# Basic Reference

At present this reference is for version 0.34

!

! is an indirection operator that does a similar job to PEEK and POKE, e.g. accesses memory. It can be used either in unary fashion (!47 reads the word at location 47) or binary (a!4 reads the word at the value in address a+4). It can also appear on the left-hand side of an assignment statement.

!a = 42 print !a print a!b a!b=12

%

Binary constant prefix. %101010 is the same as the decimal constant 42.

%101010

&

Hexadecimal constant prefix. &2A is the same as the decimal constant 42.

&2A &FFFE

' rem

Comment. ‘ and rem are synonyms. The rest of the line is ignored. The only difference between the two is when listing, ‘ comments show up in reverse to highlight them. The remainder of the line is still tokenised, so it is advised to use alphanumerics, commas, full stops only. Some comments (e.g. rem && hello) will not tokenise.

‘ My Program Rem rem Hello world

\*

Binary multiply

4\*2

+

Binary add or string concatenation.

4+2 “Hello “+”World !”

-

Binary subtract

44 – 2

.

Sets the following label to the current assembler address

.myLabel

/

Signed binary division. An error occurs if the divisor is zero.

420/10

< <= <> = > >=

Comparison binary operators, which return 0 for false and -1 for true. They can be used to either compare two integers or two strings.

A<42 c$>=”Hello”

? print

Prints to the current output device, either strings or integers (which are preceded by a space). Print a ‘ goes to the next line. Print a , goes to the next tab stop. A return is printed unless the command ends in ; or , . ? is a synonym for print

Print 42,”Hello”’”World”

abs()

Returns the absolute value of the parameter

Abs(-4)

and

Binary and operator. This is a binary operator not a logical, e.g. it is the binary and *not* a logical and so it can return values other than true and false

Count & 7

alloc()

Allocate the given number of words of memory and return the address.

Alloc(32)

asc()

Returns the ASCII value of the first character in the string, or zero if the string is empty.

Asc(“\*”)

assert

Every good programming language should have assert. It verifies contracts and detects error conditions. If the expression following is zero, an error is produced.

Assert my.age = 42

blit

Sends one command to the display blitter. This is documented in the Eris hardware document. The parameters are x and y position, data source, mask/colour and command.

Blit 42,10,gfxData,&F6,16

call proc endproc

Simple procedures. These should be used rather than gosub. Or else. The empty brackets are mandatory even if there aren’t any parameters (the aim is to use value parameters).

Call print.message(“Hello”,42)

….

Proc print.message(msg$,n)

Print msg$+“World x “+str$(n)

Endproc

chr$()

Convert an ASCII integer to a single character string.

Chr$(42)

clear

Erase all variables. This does *not erase* integer variables A-Z which are permanent

clear

cls

Clears the text screen. Paradoxically this does not … clear the screen. Only the text screen. If there is a sprite plane, this will be unaffected by the cls command.

Cls

code

Code initialises an assembler pass. Apart from the simplest bits of code, the assembler is two pass. It has two parameters. The first is the location in memory the assembled code should be stored, the second is the mode. At present there are two mode bits ; bit 0 indicates the pass (0 1st pass, 1 2nd pass) and bit 1 specifies whether the code is listed as it goes. Normally these values will be 0 and 1, as the listing is a bit slow.

Code &6000,1

cursor

Move the text cursor to the given text coordinates

Cursor 22,10

dim

Dimension integer or string arrays with up to two dimensions

Dim a$(10),a.score(10) dim name$(10,2)

dir

Displays the local file system directory.

dir

end

Ends the current program and returns to the command line

End

exists()

Check if the file exists in the local file system

Exists(“demo.dat”)

false

Returns the constant zero.

false

fkey

Programs the function keys f1-f6 (also accessible by Ctrl 1-6). Sets the string that is put into the keyboard buffer when you press the key. To make it enter, follow with chr$(13) (Carriage Return)

Fkey 1,”List”+chr$(13) fkey 2,”\*\*\*\*”

for to step next

Loop which repeats code a fixed number of times, which must be executed at least once. The default step is 1 but can be negative. The final letter on next is optional.

For I = 1 to 10 step 2:print i:next i

frame

Draws a rectangular frame in the current ink colour. This has two forms ; with just two coordinates it draws from the last position accessed, with four it draws a frame using those two coordinates.

frame 42,120 frame 10,10 to 200,30

get$()

Get a single keystroke from the keyboard and return as a single character string

Get$()

get()

Get a single keystroke from the keyboard and return as an ascii value

Get()

gosub

Call a routine at a given line number. This is provided for compatibility only. Do not use it or I will hunt you down and torture you.

Gosub 1000

goto

Transfer execution to given line number. See GOSUB ; same comment.

Goto 666:rem “If you use Goto. You don’t need it”

if then else endif

If has two forms. The first is classic BASIC, e.g. if <condition> then <do something>

If name=”Benny” then my.iq = 70

The second form is more complex. It allows multi line conditional execution, with an optional else clause. This is why there is a death threat attached to GOTO. This is better. Note the endif is mandatory, you cannot use a single line if then else.

If age < 18 print “Child” else print “Adult” endif

ink

Sets the text drawing colour.

Ink 4

inkey$()

Sees if a new key press has occurred, if so returns it as a string, if no key available returns “”. It’s like get$() except it doesn’t wait.

Inkey$()

inkey(

Sees if a new key press has occurred, if so returns its ASCII value as an integer, if no key available returns 0. It’s like get() except it doesn’t wait. You probably guessed.

Inkey()

input

Inputs a string or an integer which can be edited using backspace and ends with return. The string cannot go beyond one line. Very limited. Deliberately.

Input a$

joyb()

Checks the joystick button. As there are two , this has a parameter , 1 or 2.

Joyb(2)

joyx()

Current x position of the joystick, which is a simple controller non analogue type (or also the keyboard), returns -1 0 or 1

Joyx()

joyy()

Current x position of the joystick, which is a simple controller non analogue type (or also the keyboard), returns -1 0 or 1

Joyy()

left$()

Returns several characters from a string counting from the left

Left$(a$,4)

len() length()

Returns the length of the string as an integer

Len(“Hello, World”)

let

Assignment statement. Can be assumed (except for PAGE).

Let a = 42 a$=”Hello”

line

Draws a line in the current ink colour. This has two forms ; with just two coordinates it draws from the last position accessed, with four it draws a line between those two coordinates.

Line 42,120 Line 10,10 to 200,30

list

Lists the program. Can start from the beginning or a given line number or a procedure. This is slightly unusual in that it clears the screen and lists from the top down and stops when the screen is mostly full. This is deliberate. The indentation is automatic and doesn’t work for else at the time of writing.

List List 1000 List myProcedure

load

Loads a file from local storage. The load address is part of the file and can be overridden so a file can be loaded elsewhere. If the file is loaded to the basic Program area (e.g. the value of PAGE) it is assumed to be a BASIC program and variables are cleared ready to run the program.

Load “myprogram” load “my.data”,&7800

local

Defines the list of variables (no arrays allowed) as local to the current procedure. The locals are initialised to an empty string or zero depending on their type. They can also be followed by an assignment statement, but only once per local (whereas unassigned locals can be a long list)

Local test$,count Local count=42

lower$()

Converts the string to lower case.

Lower$(“Hello, world !”)

mid$()

Returns a subsegment of a string, given the start position (first character is 1) and the length, so mid$(“abcdef”,3,2) returns “cd”. It does, I just tried it. The third parameter is optional and if omitted means “to the end of the string”

Mid$(“Hello”,2,3) mid$(“Another word”,2)

mod

Binary modulus operator. The second value must be non-zero.

42 mod 5

move

Moves the graphics cursor to the given coordinate position

Move 42,120

new

Erases the current program

New

old

Recovers the current program or tries to. Some things will corrupt it, and it won’t work, and the first line might be corrupted. It usually sets the line number of the first line to zero. But then using line numbers for anything other than editing is a capital offence anyway. How well this works depends on what you’ve done.

old

or

Binary or operator. This is a binary operator not a logical, e.g. it is the binary and *not* a logical and so it can return values other than true and false

Read.value or 4

page

Sets the program location or gets it. It behaves like a variable and moves the basic program address about. So, if you change it, it can do wacky things if you don’t have program there. I’m not psychic

Print page let page=$6800

palette

Change the palette colour. This is not instant as in hardware palettes because some implementation are not hardware palettes. So, this should be set before you start drawing stuff. Don’t use it to do things like flashing the screen at the end of a Pacman level, it won’t work. The parameters are a colour number, the plane (0 background 1 foreground) and a BGR value from 0-7. The range of colour numbers depends on how you set the screen up. I’ll probably write more under SCREEN if I remember.

Palette 0,0,16

paper

Sets the text drawing colour

Paper 12

peek()

Returns the contents of the given memory location. I prefer using the pling operator but if you want it it’s here

Peek(a)

plot

Moves the graphics cursor to the given coordinate position and plots a pixel in the current ink colour

Plot 42,120

poke

Sets the contents of a memory location. See peek() regarding pling.

Poke 32768,42

rect

Draws a solid rectangle frame in the current ink colour. This has two forms ; with just two coordinates it draws from the last position accessed, with four it draws a rectangle using those two coordinates.

rect 42,120 rect 10,10 to 200,30

renumber

Renumbers the program. This is a very simple renumber that renumbers the current program in steps of 10 from 1000. This does ***not*** work with GOTO and GOSUB , and this is deliberate, and it will not change.

renumber

rnd() random()

Generates random numbers. This has two forms, which is still many fewer than Odo. Without a parameter, it returns a random integer in range (-32768..32767). With two parameters it returns a number between them inclusive, so random(1,6) could be used to simulate a die. Rnd() and random() are synonyms. I like clarity.

Rnd() random(1,6)

repeat until

Conditional loop, which is tested at the bottom.

repeat

call drink.a.pint()

until alcohol.in.blood > 100

return

Return from GOSUB call. You can make up your own death threats.

return

right$()

Returns several characters from a string counting from the right

Right$(a$,4)

run

Runs the current program after clearing variables as for CLEAR. If a filename is provided, that file is loaded and run. Running programs can be stopped using the Ctrl+Space combination. If the program “autoexec.prg” exists in the local file system, it is run automatically when BASIC starts.

Run run “myProgram.bas”

save

Saves to the local store. This can either be the current basic program which is the default when just a file name is given, or any area of memory, in which case there are three parameters, the file name, the start and the length. The saved file will automatically load into the same space

Save “my.program” save “some.data”,&6800,&800

screen

The Eris display is built using two planes, the foreground which is used for sprites, and the background which is used for text and fixed graphics. The screen display allocates the 4 planes available on the base system between the background, the first parameter and the foreground, the second parameter. It is possible to have no foreground or no background if you wish. The screen is erased, and the default palette is loaded.

The system defaults to 0,4 e.g. no foreground/sprite plane and 4 text planes, which gives a 16-colour text display (because 2^4 = 16). Screen 2,2 reorganises this so there are 2 foreground planes and 2 background planes. The background plane has four colours and the foreground three (because on the foreground colour 0 is transparent).

So, you can have no sprites and 16 colours, a single sprite colour and 8 colours, 4 sprite colours and 4 background colours and so on. Later implementation may allow the use of more colours without affecting background compatibility.

This is done entirely in software ; in reality it is a single 4-bit plane with (limited) palette mapping (see PALETTE) in which any plane or planes can be written to by the blitter. The use of palettes and different bit allocations creates this 2-layer display.

The default palette is calculate from BGR where B is bit 2, G bit 1 and R bit 0, 0 = Black, 1 = Red, 2 = Green, 3 = Yellow, 4 = Blue, 5 = Magenta, 6 = Cyan, 7 = White, and is repeated where necessary. This is of course reprogrammable. As stated earlier, the reprogrammable palette should not be used for palette effects, e.g. flashing the screen by changing the palette. Direct hardware palettes are not required ; it simply affects all future writes. The emulation and ESP32 versions do this differently. If you draw something on the screen in a colour and change that colours palette definition, it will change instantly on emulation, but not on the ESP32. This is deliberate.

Screen 0,4 screen 3,1

sgn() sign()

Returns the sign of an integer, which is -1 0 or 1 depending on the value.

Sgn(42)

stop

Stops program with an error

stop

sys

Executes a machine code routine at a given address. On entry, R0 points to the fixed variable A, and variables B-Z follow in logical sequence

Sys &FFE4

sysvar()

Returns a kernel system value. These are listed in the kernel source

example

to.number() val()

Converts a number to a string. There must be some number there e.g. “-42xxx” works and returns 42 but “xxx” returns an error. It’s a compatibility thing and I’ll probably replace it with something usable.

Val(“42”)

to.string$() str$()

Converts a string to a number, in 16 bit signed decimal form.

Str$(42)

true

Returns the constant -1

True

upper$()

Converts the string to lower case.

Upper$(“Hello, world !”)

while wend

Conditional loop with test at the top

While wife.very.cross

Call buy.flowers()

Call grovel()

wend

xor

Binary exclusive or operator. This is a binary operator not a logical, e.g. it is the binary and *not* a logical and so it can return values other than true and false

A xor &0E