NextBASIC new commands and features (Updated 19 Feb 2018)

This document describes planned new commands and features for NextBASIC. Many of these are largely complete already, but some are still to be implemented; the features as described will be available for the official cased machines launch (as NextOS v2.00).

This should be read in conjunction with the other documents:

- NextBASIC file-related commands and features
- NextOS Editor features
- NextOS API
- NextOS Unimplemented features

A list of updates made to this document is now provided at the end.

New keyword tokens

The following keyword tokens are defined in addition to SPECTRUM, PLAY and all the 48K BASIC tokens:

MOD	$8b
<<	$8c
>>	$8d
UNTIL	$8e
ERROR	$8f
ON	$90
DEFPROC	$91
ENDPROC	$92
PROC	$93
LOCAL	$94
DRIVER	$95
WHILE	$96
REPEAT	$97
ELSE	$98
REMOUNT	$99
BANK	$9a
TILE	$9b
LAYER	$9c
PALETTE	$9d
SPRITE	$9e
PWD	$9f
CD	$a0
MKDIR	$a1
RMDIR	$a2

New errors

Invalid mode
Direct command error
Loop error
Missing procedure

Memory bank access

The *Next* comes with between 1MB and 2MB of RAM, divided into 16K banks. These are numbered as follows under *NextOS*:

0..7	Same as the standard RAM banks on all 128K Spectrums.
8..47	Additional RAM banks available on 1MB Nexts.
48..111	Further additional RAM banks available on 2MB Nexts.

(the remaining 256K is used for ROMs and the DivMMC interface, and is unavailable to users).

Under *NextOS* the memory capacity is shown in the on-screen menus. It can also be queried programmatically by examining the new system variable, *MAXBNK*, which contains the number of the highest usable bank in the system (normally 47 or 111).

NextOS uses the first 9 RAM banks as follows:

0	Standard 48K Spectrum memory (at 49152-65535)
1	RAMdisk
2	Standard 48K Spectrum memory (at 32768-49151)
3	RAMdisk
4	RAMdisk
5	Standard 48K Spectrum memory (at 16384-32767)
6	RAMdisk
7	Used for workspace and data structures by *NextOS*
8	Used for additional screen data (in lo-res, Timex hi-res and Timex hi-colour modes) and other data by *NextOS*

Banks 9+ are always available to the programmer, and can be accessed using the new **BANK** command (and extended **LOAD/SAVE/VERIFY..BANK..** commands seen previously).

Banks 5,2,0 (the standard 48K Spectrum memory) may also be used without restriction in the **BANK** command, but it should be noted that it is generally only safe to use the screen area (0-6911 in bank 5) plus any memory located above *RAMTOP* - the rest of the memory is managed by *NextBASIC* and should not be modified.

Banks 1,3,4,6 can be used if the **BANK 1346 USR** command has been executed.

Banks 7 and 8 are for system use only, and can never be used in a **BANK** command.

The following new commands are available to manipulate banks:

BANK *n* **POKE** *offset,value*
POKE a byte *value* at offset *offset* (0-16383) in bank *n*

BANK *n* **PEEK** *offset* **TO** *var*
PEEK a byte at offset *offset* (0-16383) in bank *n*, and store the value in numeric variable *var*

BANK *n* **COPY TO** *n2*
Copy all 16K from bank *n* to bank *n2*

BANK *n* **COPY** *offset,len* **TO** *n2,offset2*
Copy *len* bytes starting at offset *offset* in bank *n* to offset *offset2* in bank *n2*

BANK *n* **ERASE**
BANK *n* **ERASE** *value*
Fill all 16K of bank *n* with *value* (zero is used if *value* not specified)

BANK *n* **ERASE** *offset,len*

BANK *n* **ERASE** *offset,len,value*
Fill *len* bytes at offset *offset* in bank *n* with *value* (zero is used if *value* not specified)

BANK 1346 USR
Allow banks 1,3,4,6 to be used in the **BANK** command. This will delete all files on the RAMdisk and unmap it from any drive it is currently mapped to (usually M:).

BANK 1346 FORMAT
Release banks 1,3,4,6 for use by the RAMdisk again. (The RAMdisk will need to be mapped back to a drive using the **MOVE..IN** command).

BANK *n* **CLEAR**
Marks bank *n* as free for use by other parts of the system (eg dot commands). Banks are marked as used by BASIC by commands that access them (eg **BANK..PEEK/POKE/COPY/ERASE/USR/LAYER**, **LAYER BANK** and **LOAD..BANK**). Marked banks remain reserved after a **NEW** command, and are only released at a reset (or with this command).
Note that the layer 2 banks (by default 9,10,11 but may be changed using the **LAYER BANK** command) cannot be released. However, if you are not using layer 2, they can be used for other purposes (including by machine code programs).

BANK NEW var
Reserves the next available free bank number and assigns it to the numeric variable *var*, ready for use in other **BANK** commands. This command is useful for allocating banks for use in BASIC, allowing for cases where a resident machine code program has previously allocated banks for its own use. (It is not essential to use this command, as commands such as **BANK..LOAD** will automatically allocate the specified bank for use by BASIC, but only if the specified bank is not already in use by a resident machine code program.)

BANK *n* **LAYER** *x,y,w,h* **TO** *[rop] offset*
Copies data from the screen (in the current mode) to *offset* in bank *n*.
BANK *n* **LAYER** *offset* **TO** *[rop] x,y,w,h*
Copies data to the screen (in the current mode) from *offset* in bank *n*.

[rop] is an optional symbol modifier which affects how the data is copied:

TO	(no symbol)	straightforward copy
TO &		AND the copied data into the destination
TO |		OR the copied data into the destination
TO ^		XOR the copied data into the destination
TO ~		copy data into the destination unless it is equal to the global transparency colour (default **227**); in this case, leave the destination unchanged

The area of screen copied by **BANK...LAYER** is defined by the top left character position *x,y* and width *w* characters, height *h* characters. (As with windows, character positions range from x=0..31 and y=0..23 for all modes except lo-res, where they range from x=0..15 and y=0..11).

Data copied from the screen is laid out as follows, depending upon currently selected layer/mode:

Standard resolution (layer 0 or layer 1,1)
The attribute data comes first, stored as *h* consecutive rows of attributes, *w* bytes wide.
Following this is the screen data, stored as *h*8* consecutive rows of pixel data, *w* bytes wide.
The total memory used is therefore *w*h*9* bytes.

Timex hi-res (layer 1,2)

In this mode, each “character” position is 16 pixels wide, comprising a “left” and “right” half.
The screen data is stored as *h*8* consecutive pixel rows of data.
For each row, the first *w* bytes comprise the left halves of all characters.
The next *w* bytes in the row comprise the right halves of all the characters.
The total memory used is therefore *w*h*16* bytes.

Timex hi-colour (layer 1,3)
The screen data is stored as *h*8* consecutive pixel rows of data.
For each row, the first *w* bytes comprise the pixel data.
The next *w* bytes in the row comprise the attribute data.
The total memory used is therefore *w*h*16* bytes.

Lo-res (layer 1,0) and layer 2
The data is stored as *h*8* consecutive pixel rows of data.
For each row, there are *w*8* bytes, with each byte representing a single pixel.
The total memory used is therefore *w*h*64* bytes.

Palette manipulation

The *Next* provides 6 palettes: 2 palettes each (numbered 0 and 1) for sprites, ULA modes, and layer2. All can be manipulated in BASIC. Note that the Editor will use ULA palette 1 so it is safe to muck around with palette 0 without risk of being unable to see what's going on (the current palette will be restored by the Editor when BASIC is running).

The following new palette manipulation commands are available:

PALETTE DIM *n*

Palettes being specified in the **LAYER PALETTE BANK** and **SPRITE PALETTE BANK** commands use *n* bits per colour (n=8 or 9), ie 256 bytes or 512 bytes (default value is *n*=9).

PALETTE FORMAT *n*

Enable the ULANext extended palette with *n* INKs (1,3,7,15,31,63,127 or 255)
When the ULANext extended palette is enabled, BRIGHT and FLASH are not allowed (in standard and Timex hi-colour modes), and INK and PAPER accept the appropriate new range of values (layer 1 modes only - see later notes).
If *n*=0, disables the ULANext extended palette and uses standard attributes with 8 inks, 8 papers, bright and flash.

PALETTE OVER *n*

Sets the global transparency colour to *n* (default value is 227)

PALETTE CLEAR

Resets all palettes and related settings to defaults. This is also done by **NEW**.

Sprites

The *Next* provides 64 sprites (size 16x16 pixels). These can be manipulated with the following new commands:

SPRITE BANK *b*

Defines all 64 sprite patterns using the 16K of data (256 bytes per sprite) in bank *b*.

SPRITE BANK *b*,*offset*,*p*,*n*

Defines *n* sprite patterns starting with pattern *p*. Pattern data begins at offset *offset* in bank *b*.

SPRITE PALETTE *n*

Switch to using sprite palette *n* (0 or 1)

SPRITE PALETTE *n* **BANK** *b*,*offset*

Set sprite palette *n* from bank *b*, at offset *offset*. Either 256 bytes or 512 bytes of data is used, depending upon the **PALETTE DIM** setting.

SPRITE PALETTE *n*,*i*,*v*

Set sprite palette *n*, index *i* to value *v*

NOTE: *v* is always specified as a 9-bit value RRRGGGBBB (0-511) regardless of the **PALETTE DIM** setting, and can be conveniently specified using the standard Spectrum **BIN** function.

SPRITE PRINT *n*

Enable (*n*=1) or disable (*n*=0) sprites

SPRITE BORDER *n*

Enable (*n*=1) or disable (*n*=0) sprites over the border

SPRITE *s*,*x*,*y*,*i*,*f*

Set sprite *s* to image *i*, position (*x*,*y*) with flags *f*, which is a bitmask:

```
      bit 0=visible flag
      bit 1=rotate flag
      bit 2=Y-mirror flag
      bit 3=X-mirror flag
      bits 4..7=palette offset (or zero)
```

Again the **BIN** function can be used to specify this more conveniently.

SPRITE DIM *x1*,*y1*,*x2*,*y2*

Sets the clip window for sprites from (*x1*,*y1*) to (*x2*,*y2*). Any part of a sprite outside this window is not visible. Note that this has no effect if sprites over the border (**SPRITE BORDER 1**) is enabled.

SPRITE CLEAR

Resets the sprite attributes and global settings to defaults. This is also done by **NEW**.

Layers and modes

The *Next* provides various new graphics modes, to which *NextBASIC* gives access using the **LAYER** command.

There are conceptually 3 layers of graphics which can be seen on the screen at the same time (the 3 layers can be placed in any front-to-back order). The top layer is usually the sprites, which are manipulated with the **SPRITE** command. The other 2 layers are manipulated by the **LAYER** command. The "bottom" of these two layers can only be seen where the "top" layer has the transparency colour (227).

Layer 1 is the ULA screen, and by default is the bottom layer.
This can be in any of 4 different modes:

- mode 0: lo-res mode (128x96 pixels, each can be any of 256 colours)
- mode 1: standard Spectrum screen mode (256x192 pixels, with 32x24 attributes)
- mode 2: Timex hi-res mode (512x192 pixels, monochrome but with 8 different selectable global ink/paper combinations)
- mode 3: Timex hi-colour mode (256x192 pixels, with 32x192 attributes)

Layer 2 is 256x192 pixels, each can be any of 256 colours. By default it is the top layer but disabled, so does not usually obscure the layer 1 screen.

The **LAYER** command allows either layer to be selected (and for layer 1, any of the 4 available modes to be selected). After the **LAYER** command takes effect, all of the following standard Spectrum commands take place on the selected layer/mode (until another **LAYER** command is issued):

- **INK**, **PAPER**, **BRIGHT**, **FLASH**, **OVER**, **INVERSE**
- **CLS**
- **PLOT**, **DRAW**, **CIRCLE**
- **PRINT**, **LIST**, **CAT** etc (through the standard "s" channel, usually on stream 2)

NOTE: The **ATTR**, **POINT** and **SCREEN$** functions do not take account of the layer/mode settings, and only refer to the standard Spectrum screen. However, instead, the following new command is available:

POINT *x*,*y* **TO** *var*
Checks the pixel on the current layer at (*x*,*y*) and stores the value in variable *var*.
The value will be 0 or 1 for standard Spectrum modes and Timex hi-res and hi-colour modes (pixel off or on). The value will be 0-255 for lo-res and layer 2 (actual pixel colour).

BRIGHT and **FLASH** are only effective in standard and hi-colour modes (and only when the ULANext extended palette is not enabled).
INK and **PAPER** values can range from 0..255 in lo-res and layer 2.
In hi-res mode, either **INK** or **PAPER** can be used to select the appropriate colour scheme (see list later).

The **LAYER** command also allows you to select layer 0. This is the default layer/mode when *NextOS* starts and is identical to the standard Spectrum screen mode used on 48K/128K Spectrums. This is the mode you should select in order to load and run standard Spectrum software.

You can switch back and forth between layer 1 and layer 2 without affecting what is on the screen (as long as you always select the same layer 1 mode each time). This allows BASIC programs to enable and manipulate both layer 1 and layer 2 screens, and use transparent areas so that both can be seen together.

The following **LAYER** commands are available:

LAYER 0
Select layer 0, standard Spectrum mode

LAYER 1,0
Select lo-res mode

LAYER 1,1
Select standard resolution mode

LAYER 1,2
Select Timex hi-res mode

LAYER 1,3
Select Timex hi-colour mode

LAYER 2
Select layer2

LAYER 2,0
Select layer2, and disable displaying it

LAYER 2,1
Select layer2, and enable displaying it

LAYER PALETTE *n*
Switch to using palette *n* (0 or 1) for the current layer

LAYER PALETTE *n* **BANK** *b*,*offset*
Set palette *n* for the current layer from bank *b*, at offset *offset*. Either 256 bytes or 512 bytes of data is used, depending upon the **PALETTE DIM** setting.

LAYER PALETTE *n*,*i*,*v*
Set palette *n* for the current layer, index *i* to value *v*
NOTE: *v* is always specified as a 9-bit value RRRGGGBBB (0-511) regardless of the **PALETTE DIM** setting, and can be conveniently specified using the standard Spectrum **BIN** function.

LAYER AT *x*,*y*
(Layer 2 or lo-res only).
Set the display offset for the top-left of the screen for the current layer to *x*,*y*. This is used for scrolling effects.

LAYER OVER *n*
Set sprite/layer SLU ordering:

n=**BIN** 000	sprites over layer2 over ULA (layer1) - the default
n=**BIN** 001	layer2 over sprites over ULA (layer1)
n=**BIN** 010	sprites over ULA (layer1) over layer2
n=**BIN** 011	layer2 over ULA (layer1) over sprites
n=**BIN** 100	ULA (layer1) over sprites over layer2
n=**BIN** 101	ULA (layer1) over layer2 over sprites

LAYER BANK *n*,*m*
(Layer 2 only). Set current banks *n*..*n*+2 as frontbuffer (to be displayed) and banks *m*..*m*+2 as backbuffer (for rendering). These values can be the same and both default to 9.
This command always applies to the layer 2 banks, but can be executed in any mode.

LAYER ERASE *x*,*y*,*w*,*h*

LAYER ERASE *x*,*y*,*w*,*h*,f
(Layer 2 or lo-res only).
Fill region width *w* pixels, height *h* pixels, top-left corner *x*,*y* with value *f*. If *f* is not specifed, the current global transparency value (usually 227) is used.

LAYER DIM *x1*,*y1*,*x2*,*y2*
Sets the clip window for the current layer from (*x1*,*y1*) to (*x2*,*y2*). Areas of the layer outside this window are not visible. Note that all layer 1 modes and layer 0 share the same clip window; layer 2 has its own separate clip window.

LAYER CLEAR
Reset all layer information to defaults. This is also done by **NEW**.
Resets banks, mode, layer2 enable, layer offsets, layer ordering.

Differences between layer 0 and layer 1 mode 1.
Layer 0 behaves in exactly the same way as the screen always has on 48K and 128K Spectrums. Layer 1 mode 1 has the same resolution and attributes, but behaves in a slightly different manner under *NextBASIC*. It shares this same behaviour with all other layer 1 modes (and layer 2).

In layer 0, the standard Spectrum memory map is in force (ROM, RAM 5, RAM 2, RAM 0). However, in all layer 1 modes, the top 8K of RAM 5 is replaced with 8K from the *NextOS* RAM 8 bank. This is done so that BASIC still has access to the same amount of memory as usual (~41K); without this change, it would lose about 6K to the new screen modes.

The other main differences are:

Layer 0 pixel coordinates (used by **PLOT**, **DRAW**, **CIRCLE**) run from (0,0) at the bottom left on the main screen area to (255,175) at the top right. The bottom two screen lines are not normally accessible to these commands. However, in layer 1/2 modes, pixel coordinates run from (0,0) at the top left of the screen to (255,191) at the bottom right (511,191 in hi-res mode, 127,95 in lo-res mode). In layer 1/2 modes it is also allowed to draw points, lines and circles so that they go partly off-screen without generating "out of screen" errors.

Layer 0 **PRINT** coordinates (on channel "s") are in character squares, defined as (0,0) at the top left and (21,31) at the bottom right (again, the lower screen is not usually accessible).

Layer 0 only accepts standard colour ranges (0..7 for **INK/PAPER** etc). Colours from the extended ULANext palette with numbers higher than 7 can generally only be specified in layer 1 (mode 1 - standard, or mode 3 - Timex hi-colour). For layer 0 you can, however, POKE the system variable ATTR_P with the calculated attribute value required and the desired ULANext colours will be used.

Layer 1/2 modes all use a full-screen system-defined text window for any **PRINT**s directed to channel "s". Therefore they generally use the same control codes as other text windows (except justify and save/load are not available), and **AT** coordinates are defined using a pixel line number and a character position (the number of positions depending upon the character set size selected).

By default, scrolling auto-pause is turned on for the layer 1/2 mode full-screen windows, so after a screen full of text has been printed the user must press SPACE to continue. This behaviour can be disabled using control code 26, as with other windows.

Layer 1/2 modes do not support "9" to mean contrast (for **PAPER/INK**) or "8" to mean transparent (for **PAPER/INK/BRIGHT/FLASH**). These are taken to mean actual colour numbers from the ULANext extended palette.

Timex Hi-Res colour scheme

The colour scheme for hi-res mode is selected using **INK** or **PAPER** (either as a direct command or by **PRINT**ing to the hi-res screen or a window). This will immediately change the whole colour scheme. The colour schemes available (can be altered using ULANext palettes) are:

INK 0 (or PAPER 7)	black on white
INK 1 (or PAPER 6)	blue on yellow
INK 2 (or PAPER 5)	red on cyan
INK 3 (or PAPER 4)	magenta on green
INK 4 (or PAPER 3)	green on magenta
INK 5 (or PAPER 2)	cyan on red
INK 6 (or PAPER 1)	yellow on blue
INK 7 (or PAPER 0)	white on black

Tiling commands

For layer 2 and lo-res modes, there are new commands available to draw complete screens (or sections of a screen) from a set of tiles and a tilemap.

Tiles are either 8x8 pixels in size or 16x16 pixels in size. This allows a 16K bank to hold 256 8x8 tiles or 64 16x16 tiles. Tiles are numbered 0..255. Therefore, a complete set of 8x8 tiles occupies a single 16K bank, and a complete set of 16x16 tiles occupies 4 16K banks. If you use 16x16 tiles, you can restrict the tile numbers used and therefore reduce the memory requirements (eg if you need 64 or fewer different tiles, only 1 16K bank is required).

A tilemap is a linear map of 8-bit tile numbers. The user can specify any width up to 2048 tiles; each row of tiles follows directly after the previous one. The tilemap must be fully contained in a single 16K bank. This gives a maximum tilemap size of 256x64, 128x128, 2048x8 etc.

Any pixels in a tile which are the same colour as the current global transparency colour (which defaults to **227**) will not be written to the screen. If you want to draw pixels containing the global transparency colour you can temporarily change it to another colour (not used in your tiles) using the **PALETTE OVER** command before using **TILE**. Alternatively, you can use the **LAYER ERASE** command to clear regions of the screen to the global transparency colour before drawing tiles on top.

Information on layer 2 and lo-res tilemaps is stored separately, so you can use both. The **TILE** commands affect the currently selected layer/mode. They are:

TILE BANK *n*
Define bank *n* as containing the tiles (up to 4 banks *n*..*n*+3 if 16x16 tiles)

TILE DIM *n*,*offset*,*w*,*tilesize*
Define bank *n* as containing the tilemap, starting at offset *offset* in the bank. The tilemap is width *w* (1-2048) and uses 8x8 (*tilesize*=8) or 16x16 (*tilesize*=16) tiles.

TILE
TILE AT *x*,*y*
Draw entire screen from tilemap, from tile offset *x*,*y* in the tilemap (0,0 if not specified).

TILE *w*,*h*
TILE *w*,*h* **AT** *x*,*y*
TILE *w*,*h* **TO** *x2*,*y2*
TILE *w*,*h* **AT** *x*,*y* **TO** *x2*,*y2*
Draw section of screen from tilemap.
Number of tiles to draw is width *w*, height *h*.
Draw from tile offset *x*,*y* in the tilemap (or 0,0 if not specified).
Draw to tile offset *x2*,*y2* on the screen (or 0,0 if not specified).

Text window changes

There are some changes to the text window channels from those used in the +3e.

As noted earlier, there are 4 system-maintained full-screen windows which are used for all **PRINT**ing through the standard "s" channel when one of the layer 1 modes is selected, and most of the changes were made to accommodate this.

Windows can only be used in the same layer/mode that was active when they were defined. Control codes not listed here behave in exactly the same was as on +3e v1.43. The full list of original +3e window control codes is shown here: http://www.worldofspectrum.org/zxplus3e/channels.html

Control code	Differences
0	On user-defined windows, turns justification off (as +3e) On system windows, increases the current character set width (can range from 3 to 8 pixels), and moves the cursor to the start of the next line.
1	On user-defined windows, turns justification on (as +3e) On system windows, decreases the current character set width (can range from 3 to 8 pixels), and moves the cursor to the start of the next line.
2	On user-defined windows, saves window contents (as +3e) On system windows, causes the size 8 character set to be replaced with the character set defined by the CHARS system variable.
3	On user-defined windows, restores window contents (as +3e) On system windows, causes the size 3..7 character sets to be regenerated
15	Wash window. This does nothing on layer 2 or lo-res windows.
18,n	FLASH n. Ignored unless in standard or Timex hi-colour modes, and ULANext is not enabled.
19,n	BRIGHT n. Ignored unless in standard or Timex hi-colour modes, and ULANext is not enabled.
24,n	ATTR n. Ignored in lo-res, layer2 and Timex hi-res modes.
25,n	Kern adjust. This moves the position within the window left by *n* pixels, and can be used for primitive kerning. Previously this control code turned on or off extended UDGs for codes 165-255 instead of keyword tokens. Under *NextOS* extended UDGs are always used (LIST will expand keywords so keyword token codes will not normally be seen by windows anyway)
26,n	Auto-pause every n character lines. After each n character lines have been scrolled out of the window, output will automatically pause until the SPACE key is pressed (the bottom right character in the window will be flashed to indicate SPACE is being waited for). After a window has been cleared, the first pause occurs before any lines have been scrolled out; subsequent pauses wait for n character lines. Typically you would want to set "n" to the height of the window. If set to zero (the default), auto-pause is disabled.

30,n — On user-defined windows, selects justification mode 0, 1 or 2 (as +3e).
On system windows, changes the current character set width to *n* (can be 3,4,5,6,7 or 8 pixels), and moves the cursor to the start of the next line.

31,n — On user-defined windows, selects whether embedded codes are permitted in justify mode (as +3e).
On system windows, causes the size *n* character set to be replaced with the character set defined by the CHARS system variable.

User character sets

If the default character set(s) are replaced using control codes 2, 3 or 31 in a system window, any subsequent text printed in any window (which doesn't have its own user-defined character set) will use the new character set(s).

The system-defined character sets are partially shared: sizes 3 and 4 use the same set (only the leftmost 3 pixels are used for size 3), and similarly so do sizes 5 and 6. This should be borne in mind when replacing system character sets using control code 31.

Window input

Text windows now support the **INPUT** command, as in *ResiDOS*. If you use **INPUT #**, then a cursor is added to the window at the current position. The user can then input any text desired, using the left and right arrows to move along the text input so far, or the up and down arrows to move to the start or end of the text. The DELETE key deletes the character to the left of the cursor, and the ENTER key completes the input. Up to 191 characters can be accepted into each input variable.

INPUT # may also now be used with other channels such as file, memory and variable channels. In these cases it is advisable to avoid any accidental outputs to the channels, by not using any prompt strings, and by using only the semicolon as a separator. In most cases you will want to input a string using the **LINE** modifier; without this, the data in the file (or other channel) would need to be surrounded with quotes.

Window definitions

Windows are still defined using character squares as before. In lo-res mode, this means the maximum window size is 16x12 (not 32x24). In hi-res mode, "character squares" are considered to be 16 pixels wide, so the maximum window size is still 32x24 for this mode.

Memory constraints

It should be noted that saving/loading window contents (only available on user-defined windows) is a costly operation. The amount of memory required for each character square is:

- 9 bytes (standard resolution mode)
- 16 bytes (Timex hi-res or hi-colour modes)
- 64 bytes (lo-res or layer2 modes)

For example, a 10x10 window in layer2 would require 6400 bytes of available memory for saving the contents.

BASIC Program Extensions

It is now possible to write BASIC programs larger than the usual ~41K with a little extra effort. Sections of BASIC programs can be copied into any memory bank available to the user (and saved/loaded with the **SAVE/LOAD..BANK** commands), and the program can then switch between lines in the "main" program area and a bank.

The following new commands are available to manage banked sections of BASIC programs:

BANK *n* **LINE** *x*,*y*
Copies lines *x* to *y* inclusive from the main program to bank *n*. The total number of bytes used in the bank will be shown.
Once this has been done it is not possible to change or delete any lines in the banked section (except by completely overwriting the bank's contents using another **BANK...LINE** command).

BANK *n* **LIST**
BANK *n* **LIST** *l*
List lines (optionally from *l*) in bank *n*

BANK *n* **MERGE**
Copy all lines back from bank *n* into the main program

BANK *n* **GOTO** *l*
GOTO line *l* in bank *n*. To GOTO the main program from a banked section, use *n*=-1.

BANK *n* **GOSUB** *l*
GOSUB line *l* in bank *n*. To GOSUB the main program from a banked section, use *n*=-1.

BANK *n* **RESTORE** *l*
Set the DATA pointer to line *l* in bank *n*

Renumbering and erasing BASIC programs

The following additional commands are provided:

LINE *start*,*step*
Renumbers the BASIC program with new starting line number *start* and incrementing numbers by *step*.

ERASE *first*,*last*
Erases a specified section of BASIC program (from *first* to *last* line, inclusive).

ERASE
Erases the entire BASIC program (but leaves variables intact), and finish with an "OK" report.

Notes

Any **GOTO** or **GOSUB** within a banked section will go to a line in the same bank.

Any **RETURN** will always return to the calling bank.

DEF FN statements must be in the main program; they will not be searched for in banked sections.

Lines in banks can have the same numbers as main program lines.

Renumbers won't affect or take into account lines in banked sections.

Commands that affect program lines can only be used as direct commands, and not be part of a program. These are:
ERASE *first*,*last*
LINE *start*,*step*
BANK *n* **LINE** *x*,*y*
BANK *n* **MERGE**

The exception is the new command:
ERASE
which erases all lines in the BASIC program (but leaves the variables intact) and finishes with an "OK" report. This is a useful command to have as the last line of your C:/NEXTOS/AUTOEXEC.BAS file.

Sound support

The **PLAY** command has been augmented to support the Next's turbosound features (3xAY chips).

9 strings are now allows:

- strings 1,2,3 correspond to channels A,B,C of AY 1
- strings 4,5,6 correspond to channels A,B,C of AY 2
- strings 7,8,9 correspond to channels A,B,C of AY 3

The **W** (waveform) parameter affects only the AY chip for the string in which it appears, so each chip may have a different value at once.

If more than 3 strings are provided, the default volume is reduced to 13 in order to prevent clipping when many channels are playing at once. If 3 or fewer strings are provided, the default volume is 15 (as before).

3 new parameters are provided, which affect only the AY chip for the string in which they appear:

L
Restrict audio output for this AY chip to the left speaker only

R
Restrict audio output for this AY chip to the right speaker only

S
Allow audio output for this AY chip to go to both left and right speakers again

If the Next is set up with ABC stereo (which is the default), normally channel A goes to the left speaker, B goes to left and right, and C goes to right.

Therefore if the **L** parameter is used, only channels A and B from the current AY chip will be audible. Similarly, if **R** is used, only channels B and C will be audible.

BASIC program flow structures

The **IF** command now supports **ELSE**, which precedes a list of commands to be executed if the test was false. The **ELSE** must be on the same line as the **IF**, and preceded by a colon.
IF...THEN...ELSE statements may be nested. For example:

```
10 IF x=0 THEN PRINT "null":BEEP 1,0:ELSE IF x=1 THEN PRINT "one":BEEP 1,1:ELSE
PRINT "x was ";x
```

Note that this is not "true" nesting since there is no marker to indicate the end of an **IF**. When any **IF** condition fails, execution skips to the code following the next **ELSE** statement within the same line.

A new looping structure is also available. The start of the loop is marked with **REPEAT** and the end with **REPEAT UNTIL** *condition*. This will keep repeating the loop until *condition* is true (use **REPEAT...REPEAT UNTIL 0** for an infinite loop).

Optionally, any number of **WHILE** statements may be present within the loop, taking the form: **WHILE** *condition*. If the condition is false the loop is terminated immediately and execution continues after the matching **REPEAT UNTIL** statement.

These features allow you to construct loops with conditions at the top or the bottom of the loop, or at points in between (or any combination of these).

REPEAT loops may be nested to any depth.

Examples:

```
10 LET address=32768
20 REPEAT
30   READ b
40 WHILE b>=0
50   POKE address,b
60   LET address=address+1
70 REPEAT UNTIL 0
80 DATA 62,25,1,112,17,201,-1
```

```
10 REPEAT
20   INPUT "Enter a number (-1 to end): ";n
30   PRINT n
40 REPEAT UNTIL n=-1
```

```
10 LET y=0
20 REPEAT : WHILE y<22
30   PRINT AT y,0;"This is line ";y
40 REPEAT UNTIL 0
```

```
10 REPEAT
20   INPUT "Stock item: ";x$
30 WHILE x$<>""
40   INPUT "Quantity: ";n
50   PRINT x$;" : ";n
60 REPEAT UNTIL n$="END"
```

Procedures and local variables

Named procedures can now be defined, using the following command:

DEFPROC *procedurename*(*paramlist*)

where *procedurename* follows the same rules as standard numeric variables (must start with a letter, and contain only letters, numbers and spaces; when matching names, the case of letters is unimportant and spaces are ignored)

and *paramlist* is an optional list of up to 8 variable names (simple strings, numeric variables or integer variables, but not arrays of any type). These names can be used within the procedure to reference the values that are passed in by the **PROC** command.

The execution of a procedure is terminated by the following command:

ENDPROC

It is possible to have more than one **ENDPROC** in a procedure (for example an early exit can be made with a command such as **IF** *condition* **THEN ENDPROC**).

A procedure is called with the following command:

PROC *procedurename*(*expressionlist*)

The number of expressions and each of their types must match those defined in the **DEFPROC**, otherwise a *Q Parameter Error* report will be generated. (Note that you can provide an integer expression for a numeric parameter or vice versa, but strings/numbers must be correctly matched).

Optionally, a procedure can return up to 8 results to the caller, with the following command variants:

ENDPROC = *expressionlist*

PROC *procedurename*(*expressionlist*) **TO** *paramlist*

Again, the types of expressions in the **ENDPROC** must match the types of the variables in the **PROC**'s *paramlist*. It is acceptable for the **ENDPROC** to provide more results than the **PROC** requires (or even for the **PROC** not to have a *paramlist* at all): the unneeded values will just be discarded.

Within a procedure (or within a subroutine called by **GOSUB**) it is possible to create local variables, which can be used within the procedure/subroutine without affecting any existing variables with the same name (the original values will be restored at the **ENDPROC** or **RETURN** command):

LOCAL *variablelist*

As with the *paramlist* in a **DEFPROC** (which is itself a set of local variable names for the procedure, initialised with the values from the **PROC** command), only simple string, numeric and integer variables can be made local; not arrays.

Each **LOCAL** statement can contain up to 256 variable names, and multiple **LOCAL** statements may be present in a subroutine or procedure; the only limit on the number of local variables that can be created is available memory.

Local variables are initialised to zero (or the empty string) by the **LOCAL**

command.

Procedures may also be recursive. Here is a simple example:

```
  10 INPUT “Enter a number 0+:”;x
  20 PROC factorial(x) TO f
  30 PRINT “The factorial of “;x;” is “;f
  40 GOTO 10
 999 STOP
1000 DEFPROC factorial(n)
1010 IF n<0 OR n<>INT n THEN PRINT “Factorial only possible for non-negative
integers”:STOP
1020 IF n<=1 THEN ENDPROC=1
1030 LOCAL partial
1040 PROC factorial(n-1) TO partial
1050 ENDPROC=n*partial
```

Error-trapping

Any error (except "*0 OK*" which is not considered an error) can be trapped by the **ON ERROR** command, allowing BASIC to recover from expected error conditions.

NOTE: Errors generated by dot commands are *not* trapped.

To turn on error trapping, use the command:

ON ERROR *statementlist*

This will cause the statements after the **ON ERROR** command to be executed whenever an error report would normally have been displayed. Note that this command *must* be part of a program and cannot be entered as a direct command.

To turn off error-trapping again, just use:

ON ERROR

This is required if you wish to generate errors again. eg the following will display "There was an error!" and terminate with the *9 STOP statement* error when line 20 is executed:

```
10 ON ERROR PRINT "There was an error!":ON ERROR:STOP
20 PRINT 5/0
```

To generate the last error that actually occured (this does not need error trapping to be turned off), use the command:

ERROR

eg this will print the message but still give the correct *Number too big* report.

```
10 ON ERROR PRINT "There was an error!":ERROR
20 PRINT 5/0
```

You can also obtain details of the last error using the following command:

ERROR TO *codevar,linevar,statementvar,bankvar*

This will store the error code in the numeric variable *codevar*, the line number in *linevar*, the statement number in *statementvar* and the bank number in *bankvar*. Note that you do not need to supply later variable names if you do not need the information, eg all of these are valid:

```
ERROR TO e
ERROR TO e,l
ERROR TO e,l,s
ERROR TO e,l,s,b
```

Localised error-trapping

As well as (or instead of) having a global error-trapping routine for your program, each procedure, subroutine and repeat loop may have its own local error-trapping routine, simply by using the **ON ERROR** command within it.

When an error occurs within a repeat loop, subroutine or procedure, it will be trapped by its own **ON ERROR** routine if there is one. If not, the error will be passed out to the next level and trapped by any **ON ERROR** routine there and so on. Only if there is no **ON ERROR** at any level above the command that caused the error will a normal error report be generated.

For example:

```
 10 ON ERROR PRINT "Outer error handler!":ERROR
 20 REPEAT
 30   PRINT "Starting..."
 40   ON ERROR PRINT "Oops!":ON ERROR:STOP
 50   GO SUB 100
 60   PRINT "Iterating..."
 70   ON ERROR
 80 REPEAT UNTIL 0
 90 STOP
100 ON ERROR PRINT "Bad pigs!":RETURN
110 PROC myproc()
120 PRINT "Pigs:";pigs
130 RETURN
200 DEFPROC myproc()
210   LOCAL m
220   ON ERROR PRINT "Myproc died...":ENDPROC
230   PRINT "m=";m,"n=";n
240 ENDPROC
```

Note that in **REPEAT** loops it is important to turn off any local error handling for that loop before the **REPEAT UNTIL** is executed. If not, the loop start cannot be found and a *Loop error* would result (and be trapped by the loop's own error handler). Removing line 70 in the example above would demonstrate this.

Also note that any **LOCAL** commands in a procedure or subroutine must come before a local error handler (ie lines 210 and 220 in the example cannot be reversed).

Integer variables and expressions

For additional speed and memory efficiency, NextBASIC provides a new integer expression evaluator. All integer values are treated as unsigned 16-bit values, and all operations are performed modulo 65535, with no checks for overflow/underflow (except division by zero, which results in error 6, Number too big).

A fixed set of integer variables are provided: the user cannot define additional variables. The two main advantages of a fixed set of variables are:

- speed of access (all integer variables are at a known location)
- memory usage - the integer variables are stored in some of the RAM 8 bank reserved by NextOS, and hence do not use any space in the normal BASIC/variables area

All integer variables are erased to zero at **RUN**, **CLEAR** and **NEW**. Note, however, that integer variables are not saved/loaded along with BASIC programs, as is the case with normal floating-point and string variables.

There are 26 integer variables provided, named **A** to **Z** (can also be referred to in lower-case, **a** to **z**). There are also 26 integer variable arrays provided, named **A()** to **Z()** (or **a()** to **z()**), each containing 64 elements, numbered 0 to 63.

Note that array elements are numbered from 0, not 1 as in normal floating-point/string arrays. Also note that integer array element **A(0)** is *not* the same as integer variable **A**.

The following unary operator may precede any integer value or expression:

!	bitwise not

Literal numbers can be specified in decimal (the default), hexadecimal (preceded by the **$** symbol) or binary (preceded by the **@** symbol), eg:

```
32767
$ed01
@11100010
$FF
```

The following binary operators are available:

+	add
-	subtract
*****	multiply
/	divide
MOD	modulus (remainder)
<<	shift left
>>	shift right
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

The six relational operators always produce a result of $0000 for false and $ffff for true.

Operations are performed in strictly left-to-right order, unless overridden by the use of parentheses.

An integer expression can be used in any BASIC line where a numeric expression is normally expected. To indicate an integer expression instead of a floating point expression, a **%** symbol must always precede an integer expression.

Similarly, integer variables can be used in assignments (such as **LET**, **INPUT**, **READ**, **FOR**) by preceding their name with a %.

It is *not* possible to access standard numeric variables or functions within an integer expression, or to access integer variables or operations within a standard numeric expression.

It *is* possible to assign an integer expression to a standard normal numeric variable, or vice-versa, and the value will be converted appropriately. For example, all the following assignments are valid:

LET %A=2*PI*radius
assigns truncated floating point calculation to integer variable A

LET %B=%B+(A(7)<<3)
shifts integer array element A(7) left 3 bits and adds to integer variable B

LET addr=%x(1)<<8+x(0)
calculates standard numeric variable *addr* from low and high bytes in integer array X elements 0 and 1

Note that **DEF FN** does not support user-defined integer functions.

FOR..NEXT loops may be used with integer variables as the index, eg:

```
10 FOR %i=%$c9 TO 220
20 PRINT %i
30 NEXT %i
```

However, they can only be used as part of a program, and not on a direct command. Any attempt to do this will result in a "Direct command error". This restriction allows integer loops to run much faster than loops using a standard floating point index variable, especially when loops are used towards the end of long programs. Integer **FOR..NEXT** loops run at the same speed regardless of where they are located in the BASIC program, but standard **FOR..NEXT** loops become progressively slower the later they are located in the program.

Installable device drivers

NextOS allows for a number of device drivers to be installed/uninstalled at will using the .install/.uninstall dot commands (currently a maximum of 4 drivers may be installed at any one time but this could change in the future).

These are mainly intended for use as drivers for external peripherals such as printers, mice, network devices etc, but could be used for other purposes.

To install or uninstall a driver, use the following dot commands:

.install drivername.**drv**
.uninstall drivername.**drv**

The documentation that comes with the driver describes how to use it. Some drivers may make use of the new **DRIVER** command. This has the following form:

DRIVER *driverid*,*callid*[,*n1*[,*n2*]] [**TO** *var1*[,*var2*[,*var3*]]]

where *n1* and *n2* are optional values to pass to the driver, and *var1*, *var2* and *var3* are optional variables to receive results from the driver call. The documentation for each driver will describe the individual **DRIVER** commands that you can use.

Channel support

Some drivers can support input/output via the streams and channels system of the Spectrum Next. If so, the documentation will describe how to open a channel, using one of the following command variants (assuming the driver id is ASCII 'X'):

OPEN #*n*,"**D>***X*"
open stream n to simple channel for device 'X'

OPEN #*n*,"**D>***X***>***string*"
open stream n to channel described by *string* on device 'X'

OPEN #n,"**D>***X*,*p1*"
open stream n to channel described by numeric value *p1* on device 'X'

OPEN #n,"**D>***X*,*p1*,*p2*"
open stream n to channel described by numeric values *p1* and *p2* on device 'X'

CLOSE #*n*
close stream n

Once a channel is open, you can use any of *NextBASIC*'s stream input, output or pointer manipulation commands (some drivers may not support all of these; the documentation should describe what can be used). eg:

PRINT #*n;....*
INPUT #*n;....*
INKEY$ #*n*
RETURN #*n,var* (get current stream pointer to variable *var*)
DIM #*n,var* (get current stream size/extent to variable *var*)
GOTO #*n,value* (set current stream pointer)
NEXT #*n,var* (wait for next input character from stream and store in *var*)

System variable changes

The following system variables have been changed (same format as +3 manual):

1	5B5FH (23391)	INKL	INK colour for lo-res mode (was BAUD)
1	5B60H (23392)	INK2	INK colour for layer2 mode (was BAUD+1)
1	5B61H (23393)	ATTRULA	Attributes for standard mode (was SERFL)
1	5B62H (23394)	ATTRHR	Attributes for hi-res mode (only paper colour in bits 3..5 is used) (was SERFL+1)
1	5B63H (23395)	ATTRHC	Attributes for hi-colour mode (was COL)
1	5B64H (23396)	INKMASK	Softcopy of ULANext inkmask(or 0)(was WIDTH)
X1	5B68H (23400)	FLAGN	Flags for the NextOS system (was XLOC)
1	5B69H (23401)	MAXBNK	Maximum available RAM bank (was YLOC)
1	5B73H (23411)	TILEBNKL	Tiles bank for lo-res (was RC LINE)
1	5B74H (23412)	TILEML	Tilemap bank for lo-res (was RC LINE+1)
1	5B75H (23413)	TILEBNK2	Tiles bank for layer2 (was RC START)
1	5B76H (23414)	TILEM2	Tilemap bank for layer2 (was RC START+1)
X1	5B77H (23415)	NXTBNK	Bank containing NXTLIN (was RC STEP)
X1	5B78H (23416)	DATABNK	Bank containing DATADD (was RC STEP+1)
N1	5B7BH (23419)	L2SOFT	Softcopy of layer2 port (was DUMPLF)
X1	5C7FH (23679)	GMODE	Graphical layer/mode flags (was P POSN)
1	5C81H (23681)	STIMEOUT	Screensaver control (was unused)

The following system variables have been inserted where STRIP1 and STRIP2 were, within the temporary TSTACK area. This means that there are now only a guaranteed 117 bytes of TSTACK when calling +3DOS:

2	5B7CH (23420)	TILEWL	Width of lo-res tilemap
2	5B7EH (23422)	TILEW2	Width of layer2 tilemap
2	5B80H (23424)	TILEOFFL	Offset in bank for lo-res tilemap
2	5B82H (23426)	TILEOFF2	Offset in bank for layer2 tilemap
2	5B84H (23428)	COORDSX	x coord of last point plotted (layer 1/2)
2	5B86H (23430)	COORDSY	y coord of last point plotted (layer 1/2)
1	5B88H (23432)	PAPERL	PAPER colour for lo-res mode
1	5B89H (23433)	PAPER2	PAPER colour for layer2 mode
Nx	5B8AH (23434)	TMPVARS	Base of temporary system variables (space shared with bottom of TSTACK)

List of updates

Updates: 19 Feb 2018

Noted that **PLOT/DRAW/CIRCLE** in layer 1/2 modes may be drawn so that they are partly off-screen without generating "out-of-screen" errors.

Updated system variables (COORDSX, COORDSY, PAPERL, PAPER2 replacing some previously-described variables).

Added section describing enhancements to the **PLAY** command.

Updates: 12 Feb 2018

Added new procedures support (**DEFPROC**, **ENDPROC**, **PROC**, **LOCAL** commands).

Added new error-trapping support (**ON ERROR**, **ERROR**, **ERROR TO** commands).

Updated keywords list.

Updated error message list.

Changed **POINT x,y,var** command syntax to **POINT x,y TO var**.

Changed **BANK n PEEK offset,var** command syntax to **BANK n PEEK offset TO var**.

Updates: 6 Feb 2018

Replaced modulus operator % with **MOD** (new token code $8d).

Removed unary + and - operators from the integer expression evaluator.

Updates: 28 Jan 2018

Added new section on installable device drivers, with new **DRIVER** command.

Moved the new system variables INKL, INK2, ATTRULA, ATTRHR, ATTRHC, INKMASK to newly-freed system variables (previously used for printer).

Updates: 17 Jan 2018

Copied descriptions of commands in earlier updates into the main text.

Clarified that layer 2 banks cannot be released by the **BANK CLEAR** command.

Added further notes and examples for the new structured programming commands.

Added token codes for **REPEAT**, **WHILE** and **UNTIL**.

Updates: 15 Jan 2018

Clarified that all commands accessing banks mark them as "owned" by BASIC (except **BANK CLEAR** which releases them).

Clarified that **LAYER BANK** may be executed in any mode, but always applies to layer 2.

Added new **ERASE** command which erases all lines of a BASIC program. It may be used within a program, so is suitable as the last line of a C:/NEXTOS/AUTOEXEC.BAS file.

Updated description of **LINE** command, which now just allows the whole program to be renumbered (but with any starting number and step).

Added new **REPEAT...[WHILE]...REPEAT UNTIL** looping structure (see main text for full description).

Updates: 23 Dec 2017

Added new "Loop error" error and reworded "Direct command only" to "Direct command error".

Clarified behaviour of integer FOR/NEXT loops.

Added new **ELSE** command and token.

Updates: 12 Dec 2017

A new command (now in main text of the editor changes document) has been added: **SPECTRUM SCREEN$** *n,t*

Updated system variables with new **STIMEOUT** system variable.

Updates: 30 Nov 2017

The **BANK** command can now use banks 5,2,0 (the standard 48K memory) without restriction.

Added new **BANK...LAYER** command.

Noted that transparent pixels are not drawn by the **TILE** command.

Updates: 23 Nov 2017

Updated the notes on **INPUT #** which can now be used with other channels (file, memory, variable) as well as windows and the standard "K" channel.

The **REMOUNT** command should be entered when the user wishes to change the SD card. When the prompt "Remove/insert SD and press Y" is shown the SD card may be changed, and then the **Y** key should be pressed.

Updates: 14 Nov 2017

The auto-pause window control code (26,n) is changed: "n" is the number of character lines to be scrolled between pauses, not pixel lines. Also the bottom right-hand character square is now flashed rather than inverted to indicate when the window is paused and waiting for SPACE to be pressed.

By default, scrolling auto-pause is turned on for the layer 1/2 mode full-screen windows, so after a screen full of text has been printed the user must press SPACE to continue. This behaviour can be disabled using control code 26, as with other windows.

Updates: 6 Nov 2017

Removed "bright magenta" from description of the transparency colour (227) since the default value for ULA's bright magenta has been changed to 231 and so it no longer acts as a transparency.

Note that the same clip window (as specified by the **LAYER DIM** command) is shared between layer 0 and all layer 1 modes. Layer 2 has its own clip window (as do the sprites, this being specified by the **SPRITE DIM** command).

Added new command **REMOUNT** (token code $99, with tokens **<<** and **>>** now moved to $97 and $98 respectively). The **REMOUNT** command (no parameters) re-initialises the filesystem following an SD card change.

The following new command (now in the main text) will be added:
BANK NEW var

Behaviour of **NEW** in relation to banks has now changed: a **NEW** now does not mark banks reserved by BASIC as free again; this only happens at a reset.

Updates: 21 Oct 2017
STRIP1/STRIP2 removed from system variables and replaced with TMPVARS.
INKHR system variable replaced with ATTRHR.

Clarified that the extended ULANext colour ranges are only allowed to be specified with **INK/PAPER** in layer 1 modes (mode 1 - standard, or mode 3 - Timex hi-colour). In layer 0 only the standard colour ranges 0..7 can be specified for **INK/PAPER** (although any desired ULANext colour scheme can be selected for use in layer 0 by POKEing the calculated attribute value into the system variable ATTR_P).

Clarified that hi-res colour schemes can be chosen using either **INK** or **PAPER**.

Clarified that **FLASH**, **BRIGHT** and **ATTR** commands and window control codes are ignored unless used in standard or Timex hi-colour modes (with **FLASH** and **BRIGHT** always ignored if ULANext colours are enabled).

Updates: 10 Oct 2017
There will no longer be a restriction on the address for user-defined character sets.

Clarified that changing character set size also causes the window print position to be moved to the start of the next line.

Clarified that window save/load is costly in terms of memory.

Clarified that all commands using the standard "s" channel (not just **PRINT**) will operate in the currently-selected layer/mode.

Window control code 26 is now "auto-pause" instead of "fill with byte".

Added new integer expressions section.

Added new token codes for >> and <<.

The following new command (now in the main text) will be added:
BANK *n* **CLEAR**

Updates: 5 Oct 2017
Updated text to clarify some details of how different attributes are handled in different modes.

The following new commands (now in the main text) will be added:
LAYER DIM *x1*,*y1*,*x2*,*y2*
SPRITE DIM *x1*,*y1*,*x2*,*y2*
POINT *x*,*y*,*var*