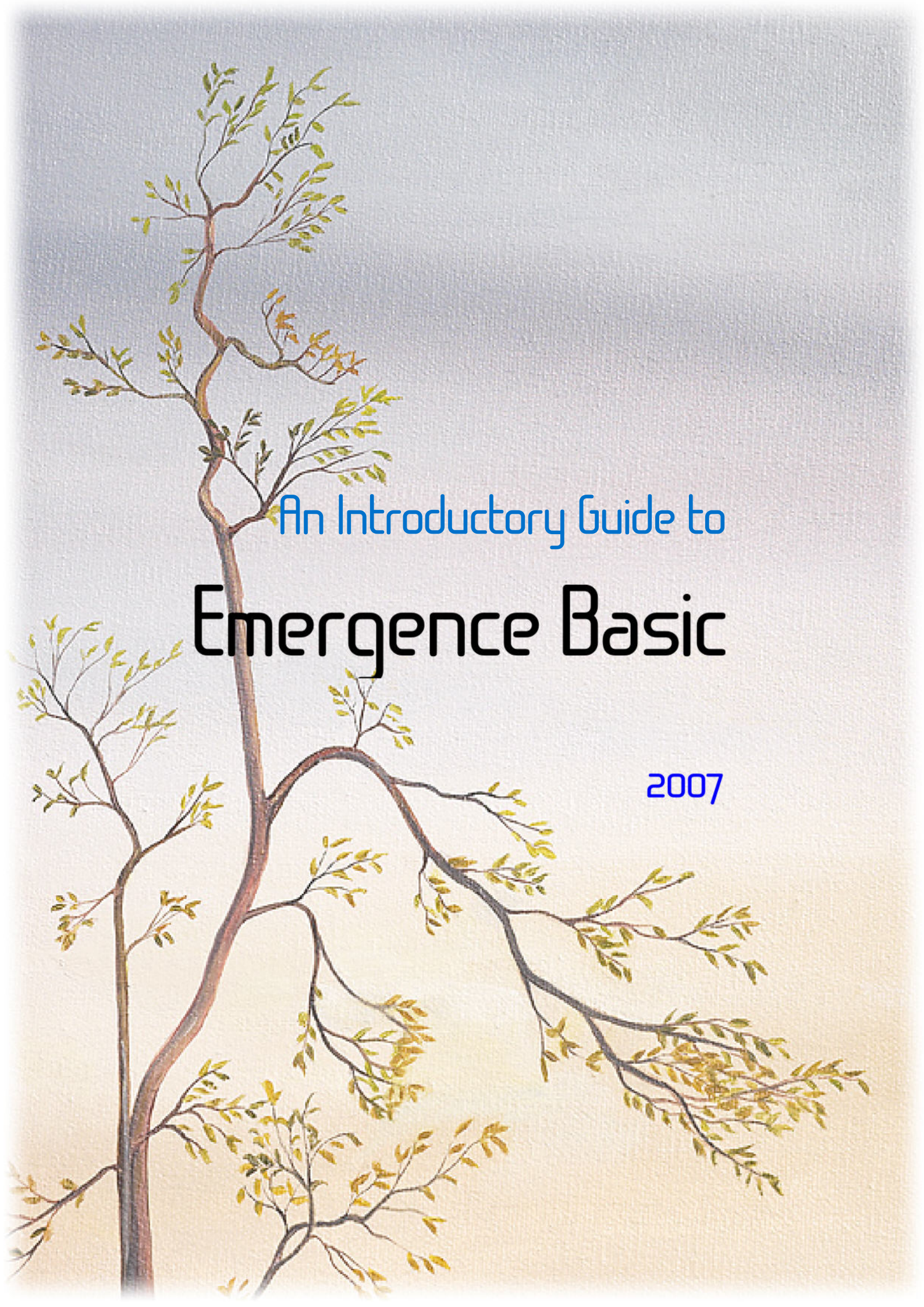


An Introductory Guide to

Emergence Basic

2007

Hello

Most programming guides are incredibly boring. So I shall make an attempt to avoid that.

There's no chance of finding an "Hello World" program here. Maximising information transfer is the objective.

If you're new to programming, and to Emergence Basic in particular, these words are for you.

If you are an experienced programmer, you probably know more than I do, and you probably have a large library of incredibly boring textbooks. I shall pretend you're not there.

I aim to consign all boring details about bits and bytes and stuff, to ..

You can find this section at the end of the document.

If you venture down there, do take care, the place is a terrible mess of boring, depressing, excruciatingly detailed information.

To be honest, I'm hoping to avoid going down there very often and adding to the clutter.

Programming is a pleasant occupation, bringing delight and happiness to many people.
Only serious developers feel the need to get involved in nitty-gritty stuff on a regular basis.

We need to thank Paul Turley (the creator of Emergence Basic) for this Windows programming tool. Simple, effective, and inexpensive, what more could we ask.

The EBasic Windows programming software provides all the tools you need to create serious Windows programs and fun games.

At the time of writing, we shall be using the hot-off-the-press first edition of EBasic.
Many developments are planned which will expand the facilities considerably.

So let's get the programming on the road.

First though, we have to install Emergence Basic. Can't do much without that.

Getting Started

Installing EBasic

Installation is very easy. Unzip the ebsetup.zip file, which you downloaded, into a temporary directory, and run the ebsetup.exe file. Choose a location in which to install the program. That's it, installation will proceed automatically.

When you first run the program, you will be asked to enter your user name and key.

Emergence Basic is now ready for use.

But suppose we want to uninstall EBasic (gasp!), or more likely, to update it.

Uninstalling and ReInstalling

To uninstall EBasic, use the **Start – Settings – Control Panel – Add or Remove Programs** menus. Locate Emergence BASIC – and click on **Remove**.

All EBasic files will be deleted except for any example files that you may have used. So you may have to delete the main EBasic directory 'EBDev' in order to remove any remaining files.

To re-install Emergence basic, just follow the installation process as for a new version.
You will not be required to re-enter your user name and key.

Learning Emergence Basic

Many sample programs are provided. Run them to see what they do. Examine the code to see how the effects are produced.

Read the EBasic help files. Even experienced programmers may learn something new, when they look through the help files.

The IonicWind Forum is a great source of help and ideas. You'll probably spend a lot of time there.

First Impressions

I assume you've managed to start EBasic, and are now looking at the editor screen.

It's a nice screen, but once you've seen one editor screen you've seen them all.

It's got a menu bar and a toolbar. Most user guides would now spend 20 pages discussing how to use all the menu and toolbar facilities. Not this one. Manuals are only to be read as a last resort, or when laying in bed with nothing better to do.

I would just mention that you will probably find the **Help** menu very useful.

Whether you leave the **Output** messages and **Project** details panels open is up to you.
Toggle them on and off using the View menu. I usually turn them off until I need them.

Well there you are, the editor screen is open, (and empty), so off you go and write your program.

What, you don't know what to write? You've never written a program before?
Hmm ... that's a bit of a problem.

I suppose this is where I have to earn my keep, and give you a few clues how to get started.

It's not going to achieve much if you type 'Ipso facto splurge'. I'm afraid the compiler just won't understand. You have to type specific and accurate instructions that it will understand.

So what are these instructions?

I can't tell you about all of them, it would take all day. This is where the **Help** menu comes into its own. Not even an experienced programmer can remember every instruction.

Open the Help menu, and select the **Alphabetical Command Reference** item.

Well there you are - a list of about 250 commands. Each of them performs a specific task, and many require several other items of information in order to work.

You'll have to learn all those off by heart before you can do any programming - ***just kidding***!
I don't imagine anyone remembers all those details – that's what the help facility is for.

Regard the commands as a set of tools to be selected and used when a particular job is to be done.

Writing a Program

Now what? You've got all those tools and you still don't know how to proceed?

Well no one said it would be easy.

This is like a writer of a book, facing his first empty page. So many things are possible – but where to start?

My advice would be not to attempt anything too ambitious to begin with.

With any new program, you need to form a clear mental picture of what you are trying to achieve.

Since we want to program in Windows, what better place to start than to create a Window.

That sounds easy enough, but a normal Window is a pretty boring object - no colour, no soul - so we'll try to make an interesting one.

Before we start, create a new directory in which the program can be stored – call it something like 'EBWin1', so that you can recognise it.

What we are aiming for, is a colourful window with an 'Exit' button to close the window again.

Seems easy, but we will add a few tricks, which of course I've had to imagine, before I could write the source code which follows.

Before I show you the code, I will just say that at this point most guides will launch into a lengthy discussion of many of the available instructions - the maths functions - the logic operators etc.

I shall follow the idea of explaining them 'in context' as we find we need them ..

So don't fall off your chair when you see the following small program. I will take you through it and reveal it's structure.

It does have a pattern which you will see repeated in all the programs you write. Once you grasp that pattern, you will find it easy to write your own code.

All programs are written a bit at a time - each part doing a small part of the whole application.

Each part can usually be tested to see that it's doing what you require before you write some more code.

The small parts soon begin to accumulate into a sizeable chunk of code, and before you know it, you've written a program of impressive size ...

A First Window Program

Here is the main part of the program.

To start a new program, click on '**File - Close**' to close any current program in the editor window. Then click '**File - New - Emergence Source File**' to create an empty window.

Then just copy and paste the following program into the EBasic editor window.

```
' An Interesting Window Example ..
' GWS -Jan 2007
window w

int run, band, wid, hgt, i, j
int meanx, meany, sdev, c, x, y
float pstep, pi

autodefine "OFF"

pi = 4 * atan(1)                    : ' gives an accurate value of pi (3.14 .. )

wid = 800
hgt = 600
openwindow w, 0, 0, wid, hgt, @SYSMENU, 0, "My First Window", &msghandler

control w, @BUTTON, "Exit", 460, 350, 70, 30, @ctlbtnflat, 1
setcontrolcolor w, 1, 0, rgb(120,140,220)

' draw a gradient fill ..
pstep = hgt / 256 + 2
for band = 0 to 255
        rect w, 0, int(band * pstep), 800, int((band + 1) * pstep), rgb(0,0,band), rgb(0,0,band)
next band

' create some coloured spots ...
for i = 1 to 10
        meanx = rand(wid - 100) + 50
        meany = rand(hgt - 250) + 50
        sdev = 20 : ' controls the spread of the pixels ...
        c = rgb(rand(255),rand(255),rand(255))
        for j = 1 to 150
                x = splot() * sdev + meanx
                y = splot() * sdev + meany
                pset w, x, y , c
        next j
next i

' display some text ..
setfont w, "Arial", 20, 700, @sfitalic
frontpen w, rgb(130,130,230)
drawmode w, @transparent
move w, 50, 450
print w, "EBasic is Fun"

starttimer w,1000

run = 1

waituntil run = 0
stoptimer w
closewindow w
end
```

The main part of the program refers to two '**Subroutines**', which are just auxiliary blocks of code kept separate for clarity. Here they are - just copy and paste them at the end of the main program ..

```
sub msghandler
        select @message
                case @idclosewindow
                        run = 0
                case @idcreate
                        centerwindow w
                case @idtimer
                        setsize w,rand(wid - 125) + 25,rand(hgt - 100) + 25,rand(50) + 40,rand(30) + 20, 1
                case @idcontrol
                        if (@controlid = 1) then run = 0
                case @idchar
                        if (@wparam = 27) then closewindow w
        endselect
return
endsub

sub splot(),float
float x1, x2, ret
' routine to generate a random value from the normal distribution ..
x1 = rnd(1.0)
x2 = rnd(1.0)
' catch very small values of x1 ..
if (x1 < 1.0e-5) then x1 = 1.0e-5
' the Normal curve value ..
ret = sqrt(-2.0 * log(x1)) * cos(2 * pi * x2)

return ret
endsub
```

You should now have the whole program in the editor window.

In order to test it, we have to save the code into the directory you set aside for it.
Click on '**File - Save As**' and save the program giving it a name such as 'WinEx1.eba'.

The program source code is now saved in a directory where you can find it, and now we need to compile and run it.

The '**compiler**' has the job of checking your code for errors, and generating a few more files including the final executable program which (if you used the suggested source file name above) will be named '**WinEx1.exe**' . This is the program that Windows runs to display your creation.

All files created by the compiler will by default reside in the directory you created for this program.

Use the '**Build – Build Single**' menu option. In the Executable Options dialog, check that the Executable File Name is set to the directory and executable program name you require. Then select '**Windows EXE**' as the Target. (This tells the compiler what type of program it is to create). Now just click the Create button to generate the program and run it.

If all has gone well, you should see an interesting window (well I hope you think it's interesting) – and look, you've hardly had to do any work yourself – so far.

How Does it Work

If you've managed to stop it (tip: press ESC if you're not too good with a mouse), we'll take a look at how it works.

Firstly, step well back from it, and just look at the screen. Assuming that you've kept the default editor colours, you will see lots of blue bits, a few bright red words, a few orange words, quite a lot of green (comments), and the remaining items will be black.

The blue words are the in-built EBasic commands, so this gives you an idea how often EBasic tools have been used in this application.

The red words, indicate data type definitions. These occur mostly at the beginning of any program or subroutine.

Orange items are what are known as system constants, used mainly when setting up controls, or checking system messages. These are also built into EBasic. (see the Help Appendix for a list).

There will usually be quite a lot of green words if you follow my advice, and include lots of comments to explain what is going on. (I didn't use too many here to keep it short).

The black items are very interesting, because these represent your own variables and data used for this application.

Now we will look a little closer to establish the parts of the program. Don't look at the detail - just try to see the overall structure.

Comments

The first two lines are a small block of comments. Every program should have some comments at the beginning to describe the program, the author and the creation date.

Variable Type Definitions

Then follows a small block of variable type definitions. You will hardly ever write all these in one go. As you are writing the program, you will often think - Oh, I need a new variable here. Then you will think of a name for it, and what type of data it will hold. Then you add the name to an appropriate definition list at the beginning of the program.

Goodbye Autodefine

Then follows the Autodefine "OFF" statement, which makes sure you don't forget to specify what type of data your variables will hold.

Open the Window

The next three lines just set up some useful data values, and then we find the OpenWindow command to open the window. This is a key instruction, because without the window we could do nothing else.

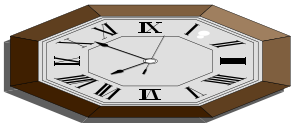
Position Controls

Once the window has been requested, we usually add the controls that we need for this window. In this small application there is just one window and one button - the 'Exit' button.

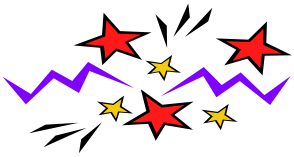
Various bits of initialising Code

Next come three small sections of program, implementing things I want to do with the window. Don't try to understand these at this point.

The first block applies a colour gradient to the window. The second block requests a number of coloured spots on the screen, and the third block writes some text to the screen.



For this application I needed a '**timer**'. This is a Windows resource that generates a message every so many milliseconds - in our case once every 1000 milliseconds (ie once a second).
We start a timer with the 'starttimer' command, and it will keep going until the timer is stopped - in this case when the program ends.



Then comes an important bit. The 'run = 1' and the 'waituntil run = 0' statements, because once the condition 'run = 0' occurs, the program will stop the timer, close the window, and END.

However, we'll leave discussing what is happening here until we have perceived the rest of the program structure.

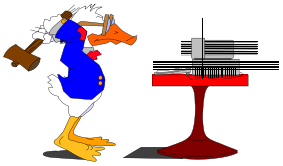
There is in fact, only the two subroutines left to consider. The last one is Subroutine Splot, and since this is just a bit of stuff I created to get a nice graphics effect - you can ignore this one for the moment.

Process the Messages

The other Subroutine msghandler is always the most important part of every Windows program you write. (It won't always be named 'msghandler' though - you can call it whatever you like - it's what it does that's important.

There we are then - do you see the pattern? In the main body of the program - Comments, Data, OpenWindow, set up controls, some sections of your own code, a 'WaitUntil' statement, followed by Closedown. Then the 'msghandler' subroutine for the window, and finally, any number of other subroutines you need for this application.

Messages and the Secrets of Windows



You need to concentrate on this bit. If you can see how messages work, you are well on the way to writing any Windows program in EBasic.

When you open a window, Windows begins generating messages such as - 'the window has been created' - 'the window has be re-sized' - 'a control has been clicked' - and so on.

The **Message Handler** subroutine in your program can intercept relevant messages and do something based on what has occurred.

If you look again at the **Subroutine msghandler** in our program, you will see we first process a message (when it occurs) that the window has been closed.

We respond by setting the variable 'run' equal to '0'.

Now do you see the point of the statement '**waituntil run = 0**' in the main part of the program. The main program has been cycling quietly to itself, just waiting for the condition '**run = 0**' to be true. Then the program drops through to stop the timer, close the window, and END.

When a 'window has been created' message is detected, we respond with a '**centerwindow w**' command.

In a similar way, we deal with other types of message concerning the Timer, a Button being clicked, and a key being pressed on the keyboard.

There will be many other types of messages which we have chosen to ignore in this simple application. You will find lots of information about the messages associated with windows and controls in the **Help** system.

Once it has been set going, the program has become '**Event Driven**'. We respond to whatever the user does.

Each window and each control will send messages that can be intercepted and acted upon by your message handling subroutine. This subroutine becomes the heart of your program which will process whatever code, and call whatever other subroutines you wish.

If you grasp this messaging concept, you will be able to tackle any EBasic program.

The Crypt

Oh Dear! .. You came down here then ?

Some people who came here have never been seen again .. Just follow the markers, and hopefully you will find what you are looking for.

Fundamentals

Elements of a Program

Every program will comprise a number of **STATEMENTS**.

Statements are often grouped into logical blocks referred to as **SUBROUTINES** or **FUNCTIONS**.

Statements can take a number of forms, the simplest being a **COMMAND** such as:
END

A very common form of statement looks like a mathematical equation, but works in a slightly different way. An example might be:

```
X = Y + 2
```

Here, '**X**' and '**Y**' are examples of **VARIABLES**.
The value '**2**' is an example of a **NUMERIC CONSTANT**.
'+' is an example of a mathematical **OPERATOR**.

The right-hand side of the 'equation' '**Y + 2**', is an example of an **EXPRESSION**.

When the statement is executed, the value currently stored in variable '**Y**' is added to the constant value '**2**', and the result is then stored in the variable '**X**'.

You will often see statements like **X = X + 1,** which means take the current value of variable 'X', add one to it, and store the result back into variable 'X'.

So if the value of 'X' is initially 3, the program will add one to it, giving 4, and this is then stored back into 'X'.

If you could watch what was happening to the value held in variable 'X', it would be '**3**' just before the statement **X = X + 1** was executed**,** and **'4'** immediately afterward.

Variables

Variables are the quantities used in your program to represent values such as speed, price, diameter, etc.

A program will use the computer's working memory to store information during processing.
This information is referred to as **DATA**.

An item of information stored in this way is known as a variable. It is a variable because the computer can alter the stored value whenever instructed by a program statement.

Variable Names

Each variable is given a name of your choice, and this can be any combination of letters and numbers (but no special characters other than Underscore are allowed).

Variable names must begin with a letter (or Underscore _).
They are not case sensitive, and names can be up to 64 characters long.

You can write for example: **costprice**, **COSTPRICE**, or **CostPrice**.

All of these are acceptable names, and EBasic regards all of them as the same quantity.

CostPrice is probably the preferred form for readability.

Variables are an important part of every program. They represent the quantities you wish to process to obtain a desired result. You can have as many variables in a program as you wish.

Variable Type Definition

Every variable needs to be defined to hold a particular data type using a definition statement .

It is best to **explicitly define** all your variable types before they are used. This helps to avoid hard to locate errors in your programs. By default, the compiler will automatically set the type of each variable based on the type of data first loaded into it.

Use the statement: autodefine "OFF" placed near the start of your program.

If Autodefine is set to "OFF" as suggested, all variables must then be defined before they are used. Otherwise, the compiler will complain to point out the omission.

EBasic has many data types. We will examine just some of the most commonly used ones.

Integer Data

Here is a small program to demonstrate the use of integer variables.

```
int i, j, total
i = 5
j = 3
total = i + j
print "The total is: ", total
```

It **specifies** (defines) three variables named '**i**' ,'**j**', and '**total**' as **int**(egers).

(An alternative definition format is: DEF i, j, total AS int)

Variable 'i' is then loaded with the integer value '5', and variable 'j' with the value '3'

The variable 'total' is then loaded with the sum of 'i' and 'j'.

The 'print' command prints the result stored in variable 'total', which will be '8'

As you can see, it is possible to use descriptive text to explain the quantities that are shown on the screen.

Notice that the calculation of 'total' must be placed after the statements defining 'i' and 'j'.

Do not use any variable, which has not been previously set (initialised) on the left-hand side of a statement.

It may contain zero, or any indeterminate quantity that happens to exist at that memory location .

Integer variables can hold either positive or negative whole-number values in the range –2,147,483,648 to 2,147,483,647.

(This should be sufficient capacity for most programs, but if you need to use even larger numbers, there is a long integer type (defined as int64) which can hold positive or negative numbers as large as 9,223,372,036,854,775,807 .. phew!)

If you are sure your application will only require **positive** whole numbers, you can specify 'unsigned' integer variables uint (range 0 to 4,294,967,295) or uint64 (range 0 to 18,446,744,073,709,551,615).

(20 million, 100Gb hard drives, would hold 16,000,000,000,000,000,000 data storage bits).

Integer calculations can be relied upon to give absolutely correct answers (assuming your maths is correct).

Decimal Data

Now let us examine another important type of variable - the decimal or **'float'** type.

The strange name goes back in history as the 'floating-point' type - meaning that the decimal point can be moved to represent very large, or very small numbers.

Consider this small snippet of code:

```
float  x, y, product
setprecision 8
x=5
y=3.303
product = x * y
print "The product is ", product
```

When you run this program, you get the answer 16.51499939, not the 16.515 you were expecting. (By the way, if you don't specify a decimal precision, EBasic defaults to only 2 decimal places for display - of course it is working to greater precision behind the scenes).

Decimal numbers are tricky and have to be handled with care. Even if you increase the 'setprecision' statement value to 15 decimal places, you will still not get the correct answer.

You are encountering a disturbing feature of decimal numbers called **'precision'**.

A variable of type '**float'** can only represent up to **8 significant figures**. (In our example, the calculation can only accurately represent 16.514999). Any additional digits will not be accurate.

If you change the variable definition line to read: **double x, y, product**
you will get the correct result 16.5150000

We have now defined the working variables as **'double'** precision type, and they are now accurate to **16 significant figures**.

Normally, most mathematical calculations involving decimals will give accurate answers if you use **'double'** precision variables. However, if you need to carry out lengthy, high-precision calculations, always be on the lookout for loss of precision.

If you are happy with 'slide rule' accuracy, the 'Float' type variable will be more than sufficient.

String Data

String variables store text and get their name from the term "a string of characters".

Here is a small snippet of code to illustrate string data:

```
string s,t
s = "Here are 3 percentage signs - "
t = "%%%"
print s + t
```

Note that strings are enclosed in quotes.

When we print the result, the second string value follows the first. Why is this?

The print instruction appears to be adding two strings - in fact, this operation is called "concatenation". It simply appends one string to another.

String variables can hold up to **254 ASCII** characters.

EBasic has an advanced string type known as an **ISTRING**. This type is in fact an array of characters, so you need to specify the maximum number of characters when you define an **ISTRING** variable.

For example: istring name[30]

The variable 'name[]' can now hold up to 30 characters. Like all arrays, the **ISTRING** variable is zero-based.

You use a statement like: name[0] = "Donald Duck" to load the entire name.

You can then have a statement like: char = name[2] , and this would place the letter 'n' in the variable 'char'.

A variable defined as type **ISTRING** can hold as many characters as can fit in memory.