F256 SuperBASIC Reference Manual

Paul Robson

30-08-2025

Contents

Introduction

This manual provides a reference for SuperBASIC, a beginner-friendly programming language for the Foenix F256 series of personal computers. It is not a tutorial and assumes the reader has some prior knowledge of programming in general and of BASIC in particular.

SuperBASIC builds upon the heritage of classic BASIC dialects while introducing a wide range of enhancements that make programming even more fun and accessible. It is tightly integrated with the advanced hardware capabilities of the F256 platform, offering first-class access to graphics, sprites, game controllers, sound, and more. Notable language additions include structured control flow, named subroutines, inline assembly, and an enriched standard library.

We've worked hard to make SuperBASIC a powerful and approachable language for curious minds of all ages. We hope you enjoy using it as much as we enjoyed creating it!

1 Getting started

1.1 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 dashes, as some BASICs do, e.g. `list 100,300`.

It is easy to cross-develop in SuperBASIC (see Chapter 9), writing a program on your favourite text editor, and transfering it to the F256 over USB using the 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 are considered to be the same, so variables `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` commands.

The `documents` directory in the SuperBASIC GitHub repository has a simple syntax highlighter for the Sublime Text editor.

2 Identifiers, Variables and Typing

2.1 Procedure and variable naming

SuperBASIC lets you give long, descriptive names to both variables and procedures (also known as subroutines). Using clear names makes your programs easier to read and understand.

A name must start with a letter or an underscore (`_`), and it can continue with any combination of letters, numbers, or underscores. A name may also end with a type character (`$` or `#`), which shows the kind of data stored in the variable (see the next section for more details).

Here are some examples of valid names:

```
n
count17
number_of_lives
str2num
name$
average#
```

Here are some examples of *invalid* names:

```
2string      ' cannot start with a number
$name        ' $ character must be at the end
total#sum    ' # character must be at the end
```

When a program is stored in memory, each identifier name is stored only once, regardless of how many times it appears in the program. Thus, aside from that single instance, using shorter identifier names provides no additional space savings.

2.2 Types

SuperBASIC supports three variable types: integers, floating-point numbers, and strings.

By default, a variable is an integer. Integers are whole numbers, such as −5, 0, or 4200. In SuperBASIC, they can go up to about 2 billion or down to −2 billion.

A variable name ending with `#` represents a floating-point number (a decimal). Floating-point numbers let you use very large, very small, or fractional values (like 178.2). They are more flexible than integers, but sometimes less exact and a little slower to calculate.

A variable name ending with `$` represents a string. A string is simply arbitrary text, up to 253 characters long. In program code, string values—called *string literals*—are written inside quotation marks. For example, `"Arthur Dent"` is a string literal.

Here are some examples:

```
100 count   = 19
110 height# = 178.2
120 name$   = "Arthur Dent"
```

2.3 Arrays

An array is a collection of related variables that share the same name and are stored together. Arrays are created using the `dim` statement, followed by the array name and the number of elements. Each individual element is accessed by giving its index inside parentheses. Array indexes always begin at zero:

```
100 dim fruits$(3)
110 fruits$(0) = "apple"
120 fruits$(1) = "orange"
130 fruits$(2) = "banana"
140 print fruits$(0)   ' prints "apple"
150 print fruits$(2)   ' prints "banana"
```

SuperBASIC supports both one-dimensional and two-dimensional arrays, with up to 254 elements in each dimension.

When first created, string array elements are empty strings, and number array elements are set to zero.

Here is a two-dimensional array of numbers, which you can think of like a grid with rows and columns:

```
100 dim grid(8,8)     ' 8 by 8 grid of numbers
110 grid(4,3) = 17
120 print grid(4,3)   ' prints 17
130 print grid(7,7)   ' prints 0
```

3 Structured Programming

SuperBASIC is built to help you write programs that are easy to read and easy to change later. If you've used BASIC on another computer, you might be used to steering your program with commands like `goto`, `gosub`, and `return`. These work fine in small programs, but once your code grows, they can make things messy and hard to follow.

SuperBASIC still lets you use those commands if you want, but it also offers tools you'll probably prefer as your programs get bigger. With loops, procedures, and multi-step conditionals, your code can flow more naturally—making it simpler to read, easier to fix, and more fun to work with.

3.1 Named procedures

A *named procedure* is simply a block of code that has a name. Once you've defined a procedure, you can run it—also called *calling* it—anywhere in your program just by using its name. This often makes your code easier to follow (assuming you choose clear, descriptive names), and saves you from writing the same steps over and over again.

A procedure is defined using the `proc` keyword, followed by the procedure's name and a pair of parentheses, followed by the code to be executed when the procedure is called. The definition is closed with the `endproc` keyword:

```
200 proc greet()
210   print "Hello!"
220   print "How are you?"
230 endproc
```

The code inside a procedure—in this case, lines 210–220—is called the *body* of the procedure. This is the part that actually runs when the procedure is called. Note that the body does not execute until you explicitly call the procedure.

In SuperBASIC, procedures must be defined at the end of your program, after the `end` keyword, but you can call a procedure from anywhere in your code—including from within other procedures.

A procedure is called by writing its name followed by parentheses:

```
100 greet()             ' prints "Hello!" and "How are you?"
110 print "Bye-bye!"    ' prints "Bye-bye!"
120 end                 ' end of program
200 proc greet()
210   print "Hello!"
220   print "How are you?"
230 endproc
```

Here, lines 200–230 define the procedure, and line 100 calls it. When the program encounters the `greet()` call in line 100, it jumps to the first line of the procedure's body (line 210, `print "Hello!"`), executes the entire body, and then returns to line 110 to continue with the rest of the program.

Procedures with parameters

Procedures become even more useful when they can accept *parameters*. A parameter is simply a placeholder for a value that we'll pass to the procedure when we call it.

Let's tweak our `greet` procedure to greet someone by name:

```
100 greet("Alice")      ' prints "Hello, Alice!"
110 greet("Bob")        ' prints "Hello, Bob!"
120 print "Bye-bye!"    ' prints "Bye—bye!"
130 end                 ' end of program
200 proc greet(name$)
210   print "Hello, " + name$ + "!"
220 endproc
```

Here, we've added a parameter called `name$`, indicating that the procedure expects to receive a string value that is a person's name.

When the procedure is called on line 100 with the argument `"Alice"`, `name$` is assigned that value, and line 210 prints `Hello, Alice!`. When control returns to line 110 and the procedure is called with `"Bob"`, `name$` becomes `"Bob"`, and line 210 prints `Hello, Bob!`.

Multiple parameters

To define a procedure that takes more than one parameter, simply separate the parameter names with commas, and do the same when providing arguments in the procedure call:

```
100 greet("Alice", "morning")    ' prints "Good morning, Alice!"
110 greet("Charlie", "evening")  ' prints "Good evening, Charlie!"
120 print "Bye-bye!"             ' prints "Bye—bye!"
130 end                          ' end of program
200 proc greet(name$, time_of_day$)
210   print "Good "; time_of_day$; ", "; name$; "!"
220 endproc
```

A procedure can accept up to 13 parameters.

SUMMARY

- A named procedure is a snippet of code that has a name.
- When you call a procedure, you give it values (called arguments), which get assigned to the parameters. A parameter is like a local variable that lives inside a procedure.
- A procedure can have no parameters, one parameter, or as many as you need.
- Procedures make your code more flexible—you can reuse the same procedure for lots of different inputs.

3.2 `for` loops

A `for` loop repeats a block of code a fixed number of times. When you know in advance how many times you want something to run, a `for` loop is usually the clearest and most concise option.

The loop definition starts with the `for` keyword, followed by a loop variable, an equals sign, and a range of values to count over:

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

The loop is closed with the `next` keyword. The code inside the loop—in this case, line 110—is called the *body* of the loop. The body executes once for each value in the loop variable's range. In the example above, the loop runs ten times, with `i` taking on the values 1 through 10, so the program prints "Hello world" ten times.

Because the loop variable changes each time, you can use it inside the body to produce different results on each pass. For example, this program prints the numbers 1 through 10:

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

Nested loops

A loop can contain another loop inside its body. This is called a *nested loop*. Nested loops are especially useful when you need to repeat an action across two or more dimensions—for example, filling rows and columns of a table.

For instance, to display a multiplication table, you could write:

```
10 cls                          ' clear the screen
20 for i=1 to 9                 ' outer loop, cycle through rows 1 to 9
30   for j=1 to 9               ' inner loop, cycle through columns 1 to 9
40     print i;"x";j;"=";i*j,   ' print one multiplication fact (i x j)
50   next                       ' go to the next column
60   print                      ' move the cursor to the next line
70   print                      ' insert a blank line for spacing
80 next                         ' go to the next row
```

As indicated by indentation, lines 30–70 form the body of the *outer loop*, while line 40 is the body of the *inner loop*.

Let's break down how this program executes, step by step:

- When the program first enters the outer loop (`i=1`), the inner loop in lines 30-50 runs through all values of `j` from 1 to 9, printing the results of `1 × j` .
- After the inner loop finishes, execution returns to the outer loop's body. Lines 60 and 70 add row spacing, and then the `next` statement in line 80 increases `i` by one, and the process repeats with `i=2` .
- This continues until the outer loop has cycled through all its values, producing the full table.

Notice that the inner loop restarts for each new row, allowing the program to cover every combination of two ranges of values, creating 81 multiplication facts in total:

```
1x1=1    1x2=2    1x3=3    1x4=4    1x5=5    1x6=6    1x7=7    1x8=8    1x9=9

2x1=2    2x2=4    2x3=6    2x4=8    2x5=10   2x6=12   2x7=14   2x8=16   2x9=18

3x1=3    3x2=6    3x3=9    3x4=12   3x5=15   3x6=18   3x7=21   3x8=24   3x9=27
...

9x1=9    9x2=18   9x3=27   9x4=36   9x5=45   9x6=54   9x7=63   9x8=72   9x9=81
```

Nested loops aren't limited to two levels—you can nest three or more if the problem naturally has more dimensions. Be aware, though, that readability drops quickly as nesting grows. In such cases, it is often clearer to move the inner logic into a *named procedure*. This allows the outer loop to function as a high-level outline, while the steps of the inner loops are contained in a separate, well-labeled block of

code.

For example, our multiplication-table program can be rewritten to move the inner loop into its own procedure:

```
10 cls                              ' clear the screen
20 for i=1 to 9                     ' loop through rows 1 to 9
30   print_row(i)                   ' print multiplication facts for the row
40   print                          ' insert a blank line for spacing
50 next                             ' go to the next row
60 end
100 proc print_row(i)               ' print one row of the table
110   for j=1 to 9                  ' loop through columns 1 to 9
120     print i;"x";j;"=";i*j,      ' print the multiplication fact
130   next                          ' go to the next column
140   print                         ' move the cursor to the next line
150 endproc
```

This version produces the same results as before, but the outer loop now reads like a high-level outline: “for each row, print the row, then add spacing.” Meanwhile, the detailed logic for printing a row is encapsulated in a self-contained procedure.

Counting backwards

You can also make a loop count backwards by using the `downto` keyword instead of `to`. This version prints the numbers from 10 down to 1:

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

> **COMPATIBILITY WITH OTHER BASICS**
>
> – There is currently no `step` keyword. Loops always count either upwards by 1 or downwards by 1.
>
> – In some BASIC dialects, the loop variable must be written again after the `next` keyword (for example, `next i`), and intricate behaviors are triggered if a loop is closed out of order. SuperBASIC simplifies this by requiring only a plain `next`, with no variable name, and it does not support or allow those peculiar behaviors.

3.3 `while` and `repeat` loops

`while` and `repeat` are a structured way of doing something repeatedly, until a condition becomes either true or false.

A `while` loop checks its condition before entering the loop. For example:

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

Here, the program keeps calling `playgame()` while the variable `lives` is greater than zero. If the test on line 110 fails immediately, the loop body will never run. Notice how indentation, shown when the program is listed, helps to make the repeated block visually clear.
A `repeat` loop, on the other hand, always runs its body at least once, because the test is checked only at the end:

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

This example produces the same behaviour as the `while` loop above, but with a different control flow: the loop executes `playgame()` once before checking whether `lives = 0`.

3.4 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."
```

(the built-in instruction `explode` plays a simple explosion sound effect).

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
```

This prints whether a number is even or odd, based on the value of `n`. You can include multiple lines of code in either the IF or ELSE clause.

The ELSE clause is optional.

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.

4 Graphics

4.1 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.

4.2 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 F256 can have up to three bitmaps, but only one is supported using BASIC commands.

4.3 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 4`	Synonyms, sets the current drawing colour from LUT 0, which is set up as RRRGGGBB.

4.4 Some useful examples

All these examples start with `bitmap on:cls:bitmap clear 3`, which enables the bitmap display, clears the screen, and fills the bitmap with color value 3. In binary, this is `00000011`, and since colors are encoded as `RRRGGGBB` by default, this corresponds to 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.

4.5 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).

Colors are values from 0 to 255, interpreted in binary as RRRGGGBB by default.

5 Tiles and Tile Maps

5.1 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.

5.2 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
```

5.3 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 plot 10,11 line 3,10
110 tile to 14,12
120 t = tile(5,5)
```

5.4 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).

6 Sprites

6.1 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.

6.2 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.

6.3 Getting images

Sprite data is built from PNG images up to 32x32. There are some examples in the `games/solarfox` directory in the SuperBASIC repository on GitHub.[1]

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.

[1] https://github.com/FoenixRetro/f256-superbasic/tree/main/games/solarfox

6.4 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.

6.5 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.

7 Sound

7.1 Introduction

Depending on the hardware model and generation, an F256 machine includes a stereo SN76489 sound chip, and may also feature additional audio hardware such as SID, OPL3, and MIDI chips.

In SuperBASIC, sound output uses only the SN76489 chip, with the same tones played simultaneously on both the left and right channels.

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).

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.

7.3 Easy commands

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

8 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). Assembled routines can be called via the CALL statement.

8.1 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`.

8.2 The Assemble command

Assembly is controlled by the ASSEMBLE command, which takes two parameters: the first specifies the memory address where the code will be assembled, 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.

8.3 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 why: on the first pass, the assembler doesn't yet know where `forward` is. On the second pass, it has been defined by line 140, so the branch can be calculated correctly.

```
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.

8.4 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 off using an assembler like 64tass or acme, and loading the generated code into memory with the BLOAD command.

9 Cross-Development

Cross-development offers an alternative to the traditional way of programming the F256, where code is typed directly on the machine. Instead, you write code on a separate computer with greater computing power and development resources, then upload it via the USB debug port. This approach can be used with BASIC, machine code, graphics, and other data.

9.1 Assistance

Each release in the SuperBASIC repository includes the file `howto-crossdev-basic.zip`, which contains everything you need to cross-develop in BASIC, along with some example programs.

9.2 Connection

To connect your F256 to a PC (Windows, Linux, or Mac), you'll need a standard USB data cable: Micro USB for the F256J/K, or USB-C for the F256J2/K2. The other end of the cable should match your development machine's USB port. Make sure to use a data-capable cable, as some only provide power. Connect the Micro USB or USB-C end to the F256, and the other end to your other computer.

9.3 Utility Software

There are two ways to program the F256 over USB. I prefer using FnxMgr,[1] a Python script that runs on all platforms and allows you to automate code uploading. Alternatively, cross-developed code can be uploaded using the Foenix IDE on Windows.

9.4 Cross-developing in SuperBASIC

You can cross-develop your program using any standard text editor. Line numbers aren't required during development, but you can include them if you prefer or need to—for example, when porting older code.

Example: Print to the screen and make a silly sound effect

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

[1] https://github.com/pweingar/FoenixMgr

However, line numbers must be added before uploading, as the SuperBASIC interpreter on the F256 uses them to organize and edit the program. The file must also end with a character whose ASCII code is greater than 127. Running your program through the `number.py` script in the `howto-crossdev-basic.zip` archive ensures both requirements are met.

If your program already includes line numbers, you can add the end-of-file marker manually by copying it from one of the examples in the archive.

9.5 Uploading and running

Note that this section assumes you're using a machine that starts up directly into SuperBASIC. If you're booting from RAM, the process may differ slightly.

Uploading works by loading an ASCII text file into memory, which is then effectively "typed in" via either the XLOAD or XGO command. XLOAD loads the program into the interpreter, allowing you to list, edit, or run it as usual. XGO does the same but immediately runs the program afterward.

To load your program into memory for use with XLOAD or XGO, use a command like one of the following. The first works on an Arch Linux system; the second is an untested example for Windows. On Windows, you can identify the correct COM port using Device Manager; on Linux, use `lsusb` or `dmesg`.

Example: Linux Upload

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

Example: Windows Upload (untested)

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

9.6 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.

9.7 Sprites

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

10

Keyword, Operator and Symbol Reference

This section lists all the keywords, operators and symbols used in SuperBASIC. While everything is included here for completeness, some topical keywords—such as graphics commands—are explained in more detail in their dedicated chapters.

! (exclamation mark)

A 16-bit indirection operator. Reads or writes a 16-bit word at a specific address or offset, similar to `peekw` and `pokew`. It has two forms:

- Unary: `!47` reads the 16-bit word stored at address 47 (i.e., it reads the byte at 47 and the next byte at 48 as a single word).
- Binary: `a!4` reads the word at address `a + 4`, making it easy to index into structured data like tables or arrays.

When used on the left-hand side of an assignment, it behaves like `pokew`, writing a 16-bit value to memory in little-endian order (low byte first, then high).

Example

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

@ (at sign)

The address operator. Returns the memory address of an expression that has a fixed location in memory—typically a variable or an array element. This address can be assigned to another variable and used later to access or modify the original value.

For example, `@fred` gives the address where the variable `fred` is stored. You can store that address in another variable—often called a pointer—and later use it with one of the indirection operators (`?`, `!`) or with `poke` to read or change the contents of `fred`.

Example

```
100 print @fred, @a(4)
110 fred = 17
120 ptr = @fred
```

```
130 ?ptr = 42
140 print fred
```

(number sign)

Identifier suffix designating a floating-point variable. Floating-point values are stored as a 32-bit signed mantissa and an 8-bit exponent, and can represent numbers approximately between -3.4E+38 and +3.4E+38, with around 9 decimal digits of precision. The type suffix is considered part of the variable's name; for example, `age` and `age#` are treated as two distinct variables (integer and floating-point, respectively). See Chapter 2 for an in-depth exploration of variable types.

Example

```
100 an_integer = 42
110 a_float# = 3.14159
120 print an_integer, a_float
```

$ (dollar sign)

Depending on where it's encountered, the dollar sign can play one of two roles:

When placed before a numeric constant, it designates a hexadecimal value. For example, `$10` is interpreted as a hexadecimal constant with the decimal value 16. Similarly, `$2A` represents the decimal value 42:

Example

```
100 n = $2A
110 if n = 42 then print "Found it!"
120 !$7ffe = 31702
```

When used as a suffix in an identifier, `$` designates a string variable. The suffix is considered part of the variable name, so `id` and `id$` are treated as two distinct variables—an integer and a string, respectively. See Chapter 2 for an in-depth discussion of variable types.

Example

```
100 an_integer_id = $FFFF
110 a_string_id$ = "Hello world"
120 print an_integer_id, a_string_id$
```

% (percent sign)

Remainder operator. Returns the non-negative remainder from dividing the first operand by the second. Both operands must be integers. The second operand must be non-zero.

Example

```
100 print 42 % 5    ' prints 2
110 print -17 % 5   ' prints 2
```

^ (caret symbol)

Bitwise XOR operator. Performs an exclusive OR operation on each bit of its operands. Both operands must be integers.

Example

```
100 print a ^ $0E
```

& (ampersand)

Bitwise AND operator. Performs an AND operation on each bit of its operands. Both operands must be integers.

Example

```
100 print count & 7
```

| (vertical bar)

Bitwise OR operator. Performs an OR operation on each bit of its operands. Both operands must be integers.

Example

```
100 print value | 4
```

* (asterisk)

Integer or floating-point multiplication. If either operand is floating-point, the result is floating-point.

Example

```
100 print 4 * 2      ' prints 8
110 print 4.00 * 2   ' prints 8.00000
```

/ (forward slash)

Floating-point division. The result is always a floating-point value, even if both operands are integers. An error occurs if the divisor is zero.

Example

```
100 print 22 / 7    ' prints 3.14285
110 print -42 / 7   ' prints -6.00000
```

\ (backward slash)

Integer division. Both operands must be integers. If the first number doesn't divide evenly by the second, the result is the whole-number part of the answer (rounded toward zero). An error occurs if the divisor is zero.

Example

```
100 print 22 \ 7    ' prints 3
110 print -27 \ 7   ' prints -3
120 print 42 \ 7    ' prints 6
```

- (minus sign)

Integer or floating-point subtraction. If either operand is floating-point, the result is floating-point.

Example

```
100 print 44 - 2       ' prints 42
110 print -40 - 2.00   ' prints -42.00000
```

+ (plus sign)

Integer or floating-point addition, or string concatenation. If either of the numeric operands is floating-point, the result is floating-point. Adding a number to a string produces a type error. See `str$` for an easy way to convert a number to a string.

Example

```
100 sum = 4 + 2                  ' sum = 6
120 total# = 17.5 + 42.5         ' total# = 60.00000
130 prompt$ = "Hello " + "Bob"   ' prompt$ = "Hello Bob"
140 print sum, total#, prompt$
```

? (question mark)

An 8-bit indirection operator. Reads or writes a single byte at a specific address or offset, similar to `peek` and `poke`. It works like `!`, but operates at the byte level instead of 16-bit words.

Example

```
100 a = 17          ' a is a 32-bit integer
110 ptr = @a        ' ptr holds the address of a
120 print ?ptr      ' prints 17 (least significant byte of a)
130 ptr?0 = 255     ' modify a's least significant byte
140 print a         ' prints 255
```

: (colon)

Statement separator. By default, only one statement is allowed per line, but you can include multiple statements by separating them with a colon. Typically used for semantic grouping of related statements.

Example

```
100 cls: print "Hello"  ' clears the screen, then prints "Hello"
```

. (period)

Assembly label operator. Assigns the label following the period to the current assembly address. The label acts as an integer variable that holds this address:

Example

```
100 ' Fills the top screen row with the character in A
110 assemble $6000, 0: .fill_top_row ' start an assembly block
    and label it
120 ldy $0001                        ' save I/O page in Y
130 pha: lda #2: sta $0001: pla      ' switch to I/O page 2
```

```
140 ldx #80                          ' number of columns to fill
150 .fill_loop                       ' define the loop label
160 dex                              ' move (backwards) to the next
    column
170 sta $c000,x                      ' write the char to column X
180 cpx #0                           ' are we done?
190 bne fill_loop                    ' if not zero, keep going
200 tya: sta $0001                   ' restore original I/O page
210 rts                              ' return from the routine
220 cls                              ' clear the screen
230 call fill_top_row, asc("#")      ' fill the top row with #
```

Labels can be referenced before they are defined, but doing so requires a multi-pass assembly block—see Chapter 8 for details.

Since labels are variables, they must be globally unique.

' (apostrophe)

Marks the beginning of a comment. Equivalent to `rem`, but more concise and does not require a colon (`:`) when used after other statements. Everything following the apostrophe on the same line is ignored.

Example

```
100 ' This is a title comment
110 print 2*2  ' prints 4
```

" (quote)

Delimits a string literal. A string must begin and end with a quote on the same line. To construct a string that spans multiple lines, use string concatenation with the newline character (`chr$(13)`). To include an actual quote character inside the string, use two consecutive quotes (`""`).

Example

```
100 print "Hello, world!"                    ' Hello, world!
110 print "She said ""hi"" and walked away."  ' She said "hi"
    and walked away.
120 greeting$ = "Hi, Bob!" + chr$(13) + "How are you today?"
130 print greeting$
```

< (less than)
<= (less equal)
= (equal)
<> (not equal)
> (greater than)
>= (greater equal)

Numeric and string comparison. Each operator returns 0 if the condition is false and -1 if true. Comparing a number to a string produces a type error. Use `str$` and `val` for type conversion.

Example

```
100 input "Enter your answer:", a
```

```
110 if a <> 42: print "Incorrect": else: print "Correct!": endif
120 if name$ = "" then input "Enter your name:", name$
130 print "Hello, "; name$
```

<< (left shift)
>> (right shift)

Bit-shift operators. Shift an integer left (`<<`) or right (`>>`) by a given number of bits. Both operands must be integers, and the result is always an integer. Often used as a fast alternative to multiplying or dividing by powers of two.

Example

```
100 n = 42
110 print n << 2   ' prints 168 (42 * 4)
120 print n >> 1   ' prints 21 (42 \ 2)
```

abs()

Returns the absolute value of a number.

Example

```
100 print abs(-4)         ' prints 4
110 print abs(-3.14159)   ' prints 3.14159
```

alloc()

Allocates the specified number of bytes in memory and returns the starting address of the allocated block. Useful for efficient byte-level storage, custom data structures, or program memory for assembly instructions.

The following example uses `alloc()` to build a table of printable ASCII characters from 32 to 127:

Example

```
100 char_start = 32: char_end = 127      ' define ASCII range
110 buffer_size = char_end - char_start
120 buffer = alloc(buffer_size)          ' allocate memory buffer
130 ' fill buffer with ASCII codes from start to end
140 for n = 0 to buffer_size - 1
150   poke buffer + n, char_start + n
160 next
170 ' print characters stored in buffer
180 for n = 0 to buffer_size - 1
190   print chr$(peek(buffer + n));
200 next
```

asc()

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

Example

```
100 print asc("*")
```

assemble

Initialises an assembler pass. Apart from the simplest bits of code, the assembler is two pass. It has two parameters. The first is the location in memory the assembled code should be stored, the second is the mode. At present there are two mode bits; bit 0 indicates the pass (0 1st pass, 1 2nd pass) and bit 1 specifies whether the code is listed as it goes. Normally these values will be 0 and 1, as the listing is a bit slow. 6502 mnemonics are typed as is. Two passes will normally be required by wrapping it in a for/next loop

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
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 layer on or off, clears it, or sets its memory address (`$10000` by default). Only one bitmap layer is supported. Modifier keywords include `on`, `off`, `clear` <*color*>, and `at` <*address*>, and may be chained as shown in the example below. Using `on` or `off` without `at` will reset the bitmap address to its default. See Chapter 4 for more details.

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 setup, for the RAM area (0000-7FFF) this will however be the same.

The example below loads the binary file `mypic.bin` into the bitmap layer, 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 setup, for the RAM area (0000-7FFF) this will however be the same.

Example

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

call

Calls an assembly subroutine at the specified address. Optionally, you may provide up to three arguments, which will be loaded into the A, X, and Y registers, respectively.

Example

```
100 call $4000
110 call $5000, $ff, $f0
```

chr$()

Convert an ASCII integer to a single-character string.

Example

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

circle

Draws a circle. The vertical height defines the radius of the circle. See Chapter 4, *Graphics* for drawing options.

Example

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

cprint

Operates the same as the `print` command, but control characters (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

Lists 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` and `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()

The `event()` function tracks elapsed time and is commonly used to trigger movement or actions in games. It returns `true` at predictable intervals, based on a 70Hz timer (i.e. 70 ticks per second). It takes two parameters: a variable and a duration in ticks.

If the variable is zero, the function waits the specified number of ticks before returning `true` once. If the variable is non-zero, it tracks repeated events. For example, `event(evt1,70)` returns `true` once per second (since 70 ticks = 1 second).

Note: Event timing continues even if the game is paused. So if an event is set to occur every 20 seconds, it may appear to fire during the pause. To avoid this, reset the event variable to zero when unpausing—this restarts the timer.

If the event variable is set to -1, it disables the event. This can be used to implement one-shot timers by setting the variable to -1 once the event has fired.

The example below 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 that repeats code a fixed number of times. The loop body will 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
110 for i = 10 downto 1: print i: next
```

frac()

Return the fractional part of a number.

Example

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

get() / 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 a three-parameter command directly to the graphics subsystem. The last two parameters are often coordinates, although not always.

This is a low-level call to the graphics library and is generally discouraged for regular use. The command parameters are documented in the `graphics.txt` document in the SuperBASIC repository.[1] Use of this function is uncommon.

Example

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

gosub

Calls a routine at the specified line number. Provided for compatibility with older BASIC programs. For new code, use named procedures instead (see Chapter 3).

Example

```
100 gosub 1000
```

goto

Transfers execution to the specified line number. Consider using structured programming constructs (Chapter 3) instead.

Example

```
10 print "Hi there! ";
20 goto 10
```

hit()

Tests whether two sprites overlap, using a bounding box test based on their size (e.g., 8×8, 16×16, 24×24, or 32×32).

Returns zero if there is no collision, or the smaller of the two coordinate differences from the center.

This function only works for sprites positioned using the graphics system; there is no way to read sprite memory directly to determine their on-screen positions.

Example

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

if ... then
if ... else ... endif

The `if` statement. Comes in two forms:

The first is the classic BASIC style: `if` <*condition*> `then` <*statement*>. All code must be on a single line, and `then` is mandatory:

[1] https://github.com/FoenixRetro/f256-superbasic/blob/main/documents/graphics.txt

Example

```
100 if name$ = "Alice" then age = 7
110 if name$ = "Dinah" then goto 200
```

The second form, `if` ... [`else` ...] `endif`, supports multi-line conditional logic and includes an optional `else` clause. The `endif` keyword is mandatory, even if no `else` is used. This form can also be written on a single line by separating each statement with colons:

Example

```
100 if age < 18: print "child": else: print "adult": endif
110 if name$ = "Bob"
120   print "Hello, Bob!"
130 else
140   print "Hello, stranger!"
150 endif
```

image

Draws a sprite image onto the bitmap, with optional scaling or flipping. Flipping is controlled by bits 7 and 6 of the mode byte (i.e., $80 for horizontal flip, $40 for vertical flip) in the colour parameter. Both the sprite and bitmap systems must be enabled. For more details, see Chapter 4.

Example

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

inkey()
inkey$()

Returns the most recent key press, if any. `inkey()` returns the ASCII code of the key, while `inkey$()` returns the character as a string. If no key has been pressed, they return `0` and `""` respectively.

These functions check for past key presses—they do not detect whether a key is currently being held down. To check the current state of a key (up or down), use `keydown()` instead.

Example

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

input

Inputs numbers or strings from the keyboard. The syntax is identical to `print`, but instead of displaying values, it reads input into the specified variables. Wherever a variable appears, a value will be read from the keyboard and stored in that variable.

Example

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

int()

Returns the integer part of a number.

Example

```
100 print int(3.14159)
```

isval()

Returns `true` if a string represents a valid numeric value, `false` otherwise. A companion to `val()`.

Example

```
100 print isval("42")
110 print isval("I like chips with salsa")
```

itemcount()

Takes a string and a separator, and returns the number of items found by splitting the string at each occurrence of the separator. A companion to `itemget$()`.

The example below prints `2`, as the string contains two items, `"hello"` and `"world"`, separated by a comma.

Example

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

itemget$()

Extracts the specified item from a string split using a separator. Takes three parameters: the input string, the index of the desired item (starting from 1), and the separator. Returns the specified sub-item. If the index is out of range, a range error is raised. See also `itemcount()`.

The example below prints `"lizzie"`, which is the third item in the list.

Example

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

joyb()

Returns a integer value indicating the status of the fire buttons on a joystick or gamepad. Bit 0 corresponds to the main fire button. Takes a single parameter, the gamepad number.

Keyboard keys Z / X / K / M / L (left / right / up / down / fire) are also mapped to this input, so a physical gamepad is not required.

Example

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

joyx()
joyy()

Return the directional value of a joystick or gamepad along the X or Y axis. A value of `1` indicates movement to the right (X) or down (Y), `-1` indicates movement to the left or up, and `0` means no movement. Each function takes a single argument, the gamepad number.

Keyboard keys Z / X / K / M / L (left / right / up / down / fire) are also mapped to this input, so a physical gamepad is not required.

Example

```
100   cls: cursor off
110   while true
120     print at 0,0;"x:";joyx(0);" y:";joyy(0);"   ";
130   wend
```

keydown()

Checks whether a key is currently being pressed. The function takes a single parameter, the raw key code. The example below also serves as a simple tool for identifying these 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 a specified number of characters from the beginning of a string.

Example

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

len()

Returns the length of the string, that is, the number of characters in the string.

Example

```
100 print len("hello, world")   ' prints 12
```

let

Assignment statement. In SuperBASIC, the `let` keyword is optional and is provided for compatibility with older BASIC dialects.

Example

```
100 let a = 42   ' you can use LET
110 b = 17       ' ... but you don't have to
```

line

Draws a line on the bitmap layer. See Chapter 4 for more details.

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)   ' prints 5
```

mdelta

Gets the current delta status of the 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'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 F256. 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.

Example

```
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 below 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 specified memory address.

Example

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

print

Displays text or numbers on the screen at the current cursor position. You can print strings, numbers, variables, or results of calculations, and you can mix them together in the same statement.

Example

```
100 print "2 + 2 = " 4
110 print "Hmm,"; (17 - 10) * 6; " sounds familiar..."
120 input "What's your name?", $name
130 print "Hello, "; $name; "!"
```

Items separated by a space or a semicolon (`;`) are printed directly after one another, while a comma (`,`) places the next item at the next tab position.

Normally, `print` finishes by moving the cursor to a new line, but if it ends with a semicolon (`;`) or comma (`,`), the next `print` will continue on the same line.

Example

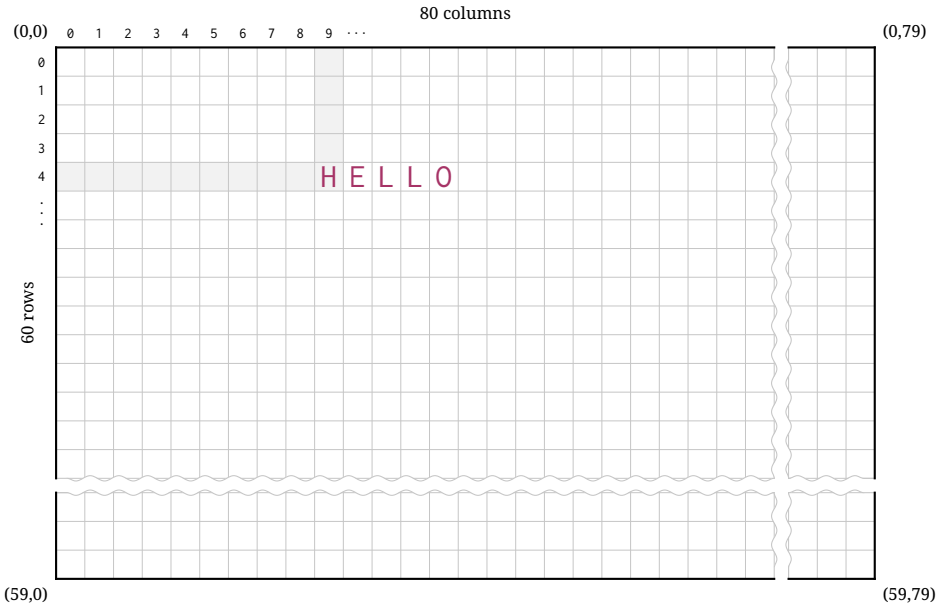
```
100 print "Goodbye, ";
110 print "feet!"
```

`at` modifier

You can use the `at` modifier to position the cursor and the following `print` output at specific screen coordinates. SuperBASIC's default text mode fits 80 characters across and 60 lines down. The `at` coordinates are zero-based and given as `row`, `column`:

Example

```
100 cls
110 ' print at row 4, column 9
120 print at 4, 9; "HELLO"
```



The following program places an asterisk in each corner of the screen, then moves the cursor back to the top-left corner:

Example

```
100 cls
110 print at 0, 0; "*";
120 print at 0, 79; "*";
130 print at 59, 0; "*";
140 print at 59, 79; "*";
150 print at 0, 0;
```

Note that when control returns to SuperBASIC, the system will place its input prompt where the cursor was last positioned. If you delete line 150 and run the program again, because line 140 positions the cursor on the very last character of the screen, the system will have to scroll the output to make room for the prompt, erasing the asterisks at the top.

The `at` modifier doesn't have to appear immediately after `print`, may be used multiple times in the same statement, and can be freely mixed with other arguments:

Example

```
100 cls
110 print "Hello"; at 2,4 "from"; at 4,8 "Canada"
```

For the reverse of `print at`, that is, *reading* the character at a specific screen position, see `screen$()`.

proc ... endproc

Simple procedures. These should be used rather than `gosub`. The empty parentheses 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
```

rem

Comment. `'` and `rem` are synonyms. The rest of the line is ignored.
Short for remark, marks the start of an explanatory comment.
It's considered to These may be included anywhere in the source code and extend to the end of the line. Comments are ignored when the program is run.

Example

```
10 ' This is a title comment
20 rem
30 rem Another comment
```

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.

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 real-time clock to the given date; the parameters are the day, month and year (00-99).

Example

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

settime

Sets the real-time clock to the given time; the parameters are hours, minutes, and seconds.

Example

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

sgn()

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

Example

```
100 print sgn(42)
```

screen()
screen$()

Returns the character located at the given screen coordinates. `screen()` returns the ASCII code of the character, while `screen$()` returns the character itself as a string. The coordinates are given as `row`, `column`, and use the same coordinate system as `print at`:

Example

```
100 print at 15, 10; "#"
110 print screen$(15, 10) ' prints "#"
120 print screen(15, 10)  ' prints 35
```

See `print` for a detailed visual explanation of the screen coordinates.

Together, `screen` and `print at` let you treat the screen as a big grid of characters, making them great tools for simple character-based game mechanics such as placing and moving objects on a map and detecting collisions:

Example

```
100 cls
110 print at 15, 30; "@"
120 input at 0, 0; "Guess a row (0-59): "; row
130 input at 0, 0; "Guess a column (0-79): "; col
140 if screen$(row, col) = "@"
150   print at row, col; "x"
160   print at 0, 0; "*** You hit the target! ***"
170 else
180   print at 0, 0; "=== Oh no, you missed! ==="
190 endif
```

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.

Example

```
100 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 example below, lines 110 and 120 do the same thing.

Example

```
100 tile to 12,0
110 tile at 4,5 plot 10,11,11,11,10
120 tile at 4,5 plot 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 number to a string, formatting it 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
explode

Simple commands that generate a simple sound effect.

Example

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