NextBASIC New Commands and Features (Updated 9 Mar 2024)

This document describes new commands and features for NextBASIC (except file-related commands and editor features, described in separate documents) as at NextZXOS v2.09.

This should be read in conjunction with the other documents:

- NextBASIC file-related commands and features
- NextZXOS Editor features
- NextZXOS and esxDOS APIs

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

Variables

Any variable (strings, arrays, loop indexes) can now have a long name. This does not apply to integer variables which are still pre-allocated single-letter names only.

Assignments

LET is now optional and may be omitted.

Multiple assignments

Assignments now support multiple destinations. This could be used to swap the contents of 2 variables without using an intermediate, or simply to assign multiple variables in the same statement.

```
eg:   LET x,y = y,x
      a,b,c,d$,e$,f = 1,2,3,"xyz","zzz",g*h
```

If there are more destinations than source expressions, the final source expression is assigned to all the extra destinations. This makes it easy to assign the same value to multiple variables in one go.

```
eg:   a,b,c,d = 0
```

Accumulation assignments

Assignments may also be used using any of the binary arithmetic operators. This performs the arithmetic operation between the destination and the source.

```
eg:   a += 1
      LET x$ += "..."
      x -= dx
      y$ *= 2
```

The full list of accumulation assignments is as follows.

For numeric destinations (all requiring a numeric source):

```
      +=
      -=
      *=
      /=
      ^=
      &=
      |=
      ^|=
      <<=
      >>=
      MOD=
```

For string destinations:

```
      +=                    string concatenation: takes a string source
      *=                    string replication: takes a numeric source
```

Accumulation assignments may be performed with multiple destinations as well.

```
eg:   x,y,z >>= 1
      a$,b$ += c$,d$
```

Labels

Labels have now been added. These start with @ and follow the same naming conventions as procedures. They may occur anywhere in the program. **GOTO, GOSUB, RESTORE, LIST, BANK..LIST, SAVE..LINE** and **EXIT** all support label targets, eg:

```
100 RETURN: @alice: PRINT “At @alice”
GOTO @bob
GOSUB @cyril
LIST @david$
BANK 20 LIST @emma
EXIT @fred
SAVE “program” LINE @gerald
```

NOTE: For **SAVE..LINE @label**, the saved program will autostart at the start of the line containing the label, even if the label is not at the start.

Conditional flow structures

Short-form **IF..THEN..ELSE**

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.

Long-form **IF..ELSE..ENDIF**

A long-form **IF**..**ELSE**..**ENDIF** is now provided. This allows properly-nested conditions, unlike the single-line **IF..THEN..ELSE**.

The format is as follows. **IF**, **ELSE** and **ENDIF** must each be the first statement of a line. No **THEN** statement is used in the **IF** (this is what determines whether it is a short-form or long-form **IF**). **ELSE** and **ELSE IF** clauses are optional; you can have as many **ELSE IF**s as you want.

```
100 IF x>7:PRINT “x>7”
105     IF x>1000:PRINT “In fact it's huge”
110     ELSE PRINT “But not too big”
115     ENDIF
120 ELSE IF x>3:PRINT “x>3 but x<=7”
140 ELSE IF x=3:PRINT “x=3”
160 ELSE
170   PRINT “x is too small to bother with”
180 ENDIF
```

Selection structures

ON *n*:<*stmt0*>:<*stmt1*>:<*stmt2*>:...:<*stmtlast*>:**ELSE** <statements>

A new **ON** *n* command is provided. This rounds *n* to the nearest integer and executes the *n*th statement on the same line following the **ON** (starting with statement 0). After executing the single statement, the rest of the line is skipped and the following line is executed (unless the statement was a non-returning jump such as **GOTO, EXIT, RETURN** or **ENDPROC**).

If **ON** ran out of statements before *n* was matched (or *n* was negative) and the optional **ELSE** clause is present, all statements following the **ELSE** will be executed.

eg:

```
    100 ON x:PROC xwaszero():GOSUB @xwasone:GOTO @xwastwo:ELSE BEEP 1,0:PRINT “x was > 2”
```

REPEAT loops

A new **REPEAT** looping structure is 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 x$=”END”
```

Exiting from loops

A new **EXIT** command is now provided to cleanly exit from the current **FOR** or **REPEAT** loop. This is the recommended way to exit from loops - **GOTO** should not be used. **EXIT** on its own will jump to the first statement after the end of the loop; it can alternatively specify an optional line number (or label). eg:

```
100 REPEAT
110   INPUT n
120   IF n=33 THEN EXIT 150
130 REPEAT UNTIL n<0
140 PRINT “Loop completed normally”:STOP
150 PRINT “Loop ended early”:STOP
```

EXIT can be used to exit from multiple levels of nested loops. To do this, use one **EXIT** statement per level on the same line. The final **EXIT** (only) may include an optional line number/label. eg:

```
100 FOR i=1 TO 10
110   FOR j=i TO 10
120     PRINT i,j
130     IF j*i>80 THEN EXIT:EXIT 170
140   NEXT j
150 NEXT i
160 STOP
170 PRINT “Product exceeded 80”:STOP
```

A conditional **RETURN** or **ENDPROC** can also be used within loops to exit all currently-active loops within the current subroutine or procedure and return to the caller.

Procedures and local variables

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

DEFPROC *procedurename***(***paramlist***)**

procedurename follows the same rules as standard numeric or string 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). The optional trailing '$' has no effect on functionality.

paramlist is an optional list of parameter names (simple strings, numeric variables, numeric arrays, character arrays, integer variables or integer arrays).

Array parameters are indicated by directly following the array name with **()**. eg

```
110 DEFPROC x(inp(),size,y)
```

Default values can be provided for non-array parameters, by following the parameter name with **=** and an expression for the default value. The default value is assigned to the parameter if the **PROC** omits any value for the parameter in that position.

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). Any parameter which has a default value specified in the **DEFPROC** may be omitted, eg

```
100 PROC x(12,,”bob”):STOP
110 DEFPROC x(a=1,b$=”alice”,c$,d=-5)
120 LOCAL e=3
130 PRINT a,b$,c$,d,e
140 ENDPROC
```

will output: 12, alice, bob, -5 and 3.

Parameters can also be passed by reference. This is mainly intended for arrays, as the default of passing-by-value will be slower (and consume more memory), but it can also be useful for strings. It is not recommended for numeric variables, which are faster if passed by value. Any changes made to a value passed by reference will of course be seen by the caller. The **REF** keyword is used in the DEFPROC to make a parameter a reference; such parameters must be passed by the calling PROC as a variable/array name only. It is not permitted to have default values for **REF**erenced parameters. eg:

```
100 PROC tl$(x$(),5):STOP
110 DEFPROC tl$(REF inp$(),index=1)
120 input$(index)=inp$(index)(2 TO)
130 ENDPROC
```

Note: Integer variables and arrays can not be passed by reference.

Optionally, a procedure can return multiple numeric or string 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.

LOCAL variables

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), any type of variable can be made local. Initial values can be provided for numeric or simple string variables in the same way as default values in **DEFPROC** (zero or the empty string is used as an initial value if no default is provided in the **LOCAL** statement).

Each **LOCAL** statement can contain multiple variable names separated by commas, 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.

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
```

PROCs may call DEFPROCs with more parameters than the DEFPROC requires. In this case, if the DEFPROC has the keyword **DATA** as its final parameter, the procedure may read the additional parameters one at a time using **READ**. The new function **DATA** can be used to determine if there are further PROC parameters left to read. eg:

```
100 PROC printem(“Digits”,0,1,2,3,4,5,6,7,8,9):STOP
110 DEFPROC printem(name$, DATA)
115 LOCAL n
120 PRINT name$
130 REPEAT: WHILE DATA
140   READ n:PRINT n
150 REPEAT UNTIL 0
160 ENDPROC
```

PRIVATE variables

A new **PRIVATE** statement is provided for use by procedures and subroutines. This works in the same way as LOCAL except that the variable retains its last value on subsequent calls. The initial values of all **PRIVATE** variables can be reset (to zero, or the default value given in the **PRIVATE** statement) with the **PRIVATE CLEAR** command.

If the **PRIVATE CLEAR** command is executed from within banked code, the privates for that bank are reset; otherwise, the privates for the main program are reset.

Only numeric **PRIVATE** variables are supported.

eg.

```
100 PRIVATE CLEAR
110 FOR i=1 TO 10:PROC iterate():NEXT i:STOP
120 DEFPROC iterate()
130 PRIVATE callcount=0
140 callcount=callcount+1
150 PRINT “I have been called “;callcount;” times”
160 ENDPROC
```

User-defined functions

User-defined functions now work similarly to procedures, following the same naming conventions and parameter-passing.

```
eg: DEF FN gladys(harold)=harold+2
    DEF FN ian$(REF jenny$(),index)=jenny$(index)
```

User-defined functions can now be recursive.

```
eg: DEF FN factorial(n)=n?(1,1,n*FN factorial(n-1))
```

The program can also be listed starting at any user-defined function:

```
      LIST FN ian$()
      BANK 20 LIST FN lydia()
```

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.

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
```

NOTE: Any errors generated by a dot command are indicated with an error code of 255 (“Dot Command Error”).

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).

Standard expression evaluator

Many new functions and operators have been added to the standard (non-integer) expression evaluator.

SPRITE s
SPRITE CONTINUE s
SPRITE AT(s,c)
SPRITE OVER(s, start [**TO** end] [, overlapXY [, overlayY]])
Returns information about sprites (see sprites section for details).

REG n
Returns the current value of the NextReg register *n*

RND(n)
Returns a random integer between 0 and n-1 (like **%RND** n in the integer expression evaluator). This is recommended over using the standard fractional floating-point function **RND** since it doesn't suffer from the biasing inherent in converting a fractional random number to an integer with multiply and truncation steps.

DATA
Returns: 1 if the next DATA item (or PROC parameter to be read) is numeric
2 if the next DATA item (or PROC parameter to be read) is a string
0 if there are no further DATA items (or PROC parameters)

ERROR
Returns last error code

ERROR(*n*)
Returns:

last error code	(if n=0)
line number where last error occurred	(if n=1)
statement number where last error occurred	(if n=2)
bank number where last error occurred	(if n=3)

ERROR$
Returns last error message

TIME
Returns the current value of the system variable FRAMES. Intended for use in timing operations (see new **TIME** command later).

TIME$
Returns the current date and time (in the same format as shown on menus, with the addition of a seconds field). If there is no RTC, returns an empty string.
eg TIME$ returns “2020-10-10 18:41:03”

POINT #*n*
Returns the current stream pointer for stream *n*

NEXT #*n*
Returns the next character from stream *n* (waiting until the char is available)

DIM #*n*
Returns the extent (size) of stream *n*

DIM(*arrayname[$]***()** *[, dimension]***)**
Returns the number of elements in the specified dimension of the array (*dimension* defaults to 0 if not specified).

If *dimension*=0, instead returns the number of dimensions in the array.

Simple strings are treated as single-dimension character arrays, returning 1 as the number of dimensions and the current string length as the number of elements in dimension 1.

eg: For an array previously declared with DIM a(100,10,5):
```
      DIM(a())    gives 3
      DIM(a(),1)  gives 100
      DIM(a(),2)  gives 10
      DIM(a(),3)  gives 5
```

STR$(n, base [, places])
Returns a string representation of number *n* in the specified base (2-36). For bases > 10, digits larger than 9 are represented with capital letters. If *places* is present, then a fractional part of *places* digits is also output.
eg:
```
STR$(201.5, 16)   gives “C9”
STR$(3.5, 2, 4)   gives “11.1000”
```

IN(*source$, match$ [, startpos[, wild$]]***)**
Returns the character number where *match$* was found in *source$.*

Optional *startpos* parameter determines the position within *source$* to begin the search (default=1).

If *startpos* is negative, the search starts from position ABS(*startpos*) and proceeds backwards.

The value returned will be between 1 and LEN *source$* if a match was found. Returns 0 if match not found, *startpos*=0, *match$*=”” or *source$*=””.

Any characters in *match$* which are the wildcard character will match any character in *source$*.

The wildcard character is the first character of *wild$*, if present, or the copyright symbol (ASCII 127) otherwise. If *wild$*=””, the wildcard character is ASCII 0.

eg:
```
IN(“A very long string”,” “)       gives 2
IN(“A very long string”,” “,3)     gives 7
IN(“A very long string”,” “,-17)   gives 12
IN(“A very long string”,”s**ing”,1,”*”)  gives 13
```

BANK n **PEEK** addr
Returns value of byte in bank

DPEEK addr
BANK n **DPEEK** addr
Returns value of 16-bit word in memory or bank

PEEK$(*addr,len***)**
BANK *n* **PEEK$(***addr,len***)**
Returns memory region (or region of bank) as a string

PEEK$(*addr,~***)**
BANK *n* **PEEK$(***addr,~***)**
Returns a bit7-terminated string from memory or bank (NOTE: bit 7 will still be set on the string that is returned)

PEEK$(*addr,~code***)**
BANK *n* **PEEK$(***addr,~code***)**
Returns a string terminated with the character value *code* from memory or bank (NOTE: the string that is returned excludes the terminating character)

USR$ *addr*
BANK *n* **USR$** *offset*
Calls the machine code routine at *addr* (or *offset* in bank *n*). Instead of returning the 16-bit number found in BC (as with **USR** *addr*), it returns a string, defined by the start address returned by the machine-code routine in DE and length in BC.

USR(*addr, param1[, param2 [, param3...]]]***)**
USR$(*addr, param1[, param2 [, param3...]]]***)**
BANK *n* **USR(***addr, param1[, param2 [, param3...]]]***)**
BANK *n* **USR$(***addr, param1[, param2 [, param3...]]]***)**
As the single-argument **USR** functions, but multiple parameters can be passed to the machine-code routine (instead of just the start address in BC).

If a single additional parameter (*param1*) is present, this is passed in BC (if it is numeric) or as an address DE and length BC (if it is a string).

The type of the parameter passed to the routine is indicated by the zero flag: if set, the parameter is a string (in DE,BC); if clear, the parameter is a number in BC.

Additionally, the type of the expected result is indicated by the carry flag: if set, the expected result is a number (in BC); if clear, the expected result is a string (in DE,BC).

All further parameters are left on the calculator stack for the machine-code routine to use with calculator operations or retrieve using standard ROM routines such as FIND-INT1, FIND-INT2, STK-FETCH. On entry, A contains the number of additional parameters on the calculator stack (0-16) and HL contains a bitmask indicating the type of each parameter. Bit 15 indicates the type of the final parameter (and will be the first to be retrieved from the calculator stack), so types can be read by shifting each bit in turn to the carry flag with ADD HL,HL. Type bits are 0 for string, 1 for numeric. Routines must remove all additional parameters from the calculator stack, otherwise it will be unbalanced and the expression may be calculated incorrectly.

INPUT *n*
Reads the current state of an input controller.
If *n*=1 or 2, reads the state of joystick1 (left) or joystick2 (right).
If *n*=0, reads the state of the “keyboard joystick”.

In each case, the value returned is a bitmask of the following value:

```
      bit 0 (value 1):        set if right pressed
      bit 1 (value 2):        set if left  pressed
      bit 2 (value 4):        set if down  pressed
      bit 3 (value 8):        set if up    pressed
      bit 4 (value 16):       set if fire  pressed
      bit 5 (value 32):       set if fire2 pressed
      bit 6 (value 64):       set if fire3 pressed
      bit 7 (value 128):      set if fire4 pressed
```

eg

```
      INPUT 1&8               returns false (0) if up is not pressed
                              on joystick 1, true (8 is non-zero) if it is
      INPUT 0&BIN 11110000    returns false (0) if no fire buttons are pressed
                              on joystick 0, true (non-zero) if at least one is
```

The default “keyboard joystick” is:

```
      up          Q
      down        A
      left        O
      right       P
      fire        SPACE
      fire2       M
      fire3       ENTER
      fire4       X
```

The “keyboard joystick” may be redefined with the INPUT function by specifying negative values for *n*, as follows:

```
      INPUT -1          waits for a key to be pressed and assigns to “right”
      INPUT -2          waits for a key to be pressed and assigns to “left”
      INPUT -3          waits for a key to be pressed and assigns to “down”
      INPUT -4          waits for a key to be pressed and assigns to “up”
      INPUT -5          waits for a key to be pressed and assigns to “fire”
      INPUT -6          waits for a key to be pressed and assigns to “fire2”
      INPUT -7          waits for a key to be pressed and assigns to “fire3”
      INPUT -8          waits for a key to be pressed and assigns to “fire4”
      INPUT -9          (or any other value) clears all assignments
```

The return value is the character code of the key pressed, which can be useful if you want to display the key just defined (although some special keys have codes below ASCII 32 which aren't PRINTable, so care should be taken).

eg

```
      100 x=INPUT -9: REM clear the “keyboard joystick”
      110 PRINT “Press a key for right”:x=INPUT -1
      120 PRINT “Press a key for left”:x=INPUT -2
      130 PRINT “Press a key for down”:x=INPUT -3
      140 PRINT “Press a key for up”:x=INPUT -4
      150 PRINT “Press a key for fire”:x=INPUT -5
      160 REM Can leave additional fire buttons undefined if they aren't needed
```

New operators

The standard expression evaluator also contains a number of new operators.

@n	binary number n (same as **BIN**)
$n	hex number n
!n	bitwise not
n **>>** m	shift right
n **<<** m	shift left
n **|** m	bitwise or
n **&** m	bitwise and
n **^|** m	bitwise xor
n **MOD** m	modulus (remainder)

These operators are all the same as the ones in the integer expression evaluator. However (apart from **MOD**, which always returns an integer), they all work on the fractional parts of numbers as well as the integer parts. Hex and binary numbers can also be written with fractional parts.
eg:

1>>1	gives 0.5
BIN 1.1	is the same as 1.5 (decimal)
@10.01	is the same as 2.25 (decimal)
$64.c	is the same as 100.75 (decimal)

Note that the integer expression evaluator uses ^ for xor, but this operator is already used in the standard expression evaluator for “to-the-power”. Hence **^|** is used instead (**^|** can also be used in the integer expression evaluator).

n**?(**expr0,expr1,expr2...)
The select (**?**) operator rounds n to the nearest integer and uses it to choose one of the following expressions. If *n*=0, *expr0* is evaluated etc.
If there are not enough expressions in the list to select based upon *n*, the last expression is always evaluated.
The expressions in the list may be numeric or string expressions, but they must all be the same type.
eg:

3?(“alice”,”bob”,x$,”denzil”,y$+z$)	gives “denzil”
5?(f/g,PI*2*r,200)	gives 200

a$*n
The multiply (*) operator can now be used for string replication. It takes a string as the first operand and a number as the second operand, returning a string. The sign of the number determines whether the result is mirrored or not.
eg:

“abcdefg”*2	gives “abcdefgabcdefg”
“abcdefg”*-1	gives “gfedcba”
“abcdefg”*1.5	gives “abcdefgabc”

a$[*modifierlist*]
Modifies the preceding string expression, according to which of the following characters are present within the square brackets in the *modifierlist*:

+	convert lower-case letters to upper-case
-	convert upper-case letters to lower-case
<	strip leading spaces (and control characters)
>	strip trailing spaces (and control characters)
~	strip bit-7 terminator from last character of string
^	add bit-7 terminator to last character of string
(f$,r$)	replace any occurrences of characters present in *f$* with the corresponding character from *r$* (or delete if there is no corresponding character)

The order of the modifiers is unimportant, except for “*(f$,r$)*” which, if present, must be the final modifier.

eg:

	“ Hello There! ”[<+->]
gives:	“hELLO tHERE!”
	“My typewriter is broken”[(”nore”,”dro”)]
gives:	“My typwoito is borkd”

Note that it is possible to slice a modified string, or modify a sliced string, since both () and [] continue to be evaluated following a string argument until there are no further opening parentheses or square brackets.

eg

```
a$(5)[-](3 TO 7)[<]
```

is perfectly valid.

{a$**}**
Tokenises (but does not syntax-check) the contents of the string expression contained within the curly braces. This can be useful for more easily preparing strings to be passed to VAL or VAL$.

eg:

{“sin (pi/4)”}	gives a string “SIN (PI/4)”, including the tokens SIN (code 178) and PI (code 167)
VAL{“sin (pi/4)”}	gives 0.7071

Integer variables and expressions

For additional speed and memory efficiency, NextBASIC provides a new integer expression evaluator. Usually all integer values are treated as unsigned 16-bit values (signed 16-bit calculations and comparisons can also be performed, see later), 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).

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. It is important to note that after the **%** symbol has been used, all variable names in the expression refer to the specially-provided integer variables and arrays, and not the standard floating-point numeric variables.

It is also possible to embed an integer expression within a standard numeric expression, by enclosing it within parentheses.

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

It is *not* generally possible to access standard numeric variables or functions within an integer expression (but see the special **INT{ }** notation later), 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
```

STEP values up to 32767 are allowed, as well as negative **STEP** values from -1 to -32767.

Note that although the loop index in an integer loop is treated as an unsigned value, this will actually work for signed loop limits as well (eg **FOR %i=-3 TO -7 STEP -1**). The one restriction is that both loop limits must be either positive or negative (eg **FOR %i=-3 TO 3 STEP 1** will not work).

Integer variables

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 additional RAM reserved by NextZXOS, 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. Therefore they survive a **LOAD** and can potentially be used to communicate information between BASIC programs.

There are 26 integer variables provided, named **A** to **Z** (can also be referred to in lower-case, **a** to **z**).

Integer arrays

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**.

It is not possible to re-dimension integer arrays: for speed, these are pre-allocated. However, to provide a little more flexibility it is possible to treat the array memory in different ways using **[]** subscript notation instead of the normal **()** notation.

When using a single **[]** subscript, you can treat an integer array as having more than 64 elements. Additional elements are taken from subsequent arrays in memory. Accessing **C[***n***]** would use memory in arrays C,D,E.. in turn, depending upon the maximum value of *n* used. For example, if you required 200 elements so *n* ranged from 0..199, then 4 arrays C(), D(), E() and F() would be in use (200/64=3.13).

When using a double **[][]** subscript, you can treat a range of integer arrays as a single two-dimensional array. In this case, the first subscript indicates the integer array to be used (0=the named array, 1=the next array etc) and the second subscript is the element number (0..63). For example, accessing **G[***n***][***m***]** where *n* ranges from 0 to 7 would use the 8 arrays G()..N().

Available operators and functions

The following unary operators may precede any integer value or sub-expression:

!	bitwise not
NOT	logical not (gives 1 if argument is zero; otherwise gives 0)
-	two's complement (negate)

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
^|	bitwise XOR synonym (to match standard expression evaluator)
<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

AND logical and (if 1st arg is 0, gives 0; otherwise gives 2nd arg)
OR logical or (if 2nd arg is 0, gives 1st arg; otherwise gives 1)

The six relational operators always produce a result of 0 for false and 1 for true.

The following integer functions are available (parentheses are optional):

IN *port*	read value from hardware port
REG *reg*	read value from Next register
PEEK *addr*	read byte from memory
DPEEK *addr*	read double-byte (16-bit word) from memory
USR addr	execute m/c routine and return value left in BC
BANK *n* **PEEK** *o*	read byte from offset in bank
BANK *n* **DPEEK** *o*	read double-byte (16-bit word) from offset in bank
BANK *n* **USR** addr	execute m/c routine in bank and return value left in BC
RND *n*	return random number
BIN *n*	synonym for @*n*, specifying a binary value
SPRITE *s*	return 1 (true) if sprite is visible, 0 otherwise
SPRITE CONTINUE *s*	returns sprite movement status (see sprites section)
SPRITE AT(*s,c***)**	returns sprite coord/movement data (see sprites section)
SPRITE OVER(..)	returns collision detection result (see sprites section)
INPUT *n*	returns/defines controllers (see standard expressions)

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

%RND *n* by default returns a random number in the range 0..n-1 (so that **%RND** *n* gives the same as the floating-point expression **INT(RND***n***)**). This behaviour can be changed (see “Code options” below).

Code options

A pseudo-variable **%CODE** may be set to change the behaviour of various *NextBASIC* operations. Each of the 16 bits controls a different feature.

Bit:	Feature:
0	Set for **%RND** to return values in the range 0..n (instead of 0..n-1). This allows **%RND 65535** to be used to obtain a full 16-bit range of random values (0..65535).
1	Set to disable the **BREAK** key.
2	Set to allow editor entry of legacy **DEF FN** statements that access the DEFADD system variable (disables editor entry of **DEF FN** statements using new parameter-passing features). New and legacy **DEF FN**s may be mixed in the same program - this bit only affects how statements entered in the editor (or .txt2bas) are treated.
3..15	reserved for future use (must be set to 0).

For example, using the command:

%CODE=1

sets bit 0 of the code options and adjusts behaviour of **%RND**. This should be done as part of your program’s initialisation code if it requires such behaviour.

Note that, for compatibility purposes, using the **RUN** command or **LOAD**ing a BASIC program resets all the code options to 0. Code options are not affected by the **CLEAR** command.

Signed integer expressions

As noted earlier, by default all operators and functions in integer expressions treat their arguments as unsigned, and produce unsigned results. However, it is also possible to override this behaviour and evaluate signed 16-bit integer expressions, with values in the range -32768..32767. This is done by enclosing the part of the expression which should be evaluated as signed with the special **SGN{ }** notation. An integer sub-expression or (more usually) the entire integer expression can be treated in this way. eg:

```
LET x1=%SGN{-100/3}
LET x2=%50+SGN{-100/3}
```

The only operators that treat their arguments as signed within a signed integer expression are the arithmetic operators (*** / MOD + -**) and the relational operators (**< <= = >= > <>**). All other operators and functions still treat their arguments as unsigned values.

Embedding floating-point sub-expressions within integer expressions

It is also possible to use the special **INT{ }** notation to embed an expression that returns a standard numeric value within an integer expression. This, for example, allows calculations to be included that are not supported in the integer expression evaluator, or access to standard floating-point variables.
eg:

```
%x=%x+INT{LEN x$}
%x=%x*INT{apples}
%x=%SGN{x+INT{(INKEY$="P" OR INKEY$="p")-(INKEY$="O" OR INKEY$="o")}}
```

Embedding integer sub-expressions within standard expressions

It is even easier to embed an integer sub-expression within a standard expression (for accessing the value of integer variables, for example). Simply enclose an integer expression within parentheses.
eg:

```
apples=SQR(%a)
x$=CHR$(%b|$20)
```

Memory bank access

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

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 *NextZXOS* 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).

NextZXOS 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 *NextZXOS*
8	Used for additional screen data (in lo-res, Timex hi-res and Timex hi-colour modes) and other data by *NextZXOS*

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 memory and banks:

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* **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 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* **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 *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.

BANK *n* **POKE** *offset*,*valuelist...*
POKEs a sequence of comma-separated values starting at offset *offset* (0-16383) in bank *n*. Each value in the list may be:

value	(numeric expr)	single byte
value~	(numeric expr)	16-bit word (low byte stored first)
v$	(string expr)	string (each character stored in turn)
v$~	(string expr)	string (bit 7 will be set on final char)

BANK *n* **DPOKE** *offset*,*valuelist...*
POKEs a sequence of comma-separated values starting at offset *offset* (0-16383) in bank *n*. The difference from **POKE** is that the default interpretation for a numeric value is a double-byte (16-bit word) rather than a single byte. Each value in the list may be:

value	(numeric expr)	16-bit word (low byte stored first)
value~	(numeric expr)	single byte
v$	(string expr)	string (each character stored in turn)
v$~	(string expr)	string (bit 7 will be set on final char)

POKE *addr*,*valuelist...*
DPOKE *addr,valuelist...*
Equivalents of **BANK..POKE** and **BANK..DPOKE** for the standard memory.

examples:

```
BANK b POKE “A null-terminated string”,0,”A bit7-terminated string”~,1000~
BANK 25 DPOKE 12345,33206,12354
DPOKE 23606,32768-256
POKE USR “a”,1,3,7,15,31,63,127,255
```

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 128 sprite objects and 64 sprite patterns (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 pattern) 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*,*p*,*f*,*rf*,*mx*,*my*
Set sprite *s* to position (*x*,*y*), pattern *p*, x-scaling *mx* and y-scaling *my*.
If the sprite id *s* is negative, this sprite is a *relative* sprite, and its position is relative to the previous *anchor* sprite (defined with a positive sprite id). See later for more information on relative sprites.

The flags parameter, *f*, 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)

The relative-flags parameter, *rf*, is a bitmask:
bit 0: type: 0=composite, 1=unified [only valid for anchor sprites]
bit 1: pattern is relative to the anchor [only valid for relative sprites]
bit 2: palette offset is relative to the anchor [only valid for relative sprites]

The scaling parameters *mx* and *my* are: 0=1x (no scaling), 1=2x, 2=4x, 3=8x.

Any parameter(s) in the **SPRITE** command can be omitted, and its value will be left unchanged from the last time it was explicitly specified.

(Again, the **BIN** function can be used to specify the flag parameters 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**.

Relative sprites

Sprites can be grouped together to form “composite” or “unified” sprites. Each such grouping consists of a single “anchor” sprite (this is always the sprite with the lowest sprite id in the grouping) followed by any number of “relative” sprites, with sprite ids following the anchor sprite in sequence.

When an anchor sprite is moved or made invisible, all the associated relative sprites are also moved or made invisible. It is also possible for individual relative sprites to be made invisible or visible. The rule is that a relative sprite is only visible if its own visibility flag is set **and** the visibility flag of the associated anchor sprite is set.

To define a relative sprite, simply specify its sprite id as a negative number (eg specifying **SPRITE -1,....** defines sprite 1 as relative to the preceding anchor sprite, 0). Any number of relative sprites can follow an anchor sprite.

The x and y coordinates specified in **SPRITE** commands for relative sprites are not actual coordinates, but signed offsets in the range -128 to +127 from the coordinates of the anchor sprite.

Additionally, if the “pattern relative” flag is set for a particular relative sprite, its pattern number is added to the pattern number from the anchor sprite (wrapping round if the sum exceeds 64). Using this, it is easy to animate an entire composite/unified sprite simply by changing the pattern of the anchor sprite.

Similarly, if the “palette relative” flag is set for a particular relative sprite, its palette offset is added to the palette offset from the anchor sprite (wrapping round if the sum exceeds 16).

Composite vs Unified sprites

The type of a grouping of sprites is determined by the “type” flag of the anchor sprite.

For composite sprites, the remaining sprite parameters (rotation, x/y mirrors and x/y scaling) are independent for each relative sprite. This allows creation of a composite sprite where individual relative sprites can be rotated etc for animation purposes.

For unified sprites, the rotation and x/y mirrors of the relative sprites are relative to that of the anchor sprite. Therefore, when the rotation or x/y mirrors of the anchor are changed, all the relative sprites rotate or reflect about the anchor.

Additionally, for unified sprites, the x/y scaling of the individual relative sprites is ignored, and the x/y scaling of the anchor sprite is applied to all sprites in the grouping, allowing the entire grouping to be scaled just by changing the scaling of the anchor. (This scaling also applies to the relative x/y coordinate offsets).

Batching

The standard **SPRITE** command normally has immediate effects to what is displayed on the screen. However, it is also possible to place **NextBASIC** into *batching* mode. In this mode, the standard **SPRITE** command has no immediate effect, but the changes specified are remembered. When all the required changes have been made (using multiple **SPRITE** commands) they can all be applied to the screen at once, giving a more synchronised look to your game. The following commands control this mode:

SPRITE STOP

Turns off immediate updates and enables batching mode.

SPRITE RUN

Turns immediate updates back on, disabling batching mode (this is also done by

the **SPRITE CLEAR** command).

SPRITE MOVE
When batching mode is enabled, sends all outstanding sprite changes to the hardware immediately.

SPRITE MOVE INT
The same as **SPRITE MOVE**, but first waits for the 50/60Hz interrupt. This can be used to synchronise your game to the framerate.

SPRITE MOVE INT *y*
The same as **SPRITE MOVE INT**, but changes are not sent to the hardware until after the TV scanline corresponding to sprite coordinate *y*. This can be used to avoid flicker by ensuring sprite changes don't occur whilst the TV is midway through displaying the sprite(s) in question.

Automatic sprite movement
In order to reduce the amount of work a **NextBASIC** program needs to do to animate and move sprites, commands are provided to allow some or all of this work to be done automatically whenever a **SPRITE MOVE** command is issued. Any sprite can have automatic movement or animation applied to it, and the standard **SPRITE** command can still be used to perform any other changes when they are needed.

The main command for setting up automatic sprite movement is the **SPRITE CONTINUE** command:

SPRITE CONTINUE *s*, [*x1* [**TO** *x2*]] [**STEP** *xs*] [**RUN** or **STOP**],
[*y1* [**TO** *y2*]] [**STEP** *ys*] [**RUN** or **STOP**],
[*p1* [**TO** *p2*]],
[*f*], [*r*], [*d*]

As indicated by the square brackets, each parameter (or sub-clause of a parameter) is optional. If not specified, the previous value will be retained.

Movement in the x-direction is specified with:
x1: minimum value for x-coordinate
x2: maximum value for x-coordinate (=*x1* if not provided)
xs: signed step in pixels (-127..+127) for every move
RUN: indicates movement in the x-direction is initially **on**
STOP: indicates movement in the x-direction is initially **off**

Movement in the y-direction is specified with:
y1: minimum value for y-coordinate
y2: maximum value for y-coordinate (=y*1* if not provided)
ys: signed step in pixels (-127..+127) for every move
RUN: indicates movement in the y-direction is initially **on**
STOP: indicates movement in the y-direction is initially **off**

Pattern animation is specified with:
p1: minimum value for sprite pattern
p2: maximum value for sprite pattern (=*p1* if not specified)

Movement rates are controlled with:
r: rate at which sprite moves/animates (0-255)
0=on every **SPRITE MOVE** command
1=skip 1 **SPRITE MOVE** command after moving
2=skip 2 **SPRITE MOVE** commands after moving
etc

d: delay before initial movement (0-255)
0=move on the first **SPRITE MOVE** command
1=skip 1 **SPRITE MOVE** command before the first move
2=skip 2 **SPRITE MOVE** commands before the first move
etc

The flags parameter, *f*, is an 8-bit mask (best specified with BIN):

bits 1..0: behaviour when x,y limits are reached:
00 = reflect this direction
01 = stop this direction, start other direction
10 = stop this direction
11 = stop completely and make sprite invisible
bit 2: flip the Y-mirror flag when y limits are reached
bit 3: flip the X-mirror flag when x limits are reached
bit 4: behaviour of pattern change:
0 = cycle upwards, wrapping back to lower limit
1 = bounce between lower and upper limits
bit 5: if set, sprite is disabled when pattern reaches limits
bit 6: if set, update pattern even when sprite is stationary
bit 7: if set, X-mirror/Y-mirror/rotation flag are set
according to direction of travel (overrides bits 2 & 3)

The initial position, pattern and other details of the sprite are determined by the last standard **SPRITE** command. If these values are outside the maximum/minimum ranges, then (depending upon the specified step and RUN/STOP status) they will gradually change until they fall within the max/min range.

If automatic movement is specified for an anchor sprite then, as you might expect, the entire composite/unified sprite will be automatically moved.

Automatic movement can also be specified for individual relative sprites if desired (the parameter *s* is always positive for the **SPRITE CONTINUE** command, but the sprite remains relative if specified as such in the last standard **SPRITE** command). Most typically this would be done to animate the relative sprites separately, so that one relative sprite might animate whilst the others remain the same (for example). However, it can also be used to automatically move a relative sprite around within a composite/group sprite. The main restriction here is that the movement limits are unsigned, so this works best for relative sprites with positive offsets (add 256 to negative offsets when specifiying them in a movement range).

Automatic movement can be temporarily suspended for particular sprites if desired:

SPRITE PAUSE *s1* [**TO** *s2*]
Turns off automatic movement for a single sprite or range of sprites.

SPRITE CONTINUE *s1* [**TO** *s2*]
Turns automatic movement back on for a single sprite or range of sprites.

Sprite functions

Several functions are provided to return details about sprites, and to make collision detection checks.

NOTE: All these functions are available in the standard expression evaluator and the integer expression evaluator.

SPRITE *s*
Returns 1 (true) if sprite is visible, 0 (false) if not visible

SPRITE CONTINUE *s*
Returns a bitmask describing the automatic movement enabled for this sprite:

bit 0: set if automatic movement is enabled
bit 1: set if currently moving in the Y direction
bit 2: set if currently moving in the X direction

(If bit 0 is set but neither bit 1 or 2 is set, only the pattern is being animated).

SPRITE AT(*s*,*c*)**
Returns a coordinate or other movement-related value for the sprite:

SPRITE AT(*s*,0**)**	returns x coordinate
SPRITE AT(*s*,1**)**	returns y coordinate
SPRITE AT(*s*,2**)**	returns pattern number
SPRITE AT(*s*,3**)**	returns x step
SPRITE AT(*s*,4**)**	returns y step
SPRITE AT(*s*,5**)**	returns delay before the sprite next moves

(0 means the sprite will move on the next **SPRITE MOVE** command)

SPRITE OVER(*s1*, *s2* [**TO** *s3*] [,*overlapX* [,*overlapY*]]**)**
Performs bounding-box collision detection between sprite *s1*, and a single other sprite *s2* or a range of sprites *s2..s3*.

Optional acceptable overlaps (in pixels) can be provided in *overlapX* and *overlapY*. If *overlapX* is not present, 0 (no overlap) is used. If *overlapY* is not present, then the value of *overlapX* is used. Overlaps should be 0..7 for an unscaled sprite *s1*, or 0..15 for a 2x scaled sprite etc.

Returns 0 (false) if there is no collision
Returns the number of the colliding sprite (*s2..s3*) if there was a collision.

NOTE:
If the colliding sprite (*s2..s3*) is id 0, 128 is returned.

NOTE:
Any relative sprites following s2 or s3 will also be checked, until the next anchor sprite not in the specified range.

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 *NextZXOS* 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.
Also resets mode windows to default settings (such as character set size, auto-pause etc).

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 *NextZXOS* 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).

In layer 1/2, therefore, **PRINT AT** *y,x* uses character coordinates (as on layer 0), but this will not always be 24 lines by 32 characters, depending upon whether different character size (and/or reduced height mode) has been selected. Note that double-width/double-height modes don't affect the coordinate system used.

In layer 1/2, it is also possible to use **PRINT POINT** *x,y* to change the print position. This uses the same pixel coordinate system used by **PLOT/DRAW/CIRCLE**.

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 5 system-maintained full-screen windows which are used for all **PRINT**ing through the standard “s” channel when one of the layer 1 or 2 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.
22,y,x	AT y,x. Position is specified in terms of character positions (dependent upon the character size currently selected and whether reduced-height text is in operation. Double-width and double-height do not affect the coordinates, however).
23,nLow,nHigh	TAB n (same as +3e, but documentation incorrect on +3e site).
24,n	ATTR n. Ignored in lo-res, layer2 and Timex hi-res modes.
25,y,xLow,xHigh	POINT x,y. Changes print position to pixel coordinates x,y. Previously this control code turned on or off extended UDGs for codes 165-255 instead of keyword tokens. Under *NextZXOS* 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.

Pointer operations

Window channels (and full-screen mode channels) now support pointer operations. The upper 16 bits contains y (in pixels) and the lower 16 bits contains x (in pixels).

eg: **DIM #2 TO wsize: height=INT (wsize/65536): width=wsize-65536*height**
RETURN #2 TO yx: y=INT (yx/65536): x=yx-65536*y
GOTO #2,65536*y+x

GOTO #2,65536*y+x is equivalent to **PRINT #2;POINT x,y;**

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. 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.

INPUT (windowed or non-windowed) now supports the following editing keys (many of which match those in the *NextBASIC* editor):

- **CURSOR LEFT/RIGHT/UP/DOWN** - Move cursor
- **EXTEND,CURSOR LEFT** - Move to start of input
- **EXTEND,CURSOR RIGHT** - Move to end of input
- **TRUE VIDEO** - Move left one word
- **INVERSE VIDEO** - Move right one word
- **DELETE** - Delete character
- **EXTEND,DELETE** - Delete character right
- **EXTEND,TRUE VIDEO** - Delete word left
- **EXTEND,INVERSE VIDEO** - Delete word right

- **EXTEND,9** - Delete to start of input
- **EXTEND,0** - Delete to end of input
- **EDIT** - Delete entire input
- **EXTEND,1** - Toggle whether keys **ASDFGYUQ** produces symbols or tokens

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*
BANK *n* **LINE** *x,y* **REM**
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).
The optional **REM** variation strips out all comments and blank lines when copying lines to the bank.

Note: The maximum length of a line that can be copied into a banked section of program is 256 bytes. This differs slightly from the number of characters in the line as seen in the NextBASIC editor, since the restriction is based on the "tokenised" length of the line: each token, such as **PRINT**, only uses up 1 byte. However, numeric literals in standard (non-integer) expressions use an extra 6 bytes for a hidden floating-point representation. Additionally there is a 5-byte overhead for the line number, line length and terminating ENTER character.

BANK *n* **LIST**
BANK *n* **LIST** *l*
BANK *n* **LIST PROC** *name***()**
BANK *n* **LIST FN** *name***()**
BANK *n* **LIST** *@label*
List lines in bank *n,* with optional starting line/procedure/function/label

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*=255.

BANK *n* **GOSUB** *l*
GOSUB line *l* in bank *n.* To GOSUB the main program from a banked section, use *n*=255.

BANK *n* **PROC** *name***(***expressionlist***)**
BANK *n* **PROC** *name***(***expressionlist***) TO** *paramlist*
Call a procedure in bank *n*. To call a procedure in the main program from a banked section, use *n*=255.

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

Working with BASIC programs

The following additional commands are provided:

LIST PROC *name***()**
LIST FN *name***()**
LIST *@label*
List program, starting from specified procedure/function/label

LINE NEW *start,step*
Starts automatic line numbering beginning with line number *start* and

incrementing numbers by *step*. If *start* is omitted, lines start after the current editor line. If *step* is omitted, numbers increment by 10.

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

LINE *n,m* **TO** *start,step*
Renumbers a block of lines within the BASIC program. All lines between *n* and *m* (inclusive) are renumbered (and, if necessary, moved), with new starting line number *start* and incrementing numbers by *step*.

LINE MERGE *first*,*last*
Merges lines from *first* to *last* into a single line (separated by colons).

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, GOSUB, RESTORE, PROC** or **FN** within a banked section will reference code in the same bank.

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

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*
LINE *n,m* **TO** *start,step*
LINE MERGE *first*,*last*
BANK *n* **LINE** *x*,*y*
BANK *n* **LINE** *x*,*y* **REM**
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:/NEXTZXOS/AUTOEXEC.BAS file.

Miscellaneous other features

CONT and **RAND** may be used as abbreviations for **CONTINUE** and **RANDOMIZE**.

; (semi-colon) may now be used as an alternative to **REM**.

This is intended for use with dot-commands which can process such lines whilst *NextBASIC* ignores them. eg an assembler might use as follows:

```
1000  .assemble
1010  ; LD BC,1234
1020  ; RET
```

The following additional commands are provided:

RUN AT *speed*
Changes the speed of the ZX Spectrum Next.
Allowable values for *speed* are:

```
0     (3.5MHz)
1     (7MHz)
2     (14MHz)
3     (28MHz)
```

REG *reg,value*
Sets a Next Register.
eg to perform a soft reset:

```
REG 2,%@0000001
```

SPECTRUM 48
Switches into 48K BASIC mode whilst retaining the current BASIC program (although NextBASIC commands and features will no longer be available). This differs from the existing **SPECTRUM** command, which leaves the Next configured as a 128K Spectrum in 48K BASIC mode. Instead, the **SPECTRUM 48** command leaves the Next configured as a genuine 48K Spectrum, with 48K timings, the original 48K BASIC ROM and with the 128K paging hardware disabled.

SPECTRUM LOAD
Reconfigures the Next as a 128K Spectrum and launches the *Tape Loader* menu option.

TIME
New command to reset the frame-counter (FRAMES) to 0. Intended for use with the corresponding **TIME** function as a timer:

```
10 TIME
20  PROC perftest()
30 PRINT “perftest() took “;TIME;” frames”
```

LOAD/SAVE/VERIFY *f$* **INT**
Load, save or verify all the integer variables and arrays.

LOAD/SAVE/VERIFY *f$* **INPUT**
Load, save or verify the definition of the “keyboard joystick” (see **INPUT** *n* function).

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.

Installable device drivers

NextZXOS 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):

X1	5B33H (23347)	CACHEBNK	8K bank for cached prog data (was REGNUOY+9)
X2	5B58H (23384)	RETVARS	Address of local variables (was TARGET)
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)
N1	5B65H (23397)	LSBANK	Temp bank in LOAD/SAVE & others (was TVPARS)
X1	5B68H (23400)	FLAGN	Flags for the NextZXOS 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)
N2	5C3FH (23615)		Reserved for system use (was LISTSP)
N2	5C6CH (23660)		Reserved for system use (was S_TOP)
1	5C81H (23681)	STIMEOUT	Screensaver control (was unused)
2	5CB0H (23728)		unused (was NMIADD)

The following system variables have been inserted where STRIP1 and STRIP2 were, within the temporary TSTACK area. This means that there are now 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)

NOTE:
NEWPPC and NSPPC can be POKEd to cause a jump to a specified line/statement. For performance reasons, USR routines that alter these system variables no longer cause a jump. However, this can be worked around by re-POKEing the low 7 bits of NSPPC after the USR routine has been executed. For example:

```
50 x=USR 30000: REM call m/c routine that alters NEWPPC/NSPPC
60 POKE 23620,PEEK 23620&127: REM activate the jump
```

New keyword tokens

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

```
TIME        $81
PRIVATE     $82
IFELSE      $83
(internal use only: displays as IF, but indicates ELSE is present on same line)
ENDIF       $84
EXIT        $85
REF         $86
PEEK$       $87
REG         $88
DPOKE       $89
DPEEK       $8a
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
No DEFPROC
Fragmented - use .DEFRAG
No ENDIF
No label
```

Layout of long-named variables in memory

NOTE: Global variables only. Local variables are stored on the return stack and probably beyond the scope of this documentation.

Long-named numeric variables (as before):

1st byte:	101 1st letter-60h
2nd-(N-1)th bytes:	2nd-(N-1)th letter/digit
Nth byte:	last letter/digit+80h

Long-named loop control variables:

1st byte:	101 1st letter-60h
2nd-(N-1)th bytes:	2nd-(N-1)th letter/digit
Nth byte:	last letter/digit+80h-20h

Long-named strings and arrays:

1st byte:	011 11111 (7fh)
2nd byte:	010 1st letter-60h (strings)
2nd byte:	100 1st letter-60h (numeric arrays)
2nd byte:	110 1st letter-60h (character arrays)
3rd-Nth bytes:	2nd-(N-1)th letter/digit
(N+1)th byte:	last letter/digit+80h

Sprite engine data

Data for the sprite engine is stored in 16K RAM bank 8. Access to this bank is restricted under BASIC (due to the presence of NextZXOS system data). However, the offset of the sprite engine data is provided (from NextZXOS v2.09) for drivers or other system extensions to make use of if desired.

```
RAM 8 offset:
$1ffe (2)               uvec8_sprite_data: offset of sprite engine data in RAM 8
uvec8_sprite_data:
+0 (128*16)             sprite0_data..sprite127_data
+$800 (32)              sprite dirty/automove flags

spriteN_data:           16-byte structure for each of sprites 0..127
                                  bit: 7  6  5  4  3  2  1  0
+0 (1)                  sprite_attrib0: X7 X6 X5 X4 X3 X2 X1 X0
+1 (1)                  sprite_attrib1: Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
+2 (1)                  sprite_attrib2: P3 P2 P1 P0 XM YM RF X8
+3 (1)                  sprite_attrib3: V  @R N5 N4 N3 N2 N1 N0
+4 (1)                  sprite_attrib4: @a @b @c XS XS YS YS @M
+5 (1)                  sprite_flags:   XR YR IS IB XM YM B1 B0
+6 (1)                  sprite_xMin: min X coord
+7 (1)                  sprite_xMax: max X coord
+8 (1)                  sprite_iMin: min image (bits 6..1), xMin8 (bit 0),
                                     update image even if not moving (bit 7)
+9 (1)                  sprite_iMax: max image (bits 6..1), xMax8 (bit 0),
                                     decrement flag (bit 7)
+10 (1)                 sprite_xStep: X step (signed)
+11 (1)                 sprite_yMin: min Y coord
+12 (1)                 sprite_yMax: max Y coord
+13 (1)                 sprite_yStep: Y step (signed)
+14 (1)                 sprite_rate: 0=every MOVE, 1=wait 1 MOVE etc
+15 (1)                 sprite_delay: # MOVEs left to wait

NOTE: sprite_attrib bytes as defined by the corresponding Next registers
($35..$39), with the following changes:
                        @R=anchor (0)/relative (1)
                        @a=palette relative flag
                        @b=pattern relative flag
                        @c=composite/unified flag
                        @M=mirror behaviour:
                              0=use auto-flip x-mirror and y-mirror behaviour
                              1=generate directional x-mirror/y-mirror/rotate

sprite_flags bits:
      XR=X running flag
      YR=Y running flag
      IS=image stop: turn off sprite if reach end
      IB=image behaviour: 0=cycle, 1=bounce
      XM=auto-flip x-mirror
      YM=auto-flip y-mirror
      B1..B0=behaviour:
                  00=reflect this dirn
                  01=stop, restart other dirn
                  10=stop this dirn
                  11=disable, make invisible

sprite dirty/automove flags, arranged in pairs of bytes:
+0    dirty flags (sprite data changed)
+1    automove flags

The first pair of bytes corresponds to sprite 0 (bit 7)..sprite 7 (bit 0)
The second pair of bytes corresponds to sprite 8 (bit 7)..sprite 15 (bit 0) etc
```

List of updates

Updates: 9 Mar 2024

Added details of sprite engine data vector, added in v2.09.

Updates: 12 Feb 2024

Updated DEFADD, RETVARS and CACHEBNK system variables to match v2.09.

Added **%CODE** bit 2, for allowing legacy **DEF FN** entry.

Added descriptions of new commands:
LINE NEW *start,step*
LINE *n,m* **TO** *start,step*
BANK *n* **LINE** *x,y* **REM**

Updates: 27 Nov 2023

Updated system variable changes for reinstatement of NEWPPC/NSPPC functionality in v2.09.

Updates: 24 May 2023

Added information on the new features in v2.08 (also available in separate "New in v2.08.pdf" document).

Updates: 26 September 2022

Added information on the editing keys available in **INPUT** from v2.07l.

Updates: 18 June 2022

Added note that window channels (and full-screen mode windows) now support pointer operations.

Updates: 11 March 2022

Clarified that scaling applies to x/y relative coordinates for unified sprites. Noted that relative x/y coordinates can now be in a full 8-bit range (-128 to +127), fixed in v2.07e.

Updates: 5 November 2021

Updated **%RND** description: **%RND 0** always gives 0 for compatibility with versions before v2.06J.

Added new **%CODE** pseudo-variable. Currently allows **%RND** behaviour to be changed to give random number in range 0..n (allowing full 16-bit range for **%RND 65535**).

Updates: 26 October 2020

Clarified that %RND 0 gives a random number in the full 16-bit range 0..65535.

Noted maximum banked line length is 256 bytes.

Updates: 30 March 2020

Clarified how integer loops work with negative loop limits and/or steps.

Noted that **CONT** and **RAND** can now be used as abbreviations for **CONTINUE** and **RANDOMIZE**.

Noted that **LET** is now optional (from v2.06A) and may be omitted.

Updates: 25 March 2020

Documented the new sprite-related features added in v2.06:

* there are now 128 sprite objects
* the main **SPRITE** command is extended with 3 new parameters (*rf, mx, my*)
* all parameters in the main **SPRITE** command are now optional
* added new **SPRITE STOP**, **SPRITE RUN**, **SPRITE MOVE**, **SPRITE PAUSE** and **SPRITE CONTINUE** commands
* added new **SPRITE**, **SPRITE AT, SPRITE CONTINUE** and **SPRITE OVER** functions into the integer expression evaluator

Updates: 30 January 2020

Added new **SPECTRUM 48** and **SPECTRUM LOAD** commands.

Updated NextZXOS version number referred to, for day zero release (v2.04).

Updates: 29 November 2019

Added a couple of missing changes to the system variables.

Updates: 28 November 2019

Added 23 (TAB) to list of window control code changes from +3e (not really a change, but mis-documented on the +3e site).

Added new commands (plus enhanced **POKE** and **BANK..POKE** commands):
REG reg,value
POKE addr,valuelist...
BANK n **POKE** addr,valuelist...
DPOKE addr,valuelist...
BANK n **DPOKE** addr,valuelist...

Added special-case assignment commands for reading strings from memory:
LET *stringdest* **= PEEK$(***addr,len***)**
LET *stringdest* **= PEEK$(***addr,~***)**
LET *stringdest* **= PEEK$(***addr,~char***)**
LET *stringdest* **= BANK** *n* **PEEK$(***addr,len***)**
LET *stringdest* **= BANK** *n* **PEEK$(***addr,~***)**
LET *stringdest* **= BANK** *n* **PEEK$(***addr,~char***)**

Revamped integer expressions section to include all the new features as at v2.03b, including:

alternative views of integer arrays using [] subscripting
SGN{} and **INT{}** sub-expressions
IN, REG, PEEK, DPEEK, USR, BIN, RND, BANK..PEEK, BANK..DPEEK, BANK..USR functions
NOT, - unary operators
AND, OR binary operators

Noted that **RUN AT** can now also select 28MHz.

Added new keyword token codes.

Clarified that **ENDPROC** always returns to the calling bank.

Removed deprecated **BANK..PEEK** and **BANK...USR** commands.

Updates: 23 October 2019

Added missing description of **BANK** *n* **PROC** command.

Fixed “main program” id from -1 to 255 in **BANK** *n* **GOTO/GOSUB/PROC** commands.

Updates: 18 May 2019

Clarified that integer variables survive across a LOAD command.
Fixed typo in one of the REPEAT examples.

Updates: 18 Oct 2018

Clarified that error trapping does work for dot commands.

Updates: 23 Aug 2018

Added new **BANK n USR address** command to execute machine-code routines in any bank.

Updates: 15 Aug 2018

Noted that **;** may be used as an alternative to **REM** (intended for dot command use, eg for an assembler).

Updates: 8 Jul 2018

Updated the list of new error messages.

Updates: 20 May 2018

Noted that **LAYER CLEAR** also resets mode windows to default settings.

Added **LIST PROC**, **BANK..LIST PROC**, **RUN AT** and **LINE MERGE** commands.

Updated text windows: control code 25 is now used to specify a new print position using pixel coordinates.

Updated notes on layers to reflect that **PRINT AT** now uses character position coordinates (as with layer 0) and that there is a new **PRINT POINT** facility which allows position to be specified with pixel coordinates.

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:/NEXTZXOS/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*