year = system_date[1]
curr_month = word_month[system_date[2]]
curr_day = system_date[3]
curr_hour = system_date[4]
curr_minute = system_date[5]
curr_second = system_date[6]
curr_day_of_week = word_week[system_date[7]]
curr_day_of_year = system_date[8]

if curr_hour >= 12 then
     notation = "p.m."
else
     notation = "a.m."
end if

if curr_hour > 12 then
     curr_hour = curr_hour - 12
end if

if curr_hour = 0 then
     curr_hour = 12
end if

puts(1, "\nHello!\n\n")
printf(1, "Today is %s, %s %d, 19%d.\n", {curr_day_of_week,
                                          curr_month,
                                          curr_day, curr_year})
printf(1, "The time is %.2d:%.2d:%.2d %s\n", {curr_hour, curr_minute,
                                             curr_second, notation})
printf(1, "It is %3d days into the current year.\n", {curr_day_of_year})
```

```
Hello!

Today is Wednesday, March 30, 19111.
The time is 10:45:57 a.m.
It is  89 days into the current year.
```

As a programmer, you may be interested in time of a differenct kind, as in elapsed time taken to complete a process. This is particularly the case if you are interested in learning how long a section of code in your program takes to complete, or even to delay program execution for an interval of time. Here's the library routine that can help you do this:

```
ra = time()
```

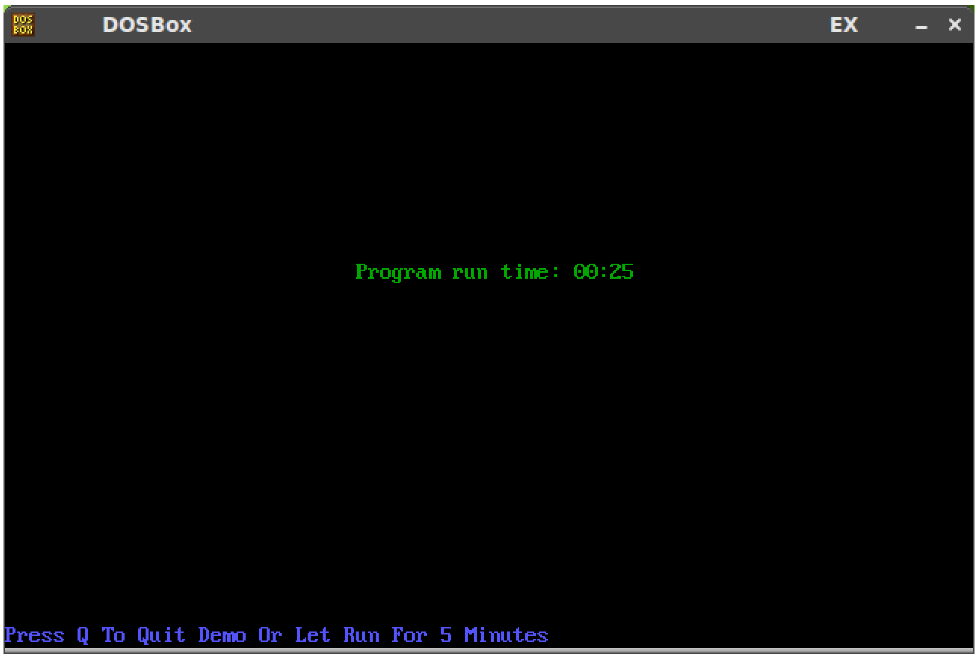
time returns an atom value, representing the number of seconds elapsed since a fixed point in time which can be stored in receiving variable ra. The fixed point in time `time()` measures against is the moment the Euphoria program started running, so executing `time()` at the very start of your program would return a value very close to, if not equal to, 0.

To measure an interval of time, or to delay program execution for a set number of seconds, you execute `time()` at two different points of the program run, storing each returned value in separate atom variables, by subtracting the first returned value from the second one, you obtain a measurement in seconds between these two points.

When you look at a clock, such as your wristwatch or a wall-mounted one, you notice that it advance one second at a time. This means its measurement of time has a resolution of one second. MSDOS on the other hand has a resolution of 0.055 seconds, so its resolution is more precise. This means the smallest amount of time you can measure using `time()`, by default, is 0.055 seconds [Note: On Win32 \ Linux \ FreeBSD it's about 0.01 seconds]. A demo program is available to show how to use `time()` as a timer.

program 69

```
include graphics.e
atom minutes, seconds, elapsed_seconds, halt_program
clear_screen()
position(25,1)
text_color(9)
puts(1,"Press Q To Quit Demo Or Let Run For 5 Minutes")
text_color(2)
halt_program = ' '
while halt_program != 'q' do
     halt_program = get_key()
     elapsed_seconds = time()
     minutes = elapsed_seconds / 60
     seconds = remainder(elapsed_seconds, 60)

     position(10, 30)
     printf(1, "Program run time: %.2d:%.2d", {minutes, seconds})
     if minutes >= 4 then
        if seconds >= 50 then
             if seconds >= 55 then
                  text_color(12 + 16)
             else
                  text_color(14)
             end if
        end if
     end if

     if minutes >= 5 then
          clear_screen()
          halt_program = 'q'
     end if
end while
```



```
Program run time: 00:25

Press Q To Quit Demo Or Let Run For 5 Minutes
```

0.055 seconds resolution is a very small fraction of time, too small to perceive, and is fine enough to serve your programming purposes.

However, for programming that requires a very fine resolution of time, 0.055 seconds may be too large. If you want to measure time in even smaller steps, you have to force the operating system to fetch the time more often per second. By default, the time is checked 18.2 times per second, here is a library routine that forces more checking per second and thus giving you a finer resolution of time:

```
include machine.e
tick_rate(i)  [Note: DOS32 only]
```

i represents the number of interrupts per second. An interrupt is a pause where the operating system stops to go get something, such as a keystroke, or, in this case, the time. Because computers run very fast, the pause is unnoticeable. The higher i is in value, the more often the time is checked, and the finer the time resolution will become.

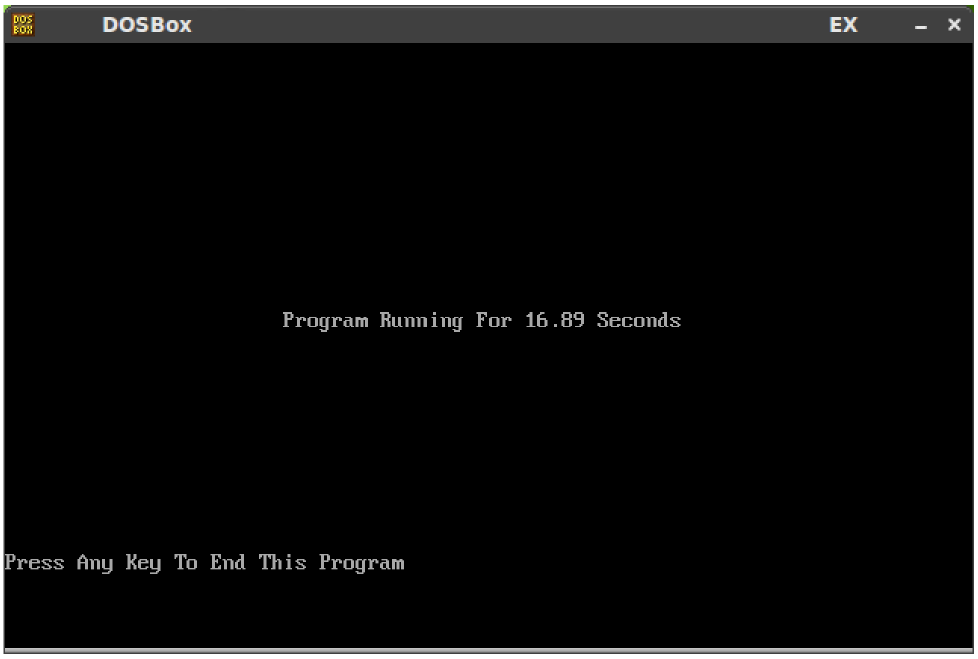
To figure out how to get the resolution you want, simply divide one by the desired time resolution. For example, if you want to have time advance by 0.025 seconds, you divide one by 0.025, giving 40. This means the operating system must check the time 40 times a second to get a resolution of 0.025 seconds, so you issue tick_rate(40).

The time resolution always resets back to 0.055 when a Euphoria program stops running (normally or by any error). You can issue tick_rate(0) to set it back to 0.055 if you want the time resolution back to default without ending the program run. While you can set the time resolution as fine as you want, you cannot change the time resolution to something more than 0.055 seconds. If the program encounters a “causeway” error, or if your system locks up, you should reboot your computer immediately to set the time resolution back to normal, or it will run too quickly.

A demo program is available to show how to properly use tick_rate()

program 70

```
include machine.e
atom precision_rate, ticker, seconds
precision_rate = .01
ticker = 1/precision_rate
tick_rate(ticker)
clear_screen()
position(22,1)
puts(1,"Press Any Key To End This Program")
seconds = 0
while get_key() = -1 do
     position(12,24)
     seconds = time()
     printf(1,"Program Running For %03.2f Seconds",seconds)
end while
clear_screen()
```



```
                       Program Running For 16.89 Seconds




Press Any Key To End This Program
```

dir() contains information on either a single file or a directory containing files (s). The information returned is a sequence value that is stored in receiving variable ro. dir() works exactly like the DOS command DIR.

If s is a directory name, a shortcut path like .. (parent directory) or . (current directory) or a wildcard filename like *.COM, the sequence returned is made up of sequence elements, where each element represents information on a file or subdirectory.

If s is a filename, like DAVID.GIF, or any wildcard filename that matches only one file, the sequence returned is made up of a single sequence element, representing information on that one file. Here is the structure of the returned sequence:

```
{{s1, s2, i3, i4, i5, i6, i7, i8, i9},
 {s1, s2, i3, i4, i5, i6, i7, i8, i9},...}
```

Each sequence element is in turn made up of 2 sequences (s1 and s2) and seven integer elements (i3 to i9). Each is explained below:

s1 - File or directory name	s2 - Attribute(s)
i3 - Size in bytes	i4 - Year modified
i5 - Month modified	i6 - Day modified
i7 - Hour modified	i8 - Minute modified
i9 - Second modified	

The attributes element (s2) contains a list of single characters that describes the file or directory (s1). Each of the characters are described below:

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

It's possible to have the attributes as a null sequence or {}, meaning a nonsystem file that is erasable, visible, and unchanged.

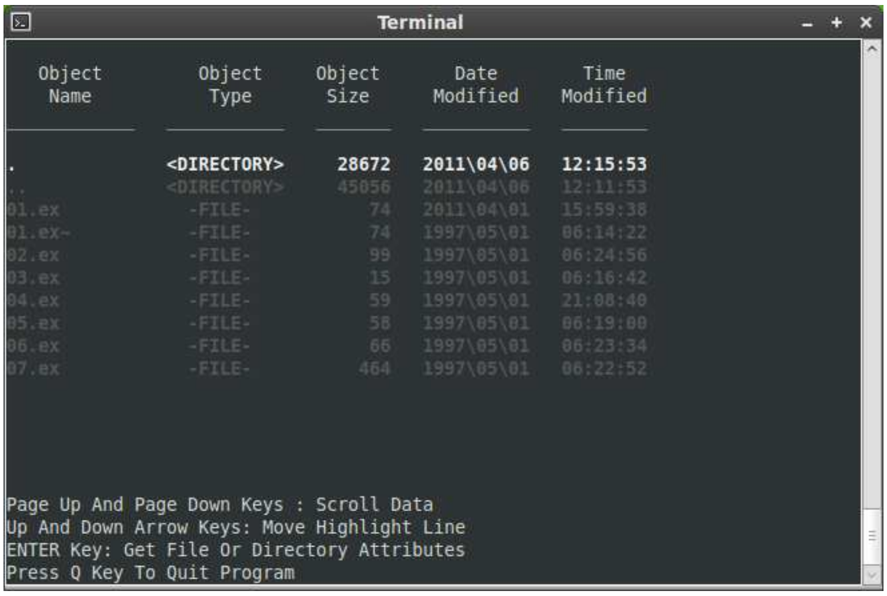
While you can pass a Windows 95 long file or directory name to dir(), the file and directory names returned are in the DOS 8.3 format. If the file or directory name passed into dir() does not exist or is invalid, the library routine will return a value of -1. A demo program is available to demonstrate the use of `dir()` to list directory files.

Under Unix, the keycodes are incorrect, so the program only shows the first screen.

program 71

```
include file.e
include graphics.e

sequence entry_type, format_string

integer keystroke, update, display_from, length_window, current_entry
object cur_dir_info

clear_screen()

position(21,1)
text_color(7)
puts(1,"Page Up And Page Down Keys : Scroll Data ")
position(22,1)
text_color(7)
puts(1,"Up And Down Arrow Keys: Move Highlight Line ")
position(23,1)
text_color(7)
puts(1,"ENTER Key: Get File Or Directory Attributes ")
position(24,1)
text_color(7)
```

```
puts(1,"Press Q Key To Quit Program")

position(2,1)
puts(1,
"   Object        Object     Object      Date        Time  \n")

puts(1,
"    Name           Type       Size      Modified    Modified\n")

puts(1,
"____________   ___________   _______   __________   ________\n")
format_string = "%-12s   %11s   %7d   %04d\\%02d\\%02d   %02d:%02d:%02d\n"
display_from = 1
length_window = 10
cur_dir_info = dir(".")
current_entry = 1
update = 'y'
keystroke = 0
while keystroke != 'q' do
     keystroke = get_key()

     if keystroke = 13 then
          position(18,1)
          puts(1,repeat(' ',78))
          if length(cur_dir_info[current_entry][2]) > 0 then
               position(18,1)
               text_color(7)
               puts(1,"Entry Attributes: ")
              for attribz = 1 to length(cur_dir_info[current_entry][2]) do
                    puts(1,cur_dir_info[current_entry][2][attribz])
                    puts(1, " ")
               end for
               text_color(8)
          end if
     end if

     if keystroke = 337 then
          if (display_from + length_window - 1) <
              length(cur_dir_info) then
               display_from = display_from + 1
               current_entry = display_from
               update = 'y'
          end if
     end if

     if keystroke = 329 then
          if display_from > 1 then
               display_from = display_from - 1
               current_entry = display_from
               update = 'y'
          end if
     end if

     if keystroke = 336 then
          if current_entry < length(cur_dir_info) and
               current_entry < (display_from + length_window - 1) then
               update = 'y'
               current_entry = current_entry + 1
          end if
     end if

     if keystroke = 328 then
          if current_entry > 1 and current_entry > display_from then
               update = 'y'
               current_entry = current_entry - 1
          end if
     end if
     if update = 'y' then

          update = 'n'
          position(6,1)
          for line = display_from to (display_from + (length_window-1)) do
               if line <= length(cur_dir_info) then
                    if find('d',cur_dir_info[line][2]) then
                         entry_type = "<DIRECTORY>"
                    else
                         entry_type = "  -FILE-    "
                    end if
                    if current_entry = line then
                         text_color(15)
                    else
                         text_color(8)
                    end if
                    printf(1,format_string,{cur_dir_info[line][1],
                                            entry_type,
                                            cur_dir_info[line][3],
                                            cur_dir_info[line][4],
                                            cur_dir_info[line][5],
                                            cur_dir_info[line][6],
                                            cur_dir_info[line][7],
                                            cur_dir_info[line][8],
                                            cur_dir_info[line][9]})
               else
                    puts(1,repeat(' ',78) & "\n")
               end if
          end for
     end if
end while

clear_screen()
```



```
   Object          Object      Object      Date        Time
    Name            Type        Size     Modified    Modified

____________    ___________  _______  __________  ________

.               <DIRECTORY>    28672  2011\04\06  12:15:53
..              <DIRECTORY>    45056  2011\04\06  12:11:53
01.ex             -FILE-          74  2011\04\01  15:59:38
01.ex~            -FILE-          74  1997\05\01  06:14:22
02.ex             -FILE-          99  1997\05\01  06:24:56
03.ex             -FILE-          15  1997\05\01  06:16:42
04.ex             -FILE-          59  1997\05\01  21:08:40
05.ex             -FILE-          58  1997\05\01  06:19:00
06.ex             -FILE-          66  1997\05\01  06:23:34
07.ex             -FILE-         464  1997\05\01  06:22:52



Page Up And Page Down Keys : Scroll Data
Up And Down Arrow Keys: Move Highlight Line
ENTER Key: Get File Or Directory Attributes
Press Q Key To Quit Program
```

```
   Object           Object       Object      Date        Time
    Name             Type         Size      Modified    Modified
____________     ___________   _______   __________  ________

.                <DIRECTORY>    24576    2011\03\30  11:10:44
..               <DIRECTORY>    32768    2011\03\30  11:07:35
01.ex              -FILE-          74    1997\05\01  06:14:22
02.ex              -FILE-          99    1997\05\01  06:24:56
03.ex              -FILE-          15    1997\05\01  06:16:42
04.ex              -FILE-          59    1997\05\01  21:08:40
05.ex              -FILE-          58    1997\05\01  06:19:00
06.ex              -FILE-          66    1997\05\01  06:23:34
07.ex              -FILE-         464    1997\05\01  06:22:52
08.ex              -FILE-         550    1997\05\01  06:46:00




Page Up And Page Down Keys : Scroll Data
Up And Down Arrow Keys: Move Highlight Line
ENTER Key: Get File Or Directory Attributes
Press Q Key To Quit Program
```

If you want to know what directory you are currently in, you can use the following library routine:

```
include file.e
rs = current_dir()
```

`current_dir()` returns a sequence value representing the current working directory. If you were running a Euphoria program in directory "C:\STUFF", and you issue `current_dir()`, the value returned to receiving variable rs would be "C:\STUFF". You could then pass this sequence value to `dir()` to obtain directory information.

A short demo is available to show how `current_dir()` works:

program 72

```
include file.e
sequence where_am_i
where_am_i = current_dir()
puts(1,"Hello!\n")
printf(1,"This demo runs from directory %s\n",{where_am_i})
```

```
mint@mint ~/Desktop $ eui 72
Hello!
This demo runs from directory /home/mint/Desktop
```

Having the ability to access all DOS commands and programs from a Euphoria program means having access to nearly all features of your computer. This gives the program incredible scope beyond the limits of the language. This library routine allows such access:

```
system(s,i)
```

`system()` will pass a string, s, representing a command for DOS to execute for you. It can be DOS command like cd, dir, rename, or deltree. The string can also be the name of a program, either written in Euphoria or in another programming language. i handles the kind of return to the Euphoria program once the command is finished running:

- 0 —clear the screen by restoring graphics mode of Euphoria program
- 1 —beep, wait for key press, and then restore graphics mode
- 2 —do not restore graphics mode

Be careful with option 2 as the choice of return. It should only be used when the graphics mode is not going to be changed by the DOS command.

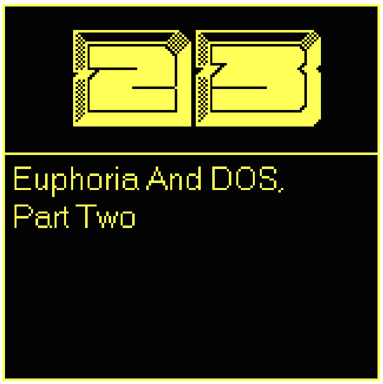
`system()` can be used to design Euphoria programs that install software, or to handle handle the programs on your computer using a flexible menu. A demo program uses `system()` to access the DOS command TYPE, in order to display your AUTOEXEC.BAT file.

program 73

```
system("type C:\\autoexec.bat | more ",2)
```

We will conclude our discussion of Euphoria and DOS in the next chapter by showing how to use DOS to control the execution of a Euphoria program, and also how to end the program in more ways than one.

23. Euphoria And DOS, Part Two



The previous chapter showed how a Euphoria program accessed the operating system for needed resources. But it is also possible to have DOS influence the way a Euphoria program runs, such as passing values to the program upon startup, like the way you pass the drive letter to FORMAT to indicate which drive to format. You can also control the way a Euphoria program terminates, even when the termination is a result of a program error.

Just as library routines can accept parameters to process, a program can accept parameters from the user in order to operate in a certain way based on the received values. For example, when you use XCOPY to copy files from one part of your hard drive to another (or to a floppy disk), you can state if you want subdirectories to be copied as well.

Even though it looks like a different kind of program is running when program parameters are used, it's really the same program running in a slightly different way.

To allow your Euphoria programs to accept parameters from the MS-DOS prompt, or from the Run window in Windows, you use this library routine:

```
rs = command_line()
```

`command_line()` returns the string used to start your Euphoria program, along with any parameters following the program name. This line is stored as a sequence value in receiving variable rs.

The sequence value returned by `command_line()` is made up of sequence elements, each element representing a word in the string used to start the Euphoria program.

`command_line()` works whether you run your Euphoria program through the interpreter EX.EXE, or as a stand-alone file .EXE created by BIND.BAT. But the returned sequence value will differ based on the method used.

If the Euphoria program is run by EX.EXE, the sequence value is:

```
{the EX.EXE file name (including the directory where it is stored),
 the name of your Euphoria program being run,
 the first parameter the program accepts,...}
```

If the Euphoria program is an .EXE file, the sequence value is:

```
{the Euphoria program name (including the directory where it is stored),
 the Euphoria program name (including the directory where it is stored),
 the first parameter the program accepts,...}
```

When `command_line()` is used in a program that is stand-alone created by BIND.BAT, the first and second elements are the same value. This ensures the the parameters following the program name are in the same element positions, no matter how the Euphoria program is started.

Depending on the number of parameters following the program name, the sequence value returned by `command_line()` can be any length in terms of elements. If no parameters are entered after the program, the smallest the sequence value can be is two elements long.

It's the elements from the third position onward that you should focus attention on, as these are where the parameters are located. You can condition groups of statements to execute only when certain parameters are received by you program.

A demo program is available to show how to use `command_line()` in a Euphoria program, but one important note: you may view both the batch file source and the source of the Euphoria program the batch file runs and sends parameters to, but do NOT run the Euphoria program itself!

program 74

```
sequence command_line_data

atom number_of_parameters

clear_screen()

command_line_data = command_line()

number_of_parameters = length(command_line_data) - 2

if number_of_parameters = 0 then
     puts(1, "\nPlease run the demo BATCH file to execute this program\n")
else
     printf(1, "\n%d parameter(s) were passed to this program\n\n",
            {number_of_parameters})
     for ix = 3 to length(command_line_data) do
          printf(1, "%s is parameter %d\n", {command_line_data[ix],
                                              ix-2})
     end for
end if
```

Here's the associated batch file:

Type the contents of the “bat” file to see the following result.

```
# demo/74.bat

@PP

74.bat
```

```
4 parameter(s) were passed to this program

cats is parameter 1
dogs is parameter 2
budgies is parameter 3
mice is parameter 4
```

Another way of passing parameters to your program is by accepting the values of environment variables in DOS. You are probably familiar with the PATH variable (where the operating system searches for a program if it is not found in the current directory you are in).

```
ro = getenv(s)    [Note: DOS32 only]
```

The value assigned to the environment variable, shown here as s, is stored as a sequence in receiving variable ro. If the environment variable has no assigned value, then a value of -1 is returned instead. Run a demo program now that gets information on the PATH variable.

program 75

```
object path_settings
clear_screen()
path_settings = getenv("PATH")
if sequence(path_settings) then
     puts(1,
     "\nThe following directories are in your DOS PATH variable:\n\n")
     for ix = 1 to length(path_settings) do
          if path_settings[ix] = ';' then
               puts(1, "\n")
          else
               puts(1, path_settings[ix])
          end if
     end for
     puts(1, "\n\nScan completed. Have a nice day!\n")
else
     puts(1, "\nNo variable PATH found. You really should set the path\n")
     puts(1, "variable in DOS. It will allow you to run programs in\n")
     puts(1, "different directories without typing the full path name!\n")
end if
```

Executed on a Unix system.

```
The following directories are in your DOS PATH variable:

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

Scan completed. Have a nice day!
```

Contolling the execution of a Euphoria program includes how it terminates. So far, the programs you have seen ended after the last statement was executed. Sometimes it is necessary to halt the program when something it requires is missing (such as a file) rather than continue onward. To do this, you would either need to segment your program as conditioned groups of code, each dependant on the outcome of processing from previous groups, or you can use this very simple library routine:

```
abort(i)
```

When executed, the Euphoria program immediately stops, and the value i is returned to the operating system. A batch file could use the returned value to proceed based on how the Euphoria program ended. While it is entirely up to you to how to assign meanings to the values returned by `abort()`, most programmers define 0 to mean the program ended normally.

The beauty of `abort()` is that the program ends quickly and cleanly immediately after it is executed, no matter how deep in the program's execution you are in. It also allows the program to communicate why it ended so you know what is wrong, not to mention forcing you to design programs to be able to handle all possible conditions. A demo program shows how `abort()` is used in conjuction with a batch file.

program 76

```
sequence parms, workarea

integer no_of_parms, bad_first, bad_second

parms = command_line()

no_of_parms = length(parms) - 2

if no_of_parms < 2 then
     abort(1)
else
     workarea = parms[3]
     bad_first = 0
     bad_second = 0
     for ix = 1 to length(workarea) do
          if workarea[ix] < '0' or workarea[ix] > '9' then
               bad_first = 1
               exit
          end if
     end for

     workarea = parms[4]
     for ix = 1 to length(workarea) do
          if workarea[ix] < '0' or workarea[ix] > '9' then
               bad_second = 1
               exit
          end if
     end for

     if bad_first then
          abort(2)
     end if
     if bad_second then
          abort(3)
     end if

     puts(1, parms[3] & parms[4] & "\n\n")

     abort(0)
end if
```

Here's the associated batch file:

```
@echo off
ex d2307a.ex %1 %2
if errorlevel 3 goto err3
if errorlevel 2 goto err2
if errorlevel 1 goto err1
if errorlevel 0 goto err0
:err3
echo Second parameter is non-numeric
goto finished
:err2
echo First parameter is non-numeric
goto finished
:err1
echo Two numbers required to join
goto finished
:err0
echo Program completed normally
:finished
```

Despite all the best planning and all the possible contingencies you have imagined, no program is perfect. There will come a time when you program will encounter an error and it will stop abruptly because of it. Normally when this happens, a file called EX.ERR is generated containing details about the problem, Also, there previous graphics mode you were in before starting the program is not restored, making characters on the screen unreadable. While a Euphoria programmmer can just look inside the EX.ERR file and find out what went wrong, suppose this program was run by a person you wrote it for? or worse, sold the program to!!!

So with the person running your program staring at the screen that either has cryptic programming diagnostics, or is totally unreadable, you can believe he or she is going to be worried about what to do next. A more appropriate way is to have the program send a screen message explaining what to do next, and who to contact for assistance. The message should also be in easy-to-understand terms. Euphoria has a library routine that sets a screen message to be displayed in case of program failure:

```
include machine.e
```

```
crash_message(s)
```

crash_message() does not display a message, s, on the screen when it is executed. Rather, it simply tells Euphoria what to display in case a syntax error (such as an undeclared variable name) occurs, and also for errors that occur during program execution, like trying to divide by zero or using invalid element numbers. These are called run-time errors. You can format the message using special characters like " \ n " or " \ t ", to give it a certain appearance. When an error occurs, the graphics mode is set to text mode before your message is displayed. It will appear at the top of the screen. You can issue crash_message() as many times as needed, but the message of the most recent crash_message() will be the one that appears if an error occurs. Still, you may want to issue crash_message() every time a section of your program begins running if each section requires different handling instructions.

Euphoria always generates an EX.ERR file whether or not you use crash_message(). Your crash_message() should include a note to send the EX.ERR file to you for analysis. A demo program is available to show how crash_message() in a divide by zero situation.

program 77

```
include machine.e

atom result

crash_message("**************************************\n"&
              "* An error has been encountered that *\n"&
              "* is so serious the software must    *\n"&
              "* stop running now.                  *\n"&
              "*                                    *\n"&
              "* Please Email the file ex.err to    *\n"&
              "* moggie@interlog.com. Thank you!    *\n"&
              "**************************************\n")



for ix = 100 to 0 by -1 do
    result = 100 / ix
    printf(1,"%d divided by %d gives %f\n",{100,ix,result})
end for
```

```
**************************************
* An error has been encountered that *
* is so serious the software must    *
* stop running now.                  *
*                                    *
* Please Email the file ex.err to    *
* moggie@interlog.com. Thank you!    *
**************************************
```

A final way to make DOS control the way a Euphoria program runs is by use of DOS' redirection symbols. Earlier in the tutorial, we stated that the number 0 was by default defined for keyboard input, and 1 and 2 were by default assigned for screen output. By using the following DOS redirection symbols below, the source of input (0) and destination for output (1 and 2) can be changed:

```
euphoria program < input file or device

euphoria program > output file or device
```

The DOS redirection symbol < means keyboard input library routines will read data from an **input file or device** other than the keyboard. The DOS redirection symbol > means (text) screen output library routines will send data to an **output file or device** other than the computer screen.

The use of the `open()` library routine in conjuction with input and output library routines is more effective than using the DOS redirect symbols. However, DOS redirect symbols serve as a handy "ad-hoc" way to force your program to handle input and output that do not involve the keyboard and screen. Your operating system manual contains more details on how to use redirection symbols and other DOS features to change data flow to and from your written programs.

Go to the next chapter now to learn about the binary number system and how it can be used with the Euphora programming language!

24. An Introduction To Bits And Bytes

If you remember from our discussion on variables, we introduced the byte, a stored value between 0 and 255. You may have asked yourself, “why 255? why not a much easier to remember range like 0 ot 10, or 0 to 100, as in the metric system?” Well, remember that the only language a computer understands is a set of instructions composed of 1’s and 0’s, or binary language. If so, one would expect that computers would use a similar standard composed of 1’s and 0’s to represent numbers.

Because computers only understand the values of 0 and 1, they use a system called the “binary system” to represent stored numbers. To understand how it works, we need to go back to school to review how humans represent numbers.

Humans use a numbering system called the decimal system, because we can relate to groups of 10 easily (no doubt because we have 10 fingers). The decimal system states each digit is the number of groups of 10, where each group is raised to a power based on the position from right to left:

```
                           5   4   1   3
5 groups of 1000 (103)-----^
4 groups of  100 (102)---------^
1 group of    10 (101)-------------^
3 groups of     1 (100)-----------------^
```

The exponent that raises 10 to a power starts at 0 from the right-most digit, and increases by a value of 1 as you go left.

With computers using the binary system, a base of 2, not 10, is used. This is because the computer only uses two digits in its numbering system. Yet the representation of a number under the binary system is the same as in the decimal system:

```
                           1  0  1  1  0
1 group of 16 (or 24)------^
0 groups of 8 (or 23)---------^
1 group of  4 (or 22)------------^
1 group of  2 (or 21)---------------^
0 groups of 1 (or 20)------------------^
```

For your information, the value 10110 is pronounced "one - zero - one - one - zero," not "ten-thousand-one-hundred-ten." The terms "thousand," "hundred" and "ten" describe groups in the decimal system.

The binary number 10110 is equal to 22. You can determine the value of any binary number by adding up all the powers of 2 represented by the binary digit 1, also known as a BIT (BInary digiT). For example, 10100 is equal to 2 + 4 +16, which equals 22.

You can also convert any decimal number to binary by following these instructions (you will need a calculator and a sheet of paper for this):

- Divide number n by 2.
- If the result of the division ends in .5, write down the value 1 on the sheet of paper, and then change the result to an integer (for example, 12.5 to 12). Otherwise, just write down the value 0.
- Take the results and make it the next number n to divide by 2, and go to step 1. Repeat these steps until you produce a value less than 1. When you are done, reverse the digits written on the paper.

For example, the value of 23 is 10111. We work it out by dividing 23 by 2 to get 11.5 (1), 11 by 2 to get 5.5 (1), 5 by 2 to get 2.5 (1), 2 by 2 to get 1 (0), 1 by 2 to get 0.5 (1). You then reverse the digits produced to change 11101 to 10111.

These bits are important when we talk about bytes. A byte, as defined by the American Standard Code for Information Interchange, is made up of 8 bits:

Binary	Decimal	Binary	Decimal	Binary	Decimal
00000000	0	00000101	5	00001010	10
00000001	1	00000110	6	:	:
00000010	2	00000111	7	11111101	253
00000011	3	00001000	8	11111110	254
00000100	4	00001001	9	11111110	255

It is at this point you understand why a byte has a value beween 0 and 255. 0 equals 00000000 and 255 equals 11111111.

There are two problems with this. The first problem is the largest value a byte can hold is 255. In order to represent larger values, you have to use more than one byte. So, putting two bytes together makes a what is called a word. Using 16 bits, a word can represent values between 0 and 65536. There is also a double word, which is made up of four bytes (32 bits) and can represent extremely large values.

The other problem is that the values bytes, words, and double words can represent are unsigned, or positive only. This problem is solved by two approaches. First, the leftmost bit position is used only to show if the number is negative or positive. Second, a method called two's complement is used to represent negative numbers.

Here is how a binary number is shown as negative using two's complement on a binary number 00010010 (34):

- First 00100010 is reversed to 11011101 (called one's complement)
- You then add binary 1 to the reversed value:

```
11011101
+00000001
--------
11011110  (-34 in decimal)
```

You'll notice that addition using binary numbers follows the same rules as with decimal numbers: when adding two numbers produces a sum larger than a single digit, you carry left to the next column. Adding 1 and 1 produces 10, where the 1 is carried over to the next column to the left.

The use of two's complement to store negative values works with 8 bit, 16 bit, or 32 bit values. The only disadvantage of using two's complement is that the range of numbers supported drops for positive numbers. For example, a byte that can handle negative numbers now only represents numbers between -128 and +127, and a word can only represent numbers between -32768 and +32767, both inclusive. however, it is up to the programmer to decide whether or not to have signed values.

The understanding of how bits and bytes work is not mandatory in order to write programs in Euphoria. After all, Euphoria handles the storage and manipulation of binary values behind the scenes so you really do not see any use of bits, bytes, words, and double words, nor do you have to convert numbers to two's complement if you want them negative.

So you probably ask, "what is the point of learning about bits and bytes if Euphoria handles all that for me automatically?"

Well, in addition to getting a better feel on how the computer really stores data, there are other benefits. First, because 8 bits make up a byte, you could use single byte to keep track of 8 different conditions in your program, where each 1 bit means a condition is true. Also, if you are interested in data compression, using less than 8 bits to represent values would be helpful.

A good understanding of bits and bytes is also handy in graphics, where you want to merge images together, without any black area around either image being merged.

This is rough territory for the person who has never programmed. If you are not ready to learn the Euphoria library routines that handle bits and bytes, use the remote to skip to "Creating Library Routines And Variable Types." Otherwise, just go to the next page!

25. Working With Bits

In this chapter, we will dig deeper into the theory of bits by actually using them in Euphoria programs. There are library routines that can convert integer values to a sequence representing binary values and back. Also, you will learn how boolean logic works with values at the binary digit level. You are already familiar with boolean logic when you learned about logical expressions. This chapter will expand a bit on this in order to handle bits.

If you plan to manipulate values at the bit level, probably for use as condition switches in your program, you need to actually see them. The best way to do this is have them shown as a sequence, where each element is an atom having a value of 1 or 0. This way, you can perform element indexing of single bits or an entire range of them.

Here is the library routine that lets you access bits in integer values:

```
include machine.e
rs = int_to_bits(a,i)
```

This returns a sequence value containing the rightmost number of bits (i) in integer value a, to be stored in receiving variable rs. The sequence is made up of atom elements representing bit values starting from the right. We use a instead of i as the integer value in case you want to work with integer values outside the range of -1073741824 and +1073741823. Only atom data objects can hold values outside that range.

The returned sequence value is actually reversed in appearance, because the rightmost bits start at element 1. For example, bit 20 is element 1, bit 21 is element2, bit 22 is element 3, and so forth.

`int_to_bits()` will return the rightmost bits of negative numbers too. Just remember that negative numbers always use the two's complement format.

The number of bits parameter depends on the size of the integer value you are accessing for bits. A byte-sized integer only needs to have a maximum of 8 bits returned, while word and double-word sized integers will require you to go as high as 16 or even 32 bits to return. A demo program shows how to return bits from different integer values.

program 78

```
include graphics.e
include image.e
atom blue, increment, status
sequence palette_mode,new_blue, previous_blue
blue = 30
increment = 1
if graphics_mode(18) then
     puts(1,"Unable to go into mode 18!")
else
     palette_mode = get_all_palette()
     position(1,3)
     text_color(11)
     puts(1,"How to build a bitmap using Euphoria, Part II")
     position(24,5)
     puts(1,"Press any key to continue demo")
     for ix = 0 to 50 by 4 do
         ellipse(2,0,{100+ix,100-ix},{199-ix,199+ix})
         ellipse(2,0,{100-ix,100+ix},{199+ix,199-ix})
         polygon(1,0,{{50,50},{50,249},{249,249},{249,50}})
     end for
     position(20,1)
     puts(1,"Define an area on the screen you want to save")
     while get_key() = -1 do
          if blue = 63 then
               increment = -3
          elsif blue = 30 then
               increment = 3
          end if
          blue = blue + increment
          new_blue = {0,0,0}
          new_blue[3] = blue
          previous_blue = palette(1,new_blue)
     end while
     all_palette(palette_mode)
     polygon(0,0,{{50,50},{50,249},{249,249},{249,50}})
     status = save_screen({{50,50},{249,249}},"d2109a.bmp")
     polygon(0,1,{{50,50},{50,249},{249,249},{249,50}})
     clear_screen()
     puts(1,"save_screen() produces the same result as save_bitmap()\n")
     puts(1,"but without the steps shown in the save_bitmap() demo.\n")
     while get_key() = -1 do
     end while
     if graphics_mode(-1) then
          puts(1,"Unable to reset mode!")
     end if
end if
```

The opposite of this is to take a binary number and convert it into an integer value. The is approach might be taken when you are using bits to represent a list of conditions, and for efficient storage want to bundle them all into a single integer value.

To convert a binary number into an integer value, you use the following library routine below:

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

`bits_to_int()` takes sequence s representing a binary number and converts it to a positive integer value, which is stored in receiving variable ra. s is made up of atom elements that represent the bits of the binary number. Each element is either 0 or 1 in value.

The elements in s representing bits appear in reverse order, where element 1 is the rightmost bit. For example, element 1 is bit 20, element 2 is bit 21, element 3 is bit 22 and so forth.

The receiving variable is an atom and not an integer for the same reason mentioned in our discussion with `int_to_bits()`. You may want to produce integer values beyond the range of -1073741824 and +1073741823. Only atom variables can support integers beyond that range.

You will notice that `bits_to_int()` only produces positive integer values. This is because the leftmost bit (the last element in the sequence) is assumed to be a part of the integer value, and not a sign bit. This shouldn't be a problem, as there wouldn't be a reason to convert a list of binary digits arranged in two's complement format if you are using each bit as an outcome of a condition test.

A demo program is available to show how `bits_to_int()` us used to store bit patterns into a single byte value.

program 79

```
include graphics.e
include image.e
include get.e
integer file_id
sequence capture_buffer, video_data

video_data = video_config()

clear_screen()
printf(1, "%d pages available, 5 pages required\n", {video_data[8]})
if video_data[8] < 5 then
     puts(1, "Sorry, you have insufficient pages on your video card\n")
else
     puts(1, "Stand By, Loading Each Virtual Page\n")

     set_active_page(1)
     file_id = open("now.bin","rb")
     capture_buffer = get(file_id)
     close(file_id)
     display_text_image({1,1}, capture_buffer[2])

     set_active_page(2)
     file_id = open("this.bin","rb")
     capture_buffer = get(file_id)
     close(file_id)
     display_text_image({1,1}, capture_buffer[2])
     set_active_page(3)
     file_id = open("is.bin","rb")
     capture_buffer = get(file_id)
     close(file_id)
     display_text_image({1,1}, capture_buffer[2])

     set_active_page(4)
     file_id = open("ppower.bin","rb")
     capture_buffer = get(file_id)
     close(file_id)
     display_text_image({1,1}, capture_buffer[2])

     set_active_page(0)

     clear_screen()
     puts(1, "Done....to cycle through all the four pages, press 'n'.\n")
     puts(1, "Press any key to start cycling now.\n")
     while get_key() = -1 do
     end while

     for ix = 1 to 4 do
          set_display_page(ix)
          while get_key() != 'n' do
          end while
     end for

     clear_screen()
end if

set_active_page(0)
set_display_page(0)
```

You can also reference one or more bits by a process called "masking." Masking involves comparing a value (let's call it "A") against a second value (let's call it "B") in such a way that B's bit pattern either obtains or filters out specific bits in A. B is called the mask value. Boolean logic at the bit level is used in masking. There are three types of masks, with the first two being shown below:

```
11110000 - value          11010010 - value
00001111 - OR mask        01111110 - AND mask
--------                  --------
11111111 - OR result      01010010 - AND result
```

In an OR mask, the result bit positions only contain 1 if the value, the mask, or both, have matching bit positions containing 1. In and AND mask, the result bit positions only contain 1 if both the value and the mask have matching bit positions containing 1.

The third type of mask is called XOR (rhymes with "sore" but starting with a "Z" sound). It can best be described as a cross between an OR and something like a backwards AND where two 1 bits result in a 0. Here's how it works below:

```
11011100 - value
00011100 - XOR mask
--------
11000000 - XOR result
```

In an XOR mask, the result bit positions only contain 1 when either the value or the mask (but not both!) have matching bit positions containing 1.

With masks now understood, let's show some library routines that perform AND, OR and XOR bit operations in Euphoria.

```
ro = and_bits(o1,o2)
```

`and_bits()` performs AND operations using values o1 and o2 to create a result that is stored in receiving variable ro. The bits in the result stored in ro will only be 1 if the matching bit positions in o1 and o2 are both 1. o1 and o2 can be atoms or sequences containing atom elements.

If o1 is an atom and o2 is a sequence (or vice-versa), then the rule of mixing atoms and sequences in a binary expression (where the atom value is converted to a sequence having the same length as the other sequence value, and made up of elements having the value ofthe original atom) applies. `and_bits()` can handle any accepted values up to and including 32 bits in size.

The result produced by `and_bits()` may be a negative value if the process causes the leftmost bit to be set to 1. This is because the leftmost bit is considered a sign bit. A demo program shows how `and_bits()` works with some atoms and sequences.

program 80

```
include machine.e
atom precision_rate, ticker, seconds
precision_rate = .01
ticker = 1/precision_rate
tick_rate(ticker)
clear_screen()
position(22,1)
puts(1,"Press Any Key To End This Program")
seconds = 0
while get_key() = -1 do
     position(12,24)
     seconds = time()
     printf(1,"Program Running For %03.2f Seconds",seconds)
end while
clear_screen()
```

```
   ro = or_bits(o1,o2)
```

`or_bits()` performs OR operations using values o1 and o2 to create a result that is stored in receiving variable ro. The bits in the result stored in ro will only be 1 if the matching bit positions in either o1, o2, or both, is a value of 1. o1 and o2 can be atoms or sequences containing atom elements.

if o1 is an atom and o2 is a sequence (or vice-versa), then the rule of mixing atoms and sequences in a binary expression (where the atom value is converted to a sequence having the same length as the other sequence value, and made up of elements having the value of the original atom) applies. `or_bits()` can handle any accepted values up to and including 32 bits in size.

The result produced by `or_bits()` may be a negative value if the process causes the leftmost bit to be set to 1. This is because the leftmost bit is considered a sign bit. A demo program shows how `or_bits()` works with some atoms and sequences.

program 81

```
include file.e
sequence where_am_i
where_am_i = current_dir()
puts(1,"Hello!\n")
printf(1,"This demo runs from directory %s\n",{where_am_i})
```

```
ro = xor_bits(o1,o2)
```

`xor_bits()` performs XOR operations using values o1 and o2 to create a result that is stored in receiving variable ro. The bits in the result stored in ro will only be 1 if either of the matching bit positions in o1 and o2 are 1, but not both. o1 and o2 can be atoms or sequences containing atom elements.

If o1 is an atom and o2 is a sequence (or vice-versa), then the rule of mixing atoms and sequences in a binary expression (where the atom value is converted to a sequence having the same length as the other sequence value, and made up of elements having the value of the original atom) applies. `xor_bits()` can handle any accepted values up to and including 32 bits in size.

The result produced by `xor_bits()` may be a negative value if the process causes the leftmost bit to be set to 1. This is because the leftmost bit is considered a sign bit. We've modified the `or_bits()` demo program to use `xor_bits()` instead, to show the one difference between XOR and OR.

program 82

```
system("type C:\\autoexec.bat | more ",2)
```

The last bit-handling library routine for this chapter is listed below:

```
ro = not_bits(o)
```

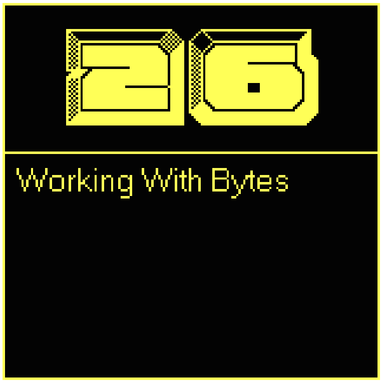
not_bits() reverses each bit in value o to its opposite state (for example, 1 to 0 or 0 to 1). o may be an atom value, or a sequence made up of atom elements. not_bits() can handle accepted values up to and including 32 bits. The inverted result is stored in receiving variable ro. If the leftmost bit is changed to 1, the value placed in ro will be negative because this bit is the sign bit. A demo program is ready to show how not_bits() works with sequence and atom values.

program 83

```
sequence command_line_data

atom number_of_parameters

clear_screen()

command_line_data = command_line()

number_of_parameters = length(command_line_data) - 2

if number_of_parameters = 0 then
     puts(1, "\nPlease run the demo BATCH file to execute this program\n")
else
     printf(1, "\n%d parameter(s) were passed to this program\n\n",
            {number_of_parameters})
     for ix = 3 to length(command_line_data) do
          printf(1, "%s is parameter %d\n", {command_line_data[ix],
                                               ix-2})
     end for
end if
```

The next chapter will show you how you can save numbers larger than 255 outside your program, whether it is an integer or a floating point number.

26. Working With Bytes



The `puts()` library routine is handy for sending character output to files and the screen. But the one drawback it has is that only byte values, or values between 0 and 255, can be used. Any attempt to send a larger value out to a file or screen will result in data loss. This occurs because only the lower 8 bits are sent. However, Euphoria has a set of library routines that convert large integer and even floating point numbers to a series of bytes. One option of sending values larger than 255 to the screen or file is to use the `print()` library routine. The end result, in the case of files, is an outputted character string (for example, -2453 is stored as a 5 byte string "-2543"). It works, but it is very wasteful in terms of data storage. So it stands to reason that if a value like 255 can be represented as a single byte, then values like 65535 can be represented using only two bytes rather than 5 bytes when using `print()`. Here is the library routine that can help you do this:

```
include machine.e
rs = int_to_bytes(a)
```

`int_to_bytes()` takes a signed integer value, a, and converts it into a 4 element long sequence representing 4 bytes. The integer is represented as a rather than i because `int_to_bytes()` works with 32 bit numbers, and only atom data objects can be that large. Integers are only 31 bits long. The 4 element sequence is returned to the receiving variable rs. Each element is a bundle of 8 bits, with the lowermost 8 bits (2 to the power of 0 to 2 to the power of 7) starting in the first element. To clarify, the structure of the 4-element sequence looks like this, with meanings shown for each element:

```
                 {byte, byte, byte, byte}
bits 20 to 27 ----^
bits 28 to 215----------^
bits 216 to 223----------------^
bits 224 to 231----------------------^
```

Once an integer is converted to a series of bytes, you can write each byte out to files using `puts()` without any risk of data loss. If you want to handle extremely large numbers that require 64 bits instead of 32 bits, then `int_to_bytes()` will return the lowermost 32 bits of these numbers.

So now you have a way to convert large integer values into a series of bytes. It would be just as handy to have the ability to take those same bytes and convert them back into the original integer value. Here is a library routine that can do this for positive integers:

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

`bytes_to_int()` takes a sequence value, s, representing a 32 bit number, and converts it to a positive integer. The positive integer value is then stored in receiving variable ra. ra is an atom because `bytes_to_int()` works with 32 bit long numbers. The sequence passed to `bytes_to_int()` is made up of 4 atom elements, where each element is a bundle of 8 bits, starting with the lowermost bits (bit 20 to 27) being the first element. The structure of the sequence is the same as introduced in `int_to_bytes()`. As a matter of fact, you can use the sequence generated by `int_to_bytes()` as a parameter for `bytes_to_int()` if you are working with positive numbers.

However, `bytes_to_int()` does not convert properly when you are trying to bring back a negative number previously into 4 bytes by `int_to_bytes()`. This does not mean, however, you cannot bring back the negative integer value. It means you will have to do a little extra work in order to bring it back.

To bring back a negative number previously converted by `int_to_bytes()`:

- Use int_to_bits() to convert the 4th element of the sequence created by int_to_bytes() into a 32 element sequence, and look at the 32nd element. If it is 1, you have a negative number.
- Use not_bits() to reverse the bits in the int_to_bytes() sequence.
- Use the result of not_bits() as the sequence you pass to bytes_to_int().
- Add 1 to the integer created by bytes_to_int(), then multiply the integer by -1. The integer should be the correct value.

Run a demo program now that uses `int_to_bytes()` and `bytes_to_int()` to store positive and negative integers to a file on your computer.

program 84

```
object path_settings
clear_screen()
path_settings = getenv("PATH")
if sequence(path_settings) then
     puts(1,
     "\nThe following directories are in your DOS PATH variable:\n\n")
     for ix = 1 to length(path_settings) do
          if path_settings[ix] = ';' then
               puts(1, "\n")
          else
               puts(1, path_settings[ix])
          end if
     end for
     puts(1, "\n\nScan completed. Have a nice day!\n")
else
     puts(1, "\nNo variable PATH found. You really should set the path\n")
     puts(1, "variable in DOS. It will allow you to run programs in\n")
     puts(1, "different directories without typing the full path name!\n")
end if
```

The numbers we have worked with have been integer values. Remember also that programs work with floating-point (numbers with a decimal) as well. If you remember from the start of the tutorial, we represent very large and very small numbers using standard notation:

```
6.13451e+009 (meaning 6.13451 Ã˙ 1000000000, or 6134510000)
4.52e-005    (meaning 4.52 Ã˙ .00001, or.0000452)
```

The decimal number being multiplied by the power of 10 is called the mantissa, and is never larger than 10. In the binary numbering system, there is no such thing as a decimal point, so you can't have numbers like 101111.01 for example. Instead, an organization in the U.S.A. called the Institute of Electrical and Electronic Engineers (IEEE) created a floating point standard that addresses this problem nicely. The floating point standard created by the IEEE comes in two sizes, on using 32 bits, and the other using 64 bits. Please note these formats are being introduced to you for your personal interest only:

```
1 bit      +     8 bits     +     23 bits     =     32 bits
(sign bit)       (exponent)       (mantissa)

1 bit      +     11 bits     +     52 bits     = 64 bits
(sign bit)       (exponent)        (mantissa)
```

Because Euphoria automatically handles how the exponent and mantissa portions are created and used in the representation of binary floating numbers, we will not go any further at this point. All you need to know is what the IEEE 32 bit and 64 bit floating point formats are when mentioned in the library routines you will learn next. If you are interested, there are FAQ's about IEEE floating points on the internet.

To convert a floating point number to a 4-byte (32 bit) IEEE format, you use the following library routine:

```
include machine.e
rs = atom_to_float32(a)
```

atom_to_float32() will convert a floating point value, a, to a 4 element long sequence value, each element being an atom. The sequence will be stored in receiving variable rs. The sequence value represents the 32 bit IEEE format introduced previously. a can be a negative or positive value, and can even be an integer value, though it will still be converted to the 32 bit IEEE floating point format. To convert the 4 element sequence back to the original value, you do the following:

```
include machine.e
ra = float32_to_atom(s)
```

To convert a floating point number to an eight-byte (64 bit) IEEE format you use the following library routine:

```
include machine.e
ra = atom_to_float64(a)
```

atom_to_float64() will convert a floating point value, a, to an 8 element long sequence value, each element being an atom. The sequence will be stored in receiving variable rs. The sequence value represents the 64 bit IEEE format introduced previously. a can be a negative or positive value, and can even be an integer value, though it will still be converted to the 64 bit IEEE floating point format. To convert the 8 element sequence back to the original value, you do the following:

```
include machine.e
ra = float64_to_atom(s)
```

The use of these powerful floating point library routines allows for efficient storage of floating point data in files. Once you use either atom_to_float32() or atom_to_float64(), you can use puts() to write each element of the sequence. When it is time to bring the data back from the file into the data, float32_to_atom() or float64_to_atom() can be used once all the bytes previously written out are obtained using the . library routine.

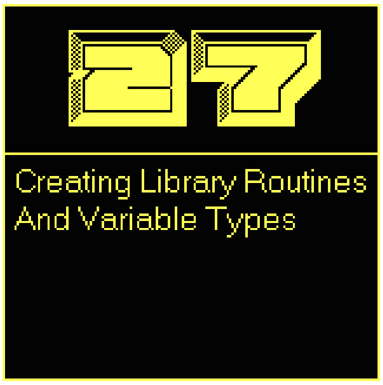
You should be careful not to use atom_to_float32() on floating point numbers that, because of size and accuracy, must use the IEEE 64 bit format. There is a risk you could lose data accuracy (if not part of the data value itself) if you are not careful about this. A demo program uses these 4 library routines to save data to a file.

program 85

```
sequence parms, workarea

integer no_of_parms, bad_first, bad_second

parms = command_line()

no_of_parms = length(parms) - 2

if no_of_parms < 2 then
     abort(1)
else
     workarea = parms[3]
     bad_first = 0
     bad_second = 0
     for ix = 1 to length(workarea) do
          if workarea[ix] < '0' or workarea[ix] > '9' then
               bad_first = 1
               exit
          end if
     end for

     workarea = parms[4]
     for ix = 1 to length(workarea) do
          if workarea[ix] < '0' or workarea[ix] > '9' then
               bad_second = 1
               exit
          end if
     end for

     if bad_first then
          abort(2)
     end if

     if bad_second then
          abort(3)
     end if

     puts(1, parms[3] & parms[4] & "\n\n")

     abort(0)
end if
```

The next chapter of this tutorial will show you how to create your own library routines and variable types!

27. Creating Library Routines And Variable Types



The library routines and variable types in Euphoria should be more than enough for anyone to write full-featured programs. However, some exceptions may arise where the programmer needs to design custom library routines and variable types. In addition, custom library routines can also help a programmer design programs in a modular manner, organizing very large programs into easily identifiable sections of code that the programmer can keep better track of. Another advantage of using custom library routines is that they can be used in other programs without having to re-invent the wheel every time you write a new program. Also, you can insert library routines created by other Euphoria programmers into your program. Custom library routines save time and effort in software creation!

You will remember from our introduction to library routines that library routines are either procedures or functions. As a result, you can either create a function or a procedure. Any library routines you create must be declared in the program, just like variables, before being used. When declaring a custom library routine, you must state the following:

- Whether the library routine is a procedure or a function.
- The name of the library routine.
- The number and type of parameters the library routine accepts. This is optional.
- The programming statements that will process the parameters accepted by the library routine.
- The value returned if the library routine is a function.

To declare a library routine that is a procedure, here is the syntax required:

```
procedure name(parameter declaration, parameter declaration, ...)
   one or more Euphoria statements to execute
end procedure
```

To declare a library routine that is a function, here is the syntax required:

```
function name(parameter declaration, parameter declaration, ...)
   one or more Euphoria statements to execute
   return value
end function
```

The custom functions and procedures you create will of course have a name, just as variables have a name. If your procedure or function accepts values, probably for the purpose of processing them, you need to have parameter declarations as well. Declaring parameters is the same as declaring variables, which makes sense. Parameters are in fact variables that accept the values passed to functions and procedures. The one or more Euphoria statements to execute is the actual muscle of the library routine, which can consist of assignment statements and other library routines, either supplied by Euphoria or created by other programmers. The end of the library routine is always terminated by a end procedure or end function. The return statement is used to end the library routine's execution the moment it is executed. In procedures, it's rare you would need it, unless you plan to end the procedure in midrun. However, in functions, it's needed, because the "`return`" statement returns a value (either an atom or a sequence) back to the program that called the function.

Here's the simplest example of a custom procedure:

```
procedure print_some_lines()
   puts(1,  " lines\n " )
   puts(1,  " lines\n " )
   puts(1,  " lines\n " )
end procedure

print_some_lines()
```

The first five lines define a procedure called print_some_lines(). This procedure accepts no parameters. When executed, it will print three lines of text. The procedure starts running when it is called by name (the print_some_lines() following the declaration portion). Let's build on it a bit more by adding parameters:

```
procedure print_some_lines(integer repeat, sequence line_to_print)
   for count = 1 to repeat do
      puts(1, line_to_print &  " \n " )
   end for
end procedure

print_some_lines(10,  " Hi There! " )
```

The procedure can now accept two parameters. The first one, “repeat,” controls the number of lines to be printed. The second one, “line_to_print,” is what is printed “repeat” times. These parameters are declared inside the procedure, as part of the declaration of the procedure itself. Now we have a procedure where its run can be controlled by the parameters it receives. Let’s change it to a function to have it return a value back:

```
integer status_of_print

function print_some_lines(integer repeat, sequenc line_to_print)
   if repeat > 50 then
      return 2
   else
      for count = 1 to repeat do
         puts(1, line_to_print &  " \n " )
      end for
      return 1
   end if
end function

sta-
tus_of_print = print_some_lines(10,  " Hi There " )
```

Now the library routine will only print a maximum of 50 lines, returning a value of 1 to let the program that called it know that it printed some lines. Any attempt to print more than 50 lines will cause the library routine to return a value of 2, and print nothing at all. You will notice the way we execute print_some_lines() has changed, by making it a part of an assignment statement, which is how functions in general are run. After print_some_lines() finishes executing, the returned value is stored in variable “status_of_print.” A demo program is available to show how to create a function that draws a text square on the screen:

program 86

```
include machine.e

atom result

crash_message("**************************************\n"&
              "* An error has been encountered that *\n"&
              "* is so serious the software must    *\n"&
              "* stop running now.                  *\n"&
              "*                                    *\n"&
              "* Please Email the file ex.err to    *\n"&
              "* moggie@interlog.com. Thank you!    *\n"&
              "**************************************\n")


for ix = 100 to 0 by -1 do
    result = 100 / ix
    printf(1,"%d divided by %d gives %f\n",{100,ix,result})
end for
```

You can also design custom variable types in the same manner you design custom library routines. As a matter of fact, declaring a variable type is like designing a one parameter function.

```
type name(parameter declaration)
   one or more Euphoria statements to execute
   return true or false value
end type
```

Declaring a custom variable type first involves choosing a name for it. Next, a parameter declaration is required to accept the value the custom variable type will hold. one or more Euphoria statements to execute will define what values the custom variable type is allowed to hold. This is done by running a series of tests on the parameter accepted. The outcome of these tests will determine whether a true or false value is returned once the group of code is finished running. At this point, we have two questions to ask: how do we declare a variable of a custom type instead of integer, sequence, atom, or object, and how do we actually execute the "type-end type" group?

If you created a variable type called "housepet," and wanted to declare a variable called "cat" of type "housepet," you would say:

```
housepet cat
```

As to how to execute "type-end type" group, it is started every time we attempt to assign a value to a variable using that custom type:

- Euphoria starts the type-end type group declaring the custom variable type, using the assigned variable as the parameter.
- The Euphoria statements executed within the type-end type group tests the parameter value, and a value of 1 or 0 is returned.
- If the value returned is 1, the assignment statement that triggered the type-end type sets the variable to the assigned value, otherwise the program will end with a type check error message.

Let's put it altogether in this program example:

```
type fruit(sequence unknown_fruit)
   sequence valid_fruit
   valid_fruit = { " bananas " ,  " apples " ,  " oranges " ,  " pears " }
   return find(unknown_fruit, valid_fruit)
end type

fruit fruit_check
fruit_check =  " pears "
puts(1,  " Program Run Completed\n " )
```

We've created a custom variable type called "fruit," and declared a variable called "fruit_check" of type "fruit." When we attempt to assign a value to "fruit_check" with the value "pears," the "type-end type" group is executed using "pears" as the parameter. As you can see in the "type-end type" the returned value is the result of a `find()` library routine. If `find()` cannot find "pears" in the sequence value stored in the variable "valid_fruit," a 0 will be returned. However, we can tell that 1 instead will be returned because "pears" is in the list. As a result, no type check error will occur, and the value "pears" is assigned successfully to "fruit_check" when the "fruit" type group finishes running. The next statement following the assignment of "fruit_check" with "pears" is then executed. Had we used the following assignment statement instead:

```
fruit_check =  " yams "
```

then the "type-end type" group would have returned a 0, and the program would immediately halt with the following message:

```
type_check failure, fruit_check is {121'y',97'a',109'm',115's'}
```

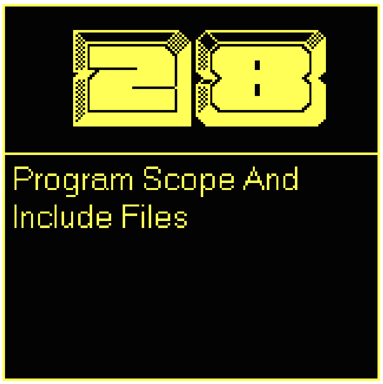
Custom variable types can help detect any problems during program design. A programmer can state certain values like time, phone numbers, postal or zip codes, etc, which must follow specified value ranges. If a program process takes a value outside an acceptable range, the "type-end type " group will return a 0 value, causing the program to halt on a type check error. A demo program shows another example of using variable types created by the programmer, this time using helpful diagnostic messages.

program 87

```
include machine.e

sequence actual_binary_number, binary_bits, series_of_values

clear_screen()

puts(1,"A Simple Example Of Using int_to_bits()\n")

puts(1,"======================================\n\n")
puts(1,"Decimal                 Binary\n")

puts(1,"=======    ================================\n\n")



series_of_values = {1,-1,500,-500}

for element = 1 to length(series_of_values) do

     binary_bits = int_to_bits(series_of_values[element],32)

     actual_binary_number = {}

     for bits = length(binary_bits) to 1 by -1 do

          actual_binary_number = actual_binary_number &

                                 (binary_bits[bits] + 48)

     end for

     printf(1,"%5d      %32s\n",

     {series_of_values[element],actual_binary_number})

end for

puts(1,"\n\n")

puts(1,"(Note: int_to_bits() returns the bits in reversed sequence.\n")

puts(1," The output displayed has been adjusted to show the bits as they\n")

puts(1," are meant to appear in a binary number)\n")
```

But your learning of custom library routines does not end at this point. There are other questions that need to be answered. For example, are any variables used in one library routine accessible by other library routines, or even the entire program? Do we have to keep everything that makes up a program in a single source file, or can certain parts be stored in several source files? The next chapter will address this when we learn about scope and using include files.

28. Program Scope And Include Files



The programs you have been writing so far have used variables that are accessible anywhere within the program. This is convenient, but if you plan to use a variable for a short process, wouldn't it be memory efficient to be able to "undeclare" it? Also, it would be faster to design a program that uses plug-in sections that are included into the program with a single line, instead of retyping or copying source code that already exists elsewhere. Euphoria can handle both problems very easily!

Previously, we briefly touched on scope when we were discussing the "for" statement, where the variable it automatically creates and increments during the loop only exists while the loop repeats. When the "for"statement completes executing, the variable is no longer available for use. This variable, therefore, is said to have "scope."

Scope coverage has been greatly expanded. See the documentation.

All Euphoria symbols, such as variables, library routines and type groups, has a scope, or a range limit in the program where it can be referenced. The scope begins the moment it is declared. This means it is impossible to use something before it is declared, so the example listed below is incorrect:

Scope of routines is now for the entire file. It is possible to use a routine that is declared **later** in the file—a huge change from Eu.

```
    marbles = marbles + 1
    atom marbles
```

In addition, the following listed below is also incorrect:

```
atom count_3s
for counter =  1 to 500 do
    count_3s = count_3s + 3
end for
printf(1, " %d %d\n " ,{count_3s, counter})
```

When run, you should get a message saying that variable "counter" was not declared. In actual fact, it was, but it only stayed around long enough for the "for" statement to complete counting to 500. The moment the "for" statement stopped, the variable "counter" was, in essence, "undeclared." By the time program execution reached the line where `printf()` is used, the variable "counter" did not exist any more.

Which brings us to an important question: just how far does a variable, library routine, or type group's scope extend?

If a variable is declared automatically by a "for" statement, its scope ends at the "end for" line, and any program statements following can no longer reference it. If a variable is declared inside a function, procedure, or type group, its scope ends at either the "end type," "end procedure," or "end function" line. This means other library routines, type groups, or even the main program can't access it. These kinds of variables are referred to as "private" variables, because only the "for" statement, library routine, or type group the variable was declared inside can reference it. Once any of these processes stop running, the variable is "undeclared," freeing up memory.

If a variable is declared outside a library routine or a type group, then the variable's scope extends from the point of declaration to the end of the program. These kinds of variables can be referenced inside a library routine, a type group, or by programming statements being executed inside a "for" statement, and are called "local" variables. The demo programs you have run up to now use local variables.

It's possible to have a local variable and a private variable with the same name. When executing the library routine or type group it is declared in, the private variable is dominant over the local variable.

Type groups and library routines declared in your program have a scope that starts at the point of declaration to the end of the file they are declared in. They can be referenced by other library routines and type groups, but only if they are declared AFTER. Look at the following example below:

No longer a restrictin in oE4

```
procedure alpha()
   clear_screen()
end procedure

procedure beta()
   alpha()
end procedure
```

It is perfectly legal for procedure "beta" to call procedure "alpha" because "alpha" was declared before "beta."

A routine can call any routine that is declared in the same file.

The scope of variables, library routines, and type groups need not be bordered within the confines of a single source file. This means you could reference variables, library routines and type groups that are not even present in your program, but in another source file. This is where the include files comes in.

You can place any variable, library routine, or type group declarations inside include files (.e) that are separate from the program. To use any of these in your program, you reference the include file with the include statement. In the include file where any variables, library routines or type groups are stored, you have to use the word "global" as part of the declaration. global means the scope of the variable, library routine or type group extends indefinitely. It's important to use lobal in any variable, library routine or type group that is declared in an include file, otherwise the scope of each is assumed to be local.

Global is still a keyword but not recommended for most use. Instead, use export in this situation.

export is preferred to global

Here are some examples of declarations with the word global:

```
global object grab_bag

global function add_two(atom first, atom second)
   atom sum
   sum = first + second
   return sum
end function
```

As a result, include files allow you to design programs that are made up of plug-in code, referenced by the include statement at the top of the program. The file name the include statement references can be be an absolute file and directory path (c:\utilities\icons.e), or a relative file name (graphics.e). If a relative file name is used, Euphoria will first search for it in the same directory the program is currently in. If not found, it will then look in "euphoria\include". The environment variable EUDIR sets the exact path of "euphoria."

more nesting levels now

Include files can in turn contain include statements referencing other include files, up to 10 levels deep of nesting. Each include statement must be on a single line by itself.

A demo program is available to show how to use include files containing externally defined variables and a procedure.

program 88

```
include machine.e
sequence bit_patterns
atom integer_value
clear_screen()
puts(1,"How bits_to_int() is used to convert a sequence of 8 bits into\n")
puts(1,"a single byte value.\n")
puts(1,"==========================================================\n\n")


bit_patterns = {{1,1,1,1,0,0,0,0},
                {1,0,1,0,1,0,1,0},
                {1,1,1,0,0,1,1,1},
                {1,0,0,0,0,0,0,1},
                {1,1,1,1,1,1,1,1}}

for bit_groups = 1 to length(bit_patterns) do
    integer_value = bits_to_int(bit_patterns[bit_groups])
    print(1,bit_patterns[bit_groups])
    printf(1," can be stored in a value of %3d\n",integer_value)

end for
```

Before running the above demo, copy and paste the following into your editor and name the file ’d2808a.e’. Then put the file in your euphoria \ include directory:

use export instead of global

The next chapter is the final one in this tutorial, so its purpose is to introduce library routines that were not easily classified into the subjects you have covered. Turn the page now to begin wrapping up your introduction to the Euphoria programming language!

29. Wrapping It Up With Mouse And Sound Support

This last chapter of the tutorial will teach you how to use other input and output devices besides the keyboard and screen. Most users today prefer the use of a mouse over a keyboard because it does not involve memorizing complex commands. Just move the mouse pointer to a graphics icon and click. Also, nothing adds a little dimension to a game than sound. Sound allows the programmer to get a person's attention by emitting a beep from the speaker, along with a message on the screen. Euphoria programs can use a mouse easily, without any need to understand device driver programming. It is available in all graphics modes. In a pixel graphics mode, the mouse pointer appears as an arrow pointing toward the top left. In text modes, it appears as a solid block. Mouse usage in modes beyond 640 x 480 pixels is possible only in Windows/95+

oE does not do Dos sound routines

oE does not do DOS mouse routines.

A mouse point is displayed on the screen using the same coordinate system that pixels use, namely a pixel column first, pixel row last pair. Whenever the mouse is used, something called an event is generated. For example, an event can be the mouse being moved across the pad, or one of the mouse buttons being clicked. Your system can only keep track of one event at a time, so previous events are always being replaced with new ones. A Euphoria program can get the last event generated when a mouse is used by using this library routine:

```
include mouse.e
ro = get_mouse()
```

get_mouse() will return a three-element long sequence that represents the last even generated, to be stored in receiving variable ro. Each element of the sequence is an atom value, the meaning of which is shown below:

```
{event, pixel column, pixel row}
```

The event is an integer value that describes what the mouse did:

```
1  --- Mouse moved
2  --- Left button pressed
4  --- Left button released
8  --- Right button pressed
16  --- Right button released
32  --- Middle button pressed
64  --- Middle button released
```

The pixel column and pixel row is where the mouse pointer was displayed when the event occured. It's possible to have an event integer that is the sum of two actions happening simultaneously, such as the right button being held down while the mouse moves across the pad (8 + 1 = 9). The pixel column and pixel row returned could either mean the very tip of the mouse pointer, or the pixel locaton the mouse pointer is pointing at. The only way you can determine this is by testing. If it is the very tip of the mouse pointer being returned, and you want to use get_pixel() to return the colour of the pixel, you have to subtract 1 from both the pixel column and pixel row to get the correct pixel location. If you want to get the actual text column and text row of the mouse pointer, you may have to scale the returned pixel column and pixel row using division.

get_mouse(), when used for the very first time, will turn on your mouse pointer. Your mouse driver must be loaded before you can get mouse events. If no mouse event has been generated since the last use of get_mouse(), a value of -1 is returned. get_mouse() by default returns every event that occurs when your mouse is used. Sometimes this is not practical, as a lot of events can occur in a short period of time, causing you to miss the events that you wan to check for. It would be nice to screen out those mouse events you are not interested in, concentrating on the mouse events that are important.

```
include mouse.e
mouse_events(i)
```

mouse_events() allows you to select which events (i) get_mouse() should report. i is actually a sum of the specific events you want reported. For example, mouse_events(42) means you want get_mouse() to report only events that involve the left, middle, or right buttons being pressed (2 + 8 + 32). Any other events that occur will be ignored. mouse_events() can be re-issued at any time to change what events you want get_mouse() to report. The very first call to mouse_events() will turn on the mouse pointer for you.

On the subject of mouse pointers, there are times where you need to turn the mouse pointer on or off. For example, you may want to turn off the mouse pointer when you plan to use save_image() to copy an area of the screen where the mouse pointer is currently over. Here's the library routine that can do this for you:

```
include mouse.e
mouse_pointer(i)
```

i serves as the toggle switch that controls whether your mouse pointer is visible or not. A value of 0 means the mouse pointer is hidden. Any positive value will make the mouse pointer visible again. There's an important note to make regarding the multiple usage of mouse_pointer(). For example, if you issue mouse_pointer(0), say, 3 times, you must issue mouse_pointer(1) the same number of times (3 in this example) to make your mouse pointer appear again. Therefore, it is extremely important for you to keep track of where the mouse pointer is being turned off and on in your program.

Now that we've wrapped up mouse feature, let's wrap up the tutorial by introducing your last library routine to learn.

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

sound() emits a sound from your speaker. i is the frequency of the sound. The higher the value of i, the higher the pitch generated. A demo sums up all the library routines you have learned here, by using the mouse to emit different sounds by clicking parts of shape!

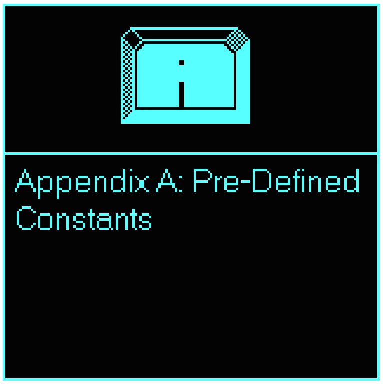
program 89

```
include machine.e

atom single_value, ANDed_atom, work_value, ANDer_atom
sequence bunch_of_values, ANDed_sequence, returned_bits

clear_screen()

ANDer_atom = 484848

bunch_of_values = {222222,333333,444444}
single_value = 123456

printf(1,"ANDing %6d and %6d\n\n",{single_value,ANDer_atom})

ANDed_atom = and_bits(single_value,ANDer_atom)

work_value = single_value
returned_bits = int_to_bits(work_value,32)
printf(1,"%6d ---------> ",work_value)
for bits = 32 to 1 by -1 do
    print(1,returned_bits[bits])
end for
puts(1,"\n")

work_value = ANDer_atom
returned_bits = int_to_bits(work_value,32)
printf(1,"%6d ---------> ",work_value)
for bits = 32 to 1 by -1 do
    print(1,returned_bits[bits])
end for
puts(1,"\n")
puts(1,repeat('-',50) & "\n")

work_value = ANDed_atom
returned_bits = int_to_bits(work_value,32)
printf(1,"%6d ---------> ",work_value)
for bits = 32 to 1 by -1 do
    print(1,returned_bits[bits])
end for
puts(1,"\n\n")
puts(1,"\nPress Any Key To Continue.......\n\n")
while get_key() = -1 do
end while

clear_screen()

ANDed_sequence = and_bits(bunch_of_values,ANDer_atom)
puts(1,"ANDing ")
print(1,bunch_of_values)
printf(1," and %6d\n\n",ANDer_atom)

for element = 1 to length(bunch_of_values) do
     work_value = bunch_of_values[element]
     returned_bits = int_to_bits(work_value,32)
     printf(1,"%6d ---------> ",work_value)
     for bits = 32 to 1 by -1 do
          print(1,returned_bits[bits])
     end for
     puts(1,"\n")

     work_value = ANDer_atom
     returned_bits = int_to_bits(work_value,32)
     printf(1,"%6d ---------> ",work_value)
     for bits = 32 to 1 by -1 do
          print(1,returned_bits[bits])
     end for
     puts(1,"\n")
     puts(1,repeat('-',50) & "\n")

     work_value = ANDed_sequence[element]
     returned_bits = int_to_bits(work_value,32)
     printf(1,"%6d ---------> ",work_value)
     for bits = 32 to 1 by -1 do
          print(1,returned_bits[bits])
     end for
     puts(1,"\n\nPress Any Key To Continue.......\n\n")
     while get_key() = -1 do
     end while
end for
puts(1,"Result is ")
print(1,ANDed_sequence)
puts(1,"\n")
```

You have reached the end of this tutorial, but not the end of your adventures in Euphoria. The best way to get a strong grasp of this powerful language is by first writing very small programs, then build larger ones. Do not be daunted by any error messages you get. They are easy to understand. However, there is a segment in the appendices at the end of this tutoria on error messages when running EX.EXE, just in case you get a little confused.

```
Good Luck In Your Euphoria
Programming!

Thank You For Using This
Tutorial!! :)
```

30. Appendix A: Predefined Constants



Most values either used by library routines, or returned by library routines, have a special meaning. For example, in graphics library routines like text, the colour blue is represented by the value 1, and obtaining a Euphoria object using gets() successfully will cause the library routine to return a value of 0. To make all these values easy to remember, a Euphoria symbol related to a variable is used to give absolute values a literal name. The declarable symbols are called constants.

Like variables, constants are literals used to reference RAM addresses using a symbolic name. Like variables, constants must be declared before use, and are assigned a value. They also follow the same naming rules that variables, library routines, and other Euphoria symbols do. But the similarities stop here at this point. While variables can have their values changed at any time, a constant can only be given a value once, and is locked with that value for the duration of the program run. As you can see, the name constant fits perfectly to describe this kind of declarable symbol.

When a constant is declared, it is given its permanent value as part of the declaration statement:

```
constant variable name = expression
```

An expression, as you remember, can be a single value, or an arithmetic, logical, or relational expression. You can use other constants, library routines, and variables as part of the expression but once you declare the constant with its value, you cannot use an assignment statement to change the constant's value again. Constants are best used for subjects that are unchangable, such as the value of PI or the speed of light. In externally declared library routines, the include files that define the library routines also contain a list of constant values you can use in place of the absolute values. For example, if you want to draw a bright magenta line, you specify in draw_line() BRIGHT_MAGENTA as the colour to use.

Constants can be local and global, but never private, so they cannot be declared inside type groups or library routines. A demo program is available not only an example of constants being used, but also lists some of the constants that are declared in include files, like image.e, graphics.e and file.e. [note: before running the demo, you will need to make sure the the file constants.txt is available, and modify the open() function accordingly]

oE now has **enum**, a way of creating constants that are sequentially numbered for you.

program 88

probably works, but not tested

```
integer constants_file, current_record, input_key, update


sequence database, input_record

constant number_of_records = 49,
         field_1_start = 1, field_1_end = 24,
         field_2_start = 25, field_2_end = 35,
         field_3_start = 36, field_3_end = 37,
         field_4_start = 38, field_4_end = 110,
         field_5_start = 111

procedure display_data(sequence record)
     sequence dline1, dline2, dline3, dline4, constant_name, include_file,
              constant_value,library_routines_where_used,
              constant_description

     dline1 = "Constant Name: %s\n"
     dline2 = "Constant Description: %s\n"
     dline3 = "Constant Value: %s\n"
     dline4 = "Declared In Include File: %s\n"

     constant_name               = record[field_1_start..field_1_end]
     include_file                = record[field_2_start..field_2_end]
     constant_value              = record[field_3_start..field_3_end]
     library_routines_where_used = record[field_4_start..field_4_end]
     constant_description        = record[field_5_start..length(record)]
     position(4,1)
     for line = 1 to 7 do
          puts(1,repeat(' ',75) & "\n")
     end for
     position(4,1)
     printf(1,dline1,{constant_name})
     printf(1,dline2,{constant_description})
     printf(1,dline3,{constant_value})
     printf(1,dline4,{include_file})
     puts(1,"\nExamples Of Library Routines Where Used:\n")
     puts(1,library_routines_where_used & "\n")
end procedure

clear_screen()
current_record = 1
database = {}
input_key = 0
update = 'y'

constants_file = open("constants.txt","r")

for ix = 1 to number_of_records do
     input_record = gets(constants_file)
     input_record = input_record[1..length(input_record)-1]
     database = append(database,input_record)
end for

position(1,1)
puts(1,"Euphoria Constants Dictionary")
position(2,1)
puts(1,"=============================")
position(13,1)
puts(1,"Press < and > to move about")
position(14,1)
puts(1,"Press Q to quit")
```

```
while input_key != 'q' do

     if input_key = '.' or input_key = '>' then
          if current_record < number_of_records then
               current_record = current_record + 1
               update = 'y'
          end if
     end if

     if input_key = ',' or input_key = '<' then
          if current_record > 1 then
               current_record = current_record - 1
               update = 'y'
          end if
     end if

     if update = 'y' then
          update = 'n'
          display_data(database[current_record])
     end if

     input_key = get_key()

end while

clear_screen()
```

31. Appendix B: Error Messages When Running EX.EXE

During your writing of Euphoria programs, you will encounter error messages. The error messages are generated when using EX.EXE (either directly by calling EX.EXE or indirectly when using BIND.BAT). While the error messages are very easy to understand, first-time users may find some difficulties solving them. This appendix will help explain what the text in EX.ERR means, and will give detailed descriptions of what some of the more commonly seen error messages are saying.

When a program aborts with an error, you will see the following information:

```
program.ex:nnnn
error message
```

The first line states the name of the program you were running, with a line number after the colon where the error occurred. The line below is the error message describing the problem. A copy of these two lines are also stored in EX.ERR, which is generated at the time of the error. There is also additional information in the EX.ERR file:

`GlobalLocal Variables (a list of variables and the values they contain follow)` Sequence variables will be shown with both actual and character values for each element. If include files, type groups, and library routines are used in your program, the variables will be grouped based on the program sections they are declared in. If the error occurred in a procedure, function, type group, whether in an included file or in the program itself, it will be shown as well. For example, you could see the following text if it occurred in a procedure:

`program.ex:nnnn in procedure procedure_name()` But all this is just a snapshot of what was going on at the time the program aborted. The really important information is of course the error message. It tells you WHY the program halted in error. While there are many error messages generated by EX.ERR, here are some that you will encounter often:

`type_check failure, variable is n` this means you tried to assign a value in a variable that does not support the data object type of the value. For example, a sequence variable may be accidentally assigned an atom value, or vice-versa, or an integer variable is being assigned a value that is larger than the accepted integer range.

`sequence found inside character string` this means you have a sequence that has one or more sequence elements. Some library routines only accept sequences that are made up of atom elements.

`subscript value n1 is out of bounds, reading from a length-n2 sequence` this means you have attempted to access an element within a sequence that does not exist. For example, if you try to reference element 10 in a 5 element sequence, you will get this message.

`slice ends past end of sequence (n1 > n2)` this means a range of elements you were trying to extract from a sequence has the end range position n1 going beyond the maximum sequence length n2.

`can't open include file path` this means the include file you supplied in the include statement was not found. You either misspelt the name, or you are using the wrong directory.

`sequence lengths are not the same (n1 != n2)` this means two sequences you are using in an operation, like addition, are of different lengths. n1 is the first sequence length, n2 is the second sequence length. You also get this message if you are using the if statement to directly compare a sequence variable with a sequence value.

`syntax error - expected to see …, not …` this means EX.EXE found a symbol that was different from what it expected. For example, if you left out then in an if statement, you would get this message. You may also see EX.EXE stating was obtained instead of an expected symbol. This could mean you left out an end if, end for, or end while.

`true/false condition must be an ATOM` this means a sequence was used in a logical or relational expression portion of a statement such as the while statement.

`unknown escape character` this means you paired the \ special character operator with another character that Euphoria does not accept as valid when paired.

`may not change the value of a constant` this means you tried to change a constant value. Constants cannot be changed to another value.

`a loop variable name is expected here` this means you used either a reserved word like global, or a symbol with an illegal character as the control variable that is incremented during the for statement loop.

`no value returned from function` this means you forgot to add the return statement in the function you created in your program.

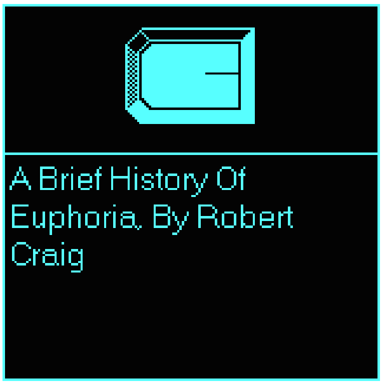
`attempt to subscript an atom` this means a value in an object variable was an atom when you tried to reference one or more elements in what you expected to be a sequence.

`slice length is less than 0 (-n1)` this means you attempted to perform a reversed range that went beyond the legal [n+1..n] limit. The value -n1 is the number of elements out of bounds of the legal reversed range limit.

`symbol has not been declared` this means you either misspelt a symbolic name (such as a variable or library routine), forgot to declare it, or forgot to use the global keyword with the variable or library routine declared in the include file.

The best way to understand errors is through experience. Good luck!!

32. A Brief History of Euphoira


A Brief History Of Euphoria. By Robert Craig

Robert Craig, is the head of RDS, and also the man who created the Euphoria programming language. Here is a brief history of the Euphoria programming language written in his own words. You'll find that when reading it, you'll learn much about the man who created the language, as well as the language itself.

A Brief History of Euphoria

Back in 1989 my brother Dave, who is also a software developer, bought an Atari Mega ST with a whopping 4 Mb of memory. I was very impressed and I wanted one too, but I didn't know how to justify buying one.

I decided I would do some kind of long term hobby project in the compiler development area, since that is what my background is. After several months of day dreaming I finally decided – I would design and implement a new programming language loosely based on my Master's thesis at the University of Toronto. In my Master's research I had examined John Backus' FP language. John Backus is the guy who developed the first Fortran compiler for IBM.

Euphoria (it didn't have a name back then – just language 'X') took the concept of atoms and sequences from FP, but is totally different in all other respects. Also, in FP an atom can be a number or a string. I didn't like that. I thought a string should be a sequence of characters, so you could apply subscripting, appending etc. to it, just like all other sequences. The idea of slicing came from Ada.

Within a few months I had a working prototype interpreter for Euphoria. I had designed it as best I could for maximum speed. I eagerly performed the first sieve benchmark test – Euphoria was 250 times *slower* than C on the Atari. I was not particularly disappointed. I thought a Euphoria interpreter *should* be tremendously slower than compiled, optimized C code, given all the flexibility, dynamic storage allocation, subscript checking etc. Over the next 3 years I continued to add to the language and to improve the speed.

Speed became something of an obsession for me. On at least 3 occasions I completely re-wrote thousands of lines of the interpreter, just to get a modest boost in performance. Euphoria is now only a few times slower than fully-optimized C on many benchmarks, even while performing subscript checking, uninitialized variable checking,integer overflow checking etc, that C doesn't do. The gap is even narrower if you program's speed is determined largely by file I/O, or calls to library routines.

My full-time day job was getting very boring at the same time that Euphoria was getting more interesting. Finally a good opportunity arose and I quit my day job so I could port Euphoria to the IBM PC, finish v1.9, and release it as shareware. v1.9 was released in July 1993. Since then, I've been very pleased with the reaction to it, and although there isn't a lot of money in it, the registration rate continues to climb.

After v1.2 (March 94) I had to go back to a real programming job for a couple of years. Version 1.3 (May 95) and 1.4 (May96) were produced while I was working full-time at a "day" job. It was quite a strain to do this. I'm glad I'm back on Euphoria full-time.

The addition of the Euphoria Web page in January 1996, and the Euphoria list server (thanks to John Kimne) in June 1996 helped a small user community to grow up around Euphoria. Up until then, users did not know each other, and all tech support had to come from me. I am very grateful to the many people who have set up their own Euphoria Web pages, and to those who have freely contributed source code and programs for other to use.

My wife, Junk Miura, left her job as a compiler developer a year ago, so she could pursue other entrepreneurial interests. Lately she has contributed to the programming effort. For v1.5 she wrote the savescreen() library routine and the HOW2REG.EX program that takes you through the options for registering Euphoria.

Euphoria (ex.exe) consists of 15,000 lines of C code that is compiled by WATCOM C/32 and is bound with the Causeway DOS extender of Michael Devore.

A Windows WIN32 version can be built from the same source by setting a C #define symbol. An alpha release of the WIN32 version should be out in a few months. It will not make the DOS version obsolete, rather, users will have a choice of which one to use for a particular application. The core language and most of the library routines will be the same. Most of the C source code will be shared.

After several years I am more excited than ever about this project, and I will continue to work on it full time for the foreseable future.

Robert Craig,
Rapid Deployment Software, February 1997