Basic Reference

Last update 8th April 2021. Reference for version 0.30

! ?

Indirection operators, modelled on the Acorn Atom/BBC Micro BASIC model, operating on bytes (?) and words (!). These can be used in unary or binary mode, so a!4 refers to the 32-bit word at a+4 and !ptr refers to the 32-bit word at ptr.

They behave like a variable, except the variable is a specific byte or word in memory.

They can also be used on the left-hand side of expressions.

For those who prefer that syntax, PEEK and POKE are implemented with 16- and 32-bit variants.

```
!ref = 0
?count = ?count + 1
record!16 = -1
```

&

& is a prefix to allow hexadecimal constants to be used.

```
count = &2a
```

' rem

This is a comment. REM and ‘ are functionally identical. This is slightly different to standard BASIC in that it can have either a quoted string, or it can be on its own (to indicate a blank line). This (along with DATA, which has a similar variation) is for syntactic consistency.

```
rem “i am a comment”.
‘
‘ “this procedure does something interesting”
‘
```

+ - * / mod^

Numeric binary operators. These behave in the usual manner with standard precedence. They will operate 32-bit integer mode, unless one of the values is a float, in which case they will operate in float mode, any integers being coerced to floats.

A consequence of this is that division of two integers always produces an integer.

They can also be used to concatenate strings.

```
a = (b + 2) * (x11 + my.age)
test$ = “hello”+”,world”+chr$(33)
```

< <= <> = > >=

Binary comparators, which can be used on strings or numbers. As with the numeric operators, they will operate in integer mode if both are integers, if one is a float the other will be coerced to be a float if it is not already.

These are also binary operators, returning the values -1 and 0 if true and false.

```
if a <> 2 then print “not two”
iszero = (count = 0)
```

<< >>

Shift operators, like C operators, which only operate on integers. These are much quicker than multiply if you are using powers of 2 (e.g. a << 1 vs a * 2). These are unsigned; hence zeros are shifted in either end.

```
a = temp << 2
print 1 << power
```

@

@ is normally used to obtain the address of a variable and returns the data relevant to that variable. This data is a 2-byte address (string) which points to a length prefixed ASCII string, a 4-byte integer, or a 6-byte float. Messing around with this is not advised. It can be used for general dereferencing, so @a!4 will return the address a+4 (where as a!4 returns the data at a+4). Could be used to pass integer references to procedures.

```
proc test(@x1)
.....
defproc test(px)
  !px = -1
endproc
```

abs(n)

Returns the numeric absolute value of its parameter.

```
print abs(42),abs(-42)
```

alloc(memSize)

allocates a block of low memory of the given size.

```
do.something.with.me = alloc(128)
```

and or xor

Binary operators. These are numeric not logical, so they calculate the actual binary value of anding, oring or xoring the integers they are given. This works with the comparison operators which return

true and false, but you cannot write “if a or b then” unless you know both a and b are true or false, you cannot assume non zero will work.

```
if age>12 and age<20 then print “get off that phone”
```

asc(string)

Returns the ASCII character code of the first character of the string. If the string is empty it will raise an error

```
print asc(“*”)
```

assembler code

The 6502 assembler is modelled on that of the BBC Micro, e.g. it is inline code, and the “2 passes” must be done manually. The main differences are the full 65C02 instruction set, there are no [and] delimiters, and that the control variables are P and O, not P% and O% as this BASIC defaults to the integer type. Labels are prefixed with a full stop. Code looks like this.

```
for o = 0 to 1
  p = &200
  ldx #0
.loop
  dex : bne loop
  rts
next o
```

Notes:

O controls the pass (bit 0 is set for the 2nd pass) and listing (bit 1 is set to list out the hexadecimal code as produced on either pass). So this produces no listing – “0 to 3 step 3” produces a pass 2 code listing. The original source code is not displayed as the BBC Micro does.

P controls the assembly location, which in this example is at &200 hexadecimal, and increments as code is generated.

As with most assemblers, the “A” is optional – you can write asl or asl a. However you cannot use variable A – it will assume you mean accumulator.

Variables are used for labels – so .loop is saying “set loop to whatever value P is”

As with the BBC Micro, you can write code in BASIC to perform macros and similar.

assert <expression>

Asserts a contract, causing an error if it fails.

```
assert ptr != 0
```

break

Enables or disables the break key that stops a program running and can interrupt a GET() GET$() or

INPUT() when waiting for user input. The break that operates to interrupt listing is always on.

```
break true
break false
```

chr$(code)

Converts a number to a one character string with the character that ASCII code represents: the inverse of asc()

```
print chr$(42)
```

clear

Resets all internals ; clears return stacks, locals, variables, resets the allocation pointer. However, it does not clear integer variables A-Z which are stored separately from all others.

```
clear
```

cls

Clears the text display.

```
cls
```

defproc name(parameters) ... endproc

Defines and ends a procedure, which can have optional parameters. At present these parameters can only be single variables, e.g. a string, integer or float. These parameters are local to this procedure so in the example below, if there is an 'n' in the caller it is not accessible.

If there are no parameters the brackets must still be present, just empty.

defproc must be at the start of a line. Which you should be doing anyway.

```
defproc printdouble(n)
  print n * 2
endproc

defproc noparameters()
defproc many.parameters(a1$,b,count.people)
```

dim

Create an array of a given dimension. Multiple dimensions are permitted. Indices in arrays start from 0 so dim a$(5) creates a six element array.

```
dim a$(5),count(22),n(3,4)
```

end

Stops the program from running and returns to the command prompt.

event(eventvar,rate)

Event is a function for activating regular events. It takes two parameters, an integer variable and an expression which is a time in 60Hz ticks. It returns TRUE at a rate equivalent to that expression e.g. the following code will print “Ping” every 1/2 second (30 ticks, at 60Hz). The maximum allowed elapsed time is 32,767 ticks, which is about 9 minutes. If you set timer variable (event.timer) to -1 that will stop it firing entirely.

```
event.timer = 0
repeat
  if event(event.timer,30) then print “ping !”
until false
```

false true

Return the constants 0 or -1 respectively. Syntactic sugar that means you can use sort of boolean variables.

```
ismale = false
if ismale then print “male !”
```

float(n)

Returns n as a float.

```
count# = float(n)
```

for ... next

Standard for/next loop, with step options, can count up and down. The index must be an integer. The index variable being specified as part of next is optional.

```
for i = 10 to 1 step -2: print i:next i
```

get() get$()

Waits for a character to be pressed (depending on the system, there may be one already in the keyboard buffer) and returns it as a character or ASCII code. The exact behaviour of this is system dependent.

```
if get() == 65 then print “you pressed a”
```

gosub ... return

Included for backwards compatibility. Don’t use them for anything else. Also, I’m not writing a renumberer that works with these awful instructions, so if you insist you are on your own. There’s a reason I spent time making PROC and LOCAL having all the bl**dy parameters and things. Think how depressed using these instructions will make me feel after all that effort. Have a heart.

goto

See GOSUB. I'd really should change the license, so it is "free, unless you use GOTO GOSUB and RETURN for new code in which case it's £1,000 to use it". This isn't 1979 you know. If you're typing in "Hunt the Wumpus" from an old Creative Computing or something, then you're let off.

if ... else ... endif

The long form if statement. This has an else clause and can spread over multiple lines and be nested. Note there is *no* THEN (this is how it tells the difference syntactically) and ENDIF is required, even if it is all one line.

You can nest If/Then within if/else/endif

```
if a$ = "y":count=count-1:else:count=count+1:endif
if count=0
  <lots of stuff,loops, calls, other ifs>
endif
```

if ... then/goto

The short form if statement. This consists of if, followed by either then or goto <line number> (see GOTO). This is a single line if typical of standard BASIC. There is no else option.

```
if a = 1 then print "a is one. whoopee !"
if a = 2 goto 13
```

ink n / paper n

Sets the text background colour or foreground colours. Colour IDs are BBC Micro ones (0-7 : Black Red Green Yellow Blue Magenta Cyan White)

```
ink 4
paper 2
```

inkey() inkey$()

Like get() except it doesn't wait for a key. If a key is in the input stream, it returns the code of that key, or the key, otherwise it returns "". This is not suitable for game writing purposes really beyond the quite simple.

```
print inkey()
```

input

Standard BASIC keyword. Input a typed line ending in RETURN. The Backspace key functions also. The parameters are a list of strings, which are printed, and variables, which are input.

```
input count
input "what is your name ",name$
```

int()

Converts a number to an integer, truncating as necessary.

```
print int(3.7)
```

left$()

Returns the left characters of a string. For example, the example below will print “abc”

```
print left$(“abcde”,3)
```

len()

Returns the length of a string as an integer.

```
print len(“abcde”)
```

let <optional>

Assignment statement, the let being optional. The left side of the equals can be a reference as well.

```
let count = 4
!ptr = 0
```

list

Lists programs to the output device. On its own lists the whole program, or an individual line, or a start and end line of a range. The start and end of the range are optional.

```
list
list 1000
list 1100,1300
```

load

Loads a basic program or a file into memory. This is currently using the standard Commodore format which has 2-byte load address on the front of a file.

```
load “myprogram.bas”
load “somedata”,&1c00
```

local

Declares a variable as local to a procedure, so it can be used without affecting variables outside it with the same name. Local variables can be assigned a value as shown in the example. They can have the same name as a parameter, but this is not advisable as you then won’t be able to access the parameters. Locals should be declared straight after the procedure definition.

```
defproc test()
  local a,b = 42
```

```
  <code>
endproc
```

locate

Set the text cursor position. The parameters are the horizontal and vertical position.

```
locate 4,2
```

lower$(string) upper$(string)

Converts a string to lower or upper case.

```
print upper$("hello, world!")
```

max() min()

Returns the largest or smallest in a series of numbers or strings. If numbers, any float will cause the whole expression to be evaluated as a float.

```
print min(1,8,12,7,-5)
print max("hello","and","goodbye")
```

mem

Returns the amount of memory currently available, this is the difference between the highest address in use for strings, and the lowest address in use for variables.

```
print mem
```

mid$()

Returns the characters from a position in the string, either to the end, or for a given number of characters. The following print "bc" and "cde" respectively.

```
print mid$("abcde",2,2)
print mid$("abcde",3)
```

new

Erase the current program, variables, everything, and warm starts.

page

Value which is constant when running a program and returns the start of the tokenised program code.

```
my.program = page
```

peek() deek() leek()

Read memory, in 1 , 2 or 4 byte chunks.

```
a = peek(&8000)
vector = deek(&fffc)
```

poke doke loke

Write memory in 1 , 2 or 4 byte chunks.

```
poke &9f23,42
doke ptr,-1
```

print

Print a sequence of expressions on the output device - standard BASIC. A comma moves to the next tab stop, and a ‘ indicates a new line to be printed. If the command does not end in , or it will print a new line at the end automatically.

```
print “my name is “;earl$ ’ “next line”
```

proc

Calls a procedure. The expressions and their type must match the ones in the definition.

```
proc do.something(2,”hello”,42)
```

random()

If no parameter is provided, returns a random 32 bit integer. If a parameter is provided, provides a random number up to that (e.g. random(6) will produce 0,1,2,3,4 or 5). Note that this uses modulus so is not fast compared to random() and 7

```
print random()
print random(6)+1
```

read ... data

These operate almost like standard BASIC read and data statements, reading data from DATA statements into BASIC variables. The difference is that string values must be placed in quotes. Data starts at the start of the program and is processed sequentially. You can return to the start with RESTORE

```
read age,name$
data 42,”dennis the menace”
```

repeat ... until

Standard repeat loop, executes the middle until an expression becomes true.

```
repeat
  proc drink.beer()
until alcohol.level > 100
```

restore

Resets the data pointer to the start of the program.

```
restore
```

right$()

Returns the right characters of a string. For example, the example below will print “cde”

```
print right$(“abcde”,3)
```

run

Clears everything and runs the program.

Save

Saves a BASIC program or area of memory. This uses the Commodore format with a 2-byte loading address, so if you save 16 bytes of memory it will create an 18 byte size file.

```
save “demo.bas”
save “rom.bas”,&c000,&e000
```

sgn()

Returns the sign of the number - 0 if it is zero, -1 if less than zero, +1 if greater than zero.

```
print sgn(42)
```

stop

Stops the current program from running with an error.

str$()

Returns a number converted to a string. Normally this is base 10, but for integers alternative bases can be requested - in the examples the last two print 2A (hexadecimal) and 10001 (binary)

```
print str$(4.7)
print str$(42,16)
print str$(17,2)
```

sys()

Run 6502 assembler. The parameter is the address to run. On entry, integers A,X and Y are placed in the A,X and Y registers, and the value returned in A is returned as a result of the function call as a byte integer (e.g. 0...255). The instruction should end with RTS.

```
print sys(&47ac)
```

timer()

Returns the number of 60Hz ticks since the start. Note, these values may not be sequential, so don't write something like repeat until timer() = 1000 but use > than (or an event() function). Some ports may scale this timer from 50Hz (3/4 is approximately right), so some values may not ever occur.

```
print timer()
```

val() isval()

These convert a string to a number. These functions replace the Microsoft BASIC equivalents. Both try to convert a string to a number, with a possible base - as with str$(), the inverse, this only works with integers. val() returns the value of that number and isval() returns True if it is a legal number.

```
print isval("cat")
print val("42")
print val("17af",16)
```

vdu

Sends the byte or word expression following to the character output. To send a word postfix with a semicolon.

```
vdu 42
vdu &2a07;9,13
```

while ... wend

Classic while loop. Repeats the loop body until the test at the top is false.

```
while is.wife.cross
  proc grovel()
  proc buy.flowers()
wend
```

xemu

Exits the emulator if running and dumps memory, used for testing. This may or may not be implemented but should only be used for development of the interpreter.

A note on tokenising

This BASIC tokenises completely differently from Microsoft BASIC. It divides things into identifiers, numbers, strings and punctuation. An identifier is a single indivisible element.

Spacing is thus required in some circumstances. In Microsoft BASIC, the variable name “total” would be tokenised as the token “TO” followed by TAL. Here it would be a variable TOTAL.

So you cannot write PRINTA, for example, which works in MS Basic, but here would be a variable called “printa”. You’d have to write PRINT A.

The upsides of this decision are that spacing is done automatically for you in listings, and you aren’t limited by variable names that start with short words like IF and TO. It also doesn’t store the spaces in the program.

An identifier begins with A-Z, and can contain 0-9, . and _ as well. It is advised, but not enforced that . notation is used for modules.

Graphics

The graphics system that draws lines, rectangles and so on is standard but optional. It requires the graphics driver code for the system which sets a position, draws a colour, moves that position and not much else. The standard graphic routines use this driver.

Graphics commands all use the same format. A command (PLOT, LINE, RECT etc.) is followed by a sequence of modifiers and coordinates, that are largely consistent. Most can be chained to do multiple actions (see the Draw Image example)

Modifiers persist from instruction to instruction (except TEXT) and are reset when the mode is changed. The last position can also be used as the start of the next position (see RECT example)

Drawing will not work if any point drawn is outside the screen area, e.g. LINE 0,0 TO 500,100 won’t show anything (500 being too large)

Modifier / Action	Effect
INK <colour>	sets the colour for drawing. These colours do not affect the console colours in the default part of the interpreter.
PAPER <colour>	sets the background colour for things like text. If zero, this is transparent.
DIM <size>	sets the scalar size for images / fonts
FLIP <value>	sets the flip value 0-3 for images and fonts. Bit 0 is horizontal flip, bit 1 is vertical flip. If possible the same as SPRITE FLIP
FROM <x,y> or just <x,y>	sets the start coordinate
AT <x,y> TO <x,y>	sets the end and action coordinate

IMAGE <n>	Sprite image n.
TEXT <text>	Text for drawing strings.

Graphics operations are not mandatory – to have them you need to implement the driver (and some form of sprites for PAINT). However, if implemented they should behave consistently, though fonts and sprite images can vary.

Drawing Functions

clg

Clear the whole graphics screen to the current or specified PAPER colour. Note that INK and PAPER are separate for text and graphics functions.

```
clg
clg paper 7
```

draw

Draws text on the display, with an optional background depending on the paper colour. Supports multiple integer scales for writing large text on the display. Can use either TEXT <text> *or* IMAGE <n> where <n> is the tile / font character number.

Currently this only works for the standard font.

```
draw text “hello world” ink 5 paper 1 dim 2 at 50,20
draw image 65 at 100,30 , image 67 at 140,30
```

frame / rect

Draws either an outline (frame) or a filled rectangle (rect) between the two corners

```
frame ink 2 from 10,10 to 100,100
rect to 200,200
```

line

Draws a single pixel size line between the two points in the current ink colour.

```
line ink 242 from 0,0 to 100,100 to 150,150
```

paint

Draws a sprite on the screen as an image – e.g. it is not an actual sprite, but uses the same image graphics. Integer scaling using DIM is permitted. The position is the top left of the graphic, not the centre as for sprites.

```
paint image 4 dim 2 at 10,10 flip 1
```

plot

Plots a single pixel at the given coordinate

```
plot ink 2 at 100,100
```

X16 Specific Extensions

clock()

Reads the RTC. The element to read is the parameter (0-5). 0-5 represent Years since 1900, Month (1-12), Day, Hours (0-23) Minutes and Seconds.

```
print clock(4)
```

hit()

The hit function checks for collision between two sprites. This is a rectangular collision, e.g. the overlapping of their 'bounding boxes' and takes no account of the actual sprite itself.

It has two or three parameters. The first two are the two sprites being checked for collision. The optional third sprite is an override range, which allows the collision distance to be changed arbitrarily – to 'hit' the distances between the two sprite centres in both the X and Y axes must be less than this range.

```
print hit(2,4)
print hit(2,5,8)
```

joy.b()

Return the status of the default joystick buttons. The parameter specifies the button 0=Start 1 = Select 2 = B 3 = A

```
if joy.b(2) then print "b pressed"
```

joy.x() joy.y()

Returns the offset on the default joystick axis, e.g. either -1,0 or 1

```
xpos = xpos + joy.x()*speed
```

mode

Mode sets the screen mode by reprogramming VERA. It has two forms.

The more complex one is a 32 bit constant where byte 3 describes the general setup, byte 2 is zero, byte 1 describes the set up for layer 1, and byte 1 describes the setup for layer 0.

Bit	Setup	Layer bits (as Vera Docs)
7	0	Map Height
6	Sprites on	
5	Layer 1 on	Map Width
4	Layer 0 on	
3	Vertical scale bits (00=x1 01=x2	Tile size (0=8x8,1=16x16)

2	10=x4 11=x8)	Bitmap Mode
1	Horizontal scale bits	Colour Depth 0-7
0		

This is obviously slightly complex than is ideal, so for mode values of 0-63 a standard mode is provided. Only one is provided at present, Mode 0, which is the normal 80x60 tile mode used to display the standard screen.

Layer 0's Map/Bitmap table is at VRAM $00000. Layer 1's Map/Bitmap table is at $10000, and Layer 1's tile lookup table is at $0F800. These are the standard X16 defaults and may change, so it is advisable to set these manually or read them from the Vera interface.

When a program stops and returns to the prompt, where you can type commands or edit lines, the system *automatically* switches to Mode 0, so you may need code to put a wait for a key press (e.g. a$ = get$()) in when you STOP or END. It does this so the mode is suitable for text editing.

```
mode &20006000
mode 0
```

palette

Defines a colour in RGB format. The colour is 0-255. The RGB format is 0-4095, usually stored in hex format &rgb where r g and b are the colour components from 0-F – so palette 1,&F80 sets palette 1 to Red=$F, Green = $8, Blue = $0 which is orange.

```
palette 1,&f80
```

playing(),playing(channel)

Playing has two forms. Without a parameter it returns the number of channels currently playing a sound. With a parameter it returns true or false depending on whether that channel is playing or not.

```
print playing() , playing(4)
```

sound

Sound allows the programmer to play one of the sixteen sound channels asynchronously. It has two forms. SOUND CLEAR turns everything off.

SOUND is followed by a series of modifiers rather like SPRITE and the graphics commands

AT <pitch> sets the pitch (see Vera documentation)

FOR <time> sets the play time in 1/10 seconds

TYPE <type> sets the type of the sound to 0-3 (see Vera documentation)

TO <channel> sets the target channel 0-15. Channel 255 means "find an unused channel"

STEP <volume> sets the volume from 0 to 63.

```
sound at 1181 for 10 step 48
```

sprite

There are two types of sprite command. The first one controls sprites in general.

```
sprite true      enables all sprites
sprite false     disables all sprites
sprite clear     erases all the sprite area to zero.
```

The second one controls an individual sprite by ID. It begins sprite <n> where <n> is the physical number 0...127. Commands follow this which can be chained together, so you can write “Sprite 4 to 100,100 image 3 flip 2” for example.

```
sprite n to x,y
```

move sprite to x,y. These coordinates are the sprite centre

```
sprite n flip f
```

set sprite flip to 0-3 (0 none 1 horizontal 2 vertical 3 both)

```
sprite n image n
```

set sprite image to ‘n’ (index in vram file) and show it.

```
sprite n true or sprite n false
```

show/hide sprite n

sprite.x() sprite.y()

Returns current position of sprite, currently the centre of the image, (e.g. the same value used in sprite n to x,y)

```
print sprite.x(4),sprite.y(4)
```

vload

Loads a .VRAM format file (see documents directory) into VRAM, decompressing as required.

VRAM files are built from the “vram” subdirectory of an application using the vramc.zip Python application, which converts graphics described by a simple text file into a convenient format. The text file is shown in the example file in source/vram. The format for VRAM files is in the file ‘vram.txt’ in the documents directory.

```
vload “demo.vram”
```

vpeek() vdeek()

These work like peek and deek but they read either a byte or word from VERAs internal RAM memory. Note that unlike X16 Basic, the address is a 17 bit address, the same as specified in the documentation.

```
print vpeek(&18703)
```

```
print vdeek(2)
```

vpoke vdoke

These work like poke and doke but they write either a byte or word to VERAs internal RAM memory. Note that unlike X16 Basic, the address is a 17 bit address, the same as specified in the documentation.

```
vpoke &18703,42
vdoke 2,&f022
```