What's new in NextZXOS v2.08

This document describes the main new features in *NextZXOS v2.08*. It is mainly intended for those people who have previously been using v2.07 and want to know what's changed.

If you are new to *NextZXOS* and *NextBASIC*, all the information in this document is also integrated into the main documentation files:

NextBASIC_File_Commands.pdf
NextBASIC_New_Commands_and_Features.pdf
NextZXOS_Editor_Features.pdf
NextZXOS_and_esxDOS_APIs.pdf

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.

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
```

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.

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
```

Conditional flow structures

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”
```

Procedures

Procedure names may now include a trailing '$' symbol if desired; this has no effect on functionality.

DEFPROC statements can now appear anywhere in the program; they no longer have to be the first statement in a line.

Arrays (string, numeric and integer) can now be passed as procedure parameters, or created as LOCALs. The syntax for this is the array name followed by (). LOCAL arrays (except integer arrays) need to be DIMmed before use. eg:

```
100 PROC x(a(),size,2) TO result:PRINT result:STOP
110 DEFPROC x(inp(),size,y)
120 LOCAL z(),tot
130 DIM z(size)
140 tot=0
150 FOR i=1 TO size:z(i)=inp(i)*y:NEXT i
160 FOR i=1 TO size:tot=tot+z(i):NEXT i
170 ENDPROC = tot
```

Default values can now be provided in DEFPROC or LOCAL statements. A PROC can omit any parameter if the matching DEFPROC has a default value in that position. 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 now 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 <u>not</u> be passed by reference.

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
```

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()
```

Assignments

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$,d$
```

Standard expression evaluator

The **PEEK$** and **BANK..PEEK$** functions have now been properly moved into the standard expression evaluator, and so can be used as part of any string expression (previously they were only allowed in a **LET** statement).

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

New functions

DPEEK addr
BANK n **PEEK** addr
BANK n **DPEEK** addr
REG n
SPRITE s
SPRITE CONTINUE s
SPRITE AT(s,c)
SPRITE OVER(s, start [**TO** end] [, overlapXY [, overlayY]])

These functions all work the same way as the existing ones in the integer expression evaluator. **DPEEK** no longer gives different results from **PEEK** since the function is now implemented in the main 48K BASIC ROM (ROM3) - the same is also true of **%DPEEK**.

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
```

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

@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 now 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 expression evaluator

These new functions/operators from the standard expression evaluator are also available in the integer expression evaluator:

n **^|** *m*	Bitwise xor (synonym of ^)
INPUT *n*	Read/define input controllers

Integer expressions can also now be used as sub-expressions in the standard expression evaluator. An integer sub-expression can be started after any opening parenthesis or separating comma.

eg.

```
x$=STR$(%a, %b, 5)
x=apples*pears+(%x(3))
```

For the new BANK.. functions in the standard expression evaluator, the bank number can be considered to be implicitly within parentheses, so it may be specified using an integer expression directly.

eg

```
x=2*PI*BANK %b PEEK myaddress
```

NextBASIC options

NextBASIC options are controlled by the special **%CODE** integer variable. This is reset to zero when a program is loaded/run.

Currently available options are (bit 0 was already present in v2.07):

Bit	Use
0	if set, **%RND *n*** and **RND(*n*)** return values between 0..*n*, rather than 0..*n*-1
1	if set, the **BREAK** key is disabled

Miscellaneous commands

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).

The following commands are no longer really needed, as equivalent functions have now been added to the expression evaluator:

RETURN *#n* **TO** *var*
NEXT *#n* **TO** *var*
DIM *#n* **TO** *var*

New keyword 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

New errors

No ENDIF
No label

System variable changes

TARGET (5B58) is replaced with:

X1	5B58	23384	CACHEBNK	8K bank id holding cached program data.
N1	5B59	23385		Reserved for system use.

DEFADD (5C0B) is replaced with:

X2	5C0B	23563	RETVARS	Address of local variables on return stack.

LISTSP, NSPPC and S_TOP (respectively) are replaced:

N2	5C3F	23615	Reserved for system use.
N1	5C44	23620	Reserved for system use.
N2	5C6C	23660	Reserved for system use.

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