Basic Reference

At present this reference is for version 0.95

As of version 0.93, variables do not auto-instantiate ; they must be defined either by assignment, local, for or as parameters, otherwise an error is reported.

!

! 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 alphanumeric, 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”
```

@

Returns the address of a l-expr, normally this is a variable of some sort, but it can be an array element or even an indirection. (print @!42 prints 42, the address of expression !42, not that it’s useful at all ….)

```
print @fred, @a(4)
```

? 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. The data source can either be a physical address, if >= 256, or an image number if 0-255

```
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. This also clears the sprite plane and resets all the sprites (to position 0,colour 0,flip 0,character 0)

```
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
```

delete

Deletes a named file. If the file does not exist, causes an error.

```
Delete “test.file”
```

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
```

draw

Draws an image on the current display in the current colour. The image can either be an index 0-255 or an address an optional dim parameter can be added which can draw the sprite double size (or single size if you use a variable)

```
draw 100,100,1   draw 200,200,&801E dim 2
```

draw on

Selects the plane that graphic operations operate on. Takes one parameter ; 0 draws on the background, the default, 1 on the foreground. This does not affect print and similar which are always on the background.

```
Draw on 1
```

end

Ends the current program and returns to the command line

```
End
```

event()

Event tracks time. It is normally used to activate object movement or events in a game or other events, and generates true at predictable rates. It takes two parameters ; a variable and an elapsed time.

If that variable is zero, then this function doesn't return true until after that many tenths of seconds has elapsed. If it is non-zero, it tracks repeated events, so if you have event(evt1,10) this will return true every 1/10 second.

Note that if a game pauses the event times will continue, so if you use it to have an event every 20 seconds, this will work – but if you pause the game, then it will think the game time has elapsed. One way out is to zero the event variables when leaving pause – this will cause it to fire after another 20 seconds.

The maximum value for the event is 32 seconds. If the event variable is set to -1 it will never fire, so this can be used to create one shots by setting it to -1 in the conditional part of the line

```
If event(event.move,10) then call move()
```

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 except for typeins of old listings or I will hunt you down and torture you. Note that renum ignores line numbers, so if you renum something with goto/gosub in it, it will stuff it.

```
Gosub 1000
```

goto

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

```
Goto 666:rem "Will happen if you use Goto. You don't need it"
```

hit()

Checks if two sprites have collided. Note, this is done on positions and a hit box, so if you move them in steps bigger than the size of the sprite the may not work. The parameters are the two sprites to check, and optionally a hit box size of 0..31, the default being 15.

```
Print hit(3,2),hit(4,2,8)
```

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$
```

is.number()

This is a support for val / to.number and takes the same parameters, a string and an optional base from 2..16. This deals with the problem with val() that it errors if

you give it a non-numeric value. This checks to see if the string is a valid number in the given base (or base 10) and returns -1 if so, 0 if it is not.

```
is.number("42",16)        is.number("I like chips in gravy")
```

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()
```

key

Checks if a key is pressed. This takes one value, which is the row number on the keyboard map in the hardware reference, multiplied by 16 and added to the column number. So 'F' is row 2 column 3 on the map, which is 2*16+3, so either 35 or &23 in hexadecimal is clearer. Returns 0 or -1.

```
if key(2,3) then print "F pressed"
```

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. The LET is optional (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. Line numbers 0 and 32768-65535 are hidden.

```
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 !")
```

max()

Returns the larger of the two parameters using signed comparison

```
print max(4,5)
```

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)
```

min()

Returns the smaller of the two parameters using signed comparison

```
print min(4,5)
```

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
```

not

Unary operator returning the logical not of its parameter, e.g. 0 if non-zero -1 otherwise.

```
Print not 42
```

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 consistently.

This might sound dumb, but it is deliberate ; because some things are harder to implement in some forms than other. With FABGL which underlies the ESP32 for example, changing the screen background colour involves changing every single pixel, because there's no palette mapping.

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. Remember memory is *words* not bytes.

```
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. The pling operator is another alternative, and as with peek() this operates on words, because eris has word memory.

```
Poke 32768,42
```

quiet()

Returns true unless the given sound channel is either playing a sound or has one queued, in which case it returns false.

```
Print quiet(2)
```

randomise

Sets the random seed. This value will cause an error if the seed is zero.

```
Randomise 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
```

renum

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, so if you use them you're on your own.

```
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)
```

rpl()

RPL is a programming language with a close resemblance to FORTH, an alternative to the inline assembler. It uses code in the same way. See the RPL documentation for more details

```
sys rpl(42 3 + . )
```

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 1,3 e.g. 1 foreground/sprite plane and 3 text planes, which gives an 8-colour text display (because 2^3 = 8). There are only 8 colours in the reference implementation :)

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. You can kill all sprites using CLS.

```
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)
```

slide

Plays a sound on the given channel, for a given time period (in 1/10s) at a varying pitch up or down. Parameter 1 is the pitch, Parameter 2 is the length. The variation is the same calculation as the value in “sound” This is rounded down by dividing by 64, so -63..63 are effectively zero. This makes weird sound effects ; best used experimentally.

```
Slide 1,-4242,10
```

sound

Plays a sound on the given channel, for a given time period (in 1/10s) at a given pitch. Parameter 1 is the pitch, Parameter 2 is the length. The pitch is a wavelength ; the frequency is 5,000,000 / n. This is rounded down by dividing by 64, so is approximate. It is accurate except at high pitches. Channel 0 is the noise channel, and the last parameter is ignored. Channel 1 and 2 are square wave tones. A table of pitches is in the documents folder.

```
Sound 1,10000,10
```

sprite

The sprite command controls sprites. Which you probably guessed.

The sprite load loads a sprite file produced by the sprite editor into the sprite image area, which supports up to 16 (at present) 16 x 16 graphic images. This can be used separately as BLIT uses the same data.

The remaining sprites require a sprite plane – which is set up in SCREEN. If you set the screen to 4,0 for some reason it will give an error. There are up to 16 (at present) independent sprites, each 16x16 or 32x32 pixels single colour. Each can be positioned on the screen, have their colour or graphic set, and be horizontally or vertically flipped, or both.

The first parameter is the sprite number, which is 0-15. The remainder are commands followed by parameter(s). TO x,y sets the position of the sprite centre, so if you do TO 0,0 you will only see the bottom right quarter, INK n sets the colour, DRAW n sets the image number and FLIP n sets the orientation (0 = normal,1 = vertical, 2 = horizontal, 3 = both). DIM sets the scale to 1 or 2, defaulting to 1. END kills the sprite

Sprites operate independently of the text screen and everything else – if you break into a sprite program the sprites will still be there and you can edit under them, which is a bit surreally. Commands can be changed – the second line below sets all four parameters in one go. Ink 0 is not black but transparent, so if you do SPRITE 1 INK 0 it becomes invisible. The sprites are normally stacked from 0-15 and this should hopefully work mostly :) 15 is the topmost. But don't rely on it without testing, though because this is all done in software it isn't implementation dependent.

```
sprite load "demo.spr"    sprite 1 ink 2    sprite 4 end
sprite 5 to 100,100 ink 3 flip 4 draw 7 dim 2
```

sprite.x() sprite.y() sprite.info()

These functions extract the position and status of the given sprite. The status word is described in the documentation.

```
Print sprite.x(1),sprite.y(1)
```

stop

Stops program with an error

```
stop
```

sub.count()

Used for data strings. These are strings of arbitrary data that can be seperated with an arbitrary character e.g. "12,42,98". Sub.count() returns the number of data items, so in the example given, it would be 3 (if the seperator is a comma)

```
count = sub.count("12,42,98",",")
```

sub.get$()

Used for data strings, in conjunction with sub.count(), returns an arbitrary element from that string list. So if the data in a$ “fred,jim,jane,jack” then sub.get$(a$,”,”,2) would return the string “jim”

```
name$ = sub.get$(“jane,paul,lizze,jack”,3)
```

sys sys()

Executes a machine code routine at a given address. The parameters are firstly the address to call, and then the values to go into R0,R1,R2,R3,R4,R5 – a maximum of 6 is allowed. These must be integers (to pass strings use @f$). Additionally R8 is set to the RPL stack and must be maintained (if it is changed it assumes the stack has not balanced in the call).

The unary function returns the value returned in R0, whereas the command version ignores it.

```
Sys &FFE4  loadaddr=Sys(&10,6,@f$)
```

sysvar()

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

Example

text

Draws text as graphics at the current position, current ink, current plane. This is completely different from PRINT which is a console text operator. It takes a coordinate pair and a string to print. Optionally you can add a scale on as an extra parameter which draws large characters

```
Text 10,10,”Hello, world!”  Text  4,5,”Ok!”,5
```

tile

Tilemaps are rather complicated. They are an array of 16x16 images drawn at a location that can be scrolled around.

There are seven parameters.

- The first two are the graphic position of the map on the screen
- The second two are the position in that map, in pixels of the top left drawn graphics position.
- The third is the size of the tilemap in tiles, e.g. 10 x 5 => 160 x 80, 16 pixels each dimension.
- The fourth is the tilemap data, which is described in the hardware reference.

The tile drawing will overflow if the tile boundaries do not align with the tiles – e.g. each offset is a multiple of 16. In this case, it will still draw the map correctly, but it will overwrite the 16 pixels on all four sides of the tilemap area. This is because if pixel tile scrolling is used, it is necessary to draw part of a pixel, and to change this in software would be too slow.

There are two solutions. One is to put a tilemap on the entire screen, so it is clipped by that. The other is to blit a rectangle to cover this onto the sprite layer – because tiles are blitted to the background, if you put a solid colour over it, it will hide the scrolling effect.

```
Tile 20,20,10,-4,10,5,tileMap
```

timer()

Returns the value of the hardware timer. This is a 16 bit signed integer which increments at a rate of 100 a second. Note that specific values are not guaranteed, and testing should be done on a signed basis, because subtraction wraps around, so use code like the example, not using equals or comparison. Comparison is signed, so -32768 ($8000 hex) is less than 32767 ($7FFF hex) even though the timer will increment from 32767 to -32768.

```
t1 = timer():repeat until timer()-t1 > 0
```

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. A second parameter can set a conversion base from 2 to 16, but defaults to 10. This is for compatibility. To make it useable use the function is.number() which checks to see if it is valid.

```
Val(“42”)   val(“4C3”,16)
```

to.string$() str$()

Converts a string to a number, in signed decimal form. A second parameter can set a conversion base from 2 to 16 and in this case it is done unsigned.

```
Str$(42)  str$(412,16)
```

true

Returns the constant -1

```
True
```

upper$()

Converts the string to lower case.

```
Upper$(“Hello, world !”)
```

wait

Wait for the given amount of centi-seconds (e.g. 1/100 of a second). While you are waiting any background things – sprite repainting, break checking continues.

```
wait
```

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
```