# BASIC Reference

At  present this reference is in significant flux.

## # and $

# and $ are used to type variables. # is a floating point value, $ is a string. The default type is integer. Variables are not stored internally by name but by reference. This means they are quick to access but means they are always in existence from the start of a program if used in it. Integers are 32 bit ; Floats have a 31 bit mantissa and byte exponent.

So variables and arrays are as follows:

```
an_integer  = 42
a_float#  = 3.14159
a_string$ = “hello world”
```

## !

! is an indirection operator that does a similar job to DEEK and DOKE, 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 statementm when it functions as a DOKE, writing a 16 bit value in low/high order.

It reads or writes a 16 bit address in the memory map.

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

## ?

! is an indirection operator that does a similar job to PEEK and POKE, e.g. accesses memory. It is the same as ? except it operates on a byte level.

```
?a = 42  print ?a   print a?b   a?b=12
```

## $

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. Remarks should be in quotes for syntactic consistency.

```
' "my program"     rem     rem "hello world"
```

## *

Multiply

```
4*2
```

## +

Add or string concatenation.

```
4+2     "hello "+"world !"
```

## -

Subtract

```
44 – 2
```

## .

Sets the following label to the current assembler address. So the example below sets the label 'mylabel' at the current address and you can write things like bra mylabel. Note also that this is an integer variable.

```
.mylabel
```

## / and \

Signed division. An error occurs if the divisor is zero. Backslash is integer division, forwar slash returns a floating point value.

```
22/7
```

## <    <=    <>    =    >    >=

Comparison binary operators, which return 0 for false and -1 for true. They can be used to either compare two numbers 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)
```

## &

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
```

## ^

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 ^ $0e
```

## |

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 | 4
```

## << >>

Binary operators which shift an integer left or right a certain number of times logically. Much quicker than multiplication.

```
a << 2  32 >> 2
```

## abs()

Returns the absolute value of the parameter

```
abs(-4)
```

## alloc()

Allocate the given number of bytes of memory and return the address. Can be used for data structures or program memory for the assembler.

```
alloc(32)
```

## asc()

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

```
asc("*")
```

## assemble

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. 6502 mnemonics are typed as is. Two passes will normally be required by wrapping it in a for/next loop

```
assemble $6000,1:lda #42:sta count:rts
```

Normally these are wrapped in a loop for the two passes for forward references.

```
for pass = 0 to 1
assemble $6000,pass *2
bra forward
<some code>
.forward:rts
next
```

This is almost identical to the BBC Microcomputer’s inline assembler.

## 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 myage = 42
```

## bitmap

Turns the bitmap on or off, or clears it. Only one bitmap is used, and it is located at $10000 in F256 Memory Space. Can be postfixed with ON OFF or CLEAR <colour>

```
bitmap on:bitmap clear $1c
```

## char

Draws a possibly scaled or flipped string from the standard font on the bitmap, using the standard syntax. Flipping is done using bits 7 and 6 of the mode (e.g. $80 and $40) in the colour option,

```
string “hello” dim 2 colour $03 to 100,100
```

## chr$()

Convert an ASCII integer to a single character string.

```
chr$(42)
```

## circle

Draws a circle, using the standard syntax. The vertical height defines the radius of the circle.

```
circle here solid to 200,200
```

## clear

Clears all variables to zero or empty string, and erases all arrays.

```
clear
```

## deek() & peek()

Deek and Peek read two or one bytes respectively from the 6502 memory

```
print deek(42),peek(1)
```

## doke & poke

Doke and Poke write two or one bytes respectively to 6502 memory

```
poke 4096,1: doke $c004,$a705
```

## dim

Dimension number or string arrays with up to two dimensions, with a maximum of 254 elements in each dimension.

```
dim a$(10),a_sine#(10)   dim name$(10,2)
```

## 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(evt1,70) this will return true every second – the clock operates at the timer rate, 70Hz.

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.

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 move()
```

## false

Returns the constant zero.

```
false
```

## for to/downto next

Loop which repeats code a fixed number of times, which must be executed at least once. The step is 1 for to and -1 for downto. The final letter on next is removed.

```
for i = 1 to 10:print i:next i
for i = 10 downto 1:print i:next
```

## frac()

Return the fractional part of a number

```
print frac(3.14159)
```

## gfx

Sends three parameter command directly to the graphics subsystem. Often the last two parameters are coordinates (not always). It is not advised to use this for general use as programs would be somewhat unreadable.

```
gfx 22,130,100
```

## 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.

```
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()

Tests if two sprites overlap. This is done using a box test based on the size of the sprite (e.g. 8x8,16x16,24x24,32x32)

The value returned is zero for no collision, or the lower of the two coordinate differences from the centre, approximately.

This only works if sprites are positioned via the graphics system ; there is no way of reading Sprite memory to ascertain where the physical sprites are.

```
print hit(1,2)
```

## 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. The instruction does not have to all be on the same line.

```
if age < 18:print "child":else:print "adult":endif
```

## image

Draws a possibly scaled or flipped sprite image on the bitmap, using the standard syntax. Flipping is done using bits 7 and 6 of the mode (e.g. $80 and $40) in the colour option. This requires both sprites and bitmap to be on.

```
image 4 dim 3 colour 0,$80 to 100,100
```

## input

Input uses the same syntax as print, except that where there is a variable a value is entered into that variable, rather than the variable being printed.

```
input a$
```

## int()

Returns the integer part of a number

```
print int(3.14159)
```

## isval()

This is a support for val and takes the same parameter, a string  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  and returns -1 if so, 0 if it is not.

```
isval("42")          isval("i like chips in gravy")
```

## joyb()

Returns a value indicating the status of the fire buttons on a gamepad, with the main fire button being bit 0. Takes a single parameter, the number of the gamepad.

```
if joyb(0) & 1 then fire()
```

## joyx() joyy()

Returns the directional value of a gamepad in the x and y axes respectively as -1,0 or 1, with 1 being right and down. Each takes a single parameter which is the number of the pad.

```
x = x + joyx(0)
```

## left$()

Returns several characters from a string counting from the left

```
left$(a$,4)
```

## len()

Returns the length of the string as an integer

```
len(“hello, world”)
```

## let

Assignment statement. The LET is optional. You can also use @a where a is a reference ; so ptr = @a ; @ptr = 42 is the same in practice as a = 42.

```
let a = 42    a$=”hello” a#=22.7
```

## line

Draws a line, using the standard syntax

```
line 100,100 colour $e0 to 200,200
```

## list

Lists the program. It is possible to list any part of the program using the line numbers, or list a procedure by name.

```
list      list 1000    list 100,200 list ,400
list myfunction()
```

## 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.

```
local test$,count
```

## max() min()

Returns the largest or smallest of the parameters, there can be any number of these (at least one). You can’t mix strings and integers.

```
print max(3,42,5)
```

## 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".

```
mid$("hello",2,3)   mid$("another word",2,99)
```

## %

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

```
42 % 5
```

## 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)
```

## palette

Sets the graphics palette. The parameters are the colour id and the red, green and blue graphics component. On start up, the palette is rrrgggbb

```
palette 1,255,128,0
```

## playing

Returns true if a channel is currently playing a sound.

```
print playing(0)
```

## plot

Plot a point in the current colour using the standard syntax

```
plot to 100,200
```

## print

Prints to the current output device, either strings or numbers (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 , .

```
print 42,"hello""world"
```

## 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).

```
printmessage("hello",42)

....

proc printmessage(msg$,n)

        print msg$+"world  x "+str$(n)

endproc
```

## rnd()     random()

Generates random numbers. This has two forms, which is still many fewer than Odo. Rnd() behaves like Microsoft Basic, Negative numbers set the seed, 0 repeats the last value, and positive numbers return an integer 0 <= n < 1. Random(n) returns a number from 0 to n-1

```
rnd(1)   random(6)
```

## read / data

Reads from DATA statements the types must match. For syntactic consistency, string data must be in quote marks

```
read a$,b  data "hello world" data 59
```

## rect

Draws a rectangle, using the standard syntax

```
rect 100,100 colour $ff to 200,200
```

## restore

Resets the data pointer to the start of the program

```
restore
```

## repeat     until

Conditional loop, which is tested at the bottom.

```
repeat

        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.

```
run
```

## sgn()

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

```
sgn(42)
```

## sound

Generates a sound on one of the channels. There are four channels, corresponding to the. Channel 3 is a noise channel, channels 0-2 are simple square wave channels generating one note each.

Sound has two forms

```
sound 1,500,10
```

generates a sound of pitch 1000 which runs for about 10 ticks (one tick is about 0.5s). The actual frequency is 111,563 / <pitch value>. The pitch value can be from 1 to 1023

Sounds can be queued up , so you can play 3 notes in a row e.g.

```
sound 1,1000,10:sound 1,500,10:sound 1,250,10
```

An adjuster value can be added which adds a constant to the pitch every tick, which allows the creation of some simple warpy effects.

```
sound 1,500,10,10
```

Creates a tone which drops as it plays (higher pitch values are lower frequency values)

Channel 3 operates slightly differently. It generates noises which can be modulated by channel 2- see the SN76489 data sheet.

However, there are currently 8 sounds, which are accessed by the pitch being 16 times the sound number.

```
sound 3,6*16,10
```

Is an explosiony sort of sound. You can just use the constant 96 of course instead.

Finally this turns off all sound and empties the queues.

```
sound off
```

## spc()

Return a string consisting of a given number of spaces

```
a$ = spc(32)
```

## sprite

Manipulate a hardware sprite using the standard modifiers. Also supported are sprite image <n> which turns a sprite on and selects image <n> to be used for it, and sprite off, which turns a sprite off. Sprite data is stored at $30000 onwards, beginning with a 512 byte index. Sprites cannot be scaled and flipped as the hardware does not permit it.

```
sprite 4 image 2 to 50,200
```

## sprites

Enables and Disables all sprites. When turned on, all the sprite records are cleared to zero.

```
sprites on
```

## stop

Stops program with an error

```
stop
```

## timer()

Returns the current value of the 70Hz Frame timer, which will wrap round in a couple of days.

```
print timer()
```

## 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.  To make it useable use the function isval() which checks to see if it is valid.

```
val(“42”)    val(“413.22”)
```

## str$()

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

```
str$(42)  str$(412.16)
```

## true

Returns the constant -1

```
true
```

## while   wend

Conditional loop with test at the top

```
while wife_very_cross
        buy_flowers()
        grovel()
wend
```

## Graphics Modifiers and Actions

Following drawing commands PLOT, LINE, RECT, CIRCLE, SPRITE, CHAR and IMAGE there are actions and modifiers which either change or cause the command to be done (e.g. draw the line, draw the string etc.). Changes persist, so if you set COLOUR 3 or SOLID it will apply to all subsequent draws until you change it.

Not all things work or make sense for all commands ; you can't change the dimensions of a line, or the colour of a hardware sprite.

These are as follows

| **Keyword** | **Purpose** |
|---|---|
| to 100,100 | Draws the object from the current point to the new point, or at the new point. |
| [from] 10,10 | Sets the current point, but doesn't draw. So you have RECT 10,10 TO 100,100. Note that PLOT requires TO to do something, which is a little odd but consistent. The FROM is optional, but must be used where a number precedes the coordinates (e.g. you can't do COLOUR 5 100,200 |
| here | Same as TO but done at the current point |
| by 4,5 | Same as TO but offset from the current point by 4 horizontal, 5 vertical |
| solid | Causes shapes to be filled in |
| outline | Causes shapes to be drawn in outline |
| dim 3 | Sets the size of scalable objects (CHAR, IMAGE) from 1 to 8. |
| colour 4<br>color 5 | Synonyms due to American mis-spelling, sets the current drawing colour from LUT 0, which is set up as RRRGGGBB. |

The range of values for draw commands is 0–319 and normally 0–239, though there is a VGA mode which is 320x200 (in which case it would be 0-199)

Sprite positions are relative to the centre of the sprite, e.g. sprite 0 to 0,0 will put the sprite centred around the top left corner of the screen.