Foenix F256jr BASIC Reference Manual

Paul Robson

15-12-2022

Contents

1 Introduction

1.1 Introduction

F256's SuperBASIC is a modernised BASIC interpreter for the 65C02 processor. It currently occupies 3 pages (24k) 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 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.

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.

It is easy to cross develop in SuperBASIC, writing a program on your favourite text editor, and squirting it down the USB cable using a Python program , or the Foenix IDE.

3

Using Procedures

4

Variables, arrays and typing

5 Graphics

5.1 Introduction

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.

5.2 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 mis-spelling, sets the current drawing colour from LUT 0, which is set up as RRRGGGBB.

5.3 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.

5.4 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 Sprites

6.1 Introduction

In SuperBASIC sprite data is loaded, with an index, to memory location $30000. This chapter explains how to create that index.

6.2 Getting images

Sprite data is built from PNG images up to 32x32. There are some examples in the Solarfox directory in the github.

They can be created individually, or ripped from spritesheets - 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.

6.3 Building a sprite data set

Sprite data sets are built using the spritebuild.py python script. 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 vflip and hflip.

```
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.

6.4 Data format

At present there is a very simple data format. The first 512 bytes are 2 256 byte records, the low byte in the first 256 bytes, the high byte in the second 256 bytes.

That byte has the following format:

00aaaaaa aaaallss

aaaaaaaaaa is the offset from the start of the file divided by 64 (e.g. shifted 6 right) ll is the LUT to use (currently only LUT 0 is available) ss is the size of the sprite (0:8x8, 1:16x16 2:24x24 3:32x323)

7 Sound

7.1 Introduction

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

For the purpose of sound these are simplified, so the same tones are played on left or right channels simultaneously.

7.2 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).

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

7.3 Easy commands

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

8 Cross Development of BASIC Programs

At present, Cross Development is the only practical way to develop BASIC on the F256 ; at the time of writing the IEC / SDCard is incomplete ao there are no options to LOAD and SAVE programs.

This should be available soon. However, this will still be available as a method of development, and it has much to commend it.

8.1 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.

8.2 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.

8.3 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

```
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.

8.4 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 **load** command or the **go** 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
```

8.5 Versions

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.

8.6 Sprites

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

9 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
```

XOR

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.

Example:

```
100 for pass = 0 to 1
110 assemble $6000,pass *2
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. Only one bitmap is used, and it is located at $10000 in F256 Memory Space. Can be postfixed with ON OFF or CLEAR with a colour.

Example:

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

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
```

clear

Clears all variables to zero or empty string, and erases all arrays. Done when a program is RUN

Example:

```
100 clear
```

deek() peek()

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

Example:

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

doke poke

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

Example:

```
100 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.

Example:

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

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

Example:

```
100 if event(event_move,10) then move()
```

false

Returns the constant zero.

Example:

```
100 print false
110 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.

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)
```

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.

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 else endif

If has two forms. The first is classic BASIC, e.g. if <condition> then <do something>

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
```

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.

Example:

```
100 input 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’’)
```

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.

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.

Example:

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

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)
```

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)
```

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)
```

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
```

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
```

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.

Example:

```
100 run
```

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 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.

Example:

```
100 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.

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 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. Sprites have their own section.

Example:

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

sprites

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

Example:

```
100 sprites 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
```

timer()

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

Example:

```
100 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.

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
```

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
```

zap ping shoot and explode

Simple commands that generate a simple sound effect

Example:

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