Basic Reference

Last update 6th March 2021. Reference for version (development)

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 MSBasic, 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.

! ?

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, similar to 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(“*”)
```

assert <expression>

Asserts a contract, causing an error if it fails.

```
assert ptr != 0
```

chr$(code)

Converts a number to an 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 seperately from all others.

```
clear
```

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 printdouble(n)
```

```
    print n * 2
endproc

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

dim

Create an array of a given dimension. Currently this only allows a single dimension, but this will be expanded to at least two if not more.

```
dim a$(5),count(22)
```

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()

Waits for a character to be pressed (depending on the system, there may be one already in the keyboard buffer) and returns it as an 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 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
```

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, otherwise it returns 0. This is not suitable for game writing purposes really.

```
print inkey()
```

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
```

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
```

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”)
```

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
```

repeat ... until

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

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

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.

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
```

X16 Specific Extensions

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
```