FUNCTIONS AND CONTROL STRUCTURE

In the first part of our appraisal of the Sinclair version of the BASIC language, we dealt with Sinclair's idiosyncratic approach to variable names and string-handling. Here we conclude our look at the dialect by considering the VAL, GOSUB and GOTO functions, and the control structures WHILE...WEND and REPEAT...UNTIL.

You may already have noticed that some functions in Sinclair BASIC do not require brackets around their arguments, unlike their counterparts in other BASICS, so that LEN(X$) can be written as either LEN X$ or LEN(X$). You should, however, use brackets if the meaning of an expression is doubtful or ambiguous.

The function CODE is the Sinclair equivalent of ASC(), and behaves in exactly the same way. The Sinclair character set, however, is standard ASCII only for values 32 to 122. So, for example, where PRINT CHR$(7) in most BASICS causes a 'beep' of some kind to be sounded, in Sinclair BASIC it causes an error message.

The function VAL is standard BASIC, but in Sinclair BASIC a statement such as VAL("a45") would cause the program to crash because the argument of the function is non-numeric. In most other BASICS this would simply return the value zero. If this particular quirk is likely to be a problem you may have to write a subroutine to replace the VAL function, or you may have to test the CODE value of the first character of the argument of VAL: if CODE A$(1) <48 or CODE A$(1) >57 then A$ is non-numeric and should not be the argument of the function VAL.

Sinclair VAL has the unusual power, however, of evaluating numeric expressions, so that:

```
LET A$="6*12":PRINT VAL A$
```

will result in 72, the value of the expression '6*12'. In most BASICS, VAL is not as powerful as this, and would return the value 6 in this case.

This ability to evaluate expressions can be put to use in many ways. A simple example is this block graph drawing program:

```
100 DIM S$(31)
200 LET S$="********************************"
300 INPUT "ENTER A FUNCTION OF X";F$
400 PRINT "Y = ";F$:PRINT
500 FOR X=1 TO 10
600 PRINT S$( TO INT(VAL F$))
700 NEXT X
800 PRINT"================================"
900 PRINT "00000000011111111112222222223"
950 PRINT "12345678901234567890123456789O"
```

When you run this program, you may type in any algebraic expression with variable X (2*X+3/X, for example), and you will see a crude block graph of that function on the screen. In this program, the y-axis is horizontal, the x-axis vertical, and graphics are pixel-size. It would not require much more code to establish the range of values of x and y immediately after the function of x was input, and then make scaling adjustments, print axes and create high resolution graphics. The result would be a very impressive graphics package; the fragment above, however, is simply designed to make clear the usefulness of VAL's power to accept expressions as its argument.

Just like VAL, in this respect, are GOSUB and GOTO. They too evaluate expressions in situations where most BASIC dialects would require that their arguments be valid line numbers. This has several advantages. You can give your subroutines names, for example, define variables with the same names and appropriate values, and then GOSUB to the subroutines by name. Here is an example of this ability:

```
100 LET INIT=1000
200 LET OUTPUT=2000
300 LET CALCULATE=3000
400 GOSUB INIT
500 GOSUB CALCULATE
600 GOSUB OUTPUT
700 STOP
1000 REM*******INIT S/R********
.....
2000 REM*****OUTPUT S/R********
.....
3000 REM****CALCULATE S/R******
```

which almost makes your program self-documenting. If you replace GOSUB INIT by GOSUB (VAR1+N*VAR2), or any valid numerical expression, the expression will be evaluated and the result treated as a line number. Furthermore, in Sinclair BASIC, if the argument of GOTO or GOSUB is a line number that does not exist, then control passes to the next valid line number. If you write, for example, GOTO 17 and line 17 does not exist but line 18 does, then control will pass to line 18 and no error will result — as it does in most BASICS.

The ability of Sinclair BASIC to handle these 'computed jumps' makes up for the lack of the ON...GOTO and ON...GOSUB structures. In another BASIC you might write:

```
2360 ON D GOSUB 100,200,300 400,500
```

meaning that if the value of D is 1 then the program will jump to line 100 (GOSUB 100), if D=2 then it will jump to line 200 (GOSUB 200), and so on. In