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.

So variables and arrays are as follows:

```
an_integer a_float# a_string$
```

!

! 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 statement. 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. 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. Note the use of quotes for syntactic consistency.

```
‘ “my program”    rem   rem “hello world”
```

*

Binary multiply

```
4*2
```

+

Add or string concatenation.

```
4+2     “hello “+”world !”
```

-

Subtract

```
44 – 2
```

.

Sets the following label to the current assembler address

```
.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
```

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
```

chr$()

Convert an ASCII integer to a single character string.

```
chr$(42)
```

clear

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

```
clear
```

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() not yet coded

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 move()
```

false

Returns the constant zero.

```
false
```

for to/downto step 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 step 2:print i:next i
```

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”
```

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
```

input

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

```
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")
```

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
```

list

Lists the program.

```
list      list 1000    list 100,200 list ,400
```

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
```

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)
```

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
```

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)
```

spc()

Return a string consisting of a given number of spaces

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

stop

Stops program with an error

```
stop
```

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
```