F256 SuperBASIC Reference Manual

Paul Robson

22-10-2023

Contents

1 Introduction

1.1 Introduction

This document is not a tutorial and assumes the reader has some knowledge of programming in general and BASIC in particular.

If you want to learn programming, I would advise you try learning using one of the numerous courses available. The course in the Vic20 manual is highly regarded and can be used with an emulator such as "Vice", and much of the knowledge will transfer directly.

F256's SuperBASIC is a modernised BASIC interpreter for the 65C02 processor. It currently occupies 4 pages (32k) of Flash mapped into 2 pages (16k) of the 65C02 memory space. At present, this is from 8000–BFFF.

Currently editing is still done using line numbers, however it is possible to cross-develop without line numbers. There are some examples of such at the primary github at http://github.com/paulscottrobson/SuperBASIC under the 'games' directory.

However, there is no requirement to actually use them in programs. GOTO, GOSUB and RETURN are supported but this is more for backwards compatibility with older programs. It is advised to use Procedures, For/Next, While, Repeat and If/Else/Endif commands for better programming and clarity.

1.2 Storage

Programs are stored in ASCII format, so in cross development any editor can be used. LOAD, VERIFY and SAVE read and write files in this format. Internally the format is quite different.

1.3 Memory usage

Memory usage is split into 2 parts.

The main space from $2000-$7FFF contains the tokenised BASIC code. Keywords such as REPEAT are replaced by a single byte, or for less common options, by two bytes.

Identifiers are replaced by a reference into the identifier table from $1000-$1FFF. The first part of this table is a list of identifiers along with the current value.

Arrays and allocated memory (using alloc() follow that).

Finally string memory occupies the top of this memory area and works down.

This should be entirely transparent to the developer.

1.4 Memory usage elsewhere

SuperBASIC uses memory locations outside the normal 6502 address space as well.

If you use a bitmap, it will be placed at $10000 in physical space and occupy 320x240 bytes.

If you use sprites, they are loaded to $30000 in physical space, and the size depends on how many you have.

If you use tiles, the tile map is stored at $24000 and the tile image data at $26000.

Finally, if you cross develop, the memory location from $28000 onwards is used to store the BASIC code you have uploaded.

If you do not use any of these features directly (you can set up your own bitmaps and sprites yourself, and enter programs through the keyboard, saving to the SD Card or IEC drive) then the memory is all yours.

2 Writing Programs in SuperBASIC

2.1 Writing programs

Programs in SuperBASIC are written in the 'classic' style, using line numbers. A line number on its own deletes a line.

LIST operates as in most other systems, except there is the option to list <procedure>() which lists the given procedure by name. (LIST also uses commas, not - as some BASICs do e.g. list 100,300)

It is easy to cross develop in SuperBASIC (see later), writing a program on your favourite text editor, and squirting it down the USB cable using the Python script fnxmgr , or the Foenix IDE. It is also possible to develop without line numbers and have them added as the last stage before uploading.

Upper and Lower case is considered to be the same, so variable myName, MYNAME and MyName are all the same variable. The only place where case is specifically differentiated is in string constants.

Programs can be loaded or saved to SD Card or to an IEC type drive (the 5 pin DIN serial port) using the SAVE and LOAD command.

There is also currently a VERIFY command whose purpose it is to check files have been saved correctly. While the SD Card and IEC code has been seen to be reliable in practice, the code is still relatively new ; so when saving BASIC programs in development, I recommend saving them under incremental names (e.g. prog1.bas, prog2.bas) , verifying them, and periodically backing up your SD card.

This may seem slightly long winded, but is a good defensive measure as there may be bugs in the kernel routines, or the BASIC routines which handle program editing.

The documents directory in the SuperBASIC github, which is publicly accessible, has a simple syntax highlighter for the Sublime Text editor.

3 Identifiers, Variables and typing

3.1 Procedure and variable naming

SuperBASIC allows full naming of both variables and procedures (a type of 'subroutine').
A name begins with an alphabetic character or underscore, and continues with either alphanumeric character or underscores. They may also have a type character ($ (or #) e.g.

```
hello_world
count09
my_name_is_earl
name$
inventory(
mean_value#
```

This allows you to write readable code ; no more trying to remember what C7 is

An implementation note: when the program is stored in memory, it only keeps one copy of the actual identifier 'text' name, irrespective of how many times you use it. So there apart from that once, there is no space saving from using long identifier names.

3.2 Types

Variable in SuperBASIC are one of three types - integers, floating point and strings.

By default a variable holds an integer, which is a number between about -2 billion and +2 billion (technically it's a 32 bit integer).

A variable name that ends with a # holds a floating point number, or a decimal, which has a much larger range, but is less accurate, and slower.

A variable name that ends with a $ holds a string , which is a sequence of characters up to 253 characters in length.

These are some examples of variables being assigned those values.

```
100 count = 42
110 height# = 162.7
120 name$ = "Jack Hobbs"
```

3.3 Arrays

Arrays are a collection of variables, stored under one name. SuperBASIC supports up to two dimensions of arrays, with a maximum of 254 elements in each dimension. Arrays are indexed from zero.

```
100 dim a$(3)
110 a$(1) = "Entry 1"
120 a$(3) = "Entry 3"
130 dim grid(8,8)
140 grid(4,3) = 42
```

4 Structured Programming

SuperBASIC is designed for better and more readable programs. If you learnt BASIC on another machines, you will mostly have used GOTO, GOSUB and RETURN.

These are terrible.

SuperBASIC does support these, but it is strongly advised you do not use them. However, they can be useful for running old code. SuperBASIC is not Microsoft BASIC compatible, but is close enough so that code will normally work with minor alterations.

4.1 Named procedures

The language supports named procedures, which are full identifiers.
sh
This means that instead of writing gosub 300, you can have a procedure called addscore or moveinvaders or whatever you like ; they can be meaningful and this enhances program readability. Whereas with "gosub 100" you have no idea what it involves.

```
100 printmessage(“hello”,42)
110 end
120 proc printmessage(msg$,n)
130  print msg$+“world  x “+str$(n)
140 endproc
```

This is a simple piece of code showing a procedure call printmessage which prints a silly message. As you can see it has two parameters, the message (in msg$) and a number (in n).

These are considered "local" to the procedure, so if you have either of them "outside" the procedure the values are not affected. If we modify the above example slightly -

```
90  n = 12
100 printmessage(“hello”,42)
105 print n
110 end
120 proc printmessage(msg$,n)
130  print msg$+“world  x “+str$(n)
140 endproc
```

This will print the message ("Hello world x 42") and after it, will print the value of n, which will still be 12. n will only be 42 inside printmessage.

If you have no parameters then the brackets must still be used e.g.

```
100 endgame()
110 end
120 proc endgame()
130  print "You lose !"
140 endproc
```

4.2 While and Repeat

While and Repeat are a structured way of doing something repeatedly, until a 'test' becomes either true or false. So at a simple level you could have

```
100 lives = 3
110 while lives > 0
120  playgame()
130 wend
```

The indentation, which is shown when the program is listed, makes it easy to see which bit is repeated. You could do this with a repeat loop e.g.

```
100 lives = 3
110 repeat
120  playgame()
130 until lives = 0
```

The difference between while and repeat loops is that the repeat loop is always done once - if the initial test on while fails, the repeated part will never be done at all.

4.3 For loops

For loops are another way of repeating code, and are found in most BASICs. This time, you know how many times the code is to be repeated. So if you want to print "Hello world" ten times, you do it like this.

```
100 for i = 1 to 10
110  print "Hello world"
120 next
```

Each time you go round the loop, i has a different value, so this code.

```
100 for i = 1 to 10
110  print i
120 next
```

prints the numbers 1 to 10. It is also possible to count backwards using downto

```
100 for i = 10 downto 1
110  print i
120 next
```

There are two variations from common BASIC. Firstly there is currently no 'STEP' ; you can only count either up by 1 , or down by 1. Secondly , some basics require the index in next (e.g. next i) and have peculiar behaviours if you change the order, which are not supported, nor should they be.

4.4 If ... Else ... Endif

If is a conditional test, allowing code to be run if some test is satisfied
e.g.

```
100 if count = 0 then explode
110 if name$ = "Paul Robson" then print "You are very clever and modest."
```

(there is an instruction 'explode')

This is standard BASIC - if the test 'passes' the code following the then is executed.
However, there is an alternate, which is more in tune with modern programming, which is if ... else ... endif

```
100 for n = 1 to 10
110  if n % 2 = 0
120     print n;" is even"
130  else
140     print n;" is odd"
150  endif
160 next
```

will print a number is even or odd, depending on the value of n. You can have many lines of code in either the 'if part' or the 'else part'.

The else part is not mandatory.

This can all be written on one line (or pretty much any way you like) e.g.

```
100 for n = 1 to 10
110  if n % 2 = 0:print n;" is even":else:print n;" is odd":endif
160 next
```

You cannot write, as you can in some BASIC interpreters, the following.

```
100 if a = 2 then print "A is two" else print "A is not two"
```

Once you have a 'then' you are locked into the simple examples above ; no else or endif.

Generally when programming you use the 'then' short version for simple tests, and the if..else..endif for more complicated ones.

Here endeth the lesson.

5 Graphics

5.1 Introduction

The graphics subsystem consists of three components, which is a subset of the full capabilities of the F256 machines.

Firstly there is an 8x8 pixel tile grid of up to 256x256 tiles, which can be scrolled about.

Secondly, on top of that is a 320x240 bitmap screen, which can have anything drawn on it.

Thirdly, on top of that, are the sprites.

The graphics are much more complex than this ; the system allows up to three tile maps for example. Those can be done in BASIC if you wish, by directly accessing the system registers, as covered in the hardware reference guide.

5.2 Bitmap Graphics

Bitmap Graphics can be done in one of three ways.

- Firstly, they can be done using BASIC commands like LINE, PLOT and TEXT. These are the easiest.
- Secondly, they can be done by directly accessing the graphics library via the GFX command.
- Thirdly, you can "hit the hardware" directly using POKE and DOKE or the indirection operators.

The latter is the most flexible. BASIC simplifies the graphics system to some extent to make it easier to use ; for example, the Junior can have up to three bitmaps, but only one is supported using BASIC commands.

5.3 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.

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 color 5	Synonyms due to American misspelling, sets the current drawing colour from LUT 0, which is set up as RRRGGGBB.

5.4 Some useful examples

All these examples begin "bitmap on:cls:bitmap clear 3" ; display and clear the bitmap, then fill it with colour 3 - this is 0000 0011 - and the colour by default is RRRG GGBB in binary - so this is Blue

Example: Some lines

```
100 bitmap on:cls:bitmap clear 3
110 line colour $1E from 10,10 to 100,200 to 200,50 to 10,10 by 0,20
```

Note how you can chain commands and also the use of the relevant position "by" which means from here.

Example: Some circles

```
100 bitmap on:cls:bitmap clear 3
110 circle solid colour $1E outline 10,10 to 200,200 solid 10,10 to 30,30
```

Rectangles are the same. Currently we cannot draw ellipses.

Example: Some text

```
100 bitmap on:cls:bitmap clear 3
110 text "Hello there" dim 1 colour $FC to 10,10 dim 3 to 10,40
```

Drawing text from the font library in Vicky. These characters can be redefined.

Example: Some pixels

```
100 bitmap on:cls:bitmap clear 3
110 repeat
120 plot color random(256) to random(320),random(230)
130 until false
```

Press break to stop this one. Note you have to write "Plot To" here.

5.5 Ranges

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).

Colours are values from 0-255 - initially this can be viewed as a binary number RRRG GGBB

6 Tiles and Tile Maps

6.1 Introduction

SuperBASIC supports a single tile map, made up of 8x8 pixel images. A tile map can be up to 255 x 255 tiles in size.

6.2 Setting up a Tile Map

The TILES command sets up a tile map. For example, the following command sets up a 48x48 tile map at the default locations (see below), and turns it on.

```
100 tiles dim 48,48 on
```

6.3 Manipulating a Tile Map

The TILE command is used to manipulate a tile map, by either scrolling its position, or by writing to it. These commands can be chained in a similar manner to the graphics drawing ones (so these two commands could be joined into one).

This example sets the draw pointer at tile 4 across, 5 down and draws the following tiles, writing horizontally, of tile 10, 3 tile 11s, and another tile 10. So it is not difficult to create maps programmatically.

Tile TO scrolls the tile map on the screen, this is in pixels not whole tiles.

The TILE() function reads the tile at the current map position(which following the code at line 100, should be 11).

```
100 tile at 4,5 draw 10,11 line 3,10
110 tile to 14,12
120 t = tile(5,5)
```

6.4 Data formats

There are two data files required for a tile map. One is the images file, which is a sequence of 64 bytes, representing an 8x8 tile. These are indexed from zero. This images file is held at $26000 by default (though it can be placed anywhere).

The second is the map file itself. This could be created in some sort of editor, and loaded manually, or could be generated randomly. This is a word sequence, which is (map width x map height x 2) bytes in size, as specified in the Hardware reference manual. This map is held at $24000 by default (it too can be placed anywhere).

7 Sprites

7.1 Introduction

Sprites are graphic images of differing sizes (8x8, 16x16, 24x24 or 32x32) which can appear on top of a bitmap but which don't affect that bitmap. It is a bit like a cartoon where you have a background and animated characters are placed on it.

The sprite command adopts the same syntax as the graphics commands in the previous section. An example is:

```
sprite 3 image 5 to 20,20
```

This manipulates sprite number 3 (there are 64, numbered 0..63), using image number 5, centred on screen position 20,20. The image number comes from the data set - in the example set below it would be enemy.png rotated by 90 - the sixth entry as we count from 0. You can change the location and image independently.

Most of the theoretical graphics options do not work ; you cannot colour, scale, flip etc. a sprite, the graphic is what it is. However, they are very fast compared with drawing an image on the screen using the IMAGE command.

7.2 Creating sprites

SuperBASIC by default loads sprite data to memory location $30000. This chapter explains how to create that data file. This can be done by the developer, or via a python script.

7.3 Getting images

Sprite data is built from PNG images up to 32x32. There are some examples in the Solarfox directory in the github https://github.com/paulscottrobson/superbasic.

They can be created individually, or ripped from sprite sheets - this is what ripgfx.py is doing in the Makefile in solarfox/graphics ; starting with the PNG file source .png it is informed where graphics are, and it tries to work out a bounding box for that graphic, and exports it to the various files.

7.4 Building a sprite data set

Sprite data sets are built using the spritebuild.py python script (in the utilities subdirectory). Again there is an example of this in the Solarfox directory.

Sprite set building is done using the spritebuild.py script which takes a file of sprite definitions. This is a simple text list of files, which can be either png files as is, or postfixed by a rotate angle (only 0,90,180 and 270) or v or h for vertical and horizontal mirroring.

```
graphics/ship.png
graphics/ship.png 90
graphics/ship.png 180
graphics/ship.png 270
graphics/enemy.png
graphics/enemy.png 90
graphics/enemy.png 180
graphics/enemy.png 270
graphics/collect1.png
graphics/collect2.png
graphics/life.png h
```

Sprite images are numbered in the order they are in the file from zero and should be loaded at $30000

When building the sprite it strips it as much as possible and centres it in the smallest sprite size it fits in. When using BASIC commands to position a sprite, that position is relative to the centre of the sprite.

7.5 Data format

At present there is a very simple data format.

```
+00 is the format code ($11)
+01 is the sprite size  (0-3, representing 8,16,24 and 32 pixel size)
+02 the LUT to use (normally zero)
+03 the first byte of sprite data
```

The size, LUT and data are then repeated for every sprite in the sprite set. The file should end with a sprite size of $80 (128) to indicate the end of the set.

8 Sound

8.1 Introduction

The F256Jr has 2 independent SN76489 sound chips, for left and right channel, and optionally 2 further SID chips or SID chip clones.

In SuperBASIC these are simplified, so the same tones are played on left or right channels simultaneously.

8.2 Channels

There are four sound channels, numbered 0 to 3. 0 to 2 are simple square wave channels, the 3rd is a sound channel.

Sounds have a queue of sounds to play. So you could queue up a series of notes to play and they will carry on playing one after the other (if they are on the same channel).

SuperBASIC does not stop to play the sounds ; it is processed in the background. Everything can be silenced with 'sound off'.

It is possible to set a parameter to automatically change the pitch of the channel as it plays to allow easy creation of simple warpy sound effects.

8.3 Easy commands

Four commands ZAP, SHOOT, PING and EXPLODE exist which play a simple sound effect.

9 Inline Assembly

SuperBASIC has a built in inline assembler, that is closely modelled on that of BASIC in the British Acorn machines (Atom, BBC Micro, Archimedes).

9.1 How it works

The way it works is that the instructions, named after their 65C02 opcode equivalents, generate code - so as a simple example the instruction "txa" generates the machine code $8A in memory. If the instruction has an operand (say) "lda #size*2" the expression is evaluated and the appropriate 2 bytes are stored in memory.

Labels are specified using .<label name>, e.g. .loop ; this is equivalent to setting the variable 'loop' to the current write address, which can then be used in expressions - such as 'jmp loop'.

9.2 The Assemble command

Assembly is controlled by the 'assemble' command. This has two parameters - the first indicates where the code is to be assembled in memory, and the second is a control byte.

This has 2 bits. Bit 0 indicates the pass, and if zero will not flag errors such as branches being out of range. Bit 1 , when set, causes the code generated by instructions like "txa".

```
100 assemble $6000,2:lda #42:sta count:rts
```

This very short example will assemble the 5 (or 4 depending on the value of count) bytes starting from $6000 - and it also outputs those bytes to the screen. Use the call statement to execute the assembled source code.

9.3 Two pass Assembly

Assemblers often require multiple passes through the source, and this one is no exception. As the inline assembler is not an assembler per se, this is done in code, by running through the assembler code with different values in the second parameter of the assemble command.

Normally these are wrapped in a loop for the two passes. This illustrates the need - the first time through the assembler doesn't know where 'forward' is - the second time round it has been set by line 140, so the branch can be correctly calculated.

```
100 for pass = 0 to 1
110 assemble $6000,pass
120 bra forward
130 jsr $FFE2
140 .forward:rts
150 next
```

Note that unlike the Acorn machines there are no square brackets to delimit the assembler code - assembler commands can be put in at any time.

9.4 Limitations

There is a minor syntactic limitation, in that instructions that target the accumulator (inc, dec, lsr, ror, asl and rol) must not use the 'A' postfix - so it is "inc" rather than "inc a".

More generally, this assembler should be used for writing supporting code for BASIC programs - to speed up a part that is too slow in an interpreted language. It could be used for writing much larger programs (I believe "Elite" was written with this sort of system), but you would be better of using an assembler like 64tass or acme, and loading the generated code into memory with the BLOAD command.

10 Cross Development of BASIC Programs

Cross development is an alternative to the classic way of programming a Home Computer, where the programmer types code directly into the machine. Cross development allows you to write the code on a Personal Computer, and upload it through the USB debug port in the F256. It is also possible to do this with machine code and graphic and other data.

10.1 Assistance

In the SuperBASIC git, https://github.com/paulscottrobson/superbasic, each release contains a file "howto-crossdev-basic.zip" which gives everything you need to cross develop in BASIC and some example programs.

10.2 Connection

To connect your F256Junior to a PC (Windows, Linux, Mac) you need a standard USB cable with a Micro USB plug. This needs to be a data cable, some cables only provide power. The Micro USB plug plugs into the board, and the USB plug into the PC.

10.3 Software

There are two ways of programming the board. I prefer FnxMgr *https://github.com/pweingar/FoenixMgr* which is a Python script which runs on all platform, and can easily automate uploading. It can also be uploaded through the Foenix IDE in Windows.

Besides Python version 3, the FnxMgr script requires pyserial.

10.4 BASIC

The input to the program is standard ASCII files, with line numbers. Line numbers are required for editing only. (There is a python script on the SuperBASIC github which adds these automatically). However, you do not need to use line numbers in programming, though GOTO and GOSUB are implemented if you wish, or want to port old software.

I would start with something simple though.

Example: Print to the screen and make silly sound effect

```
10 print "Hello, world !"
20 zap
```

Each file should end in a character with an ASCII code greater than 127, which marks the end of the file. You can copy one from the software in github.

10.5 Uploading and running

This is written for people with 'B' boards which automatically start up into BASIC. If you are booting from RAM, or have an A board, it will be slightly different.

Uploading works by loading the ASCII text into memory. It is then effectively 'typed in' by either the **xload** command or the **xgo** command. The first loads the program in (and it can then be listed or edited or run in the normal way. The second loads and runs it.

To load the program into memory to be "loaded" you need something like the below. The first one works on my Arch Linux Box. The second is simply a guess ; I do not know what the COM ports are for each system. You should be able to discover this with the Device Manager (Windows) or lsusb (Linux).

Example: Linux Upload

```
python ../bin/fnxmgr.zip --port /dev/ttyUSB0 --binary load.bas  --address 28000
```

Example: Windows Upload (not tried)

```
python ..\bin\fnxmgr.zip --port COM1 --binary load.bas  --address 28000
```

10.6 Memory Use

Initially the lower 32k of RAM (0000-7FFF) has a logical address equal to its physical address. The BASIC ROM is mapped into 8000-BFFF.

The memory block C000-DFFF is reserved by the Kernel - you can change I/O registers, but do not map RAM here and change it unless you are absolutely sure of what you are doing.

The memory block E000-FFFF contains the Kernel.

10.7 Sprites

Sprites are loaded (in BASIC) to $30000 and there is a simple index format. This is covered in the sprites section.

11 Keyword Reference

This describes the keywords in SuperBASIC. Some that are naturally grouped together, such as graphics, have their own section.

!

! 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 statements 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 6502 memory map.

Example:

```
10 !a = 42
20 print !a
30 print a!b
40 a!b=12
```

’ and 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.

Example: Simple comments

```
10 ’ "This is a title comment"
20 REM
30 REM "Another comment"
```

abs()

Returns the absolute value of the parameter

Example:

```
10 print 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.

Example:

```
10 myAssemblerCode = alloc(128)
```

asc()

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

Example:

```
10 print asc(“*”)
```

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:

Example:

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

?

? 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.

Example:

```
100 ?a = 42
110 print ?a
120 print a?b
130 a?b=12
```

$

Hexadecimal constant prefix. $2A is the same as the decimal constant 42.

Example:

```
100 print $2a
110 !$7ffe = 31702
```

*

Multiply

Example:

```
100 print 4*2
```

+

Add or string concatenation.

Example:

```
100 sum = 4+2
110 prompt$ = “hello “+”world !”
```

-

Subtract

Example:

```
100 print 44 - 2
```

%

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

Example:

```
100 print 42 % 5
```

.

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.

Example:

```
100 .mylabel
```

and

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

Example:

```
100 print 22/7
110 print 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.

Example:

```
100 if a<42 then "a is not the answer to life the universe and everything"
110 if name$="" then input name$
```

@

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 ….)

Example:

```
100 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

Example:

```
100 print count & 7
```

<Hat Character>

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

Example:

```
100 print 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

Example:

```
100 print read.value | 4
```

« »

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

Example:

```
100 print a << 2,32 >> 2
```

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

Example:

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

Normally these are wrapped in a loop for the two passes for forward references. The call statement can be used to execute the assembled source code.

Example:

```
100 for pass = 0 to 1
110 assemble $6000,pass
120 bra forward
130 <some code>
140 .forward:rts
150 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.

Example:

```
100 assert myage = 42
```

bitmap

Turns the bitmap on or off, or clears it, or sets its address (the default address is $10000). Only one bitmap is used in BASIC, but you can use others by accessing I/O. Keywords are ON OFF CLEAR <colour> AT <address> and can be chained like the example below, or not. On or Off without an at will reset the address.

Example:

```
100 bitmap at $18000 on clear $03
110 bitmap at $18000 on:bitmap clear $03
```

bload

Loads a file into memory. The 2nd parameter is the address in full memory space, *not* the 6502 CPU address. In the default set up, for the RAM area (0000-7FFF) this will however be the same.

So the example below loads the binary file mypic.bin into the BASIC bitmap, which is stored in MMU page 8 onwards.

Example:

```
100 bitmap on:bitmap clear 1
110 bload "mypic.bin",$10000
```

bsave

Saves a chunk of memory into a file. The 2nd parameter is the address in full memory space, *not* the 6502 CPU address. The 3rd parameter is the number of bytes to save.

In the default set up, for the RAM area (0000-7FFF) this will however be the same.

Example:

```
100 bsave "memory.space",$0800,$7800
```

call

Calls an assembly subroutine which is located at the address specified by the parameter. In order to return to Basic the assembly routine has to end with an rts instruction.

Example:

```
100 call $4500
```

chr$()

Convert an ASCII integer to a single character string.

Example:

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

circle

Draws a circle, using the standard syntax. The vertical height defines the radius of the circle. See the section on graphics for drawing options

Example:

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

cprint

Operates the same as the print command, but control characters (e.g. 00-1F,80-FF) are printed using the characters from the FONT memory, not as control characters. The example below prints a horizontal upper bar character, not a new line.

Example:

```
100 cprint chr$(13);
```

cursor

Turns the flashing cursor on or off

Example:

```
100 cursor on
```

dir

Shows all the files in the current drive.

Example:

```
100 dir
```

dim

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

Example:

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

drive

Sets the current drive for load save. The default drive is zero.

Example:

```
100 drive 1
```

end

Ends the current program and returns to the command line

Example:

```
100 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
An example is better, this prints Hello once a second.

Example:

```
100 repeat
110 if event(myevent1,70) then print "Hello"
120 until false
```

false

Returns the constant zero.

Example:

```
100 print false
```

for next

A 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 not supported.

Example:

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

frac()

Return the fractional part of a number

Example:

```
100 print frac(3.14159)
```

get() and get$()

Wait for the next key press then , return either the character as a string, or as the ASCII character code.

Example:

```
100 print "Letter ";get$()
```

getdate$(n)

Reads the current date from the clock returning a string in the format "dd:mm:yy". The parameter is ignored.

Example:

```
100 print "Today is ";getdate$(0)
```

gettime$(n)

Reads the current time from the clock returning a string in the format "hh:mm:ss". The parameter is ignored.

Example:

```
100 print "It is now ";gettime$(0)
```

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.
This is a direct call to the graphics library. The parameters are described in graphics.txt in the documents directory in the github. Use of this is rare.

Example:

```
100 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.

Example:

```
100 gosub 1000
```

goto

Transfer execution to given line number. See GOSUB ; same result. If it's for typing in old programs, fair enough, but please don't use it for new code.

Example:

```
100 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.

Example:

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

if then and if else endif

If has two forms. The first is classic BASIC, e.g. if <condition> then <do something>. All the code is on one line. The THEN is mandatory - you cannot write if a = 2 goto 420 (say)

Example:

```
100 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.

Example:

```
100 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. For more information see the graphics section.

Example:

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

inkey() and inkey$()

If a character key has been pressed, return either the character as a string, or as the ASCII character code. If no key is available return "" or 0.
This uses key presses, it does not detect if a key has been pressed, merely that it has been in the past. If you want to check whether a key is up or down, use keydown()

Example:

```
100 print inkey(),inkey$()
```

input

Inputs numbers or strings from the keyboard.
Input has always had a somewhat varying syntax historically. This version 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.

Example:

```
100 input "Your name is ?";a$
```

int()

Returns the integer part of a number

Example:

```
100 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.

Example:

```
100 print isval(“42”)
110 print isval(“i like chips in gravy”)
```

itemcount()

Together , itemcount and itemget provide a way of encoding multiple data items in strings. A multiple-element string has a separating character, which can be any ASCII character, often a comma. itemcount() takes a string and a separator and returns the number of items in the string if separated by that separator. The example prints ’2’ as there are two elements separated by a comma, the strings hello and world.

Example:

```
100 print itemcount("hello,world",",")
```

itemget$()

Together , itemcount and itemget provide a way of encoding multiple data items in strings. A multiple-element string has a separating character, which can be any ASCII character, often a comma. itemget$() takes three parameters, the string, the index of the substring required, which starts at ’1’ and the separator. A bad index will generate a range error. The example will print ’lizzie’, this being the third item.

Example:

```
100 print itemget$("paul,jane,lizzie,jack",3,",")
```

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.

The keyboard keys ZX KM L (left/right/up/down/fire) are also mapped onto this controller, so a Game Controller is not required.

Example:

```
100 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.

The keyboard keys ZX KM L (left/right/up/down/fire) are also mapped onto this controller, so a Game Controller is not required.

Example:

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

keydown()

Checks to see if a key is currently pressed or not - the parameter passed is the kernel raw key code. The demo below is also a simple program for identifying those raw key codes.

Example:

```
100 repeat
110 for i = 0 to 255
120 if keydown(i) then print "Key pressed ";i
130 next
140 until false
```

load

Loads a BASIC program from the current drive.

Example:

```
load "game.bas"
```

left$()

Returns several characters from a string counting from the left

Example:

```
100 print left$("mystring",4)
```

len()

Returns the length of the string as an integer

Example:

```
100 print 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.

Example:

```
100 let a = 42
110 a$=”hello”
120 a#=22.7
```

line

Draws a line, using the standard syntax which is explained in the graphics section.

Example:

```
100 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.

Example:

```
100 list
110 list 1000
120 list 100,200
130 list ,400
140 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.

Example:

```
100 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.

Example:

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

mdelta

Gets the current delta status of the PS/2 mouse. 6 reference parameters (normally integer variables) are provided. These provide the cumulative mouse changes in the x,y,z axes, and the number of times the left, middle and right buttons have been pressed.

Example:

```
100 mdelta dx,dy,dz,lmb,mmb,rmb
```

memcopy

This command is an interface to the F256 Junior's DMA hardware. A MEMCOPY command has several formats.
The first in line 100 is a straight linear copy of memory from $10000 to $18000 of length $4000.
The second in line 110 is a linear fill from $10000 , to $4000 bytes on, with the byte value $F7
The third in line 120 is a rectangular area of memory, 64 x 48 pixels or bytes, from $10000. The 320 is the characters per line, normally 320 for the Junior. This copies a 2D area of screen memory rather than a linear one.
The fourth, line 130 is a window, as defined, being filled with the byte pattern $18.
The final shows an alternate way of showing addresses. This makes use of the knowledge that this normally video memory - it doesn't have to be of course - at 32,32 and at 128,128 later, convert to the addresses of those pixels in bitmap memory.

```
100 memcopy $10000,$4000 to $18000
110 memcopy $10000,$4000 poke $F7
120 memcopy $10000 rect 64,48 by 320 to $18000
130 memcopy $10000 rect 64,48 by 320 poke $18
140 memcopy at 32,32 rect 64,48 by 320 to at 128,128
```

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".

Example:

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

mouse

Gets the current status of the PS/2 mouse. 6 reference parameters (normally integer variables) are provided. These provide the current mouse position in the x,y,z axes, and the status of times the left, middle and right buttons.

Example:

```
100 mouse x,y,z,isx,isy,isz
```

new

Erases the current program

Example:

```
100 new
```

not()

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

Example:

```
100 print not(42)
```

option

Option is used for general control functions which are not common enough to justify their own keyword.
Option 0-7 set highlighting colours for Comment Foreground, Comment Background, Line Number, Token, Constant, Identifier, Punctuation, Data respectively, the lower 4 bits setting the colour. Setting the upper bit 7 will disable the background change. (The example sets the listing to all white)

Example:

```
100 for i = 0 to 7:option i,128+15:next
```

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

Example:

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

peek() peekw() peekl() peekd()

The peek, peekw, peekl and peekd functions retrieve 1-4 bytes from the 6502 memory space.

Example:

```
100 print peekd(42),peek(1)
```

playing()

Returns true if a channel is currently playing a sound.

Example:

```
100 print playing(0)
```

plot

Plot a point in the current colour using the standard syntax which is described in the graphics section.

Example:

```
100 plot to 100,200
```

poke pokew pokel poked

The poke, pokew, pokel and poked functions write one to four bytes to the 6502 memory space.

Example:

```
100 poke 4096,1: pokew $c004,$a705
```

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 , .

Example:

```
100 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).

Example:

```
100 printmessage(“hello”,42)
110 end
120 proc printmessage(msg$,n)
130 print msg$+“world  x “+str$(n)
140 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

Example:

```
100 print rnd(1),random(6)
```

read / data

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

Example:

```
100 read a$,b
110 data “hello world”
120 data 59
```

rect

Draws a rectangle, using the standard syntax described in the graphics section

Example:

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

restore

Resets the data pointer to the start of the program

Example:

```
100 restore
```

repeat until

Conditional loop, which is tested at the bottom.

Example:

```
100 count = 0
110 repeat
120 count = count + 1:print count
130 until count = 10
```

return

Return from GOSUB call. You can make up your own death threats.

Example:

```
100 return
```

right$()

Returns several characters from a string counting from the right

Example:

```
100 print right$("last four characters",4)
```

run

Runs the current program after clearing variables as for CLEAR. Can also have a parameter which does a LOAD and then RUN

Example:

```
100 run
110 run "demo.bas"
```

save

Saves a BASIC program to the current drive.

Example:

```
save "game.bas"
```

setdate

Sets the RTC to the given date ; the parameters are the day, month and year (00-99).

Example:

```
100 setdate 23,1,3
```

settime

Sets the RTC to the given time ; the parameters are hours, minutes, seconds

Example:

```
100 settime 9,44,25
```

sgn()

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

Example:

```
100 print 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

Example:

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

generates a sound of pitch 1000 which runs for about 10 timer ticks. 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.

Example:

```
100 sound 1,1000,20:sound 1,500,10:sound 1,250,20
```

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

Example:

```
100 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.

Example:

```
100 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

Example:

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

sprite

Manipulate one of the 64 hardware sprites 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. Sprites cannot be scaled and flipped as the hardware does not permit it. Sprites have their own section. For Sprite .. To the sprite is centred on those coordinates.

Example:

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

sprites

Enables and Disables all sprites,optionally setting the location of the sprite data in memory which default to $30000. When turned on, all the sprites are erased and their control values set to zero.

Example:

```
100 sprites at $18000 on
```

stop

Stops program with an error

Example:

```
100 stop
```

text

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,

Example:

```
100 text “hello” dim 2 colour 3 to 100,100
```

tile

Manipulates the tile map. This allows you to set the scroll offset (with TO xscroll,yscroll) and draw on the time map using AT x,y to set the position and PLOT followed by a list of tiles, with a repeat option using LINE to draw on it. (In the examples, lines 110 and 120 do the same thing)

Example:

```
100 tile to 12,0
110 tile at 4,5 draw 10,11,11,11,10
120 tile at 4,5 draw 10,11 line 3,10
```

tile()

Returns the tile at the given tile map position (not screen position)

Example:

```
100 print tile(2,3)
```

tiles

Sets up the tile map. Allows the setting of the size of the tile map with DIM <width>,<height> and the location of the data with AT <map address>,<image address>, all addresses must be at the start of an 8k page. The defaults are 64 x 32 for the tile map and $24000 for the map - an array of words and $26000 for the images - an array of 8x8 byte graphics. Currently only 8x8 tiles are supported.

Example:

```
100 tiles on
110 tiles off
120 tiles dim 42,32 at $24000,$26000 on
```

timer()

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

Example:

```
100 print timer()
```

try

Tries to execute a command, usually involving the Kernel, returning an error code if it fails or 0 if successful. Currently supports BLOAD and BSAVE.

Example:

```
100 try bload "myfile",$10000 to ec
110 print ec
```

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.

Example:

```
100 print val(“42”)
110 print val(“413.22”)
```

str$()

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

Example:

```
100 print str$(42),str$(412.16)
```

true

Returns the constant -1

Example:

```
100 true
```

verify

Compares the current BASIC program to a program stored on the current drive. This command is deprecated at creation as it is a defensive measure against potential bugs in either the kernel, the kernel drivers, or BASIC itself.

Example:

```
verify "game.bas"
```

while wend

Conditional loop with test at the top

Example:

```
100 islow = 0
110 while islow < 10
120 print islow
130 islow = islow + 1
140 wend
```

xload xgo

These commands are for cross development in BASIC. If you store an ASCII BASIC program, terminated with a character code >= 128, then these commands will Load or Load and then run that program.

zap ping shoot and explode

Simple commands that generate a simple sound effect

Example:

```
100 ping:zap:explode
```