Pocket Guide

Commodore 64

Boris Allan

Programming Pocket Guides

PITMAN PUBLISHING LTD
128 Long Acre, London WC2E 9AN

Associated companies
Pitman Publishing Pty Ltd, Melbourne
Pitman Publishing New Zealand Ltd, Wellington
Copp Clark Pitman, Toronto

Consultant Editor: David Hatter

First edition, 1984

Printed in Great Britain at The Pitman Press, Bath

ISBN 0 273 02125 7

Index

How to use this Pocket Guide

To understand the workings of the Commodore 64 (henceforth shortened to the C64), the user needs a solid understanding of C64 BASIC. This Guide gives the essential knowledge necessary for that understanding.

The Guide starts with a discussion of the way in which variables and functions are stored in C64 BASIC, and in so doing enables us to investigate some of the arithmetical procedures of the computer. In this investigation certain errors appear within the C64 BASIC system.

After the discussion of variables and arrays, there is an examination of arithmetical operators, relational evaluations, and logical connectives. When we discuss logical connectives we also examine the form of computer arithmetic known as 'two's complement'.

The main part of the text examines in detail each keyword in C64 BASIC, to show the limitations and potential of the C64. For some keywords other bugs are discovered, and knowledge of these bugs may help explain some of the strange results one sometimes produces on the C64.

The abbreviations MUM and PREG are used extensively throughout the guide. MUM is the *MicroComputer User's Manual* you get together with the C64, and PREG is the *Programmer's Reference Guide for the C64*, produced separately by Commodore.

Variables in C64 BASIC

It helps in the study of C64 BASIC to clarify the nature of the items with which the language deals. The term 'item' is deliberately vague, and the purpose of this Guide is partly to make the nature of this vague term rather clearer.

An item or a collection of items is known as an 'expression', of which there are three main types.

Types of expression

(a) Constants are actual values in the line of program, such as 3, 3.4, or "HELP". They are—as the name implies—constant, and do not vary: "HELP" is always "HELP".

(b) Variables are names given to receptacles for storing values, where the values may alter. The 'assignment' X=3 stores the value 3 in the receptacle called X. X is an example of a variable, and a new assignment X=4 changes the value stored in X.

Variable names can be up to two characters long, with the first character being a letter, e.g. Y, YY, and Y1. A variable name which is more than two characters will have the rest ignored after the first two characters—COULD and COLD will be treated alike, as will COLOUR1 and COLOUR2. Three variable names are reserved (ST, TI, and TI$), and no variable name may contain within it the name of a BASIC keyword (e.g. DANDY, which would be considered by the system to be D AND Y).

(c) What we will term 'mixtures' are combinations of constants and variables (e.g. X/5), and are not to be confused with expressions which are merely complex (e.g. X*X*Y/R^T is complex, but is not a mixture, as it only contains variables).

Generally speaking, mixtures or constants produce programs which run more slowly. Impressive differences in speed can be obtained by making as many expressions as possible use only variables. It is also simpler to alter programs with many variables than those with many constants, especially if the same variable can stand for many constants of the same value.

Classes of values

Independently of the form of the expression there are four classes of expression. To help recognize the type of expression, generally, later, an integer is often shown as I0% to I9%, real numbers by X0 to X9, logical expressions by L0% to L9%, and strings by S0$ to S9$.

(i) Integer values are whole numbers with no fractional part. Integers are stored as whole numbers in two bytes (i.e. 16 bits), and integer constants might be shown as 3 or as −4567E3, for example, where each value is stored exactly, with no approximation. Integer variables are shown with a % suffix, e.g. X% or TJ%. Integers take exact (i.e. non-fractional) values from −32768 to 32767.

(ii) Real values are numbers with fractional parts, so note that 3.0 is a real number, whereas 3 is an integer. 1.23456 and 1234.56 are both real numbers, and can be written as 1.23456E0 and 1.23456E3 respectively. The portion before the E is sometimes called the 'mantissa', and the portion after the E is called the 'exponent'.

Real values are stored in the C64 in a similar manner. The number is stored as a mantissa of four bytes (32 bits, i.e. an accuracy of about 9 digits), and an exponent of one byte (8 bits, i.e. from about E−39 to E+38). One of the 32 mantissa bits, is devoted to holding the sign of the number (1 for negative and 0 for positive numbers).

Real numbers take positive values ranging from 2.93873588E−39 to 1.70141183E+38. Negative values are rather stranger.

Try the short program

```
10 T=1
20 T=T/2 : PRINT T : GOTO 20
```

and watch the succession of values as T becomes steadily smaller. Eventually we reach

```
5.87747176E-39
2.93873588E-39
0
```

and all is fine. Try running

```
10 T=-1
20 T=T/2 : PRINT T : GOTO 20
```

which reaches

```
-1.17549435E-38
-5.87747176E-39
 2.93873588E-39
 0
```

and we have discovered a bug in the C64 BASIC system. Half of −5.87747176E−39 is not (as we all know) 2.93873588E−39.

We can produce some further errors:

```
T=4.25352959E+37
PRINT T*2
```

produces an ?OVERFLOW ERROR, whereas

```
PRINT T+T
```

gives the result 8.50705917E+37, not an error. T*2 is exactly the same as T+T: thus we have discovered another bug in C64 BASIC.

Continuing with

```
T=T+T : PRINT T : PRINT T + T
```

produces the value 8.50705917E+37 for the first PRINT, and an ?OVERFLOW ERROR for the second PRINT.

```
PRINT 8.50705917E+37 + 8.50705917E+37
```

gives the answer 1.70141183E+38. To multiply by 2 (and not to add) gives an ?OVERFLOW ERROR again.

Some of the C64 BASIC routines for real arithmetic are incorrect at the extremes, and may possibly be so for less extreme examples.

(iii) Logical values are −1 (for 'true') and 0 (for 'false'). Logical values are used for evaluating the outcome of certain sets of conditions, e.g. 'If this is true and that is not true, then do this. . .'. When a logical comparison is made, both sides of the comparison are effectively turned into 16-bit integers.

Logical values can result from logical comparisons or operations, but—as they also have numerical values—logical values can be treated as integer or real values, depending on the context.

(iv) String values are ordered collections of characters, where normally characters are distinguished from variables or constants by being enclosed in quotes. "X" is a character, but X is a variable; "X/2" is a string, whereas X/2 is an arithmetical operation; and "222222" is a string, but 222222 is a constant.

Each character (e.g. "X" or "/" or "2") has associated with it two codes. The first code, the ASCII code, relates to how each character is stored in one byte of memory (most computers use ASCII codes); and the second code, the Screen Display Code, relates to how each character is presented on the display screen (these code values are specific to the C64). The two types of code should not be confused.

String variables are distinguished by a $ suffix, e.g. A$ or THIS$ (the latter is equivalent to TH$).

Computing values

Unless constrained otherwise, the C64 always behaves as if the numbers with which it is working are all real values. These are stored in memory as four bytes for the mantissa (31 bits plus 1 bit for the sign of the number), and one byte for the exponent (as we noted above).

Because all the computations are made in floating point mode (i.e. real values), there have to be procedures for converting from one type of value to another.

Examining PREG (pages 310–313) shows that at locations 3 and 4 there is a pointer (ADRAY1) to the routine which converts floating point numbers to integers, and at 5 and 6 there is a pointer (ADRAY2) to the integer to floating point routine. The need to convert from integer to floating point values explains why the use of integer variables usually slows down programs.

At location 13 there is a flag (VALTYP) indicating whether the current data type is string or numeric, and, if numeric, location 14 (INTFLG) indicates whether the number is integer or floating point.

There are two floating point accumulators (#1 and #2), and the floating point numbers stored in these locations are held in a different manner to normal floating point values. Floating point accumulator #1 is held at locations 97 to 102, and a value is stored as thus:

LOCATION	DESCRIPTION
97	Exponent
98–101	Mantissa
102	Sign byte

and the order is exactly the same for floating point accumulator #2, which extends from location 105 to location 110.

You will observe that this differs from the way in which numbers are stored in memory by the addition of an extra byte, the 'sign' byte. There is also one byte given over to storing any overflow digits from #1 (at location 104), and one byte used for rounding of the value in #1 (location 112).

After using the accumulator, the resulting number is deemed to be negative if the sign bit for the number, and the sign byte, both indicate a negative result. In the coding of the mathematical routines on the C64 (and the VIC–20) errors have been made with checks on overflows and underflows. These errors have resulted in the sign comparisons being corrupted.

This is the probable source of some of the C64 BASIC bugs, and other bugs may be due to the ways in which floating point values are stored in and retrieved from variables.

If you

```
PRINT 2^(-128)
```

the answer is zero, but

```
PRINT 2^(-127)/2
```

will produce the result 2.93873588E−39, but this is effectively the same, because 2^(−128) = 2^(−127)/2.

Though these bugs might seem trivial, they do cast doubt on some of the other mathematical routines used in the C64. If the C64 is to be used for serious purposes these bugs need eradicating.

When the C64 is programmed using machine code, the floating point routines will not affect the results of machine code programs, unless the BASIC floating point routines are used from within machine code.

Variable information

To store one floating point variable X (with its value) requires seven bytes of memory on the C64.

The number of bytes needed can be built up quite simply. As X is a floating point variable which stores the value in five bytes, five bytes are needed to store that value. As any variable can possibly have up to two characters in its name, and one byte is required for each character, that is two bytes to store the name.
Five bytes plus two bytes gives the seven bytes to store any floating point variable, including both its name and its value.
To store one integer variable, X%, also requires seven bytes on the C64.
Now, obviously, at least two bytes are necessary to store the name of the integer variable, and in fact BASIC requires no more. When the variable name for an integer variable is stored, the character value stored is modified, to distinguish between an integer name and a floating point name.
The BASIC translator does not store the % suffix. What happens is that 128 is added on to the ASCII value of each character in the name (see MUM Appendix F concerning ASCII). For example, the ASCII code for X is 88, and that is how the character X is stored for floating point variables. 88+128 is 216, and that is how the character X is stored for an integer variable.
In the case of integer variables both ASCII values are increased by 128, whereas for strings only the second value is increased by 128. If there is no second character in the variable name then the value stored is 128. For floating point variables of only one character the second value is 0.
In this way, each type of variable can be unambiguously recognized.
Returning to integer variables, two bytes from seven bytes leaves five bytes. However, as we have noted, integers only occupy two bytes (and range from −32768 to 32767). To drive this point home, enter

```
P = 1E16
P% = P
```

to which the response is ?ILLEGAL QUANTITY ERROR.
Whatever is attempted, it is not possible to squeeze more than two bytes into an integer variable. What happens to the other three bytes?

What happens to the other three bytes is that they are wasted, they do nothing. After the two bytes of the variable name, and the two bytes of the integer value, there are three bytes which contain zero.

The use of integers does not, therefore, save space, but rather the same space is used, and the program runs more slowly (the values have to be converted to floating point numbers, and vice versa).

To store a string variable X$ also takes seven bytes. Two bytes are again taken up by the name of the variable (modified to distinguish X$ from X or X%), which leaves five bytes.

The first of the remaining five bytes is taken up by the length of the string (thus a string may be of length 0 to 255, compare MUM 'Quick Reference Card'). The next two bytes act as a pointer to the location in memory at which the string starts.

The final two bytes are both set to zero.

All variables are stored at the end of the BASIC program, and locations 45 and 46 act as a pointer to the start of the BASIC variables. To find where the variables start, therefore, enter

```
PRINT PEEK(45) + PEEK(46)*256
```

and, if you subtract the start of BASIC (normally 2049) from this number, you can discover the extent of the BASIC program text.

When the BASIC translator comes across a variable name in a program line it has to find out the value stored in that variable, and to find out the value of the variable it has first to find the variable.

To do this the translator takes the pointer at locations 45 and 46, and then starts checking the variable names, commencing at the location to which 45 and 46 point.

If the first variable name does not match, the translator then moves to the next variable name, and it knows exactly where to look—it looks in steps of seven bytes at a time. C64 BASIC is a style of BASIC called 'Microsoft' BASIC, and a decision was made by its designers to speed up the search for variables at the expense of some wasted memory.

As the translator always starts from the beginning of the list of variables, programs can be speeded up by having the most commonly referenced variables defined early in the program.

If X is used before X$ then X will be before X$ in the list of variables. If, however, X$ is used far more frequently than X then (particularly in a more complex program) it makes sense to have X$ before X in the list of variables.

A short examination, of the way in which the variable names are stored, reveals that each of the two bytes given over to the name can either be 'high' (i.e. +128) or 'low' (i.e. +0). This gives four possible combinations. We have only three distinct types of variables, and so one of the possible combinations is not utilized.

The fourth combination is devoted to user-defined function names. Instead of a suffix (e.g. % or $), the functions are distinguished by a prefix (i.e. FN). The name and 'value' of a function together take up seven bytes.

The name of the function (stripped of FN) is stored in two bytes as usual, with the first byte containing the ASCII value +128. The second byte is the straight ASCII value. This is the fourth combination. To add any more data types requires a complete rewriting of the translator.

The next two bytes point to where the function is defined in the program. This is why the function has to have been defined within the program as it has run. If the function definition has not yet been reached, then an error will be generated.

```
10 DEF FNX(Y) = Y
20 W = FNX(8)
```

will work (and W will take the value 8), but

```
10 W = FNX(8)
20 DEF FNX(Y) = Y
```

will not work, because the name of the function will not be found.

The next two bytes also act as a pointer, in that they point to the variable which is used in the definition as the parameter (i.e. in FNX(Y), Y is the parameter). The final byte is unused.

The breakdown of the variable types, and the unique method by which the name is stored for each type, is

ASCII value		
Byte 0	*Byte 1*	*Variable type*
+0	+0	Floating point
+128	+128	Integer
+0	+128	String
+128	+0	Function definition

For the first three types, if a reference is made to a variable before it has been given a value, then that variable is set to a null value. For floating point and integer variables the null value is zero, and for strings the null value is the empty string.

For a function definition there is no null value, so an error is generated.

To

```
PRINT X, X%, X$, FNX(X)
```

with no program in memory, produces 0, 0, a nothing (the empty string X$), and then ?UNDEF'D FUNCTION ERROR. So be careful with functions.

This is why it is a good plan to put all the functions in one subroutine, which is called before anything else is carried out. In this way, all functions are initialized.

Arraying information

An array is an ordered collection of values, all of the same type, all called by the same general name. The collection is ordered in such a way that individual items within the collection are selected by their numerical position. (Compare MUM pages 95 to 98, and PREG pages 8 and 9.)

When the name of a variable is used, followed by an arithmetical expression in parentheses, the BASIC translator assumes that the name refers to an array. If the array has not been dimensioned, then the array is initialized to a null value.

The null value of each item within the array depends upon the type of array (i.e. floating point, integer, or string). The null/default number of items is set automatically to eleven, for all types of array. It is as if there was a default declaration

```
10 DIM X(10)
20 DIM X%(10)
30 DIM X$(10)
```

or—alternatively—

```
10 DIM X(10), X%(10), X$(10)
```

Arrays can have more than one dimension (up to the magical 255), and the number of items in each dimension can be up to (the equally magical) 32767. How an array of 255 dimensions, each with 32767 items, can be stored in memory is another problem.

Arrays are stored in memory with two bytes for the name of the array. It is possible to have an array called X as well as an ordinary variable X, because arrays are stored in a distinct portion of memory. The location at which arrays start (and variables have ended) can be found by

```
PRINT PEEK(47) + PEEK(48)*256
```

and the end of arrays (plus 1) is at

```
PRINT PEEK(49) + PEEK(50)*256
```

Following the two bytes for the name (with +0 or +128, depending upon type), there is a two byte pointer to the start of the next array. The pointer is needed because—unlike simple variables—each array will be of a different length. After those four bytes is one byte which gives the number of dimensions of the array (with the maximum being 255). The meaning of these five bytes is the same for all arrays.

The next two bytes contain the number of elements in the last dimension of the array. In the case of

```
DIM HW(1,900)
```

this will be 901. If there is more than one dimension, then the number of elements in the last but one dimension is contained in the next two bytes. This continues until there are no more dimensions.

It is at this point that the storage requirements for each type of array begin to diverge.

The arrays store the separate elements for each dimension in order, so that the values for all elements in the first dimension (commencing with element 0) are stored in succession. For example, in the case of HW, the elements are stored in the order HW(0,0), HW(1,0), HW(0,1), HW(1,1), . . . , HW(1,898), HW(0,899), HW(1,899), HW(0,900), HW(1,900).

In the case of floating point arrays each element occupies five bytes, as do ordinary floating point numbers.

Integers, however, only occupy two bytes and do not have the three spare bytes which one finds with ordinary integers. Storage for arrays of integers is thus more economical than for the storage of collections of ordinary integer variables.

Three bytes are needed for each element of a string array (apart from the contents of the string itself). One byte gives the length of the string, and two bytes act as pointer to the start of the string, in high memory. In the normal storage of strings there are then two spare and unused bytes, but for arrays there are no wasted bytes.

PREG (page 9) gives a way of calculating storage requirements for arrays, but without explanation. Allow five bytes for the name, length, and number of dimensions; two bytes for each dimension size; and then analyze the type of array.

For each element in a floating point array, allow five bytes; for each element in an integer array, allow two bytes; and for every element in a string array, allow three bytes for the length and pointer, plus one byte for each character in the string corresponding to that element.

The storage requirements for a string array can vary as the program proceeds, whereas the storage requirements for numerical arrays are fixed.

Stacking information

There are only two fixed locations (FPA #1 and FPA #2) where temporary floating point values may be stored, though the system needs rather more than two such locations. BASIC operates on two numbers at a time, but we often want to combine more than two numbers.

The problem of storing this information, information which is not needed immediately, but information which will be needed at some time, may be solved by storing values in temporary locations in main memory. To say BASIC will store values in main memory is not sufficient, for it has to keep track of where the values are stored.

To keep track of where the values are stored could be a major task in itself, so the system needs some automatic mechanism to keep track of the locations. We keep track by use of the 'stack'. On the C64 the stack is fixed in memory at locations 256 to 511, and it fills with values from location 511 downwards. There are, therefore, 256 locations which make up the stack.

The locations from 256 to 511 are common to all computers which use the MOSTEK 6502 microprocessor, though the C64 uses an improved version of the 6502 called the 6510.

Consider how the system will treat this arithmetical expression:

```
PRINT (1+(1+(1+(1+(1+1)))))
```

After examining the PRINT command, the system then examines the following expression to decide what has to be output.

First it meets the (, and it stores that information on the stack (it 'pushes' the information on to the stack). By the end of the evaluation of the expression, the routine expects to find a matching), otherwise there will be a ?SYNTAX ERROR. Next in the line is a 1, and the 1 is loaded in FPA #1.

When the + is encountered an addition routine is called, ready to add two numbers together. However, it is not a number which is next, but a (. The value from FPA #1 and the pointer to the addition routine are pushed on the stack, together with the (token.

This process is continued until the first) is encountered. At this point the two numbers and the operation are loaded from the stack (i.e. 'pulled' or 'popped') and the calculation performed. Successive encounters with) cause more information to be popped off the stack to be combined with information already in the floating point accumulator.

The stack is used not only for storing intermediate information for the evaluation of arithmetic expressions, but also for any circumstance in which the system needs to store temporary information, and where the order in which the information is stored is important.

These circumstances obviously fit for the evaluation of arithmetical expressions, but they also fit saving information about FOR loops and GOSUBs. For the latter two examples see later, when BASIC keywords are discussed.

The mathematical expression

```
PRINT 1+1+1+1+1
```

is not the same as

```
PRINT (1+(1+(1+(1+1))))
```

where the system is concerned. The second version takes much longer to process, because the stack is used extensively. The difference in time taken can be quite dramatic.

Run this program:

```
10 FOR I=1 TO 2000
20 X = 1+1+1+1+1
30 NEXT I
40 PRINT "#########"
```

which takes about 18.5 seconds from pressing RETURN to the output of #########. Modify the program slightly

```
10 FOR I=1 TO 2000
20 X = (1+(1+(1+(1+1))))
30 NEXT I
40 PRINT "#########"
```

and the time now taken is about 21.7 seconds. To use the more complicated assignment to X takes up about 17% more time.

According to PREG (page 15) there cannot be more than 10 sets of nested parentheses within an arithmetical expression, before there is no more room left on the stack. This is not quite true.

Enter

```
PRINT ((((((((((((1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1)+1)
+1)
```

and the answer is given as 13, though there are twelve sets of nested parentheses. However,

```
PRINT (1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+1)+1)
+1)+1)+1)+1)+1)+1)+1)+1)+1)+1)
```

gives an ?OUT OF MEMORY ERROR. The BASIC system has not run out of memory, as such, rather it has run out of memory locations on the stack. This example shows that where two parentheses are adjacent (e.g. "((") the routine used to evaluate arithmetical expressions acts differently from where they are separated by a value (e.g. "(1+(").

The C64 (and other machines using similar versions of BASIC) is thus fairly restricted in its ability to tackle complex mathematical expressions. Such expressions have to be analyzed into simpler component parts, and then combined to produce the result.

BASIC syntax for the C64

Arithmetical operators

There are only a few arithmetical operators in C64 BASIC, of which the most commonly used is the assignment operator (or 'equals', =).

It is called an 'operator' because it is a directive to do something—store the value of the expression on the right-hand side in the variable named on the left-hand side. For example, X = Y takes the value on the right-hand side, Y (which might be a complex expression), and stores that value in the variable named X, which is a simple variable.

The item on the left-hand side cannot be anything other than a simple variable, such as X, X(3,7), X%, X$(I), or X(I*2+1). The type of variable on the left-hand side has to be compatible with the result of the expression on the right-hand side. Possible assignments are

```
X = I%
X0 = X1
I% = X
I0% = I1%
S0$ = S1$
```

and even then there are restrictions. If X is greater than 32767, or less than −32768, then the assignment to I% produces an ?ILLEGAL QUANTITY ERROR.

Certain assignments which should be admissible are not admissible because of bugs. Consider the sequence

```
V = 1.70141183E+38 : PRINT V
V = (V/2)*2 : PRINT V
```

which prints out the correct value for V for the first PRINT, but gives a ?OVERFLOW ERROR for the second assignment—never reaching the second PRINT.

The meaning of the other arithmetical operators is fairly simple to divine, as their meaning is conventional. This simplicity is slightly misleading, because T+T is not (for example) the same as T*2 or 2*T.

Though these two processes are identical in arithmetic, they are not identical in operation on the C64. The addition sign is an operator, and it operates on two numbers to produce a result. The operation of adding two identical numbers together, however, does not always produce the same answer on the C64 as multiplying the number by two.

Multiplication is another operator, which does not work by repeated addition. In ordinary arithmetic our multiplication and addition operations are consistent with each other, but on the C64 (and many other computers) there are slight differences. For the C64, the differences can be dramatic.

The only arithmetical operator which is not totally obvious is the exponentiation operator, the up arrow ↑ . Try this

```
PRINT 15*15
```

to which the response is 225. Now

```
PRINT 15 ↑ 2
```

to which the response is also 225, and this indicates that the second value (i.e. that after the ↑) is the number of the times the first number is multiplied together. In other words, the number after the ↑ indicates the power to which 15 is to be raised.

Compare

```
PRINT 15*15*15, 15 ↑ 3
```

and both answers are 3375. Moving to a higher power,

```
PRINT 15*15*15*15*15, 15 ↑ 5
```

results in the numbers 759375 and 759375.001. The lack of equality is clear to see: the numbers differ by about .001. When we enter

```
PRINT EXP(5*LOG(15))
```

the number 759375.001 is output.

The power of a number is calculated by use of procedures similar to the EXP and LOG functions, and this explains why the answer is often slightly incorrect. The extent of inaccuracy may be gauged by this short program

```
10 FOR J=1 TO 10 : FOR I=1 TO 5 : REM J IS THE NUMBER
AND I IS THE POWER
20 N=1 : FOR K=1 TO I : N=N*J : NEXT K : REM N IS NOW
J RAISED TO THE POWER OF I
30 PRINT J,I,J ↑ I-N : REM J ↑ I-N GIVES THE ERROR
40 NEXT I : INPUT A$ : REM WAIT TO SEE THE RESULTS
FOR J, THEN HIT RETURN TO CONTINUE
50 NEXT J : REM NEXT NUMBER
```

Apart from the numbers (i.e. J) 1, 2, 4, and 8 (all powers of 2) there are always inaccuracies. The C64 is no worse than most other micros in this respect, but it has to be remembered that total perfection in calculation is impossible, and that the C64 has some extra little bugs.

Relational evaluations

The = symbol appears again when we discuss relational comparisons. This time it does not indicate assignment, rather it indicates an evaluation: does X equal Y?

To investigate the equality evaluation, try

```
I=567 : PRINT I=I, I/3*3 = I, I/3.7*3.7 = I
```

and the answers are −1, −1, and 0. If the result of an evaluation is 'true' (e.g. I=I) then the numerical value of the evaluation is −1. It is true that I=I and I/3*3=I (which is why we printed out the value −1 for both). It is not true (it is 'false') that I/3.7*3.7=I, and so the resulting numerical value is 0.

If we

```
PRINT I/3.7*3.7−I
```

the answer is −2.38418579E−07, and so we can either check for equality by subtraction or by a relational evaluation (i.e. =). In line 30, above, J ↑ I=N could have been used to check—though the size of the difference would not have been given by the relational evaluation.

The reverse of being equal is being not equal:

```
PRINT I/3.7*3.7 <> I
```

gives the result −1, it is true that the two numbers are not equal. This reversibility can save some complication in programming. Compare

```
10 INPUT N
20 IF N=100 THEN PRINT "EQUAL"
30 IF N<>100 THEN PRINT "NOT EQUAL"
```

with this program

```
10 INPUT N
20 IF N=100 THEN 50
30 PRINT "NOT EQUAL"
40 GOTO 60
50 PRINT "EQUAL"
60 REM
```

and you can see that to use two IF statements makes the first program easier to understand.

Another relational evaluation is > (i.e. 'greater than'), used in a manner such as

```
PRINT I > I/3.7 * 3.7
```

to which the response is −1 ('true'). It is true that I is greater than I/3.7 * 3.7. The reverse of > is not, however, < but rather it is <=.

If something is not greater than, it may be less than or equal. A common error is to have checks for > and for <; and miss the case when there are equal values.

As the result of a relational evaluation is a numerical value, these values can be used in arithmetical expressions.

Consider two variables X and Y: if Y is greater than X then the value of Y is stored in another variable Z, if X is greater than Y, then the value of X is stored in Z. In other words, store the maximum of X and Y in Z. If Y is the larger

```
Z = Y
```

and if X is the larger

```
Z = Y - (Y-X)
```

which is a way of writing Z is equal to X. If X is greater than Y we subtract (Y−X) from Y. To perform this subtraction why not

```
Z = Y + (X>Y)*(Y-X)
```

which, when X is greater than Y, produces (−1)*(Y−X)?

Techniques such as this can be quite useful, though sometimes confusing to the uninitiated. In one line we have two lines

```
IF Y < X THEN Z = X
IF Y >= X THEN Z = Y
```

where, notice, the reverse of < is >= (otherwise, when X=Y, there might be no assignment to Z).

The relational evaluators also work with characters, where the ordering on which the characters are evaluated is the ASCII code values for the characters (MUM Appendix F). The result of

```
PRINT "A" < "B" , "A" < ":"
```

is −1 and 0, because the ASCII code for "A" is 65, that for "B" is 66, and that for ":" is 58.

Examine the line

```
PRINT "A" < "AA", "AA" < "AB"
```

to which the response is −1, and −1. The ordering is not just ASCII for the first character in a string, rather the evaluation carries down the sequence of characters in the string until there is the first difference.

Logical functions

C64 BASIC has three logical functions: NOT, AND, and OR. The functions operate on two main types of input.

Examine this statement (where X is greater than Z)

```
IF X>Y AND Y>Z THEN MIDDLE = Y
```

and consider when the action MIDDLE = Y is activated. The condition is true when (1) The value of X is greater than the value of Y, and, in addition, (2) The value of Y is greater than the value of Z. That is, the overall condition is only true when both subconditions are true. If the overall condition is true, then Y is the middle value.

This is illustrated by what is known as a 'truth table'.

```
TRUE AND TRUE IS TRUE
TRUE AND FALSE IS FALSE
FALSE AND TRUE IS FALSE
FALSE AND FALSE IS FALSE
```

Which shows that when (and only when) both conditions are true, the overall condition is true. Thinking of individual bits in the computer:

```
1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0
```

The other main function is OR. The use of OR is shown by

```
IF X>=Y OR X>=Z THEN PRINT "X IS NOT THE SMALLEST"
```

Either if it is true that X is greater than or equal to Y and/or if it is true that X is greater than or equal to Z, then we know that X is not the smallest number. Note that the reverse of this condition is

```
IF X<Y AND X<Z THEN PRINT "X IS THE SMALLEST"
```

and try to work out why.

The truth tables for OR are

```
TRUE OR TRUE IS TRUE
TRUE OR FALSE IS TRUE
FALSE OR TRUE IS TRUE
FALSE OR FALSE IS FALSE
```

```
1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0
```

AND and OR seem to be the reverse of each other. If, in either bit table, all the 1s were changed to 0s, and vice versa, then one table is the same as the other. This simple observation is codified as De Morgan's Rules:

A AND B is the same as NOT(A) OR NOT(B)
A OR B is the same as NOT(A) AND NOT(B)

If the OR truth table for binary numbers is examined it will be seen that to make any bit equal to 1, the corresponding bit has to be ORed with 1. That is,

1 OR X = 1 (X=1 or X=0)

To leave the value of a bit unaltered we OR with 0:

0 OR X = X (X=1 or X=0)

To see how to use the OR function, enter

```
POKE 53265,PEEK(53265) OR 32
```

and funny things happen to the screen. The above line changes the display to bit map mode—at the moment this does not help, so hit STOP and RESTORE. To enter bit map mode, bit number 5 in location 53265 is changed to a 1.

Locations contain bytes, which are eight bits, and if you

```
PRINT PEEK(53265)
```

under normal circumstances, you find that the number stored at location 53265 is 27.

The decimal number 32 corresponds to the binary number 00100000. When we POKE into 53265, we POKE the value already there (i.e. PEEK(53265)) after ORing with 32. To OR with 32 is to set bit 5 to a 1, and leave all other bits as they were:

bbbbbbbb OR 00100000 = bb1bbbbb

To set bit 5 of the byte at location 53265 to a 1, is to activate bit map mode (used for high resolution graphics).

Now use

```
POKE 53265,PEEK(53265) OR 32 : FOR I=1 TO 1000 : NEXT I :
POKE 53265, PEEK(53265) AND 223
```

and the screen switches to the garbage for a short while, then it switches back.

Looking at the bit truth table for AND:

1 AND X = X	(X=1 or X=0)
0 AND X = 0	(X=1 or X=0)

To set a bit to 0, AND with 0, or to leave a bit unchanged, AND with 1. To change bit 5 to 0, and leave the other bits unchanged, AND with the binary number 11011111. This has the decimal value 223 (most easily calculated by 255−32).

In the above line of program, bit map mode was entered (by setting bit 5 to 1 by use of an OR), there was a short wait (the loop), and then bit 5 went back to 0 by use of an AND. To understand any PEEKs and POKEs, which include logical functions, remember OR to set to 1, and AND to set to 0.

In discussing floating point numbers, it was noted that they had a sign bit. Integers also have a sign bit, but the ways in which the sign bit works, though perfectly logical, seems strange at first.

An integer extends over 16 bits, and if the bit on the left (bit 15) is 0 then the integer is positive. If bit 15 is a 1, then the number is negative. As there are only 65536 different values from two bytes, a decision has to be made about how to store the absolute size of numbers.

Use two pieces of information:

$$-1 + 1 = 0$$

in normal arithmetic, and

$$65535 + 1 = 0$$

in two-byte arithmetic (something like a mileage register on a car, if the register is 999999, then one mile on it is 000000). Therefore −1 is equivalent to 65535 in internal arithmetic.

In binary arithmetic, 65535 is 1111111111111111, and as bit 15 is 1 (all bits are set to 1) this accords with the sign bit being 1. The binary number 32768 is 1000000000000000, and, as bit 15 is 1, the number is negative—it is equivalent to −32768. (Remember the limits on integers were −32768 to 32767?)

This method of storing positive and negative numbers is called 'two's complement' arithmetic, and explains why 'true' is −1. If some number is true then all bits should be true, and all those bits should be 1. The number with all bits set to 1 is 65535, which is treated as the value −1.

A bit is 'true' if it is 1, a value is 'true' if it is −1, and both are consistent with each other in two's complement arithmetic.

Remember that

```
NOT(-64) = 63
NOT(63) = -64
```

The number 63 is (as a sixteen-bit binary number) equal to 0000000000111111. To NOT every individual bit produces the binary number 1111111111000000, which as a two's complement binary number is equal to −64.

BASIC keywords

In the list of keywords which follows, there is no attempt to replicate the descriptions given in MUM (Appendix C) or in PREG (Chapter 2). The list does not give any of the logical function keywords, because their nature and use have been discussed in detail (above).

ABS(X) This function calculates the absolute value of the arithmetical expression in the parentheses. The expression can be floating point, integer, or logical, in nature.

ABS works by using the floating point accumulator #1 (FPA #1), and the routine moves all bits in the sign byte one place right, so that the sign byte then appears positive.

ASC(S$) Calculates the ASCII value of the first character of the string S$. The ASCII codes used are those particular to the C64, though for standard numbers, characters, and symbols, these are the standard ASCII values. The string can be the result of any legal set of operations to create strings.

The ASC function works by evaluating the string expression (e.g. it forms "AB" first in finding ASC("A"+"B")) and the string pointer (without name) is left on the stack. The ASC function examines the pointer, and loads the first character of the string for checking.

To apply the ASC function to an empty string (i.e. ASC("")) gives an ?ILLEGAL QUANTITY ERROR, another C64 bug and one which seems to come from BASIC on other Commodore machines. The null string (i.e. "") should have ASCII value 0, as can be seen by

```
PRINT CHR$(0);"#"
```

when the null string is output, followed (in the first column on the left) by #. (CHR$ is the reverse function to ASC.) However,

```
PRINT ASC(CHR$(0))
```

and the answer is 0.

The bug must arise from the use of the string pointers, because "" has a zero length, and the routine cannot accommodate a (null) string of length zero. CHR$ (and see later) always sets up a string of length one, even if null.

ATN(X) This function calculates the angle (in radians) which corresponds to a certain tangent value (the angle is called the arctangent). The resulting angle is always in the range −PI/2 to +PI/2, and if it is known that the result should be outside these limits, then modifications have to be made. (PI is the value of the special symbol which is produced on using SHIFT and the up arrow.)

4*ATN(1) should equal the value PI, and

```
PRINT 4*ATN(1) - PI
```

produces the result 9.31322575E−10.

Twice the arctangent of infinity is also equal to the value PI, and so

```
PRINT 2*ATN(1E38) - PI
```

where 1E38 is our approximation to infinity. The difference is again 9.31322575E−10, which shows that it is quite a close approximation.

The ATN function routine uses a special approximation formula, which works well in practice.

CHR$(X) This function converts any numeric expression, within the limits 0 to 255, into the corresponding ASCII character (if the expression is floating point, it is truncated by losing the decimal part). The character created is a string of length 1 byte, which is why the ASC function works successfully on CHR$(0).

The ASCII value for the character " is 34, and thus

```
PRINT CHR$(34); "DOUBLE QUOTES"; CHR$(34)
```

to have DOUBLE QUOTES in double quotes—i.e. "DOUBLE QUOTES".

CLOSE I% Used to 'close' files on devices, this command completes the full processing of a file, and deletes details about the file from the three special 'file tables' held in RAM. For further details see the keyword OPEN.

CLR This clears memory of variables, defined functions, strings, files, and clears anything left on the stack.

The command works by resetting a collection of pointers:

Pointer	*Location*
1. Start of BASIC program	43–44
2. Start of variables	45–46
3. Start of arrays	47–48
4. End of arrays (+1)	49–50
5. Start of strings	51–52
6. Temporary string storage	53–54
7. End of BASIC memory	55–56

When CLR is activated, pointer 1 is not changed and points to the same start to BASIC memory, and neither is pointer 2 changed, because that is where the BASIC program ends in memory. The temporary string pointer (6) points to the start of the last defined string in memory, and that is not affected either. The end of memory recognized by the BASIC system (i.e. pointer 7) is also left untouched.

Pointer 3 is set equal to pointer 2, and pointer 4 is set to one more than pointer 2. Pointer 5 is made equal to the top of BASIC memory (i.e. pointer 7), and the upshot of these changes is that the BASIC system is not aware of any variables or arrays.

The pointers can be investigated by a simple program

```
10 DEF FNP(I) = PEEK(I) + PEEK(I+1)*256
20 FOR T=43 TO 55 STEP 2
30 PRINT FNP(T)
40 NEXT T
```

which prints out all the pointer values in order. The addition of a line

```
15 FOR T=1 TO 35 : A$ = A$ + "!!!!!" : NEXT T
```

alters the readings for the string pointers more than any others.

The bottom of string storage is now much lower, and the difference between the two string pointers (5 and 6) is now 175—or the length of the last A$ (35*5). The space available for strings is taken up with all the different copies of A$—35 of them, starting with "!!!!!", then "!!!!!!!!!!", and so forth.

On entering

```
CLR : PRINT PEEK(51) + PEEK(52)*256
```

we find the location is near to the top of memory again. Pointer 5 has been reset by the use of CLR. To PEEK at locations 53 and 54 shows that the preceding CLR has not affected the temporary string pointer (number 6).

CMD I%, S$ This command initializes device number I% to take all further output, instead of the output going to the screen. The device has first to be activated by an OPEN I%, D% command, where D% is the actual device number, and I% is the file number. After the CMD command the screen is dead for output, until it is reactivated by

```
PRINT# I%, ; : CLOSE I%
```

which causes the device with file number I% to 'unlisten'. PRINT# I% allows printing to file number I%, but will still (for example) allow the program to be LISTed to the screen. CMD I%, however, takes any output (including a LIST) and sends it to the device with file number I%. (See PRINT# for further details.)

If the second (string) parameter is present then the contents of the string are output to the file specified.

CONT This restarts a BASIC program after a STOP or END in the program, or after the STOP key has been pressed. If a new program line (with line number) has been entered, CONT will not work, as pointers have been altered. They can also alter if there is a syntax error, or a CLR command.

Not all commands alter pointers, and so it is possible to LIST and then use CONT.

COS(X) This function calculates the cosine corresponding to the angle X, where X is assumed to be in radians. If the angle is in degrees, then it has to be multiplied by PI/180, e.g.

```
PRINT COS(DEGREES*PI/180)
```

The expression within the parentheses is evaluated, the result stored in FPA #1, and then the SIN routine is called. The value of COS(X) is equal to SIN(X+PI/2) and so the SIN routine operates on the expression plus PI/2.

DATA Frequently there are, within a program, certain pieces of information which never change from one operation of the program to another. For example, at the level of the machine code routines to calculate the SIN, there are several fixed values which are used each time the SIN function is called.

For the SIN, these fixed values are stored permanently in the BASIC ROM (Read Only Memory) and looked at each time the routine is used. In a BASIC program, use is made of DATA statements.

There is a permanence about the use of DATA statements which means that a mistake in the data is more easily rectified. To use INPUT statements in the processing of statistical data, for example, leaves the way open to errors of input. With DATA statements one can rectify errors by altering the appropriate statement(s).

An error in an INPUT part way through a program often means that a completely new start has to be made.

The items in a DATA statement can be anything one expects to be allowed in INPUT statements: the way in which DATA statements are READ is by use of the same set of routines as INPUT, so the same restrictions apply. An important point to remember (true for both DATA and INPUT) is that a trailing comma is taken to be either a number of value zero, or a (null) string of value "".

An important point to remember is that in the case of strings, where the strings contain commas, the string as a whole has to be enclosed in quotes. For example,

```
1000 DATA "A STRING, INCLUDING AN INTERNAL
COMMA"
```

Errors reported in DATA statements are usually mistakes in the READ statement—but see later. Placing the cursor over a READY. and pressing RETURN will produce an ?OUT OF DATA ERROR, because the BASIC system thinks READY. is

```
READ Y .
```

DEF FNYZ(X) This is the way in which user-defined functions are entered. The parameter has to be a floating point variable and strings are not allowed.

To enter 10 DEF FNII(H%) = X and then RUN produces a ?SYNTAX ERROR IN 10. A ?TYPE MISMATCH ERROR is produced if a string is used as parameter: it is invalid to define DEF FNST(A$) = ASC(A$). Only one real variable is allowed as a parameter to a function, and though the parameter may not be used, there always has to be a parameter there.

DIM X0(X1) This statement allocates space in memory for an array of the specified name, with X1+1 elements. Array elements are numbered from zero to the number in parentheses after the array name.

The example shown is a floating point array, but integer and string arrays are also possible.

The same array cannot be dimensioned twice in the same program. For a more complete examination of the topic of arrays see above.

END Ends a program, and prints READY. The program can be restarted by CONT, if certain conditions are fulfilled (see CONT, above). Compare to STOP.

EXP(X) Calculates e (2.718281828. . .) raised to the power of the value of X. For example,

```
PRINT EXP(1)
```

returns the value 2.71828183, and

```
PRINT EXP(2)
```

returns the value 7.389051.

```
PRINT EXP(1)*EXP(1), EXP(1) ↑ 2
```

returns the value 7.3890561 for both expressions.

Note that

```
PRINT LOG(EXP(X))
```

should give the value X. This program tries to check the accuracies for various X values:

```
10 FOR X=0 TO 10
20 PRINT X, LOG(EXP(X))-X
30 NEXT X
```

and only for X=3 does there seem to be any error. The error for X=3 is the (ubiquitous) 9.3122575E−10.

As a check of a different kind, it is worth remembering that EXP(88) should equal EXP(87)*EXP(1). Try to

```
PRINT EXP(88)
```

and the result is 1.65163625E+38. The (numerically) equivalent statement

```
PRINT EXP(87)*EXP(1)
```

produces—as might be expected—?OVERFLOW ERROR. The C64 BASIC arithmetical bugs strike again.

FOR X0 = X1 TO X2 STEP X3 This sequence permits the repeated processing of the lines of BASIC between this statement and the corresponding NEXT X0 statement.

The loop variable (i.e. X0 in this example) has to be a floating point variable, and not an integer. The values to start, finish, and increment the loop variable (i.e. X1, X2, and X3, in that order), need only be arithmetical expressions.

The loop variable not only has to be floating point, but it has also to be an ordinary variable—this means that X is admissible, but the complex variable X(5) is not allowed. As the loop counter is a floating point variable, there may be some errors of rounding. For example,

```
FOR I=1 TO 2 STEP .0001 : NEXT I : PRINT I
```

gives the result 2.0000983, when the answer should be 2.0001. The reason why the final value of I should be 2.0001, and not 2.0000, is made clearer by

```
FOR I=1 TO 2 : NEXT I : PRINT I
```

because the final value of I is 3.

When the BASIC translator meets a FOR statement it takes the first value for the loop counter, and always executes the loop at least once. In the case of I=1 to 2 the value of I is incremented by 1 when the NEXT I statement is reached. This incremented value is then checked against the loop limit, and if the loop variable is greater than the loop limit the loop stops.

After one activation of the loop, I=2 and this is not greater than the loop limit (which is 2), so the loop acts again. NEXT time I=3 and this is greater than the loop limit, so the loop ends—with I equal to 3 and not equal to 2.

If the loop start is greater than the loop limit (i.e. X1 > X2) then we would suppose that the STEP value (i.e. X3) must be negative. However,

```
FOR I=1 TO 0 : PRINT I : NEXT I : PRINT I
```

outputs the values 1 and 2, and then the loop ends. Even if by some mistake the start and limit values are incorrect, the loop always works once. The loop

```
FOR I=1 TO 2 STEP 0 : PRINT I : NEXT I
```

never ends, and a continual stream of 1s is printed. This is a way of producing an endless loop, or a loop which operates until something is the case. Here is a program which operates until the F key is pressed.

```
10 FOR I=1 TO 2 STEP 0
20 PRINT "#";
30 GET X$
40 IF X$ = "F" THEN I = 2
50 NEXT I
```

There is an endless loop from lines 10 to 50, which outputs #s to fill the screen. At line 30 there is a check of the keyboard to see if a key has been pressed, where the result of the enquiry is stored as a character in X$. Line 40 is the equivalent of an UNTIL statement.

IF the character stored in X$ is an F THEN the value of I is made equal to 2. When the NEXT I statement is reached, I will equal its limit parameter, and the loop will terminate.

A problem with this form of control statement is that it takes a great deal of space on the machine's stack to store information about each of the parameters (STEP is always included, even if it is the default value 1). 'Nested' loops (i.e. loops within loops) eat up stack space which will soon run out. One FOR. . .NEXT loop can use up to 18 bytes on the stack.

If there are too many FOR loops (or too many GOSUBs—see later) there may be an ?OUT OF MEMORY ERROR. Memory has not, in fact, run out, rather the stack is too full.

FRE(X) Calculates the number of bytes available to BASIC. It takes the value of the pointer to the end of arrays (see above for CLR), and the value of the string pointer, which points to the end of strings.

Before examining the pointers, FRE clears out of memory any strings to which there are no pointers from within the variables and arrays. This means that if there are several strings which are no longer active, only the active strings are considered, and kept. The latest examples of the active strings are placed together, in sequence, in memory from high memory down.

Enter

```
10 FOR I=1 TO 2 STEP 0
20 PRINT "#";
30 N$ = N$ + "!!!!"
40 NEXT I
```

and then (after the program has stopped with ?STRING TOO LONG ERROR IN 30)

```
PRINT PEEK(51) + PEEK(52)*256
```

will give some value around 30000 to 35000. To then enter

```
PRINT FRE(0), PEEK(51) + PEEK(52)*256
```

will give a negative value for FRE(0), and the new end of strings will be greater than 40000.

The difference in two values for the start of strings is an indication of the space wasted by making successive copies of N$. FRE(0) liberates that space, by a process known as 'garbage collection'. The parameter for FRE is not used in any way.

The reason why the result of FRE is negative, is that FRE gives the free space as a two's complement number. To change the negative value into the correct positive value, we use the fact that, if FRE is negative, (FRE(0) < 0) is equal to −1. (For more details on this use of relational evaluations see above.)

Use

```
PRINT FRE(0) – (FRE(0) < 0)*65536
```

to work out the ordinary value, from the two's complement version.

GET S$ This command tries to read one character from the keyboard (e.g. see FOR program emulating UNTIL, above).

As characters are typed in at the keyboard, they are stored in a portion of memory called a 'buffer' (from location 631 to location 640). As each character is used (stored as its ASCII value) all the values move along one byte, and a new character can be stored in the buffer.

In location 198 is stored the number of items in the keyboard buffer, that is, values from 0 to 10—the maximum number of values which can be stored in the buffer is 10. The number of elements in the keyboard buffer can be altered by POKEing in location 649.

```
PRINT PEEK(649)
```

normally gives the result 10.

The GET routine has a special facility which uses only the first element in the keyboard buffer. INPUT copies elements from the keyboard buffer into a system input buffer (from 512 to 600) and waits until it finds an ASCII value 13 (i.e. a carriage return). If the GET command expects a number (e.g. GET X) and a letter is entered, then there is a ?SYNTAX ERROR. This is why it is better to GET S$, and then use VAL(S$)—see later. If no key is pressed while the GET command is active either a null string, or a zero numerical value, is returned.

GET# I%, S$ Works very much like GET, but reads characters singly from the file specified instead of the keyboard. The video screen is device #3, and if this program is entered

```
 1 OPEN 1,3
10 GET# 1,S$ : PRINT S$; : IF S$ <> "$" THEN 10
```

and the program listed, when the cursor is put at the top of the screen, to RUN the program alters what is on the screen.

Many duplicated letters and symbols appear where there were characters on the screen. What has happened is that device number 3 has been assigned to the logical file number 1 by the OPEN command (see later). By positioning the cursor at the top of the screen, and then typing RUN, the screen is treated as file, and each position (in order) is examined.

As the content of the screen location obtained by GET# is then printed out, one on from where it was found, the symbols repeat. There are quite a few $ symbols in the listing of the program, and one is almost certain to be encountered, but until a $ is found the program repeats line 10.

GOSUB number This command jumps (like GOTO) to any line in a program, but if the line number does not exist there is an ?UNDEF'D STATEMENT ERROR.

Unlike the GOTO statement, the GOSUB hands control over to another portion of program, expecting that control will return to the program statement following GOSUB. For the control to be returned, there has to be a RETURN statement.

First enter

```
0 PRINT "0"
1 PRINT "1"
2 RETURN
```

and then investigate the (theoretically nonsensical) calls

```
GOSUB
GOSUB X
GOSUB GOSUB
GOSUB 0X
```

all of which are interpreted as subroutine calls to line 0, as is indicated by both the numbers 0 and 1 being output. This action is due to the fact that the operative routine to scan the number following GOSUB is a close relative of the VAL function.

Enter

```
    GOSUB 1/2
    GOSUB 1GOSUB
    GOSUB 1X
```

for which the response to each is 1, showing that all jumps are to line 1. This is again as one would expect, given the behaviour of VAL (see later). The routine which examines the content of the line following GOSUB (called CHRGET, held at locations 115 to 138) stops at the first non-numeric character.

GOSUB uses up 5 bytes on the stack, which includes a pointer to where the next character is to be read after the return, the current line number, and a special token to indicate that a subroutine has been called. To investigate how many subroutines may be on call at one time, use

```
    10 INPUT N
    20 F = 1
    30 GOSUB 100
    40 PRINT F
    50 END
    100 IF N=0 THEN RETURN
    110 N = N-1
    120 GOSUB 100
    130 N = N+1
    140 F = F*N
    150 RETURN
```

The subroutine at lines 100 to 150 calculates the factorial of N, returning the result in F. The subroutine is recursive in that it calls itself (at line 120). Values of N from 0 to 22 work (and the factorial of 22 is 1.12400073E+21), but at an input value of N=23, there is ?OUT OF MEMORY ERROR IN 100.

The subroutine shows an interesting use of two RETURNs. This is one way in which subroutines can be more flexible than GOTOs.

GOTO and GO TO Both these are direct jumps to the line number which follows the command. GOTO is not the same as the two separate words GO TO, though the effect is the same.

The BASIC translator treats GOTO as one unit but it treats GO TO as two units. GO is counted as the keyword, and when treated as a keyword it has to be followed by a space and TO. The internal code for GOTO is different from the code for GO.

The evaluation of the line number following GOTO uses the VAL style routine, as does GOSUB. Thus GOTO has similar confusions to GOSUB.

With GOSUB, the program runs more quickly if we have the most commonly used subroutines near the beginning of the program, which is why many programs start with a GOTO too near the end of the program, jumping over the frequently used subroutines. As subroutines are designed to be used from many different places in the program, it makes sense to try to pick the best average position.

The optimization of the search for line numbers when using GOTO, can have some interesting consequences—if taken to extreme lengths.

Rule 1: never jump to a line number which is before the line at which you make the call. The line number routine checks to see if the line number to which the jump is made is before or later than the present line. If the line number is later, then the search starts from the present line, and not from the beginning of the program.

Rule 2: never jump to a line which is just before the present line, jump to a line near the beginning. The closer the line is to the start, the sooner the line is found.

Rule 3: do not take rules too seriously.

In association with the IF statement, there are three possible forms of the GOTO command:

```
IF condition THEN GOTO linenumber
IF condition THEN linenumber
IF condition GOTO linenumber
```

and IF... GOTO... is faster than IF... THEN...; but IF... GO TO... is inadmissible (it produces a ?SYNTAX ERROR). This is due to the different interpretations of GOTO and GO TO; and to use IF... THEN GO TO... does not produce an error.

It is best to keep clear of GO TO, as this keyword seems to be an afterthought, and prone to bugs.

IF. . . THEN This conditional statement has two principal functions. First, it can be used to branch to a program line; or, second, it can be used to execute statements on the same line as IF. The accent is on 'the same line', so that the immediate statement

```
    IF X=X THEN : PRINT "#####"
```

will output the #####, as will

```
    IF X=X THEN PRINT "#####"
```

In the first case, however, there is no statement to follow THEN, as THEN is immediately followed by the statement separator (i.e. the colon). For a third case, try

```
    IF X<>X THEN : PRINT "#####"
```

and nothing appears on the screen.

The routine associated with IF evaluates the expression following IF, and then (provided there is also a THEN or a GOTO) the exponent of the value in FPA #1 is checked. The value in FPA #1 is the result of the expression after IF, and if the exponent is zero, the result is zero, and control jumps to the next line number, not the next statement on the same line.

If the condition is true, therefore, all the statements on that program line will be activated; if it is false, all the rest of the statements on that program line are ignored. (Compare case 1 and case 3.) There need not be any statement to follow THEN.

INPUT Has many features in common with READ (below) particularly the importance of commas, quotes, and colons.

Following the word INPUT, there may be a string to identify the input required, e.g.

```
    10 INPUT "TESTING "; A$
```

and two extra lines help to investigate the INPUT command

```
    20 PRINT A$
    30 GOTO 10
```

So try the following inputs

```
J
J,
,J
J:
J;
"J,
```

For the first entry J is printed, as with the second entry—though with the information ?EXTRA IGNORED. The routine to accept values for INPUT considers that the comma is a separator, and thus the user has entered two values. Only one value is expected, and so the extra value is ignored (it is not possible to store up values for later INPUTs).

To the third entry a null string is printed, and the extra (i.e. J in this case) is ignored. As the first item the routine encounters is a comma, it assumes a null input. The fourth entry shows that the colon has a similar effect to the comma, as J is printed and the extra ignored.

The fifth input (i.e. J;) is output exactly as that (i.e. J;). The semicolon is not a separator, in fact it is treated as an ordinary character.

The final entry shows the use of the double quotes to make the routine aware that the following set of characters is just that, a set of characters. If there is a comma (etc.) in the set, it is to be counted. In this case the output is J,.

To illustrate a bug in the INPUT routines merely hit RETURN after the last item, and the string output is J, again. Hitting RETURN on input does not enter the null string, but the string contains its previous value.

The use of quotes in INPUT allows the input of graphics commands (for example) such as CLR, and in this respect the situation has much in common with PRINT.

Change two lines in the program,

```
10 INPUT "TESTING ";A
20 PRINT A
30 GOTO 10
```

and then enter

```
1
2E3
2,
,8
5:
2/3
```

The first is non-problematical, and 1 is output; the second entry is also without any problems, and 2000 is output. The third entry gives ?EXTRA IGNORED, and then the answer 2 (the fifth entry 5: is similar). The result of entering the fourth entry is ?EXTRA IGNORED and the value of A output is 0—the comma is read as a null (i.e. zero) entry.

The input 2/3 is illegitimate, and an error message ?REDO FROM START is output. To then enter a valid value is to output the valid value. If, however, 2/3 is entered (and the error flagged) to then merely hit RETURN prints out the value 2.

This is yet another bug. What has happened is that the expression 2/3 had been evaluated as far as the / by CHRGET (in a similar way to VAL or GOSUB/GOTO), and then the error flagged. By hitting RETURN, no value is sent to A and the already existing value (i.e. the 2 of 2/3) is treated as the proper value.

When a user is entering much information, it is very easy to make such an error, and hit RETURN before the proper time.

Sometimes, when a CMD statement is operative, the INPUT tries to take data from the wrong kind of device (e.g. a printer), and ?FILE DATA ERROR is output. If there is a message with the INPUT (e.g. TESTING) then this message is sent to the device, which may produce some complications.

INPUT# This command takes data from some device, in exactly the format expected by the ordinary INPUT command, though there is no message. The file must first be OPENed.

If the data has been sent to the device by PRINT#, the format is exactly as desired because the two commands are consistent.

INPUT# is rather more touchy about data types and format than ordinary INPUT, and, though there are no warnings, extra will be ignored. The routines for INPUT and INPUT# are almost identical, apart from the setting of the file for INPUT#.

Both INPUT and INPUT# use the 80-byte buffer (see GET and GET#), and this is why INPUT cannot be used in immediate mode—immediate commands are stored in the same buffer.

INT(X) This function converts the floating point expression in parentheses into the integer value which is less than or equal to result of the expression. None of the standard restrictions on the size of the floating point expression hold (i.e. within the bounds −32768 to 32767), because the result of INT is still a floating point number, though turned into a whole number.

Note the results of the following expressions

```
PRINT INT(3.1)
PRINT INT(−3.1)
PRINT INT(4.6 + .5)
PRINT INT(−4.1 +.5)
PRINT INT(1/2)
PRINT INT(−1/2)
```

which are 3, and then −4 (INT always rounds down); next are 5 and −4 (rounded to the nearest whole number); and the next pair produce the answers 0 and −1 (note that the INT of −1 divided by 2 is −1).

The rounding down is shown by reference to the integer division

```
X%=−3 : Y%=2 : Z%=X%/Y% : PRINT Z%
```

to which the answer is −2. The two's complement form of the number −3 is 1111111111111101, and to divide a binary number by 2 we move all bits to the right, inserting a 0 at the leftmost position (e.g. 4 is 100 as a binary number and 2 is 010). Thus the number becomes 0111111111111110.

This number is now a positive number and in two's complement is 32766. It is a poor system in which half a minus number is a plus number, so to divide the sign bit reinserted in the leftmost position. The binary number formed from the division by 2 is thus 1111111111111110, and as a two's complement number is equal to −2.

The actual INT routine works by taking the result from FPA #1, converting it to a four-byte integer, and then converting the four-byte integer into a floating point number in FPA #1, keeping the old exponent.

LEFT$(S$,I%) This function takes the string S$ and extracts the first I% characters of that string. The value of I% can be from 0 to 255, and if it is greater than the length of the string all the string is returned. For example,

```
PRINT LEFT$("1234", 6), LEFT$("1234", 2)
```

outputs the string 1234 and then 12. If the length is zero, then the null string is output.

The routine works by taking the string pointers off the stack, where they are pushed as part of any string evaluation. The length of the string is compared to the value (I%) provided by the function, and the lower value is taken. The string selection routine then uses this information (which has been replaced on the stack) to set up the new string.

LEN(S$) This function finds the length of the string S$ by effectively using the byte which gives the length of the string. The length byte is popped off the stack.

LET Is not needed.

LIST This command shows the content of part or all of a program, in a form which is similar to that in which programs are entered.

There are the following alternative methods of LISTing a program:

```
LIST
```

will display all the program;

```
    LIST 200
```

will display the content of line 200;

```
    LIST 600–800
```

will display all lines from 600 to 800 (inclusive);

```
    LIST –800
```

will display all lines up to and including 800; and

```
    LIST 600–
```

will display all lines from 600 to the end of the program.

If a program has been STOPped in some manner, it is possible to LIST and then to CONT. However, if the LIST command is within a program, e.g.

```
    1  PRINT "$$$$$"
    2  REM
    3  REM
    4  LIST
    5  PRINT "#####"
```

to RUN this program, LISTs it out, but the program does not continue to line 5 (and the PRINT). To use LIST within a program stops all further computation. If you manage to STOP the listing part of the way through (easier with a longer listing), CONT then moves control to line 5, but the rest of the LIST is lost.

When CONT is used after the program has ended, the LIST is reactivated, and the program is listed again. However, the whole program is not re-run, because the first line of $$$$$ is not output—only the LIST. The CONT should start the program from where it 'finished', but using LIST confuses the issue, and LIST is repeated (and stops the program).

If line 4 is erased, and the program RUN, then to use CONT does nothing—apart from the system telling you it is READY. To alter line 4 to

```
    4  LISY
```

produces a ?SYNTAX ERROR IN 4; and then trying to CONT, gives ?CAN'T CONTINUE ERROR.

The reason for these peculiarities of LIST is that its use within a program involves extensive use of pointers, and, if CONT is used, CONT tries to use those same pointers. The status of the pointers becomes unclear. This is another bug, not serious, but one which should not happen with a clean BASIC.

It is possible to LIST to a device by OPENing a file and a device and then using CMD (see above).

LOAD This command is the reverse of SAVE, and takes a stored file (usually on disk or tape) and loads a copy of the file into memory. Normally the copy will be a BASIC program, but it can be a machine code program, or a copy of the high resolution screen, or any other such continuous set of memory locations.

Some of the more complex SAVEs use one of the KERNAL system routines, also known as SAVE. This routine is activated by SYS(65496) in conjunction with other KERNAL routines. The KERNAL is a set of machine code system routines which can be used by the programmer, and to use them usually requires knowledge of machine code.

To SAVE machine code and sets of memory locations therefore requires rather more knowledge of KERNAL routines than is possible to give here. There are more details given in PREG (pages 270 to 306).

With simple BASIC programs, the operation of LOAD can be considered in two main areas: tape and disk. In the following, where there is a string in quotes, it is possible to use a string variable (e.g. S$).

In the case of tape (i.e. cassette) all that is needed is LOAD, but there are variants,

```
LOAD
LOAD "PROG"
LOAD ""
```

the first example will LOAD the next program in order on the tape; the second will LOAD the next program on the tape which begins with PROG (e.g. PROG1, PROG2, or PROGRAM); and the third example will also LOAD the next program on the tape.

The variants for LOAD, when using disk, are not that great in number

```
LOAD "0:PROG",8
LOAD "1:*",8
LOAD "PROG*",8
LOAD "1:PRO??",8
LOAD "PRO?*",8
```

Notice the compulsory ,8 parameter to indicate that disks are to be used, and note the drive number parameter (i.e. the number followed by a colon, which precedes the name of the file).

The first LOAD takes a program called PROG from disk drive 0; the second command takes the first file from disk drive 1. The * means that any (or no) characters can be substituted in that position, to form the file name.

The third LOAD is the equivalent to the simple "PROG" name for tape, PROG1, PROG2, or PROGRAM, will satisfy the name—the first file in order on the disk, which starts with PROG, will be used. As the disk drive number is not specified, it is assumed (by default) to be drive #0.

The fourth command will accept PROG1, PROG2, PROZZ, or PROBE, because they each have five characters beginning with PRO—the first is taken. The final command will take a file off disk drive 0, and will take the first file which starts with PRO, and has at least four characters.

The LOAD command can be used both within a program, as well as in immediate mode. In the case of LOAD being in a program there are no instructions on the screen, unless the tape unit is not on "play", in which case there is the message PRESS PLAY ON TAPE. For disk there are no such messages.

The program is not only LOADed, it is also RUN: in this respect it is like the automatic load and run given by SHIFT and RUN/STOP. If the program which LOADs the other program is longer than the program loaded, then it is possible to pass values from one program to another, except strings and functions (for obvious reasons).

For example,

```
10 REM
20 REM
30 REM
40 A=10
50 B=30
60 I%=6
70 PRINT A,B,I%
80 REM
90 LOAD
```

will output the values 10, 30, and 6, so quickly you cannot see them, before the screen blanks to load in the next program. If the next program on the tape is

```
70 PRINT A,B,I%
```

then the values 10, 30, and 6, are again output.

LOG(X) This function calculates the natural logarithm (i.e. to base e) of any positive, non-zero expression. This function is the reverse of EXP (see above).

The function first tests for zero or negative values of the expression, and if the result is zero or negative then it produces an ?ILLEGAL QUANTITY ERROR. The result of the expression is left in FPA #1, and so if that value is too large (e.g. 1E39) there is an ?OVERFLOW ERROR.

The function is evaluated by use of a short series, whose constants are held in the BASIC ROM.

MID$(S$,I0%,I1%) This function selects a substring from the string S$. The starting position is given by I0%, and the length of the string is given by I1%.

If the length is not provided, or is greater than the number of characters to the end of the string, then all characters to the right are returned. This makes it similar to RIGHT$ (see later). LEFT$(S1$,I3%) is equivalent to MID$(S1$,1,I3%).

Admissible values for I0% are from 1 to 255, and values outside this range give an ?ILLEGAL QUANTITY ERROR. All values within the limits 0 to 255 are admissible for I1%, and outside this range they produce the same ?ILLEGAL QUANTITY ERROR.

The default value for the third parameter is 255, unless the parameter (i.e. I1%) is set to a value (obviously, no string can be longer than 255 characters). The correct substring is then found by manipulation of the string pointers, which are pushed on the stack. (Effectively, the LEFT$ routine is used.)

If the string is S$="" then MID$(S$,1,255) is not an error, just a null string, and if the middle parameter is changed to any value then a null string is returned. This is true for any string for which the value of the middle parameter is greater than the length of the string.

This does not make sense, and can be shown by

```
10 A$ = ""
20 PRINT MID$(A$,2,255)
30 A$ = "#"
40 PRINT MID$(A$,2,255)
```

which does not produce any errors but is idiotic. To show how idiotic, try

```
10 A$ = ""
20 PRINT MID$(A$,0,255)
30 A$ = "#"
40 PRINT MID$(A$,0,255)
```

which gives an ?ILLEGAL QUANTITY ERROR IN 20. In line 20, A$ does have a 0 length, but—as with ASC—the coding of the string routines seems to be rather suspect.

NEW This appears to erase the program from memory, but really it only changes a few pointers (see CLR), so that they point either to the start of BASIC or the end of BASIC memory. The actual program is unaltered and can be examined by changing some pointers back (a tedious task).

The changing of pointers is the basis for most systems or tool kits which have an OLD command.

NEXT X Signals the limit of a loop. Control is either passed back to the corresponding FOR, or—if the counter is now greater than the loop limit—to the next statement.

The command has an optional parameter, WHICH SHOULD NEVER BE OPTIONAL. As BASIC can become rather confused, any assistance in debugging (such as telling the system which loop is which) should not be ignored. To leave out the loop counter from a NEXT statement is exceedingly untidy, and very silly.

Not to use a loop counter after a NEXT does speed up the program, but if the program does not work then debugging will be that more difficult. The program

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 REM
40 NEXT I
50 REM
60 NEXT J
```

is wrong, as is this program

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 REM
40 NEXT
50 REM
60 NEXT
```

The first program will not run, but the second actually will—change those REMs for active statements and the second program is a recipe for disaster.

ON I% GOTO . . . ON I% GOSUB The ON command is a switching command. The value of the variable which follows the word ON directs which of the lines to follow GOTO or GOSUB is to be used.

ON can be used in two forms (in either program mode or immediate mode):

```
    10  ON X GOTO 200,300,400,500
    20  ON Y GOSUB 2000,3000,4000,5000
```

and the value of the expression following ON has to be within the limits 0 to 255.

Above, if X is 0 or greater than 4, then control passes to the next line (as there are only 4 lines from which to choose). Negative values for the expressions give ?ILLEGAL QUANTITY ERROR. If there is no such line number as one in the list, then there is ?UNDEF'D STATEMENT ERROR IN 50 when a jump is made to that line.

The alternative form GO TO cannot be used, yet another reason to ignore it.

OPEN F%, D%, A%, S$ This command sets up details of a file, its corresponding device number, and secondary address, in tables in memory, each of ten entries.

The first table extends from locations 601 to 610, and contains the logical file numbers as assigned by the OPEN command (i.e. F%). The table is given the label LAT in PREG (page 316).

The second table has ten entries, each of which corresponds to a logical file number. This table (FAT, PREG (page 316)) extends from locations 621 to 630, and contains the device numbers (i.e. D%) which correspond to the file numbers.

The final table (called SAT by PREG (page 316)) contains the 'secondary' address for each file number, and device. The table extends from locations 621 to 630. Each location (i.e. A%) corresponds with a file number, and device number.

At this point

```
    PRINT PEEK(601), PEEK(611), PEEK(621)
```

to which the response is 0, 0, and 0. The initial value for each table is 0.

Now enter

```
OPEN 3
```

be told PRESS PLAY ON TAPE, and then discover FOUND XCXC, where XCXC is the name of some program. If we then LIST, we find that the program has not been loaded.

This method reads the next header on cassette, without actually having to read in the whole program. To continue to read headers, CLOSE 3, and OPEN 3 again. OPEN 3, without CLOSEing, produces ?FILE OPEN ERROR.

To enter

```
PRINT PEEK(601), PEEK(611), PEEK(621)
```

produces the output 3, 1, and 96, and to enter

```
PRINT PEEK(184), PEEK(185), PEEK(186)
```

results in the output 3, 96, and 1. These three locations (LA, SA, and FA according to PREG (pages 314–315)) are the current values of the latest OPENed file.

```
OPEN 4,0,0
PRINT PEEK(602), PEEK(612), PEEK(622)
PRINT PEEK(184), PEEK(185), PEEK(186)
```

and the result is 4, 0, 96, from the first PRINT and 4, 96, 0, from the second.

OPEN F% is equivalent to OPEN F%, 1, 0, "". Where 1 is the default second parameter (and corresponds to cassette number 1); 0 is the default third parameter (and corresponds to an injunction to the cassette to read a file); and "" is the default fourth parameter (and corresponds to no title for the file).

For each device there are secondary addresses, and the secondary addresses control the device's use.

For example,

```
OPEN 8,1,1,"+++++"
```

produces the response PRESS RECORD & PLAY ON TAPE, and something is sent to tape. To catalog the tape by

```
CLOSE 3 : OPEN 3
```

produces SEARCHING and then FOUND +++++. The fourth parameter gives the title to a file, if the secondary address indicates that the device is open for writing.

Device	*Number*	*Secondary address*
Keyboard	0	None
Cassette #1	1	0 = Read, 1 = Write to file plus end-of-file marker, 2 = As 1 plus end-of-tape marker when CLOSEd
Cassette #2	2	As for cassette #1
Screen	3	None
Printer	4	Varies with printer
Modem	5	None
Disk Drives	8	0 = Directory read, 1 = Directory write, 15 = Error channel

All other device numbers (up to the maximum 15) are unassigned, and secondary addresses take values from 0 to 15. To read or write data to a disk requires, therefore, the use of secondary addresses 2 to 14 (inclusive). As different disk systems vary, it is usual to make the string parameter contain disk commands, concerning reading and writing.

PEEKing at the secondary addresses, shows that they start at 96 and extend to 111. To obtain the proper value for the secondary address:

```
PRINT PEEK(621) AND 15
```

will give the correct answer (as will PEEK(621) – 96).

PEEK(X) This function provides the value of the contents of a byte at the location specified within parentheses. The value is returned as a decimal number within the range 0 to 255. The location has to be within the limits 0 to 65535, otherwise an ?ILLEGAL QUANTITY ERROR is produced.

When used, the routine takes the value stored in locations 20 and 21, and pushes that value on the stack. These locations, known jointly as LINNUM (PREG page 311), are used for temporary storage of line numbers, and the pushing of the value is a safeguard.

LINNUM is then loaded with the value of the parameter (it is converted to a two-byte integer from FPA #1). Routines are then called to load the value from that memory location into FPA #1. The original contents of LINNUM are popped off the stack and returned.

The need for the saving of the contents of LINNUM on the stack are shown when we consider the twin of PEEK, that is, POKE.

POKE X,I% The POKE command replaces the contents of the location specified as the first parameter, by the value specified in the second parameter.

```
POKE 56334,0
```

and the keyboard will not respond to your attentions. The only way in which you can recover is to hit STOP and RESTORE simultaneously. Entering the value 0 into location 56334 stops Timer A, and Timer A controls the scanning of the keyboard. Stopping the timer makes the keyboard dead.

Another way of achieving the same result is to

```
POKE 56334, PEEK(56334) AND 254
```

which leaves all bits except bit 0 unchanged, and changes bit 0 to the value 0. It is bit 0 which controls the timer, and so the keyboard becomes dead again. Notice that the same line number appears twice in the same statement. See above, concerning logical operators, for more details on this use of PEEK and POKE.

If the value to be POKEd (i.e. the second parameter) is outside the bounds 0 to 255, we have the ?ILLEGAL QUANTITY ERROR, and we produce the same error if the location is outside the range 0 to 65535.

The routine works by storing the value of the address in LINNUM (i.e. memory locations 20 and 21), evaluating the expression following the comma, and storing the result of the evaluation at the address given by LINNUM.

POS(X) This is a fairly pointless function. When called, it returns the value of the current cursor position on the screen. For example

```
FOR I=1 TO 255 : PRINT "#"; : NEXT I : PRINT POS(0)
```

will output a stream of #s, and then the number 15. The number 15 is calculated by 255 = 3*80 + 15, and if the line is altered to

```
FOR I=1 TO 239 : PRINT "#"; : NEXT I : PRINT POS("")
```

the answer is 79. Note that the value in parentheses is immaterial. The strange way in which this function produces its results seems to be left over from 80 column PETs.

PRINT This is probably the most complex of all C64 BASIC commands, and it prints information on the screen.

It is possible to print strings:

```
A$ ="123" : FOR I=1 TO 100 : PRINT A$ : NEXT I
FOR I=1 TO 100 : PRINT A$, : NEXT I
FOR I=1 TO 100 : PRINT A$; : NEXT I
```

where the first line prints the string 123 on separate lines, the second prints four examples of 123 per line, and the third prints 12312312 to the end of each line and continues, where it left off, on the next line.

One can print numbers:

```
A=123 : FOR I=1 TO 100 : PRINT A : NEXT I
FOR I=1 TO 100 : PRINT A, : NEXT I
FOR I=1 TO 100 : PRINT A; : NEXT I
```

and for the first two examples the result looks the same as for the string—but not quite.

In the second example, the number 123 starts in by one space, and

```
PRINT -123
```

shows why. The first column is used by the sign of the number, a sign which does not appear if the number is positive.

In the third example the numbers 123 do not appear next to each other as with the strings 123, for there are two spaces in between. A string has no leading or trailing spaces; entering

```
PRINT A;A$
```

there is one space between the number 123, and the string 123. Each number is led by a space and followed by a space.

```
B = -10/9 : FOR I=1 TO 100 : PRINT B, : NEXT I
FOR I=1 TO 100 : PRINT B; : NEXT I
```

shows that—in the first case—there are only two numbers per line (the number being −1.11111111), because there is not enough room for four numbers of this length on a screen line at one time. Each number (including the minus sign) is eleven digits, and thus it is physically impossible to have four eleven-digit numbers on a forty-column screen.

In the second case the numbers −1.11111111 are printed out continuously, with a space between and wrap round.

Consider also the following

Instruction		*Result*
PRINT A A	produces	0
PRINT A$ A$		123123
PRINT 123. 9		123.9
PRINT 123. 9.		123.9 0
PRINT 123. .9		123 .9
PRINT 123(9)		123 9

A A is considered to be AA, which by default has the value 0; A$ A$ is considered to be the equivalent of A$;A$ and thus 123123 is output (note that this is also true for integer % and array () terminators); 123. 9 is considered to be 123.9; 123. 9.   is considered to be 123.9 and . (and . is considered to be 0); 123. .9 is considered to be 123 and .9; and 123(9) is considered to be 123 9, because (is also a separator.

If graphics symbols follow a double quotes (compare INPUT) they do not act immediately, but only operate when the string is printed. Note that if a double quote is the final character on PRINT line, it may be omitted.

The operation of the routines involved in the activation of the PRINT statement is rather complex.

PRINT# F%, This is used to send information to devices such as printers, disk or tape (see OPEN for device numbers).

The file has first to be opened, in the correct mode, before the information is PRINT#ed. The first parameter, the file number (between 1 and 255) is always followed by a comma. After the comma is the list of items to be output.

The format of the list following the PRINT# is the same as that of PRINT. As particular devices have their own particular characteristics, it is not possible to give a general format—the documentation for the printer has to be examined.

For non-Commodore devices, e.g. parallel printers, secondary addresses can be rather important, as they are often used to control the character set (i.e. SET 1 or SET 2, MUM Appendix E).

PRINT# activates the device, prints, and then deactivates the device. PRINT#21, ; does not print anything to the device, but it does deactivate the device. It is only if this is followed by CLOSE 21, that the file details are erased from the file tables. To use PRINT#21, therefore, means that file 21 is not 'listening', but the file 21 can be used again.

CMD activates the device, prints, and leaves the device active. CMD 21 ; also prints nothing but leaves the device active. To follow this command by CLOSE 21 erases all details, but the file is still active. PRINT# is the only way to 'unlisten', but leave the device on-line.

With printers it is often better to use PRINT#, rather than CMD and PRINT. In fact, in the PRINT# routine, calls are made to both CMD and PRINT.

READ This command reads in information from DATA statements (which see), and conforms in general principles to the INPUT statement (which see). Some of the quirks of the INPUT command are not repeated. For example,

```
1 FOR I=1 TO 5
2 READ Y
3 PRINT Y
4 NEXT I
5 DATA ,,,,,
```

produces a string of 0s, but to change line 5 to

```
5 DATA 1/2,,,,,
```

gives a ?SYNTAX ERROR IN 5, whereas with INPUT you would have been asked to redo.

REM This statement allows comments to be made in the program. Anything which follows the REM (even a colon) is treated as part of the comment. Like IF, the jump is to the next line number, not the next statement.

Try a line which starts with 10 REM followed by SHIFT +, then list.

RESTORE Sets the pointer which indicates the present DATA item back to the beginning of the text (locations 63 and 64 for the line number, and locations 65 and 66 for the current DATA item address, PREG page 312).

The same information can be read and re-read.

RESTORE is automatically called by any other command which sets pointers (e.g. NEW, RUN, or CLR).

RETURN Indicates the end of a subroutine, see GOSUB for more details.

RIGHT$(S$,I%) Produces the I% rightmost characters of the string S$ (compare LEFT$ and MID$). The value of I% can vary from 0 to 255, and what is returned is the lesser of I% or the length of the string S$.

This function is related to LEFT$ and MID$, and

```
A$="1234567890" : L=LEN(A$)
FOR I=1 TO L : PRINT LEFT$(A$,I)+RIGHT$(A$,L-I) :
NEXT I
```

gives a series of copies of A$. We have taken the left portion and the rest (the right portion), added them together, and produced the original string.

The routine works by evaluating the string, taking the parameters from the stack, modifying the lower parameter, and then using the routine for LEFT$.

RND(X) This function generates a 'random' (i.e. unpredictable) value between 0 and 1. No sequence of numbers generated by a fixed formula is truly random, because the sequence repeats after a certain period.

There are three classes of value for the parameter. Negative values for the parameter (whichever value they are) always produce the same random number. Positive values for the parameter follow a determinate sequence, once the initial value has been chosen.

If the parameter value is zero, then the random number takes its value from a real-time clock. This provides a random start to any sequence (the start is called a 'seed') but being based on a regular clock, the values tend to repeat. To provide an unpredictable sequence of numbers, it is best to start with RND(0) and then RND(1) or any other positive parameter.

RUN Starts the execution of a program from the beginning, or from any specified linenumber. Pointers are reset, so all previous information is effectively lost.

The command can be used in immediate and program mode, e.g.

```
RUN 5000

200 INPUT "DO YOU WANT ANOTHER GO "; A$
210 IF A$="YES" THEN RUN
```

If there is a linenumber to follow the RUN command, it is evaluated as with GOSUB and GOTO (etc.), so RUN RUN is the same as RUN 0.

SAVE S$, F%, D%, A% This command saves the content of a portion of memory (usually a BASIC program) onto some device. SAVE has the same parameters as LOAD, and shares many common characteristics (see LOAD for full details of the meaning of the parameters).

SGN(X) This function gives the sign of the arithmetic expression within parentheses.

The sign is calculated by first examining the exponent of the expression in FPA #1. If the exponent is zero, then the sign is zero. If the exponent is non-zero, then the sign byte is tested and provides the sign.

One of the bug tests can be modified

```
10 T = -1
20 T = T/2 : PRINT SGN(T) : GOTO 20
```

and produces a long stream of −1s, a final 1, and then 0s.

SGN is short for SIGNUM.

SIN(X) This function gives the sine of the angle (in radians) corresponding to the expression in parentheses.

The SIN is the function calculated to produce the COSine function (see above). Beyond a certain angle, which you may care to investigate, SIN always gives the answer 0.

SPC(I%) This function prints a number of spaces on the screen or printer. This function is used within a PRINT list of items.

The parameter can take values from 0 to 255, and it is used for formatting output. (See TAB, later.)

SQR(X) Finds the square root of the arithmetic expression in parentheses.

The value of the square root is worked out as a special case of the power operator ↑ (see earlier). The square root of X is evaluated as X ↑ .5. This value is not always totally accurate to machine precision, and the best estimate is given by

```
10 DEF FNS(X) = (SQR(X) + X/SQR(X))/2
```

and a test is given by

```
20 FOR I=1 TO 10
30 PRINT SQR(I) – FNS(I)
40 NEXT I
```

For those few values there are two errors of 9.31322575E−10. For maximum accuracy, therefore, use FNS.

ST is a reserved variable (one of three) and it provides a record of the status of the system after any input or output operation. The full name of ST is STATUS.

Many of the errors which are more difficult to detect are not treated by ST, and when one has reached the stage where one is bothered about such problems, there are better ways.

ST is stored in RAM as one byte in a special location, not with the other variables (location 144).

STEP Part of the FOR. . .NEXT loop control structure, see FOR.

STOP Stops a program and indicates at which line the program has halted. The program can be restarted using CONT.

The operation of the routine associated with this command is almost identical to that of END, with added information about the line number.

STR$(X) This function converts the numerical expression within parentheses into a string value. One of the principal uses for this function is in formatting.

The routine is fairly good, for example,

```
PRINT STR$(1234500000000)
PRINT STR$(.000000000000001)
```

produce 1.2345E+12 and 1E−15. Note that the routine inserts a leading space.

SYS X This command transfers control to a machine code routine starting at the address following the command. The machine code routine runs until a return from subroutine instruction is encountered (usually an RTS instruction).

The range of values for the arithmetical expression must be within the limits 0 to 65535, and parentheses are not needed.

To try

```
SYS 40960
```

completely kills the system, and the only way to recover is to switch off. Location 40960 is the start of the BASIC system (held in ROM) and SYS 40960 jumps to the start (where it should not start).

TAB(I%) This is a print formatting command (compare SPC) but one which cannot be used with a printer.

Unlike SPC, which moves a certain number over spaces forward from the present position, TAB moves to an absolute position. Unfortunately if the absolute position is backwards, the character is printed in the next available position.

This is illustrated by

```
PRINT TAB(3);"$#";TAB(2);"!"
```

which leaves three spaces, and then prints $#!.

TAN(X) Gives the tangent of the angle in parentheses where the angle is treated as being in radians.

The value is calculated as SIN(X)/COS(X), which effectively means (see COS) that there are two calls to the SIN routine.

TI and **TI$** Also known as TIME and TIME$, these reserved variables give a reading of the internal clock.

TI gives values in 'jiffies', i.e. 1/60 second, and to calculate in seconds (say in timing a program) use TI/60. It is not possible to assign values to TI, and a ?SYNTAX ERROR is produced.

TI$ measures time in a different manner, and also TI$ can have values assigned. The values assigned must be strings of 6 characters in the form HHMMSS, where any hours value greater than 23 (e.g. 590000) is set to 000000.

The jiffy clock is stored at locations 160 to 162 (called TIME in PREG page 314), and all three locations are set to zero when the machine is switched on. There are various routines to calculate TI$ from the TI value, and various input/output routines will cause the jiffy clock to lose time. (Technically, the jiffy clock counts interrupts, which are sometimes disabled for input/output.)

USR(X) This is an arithmetical function which uses special machine code routines. Not to be used other than by the expert.
When called by say

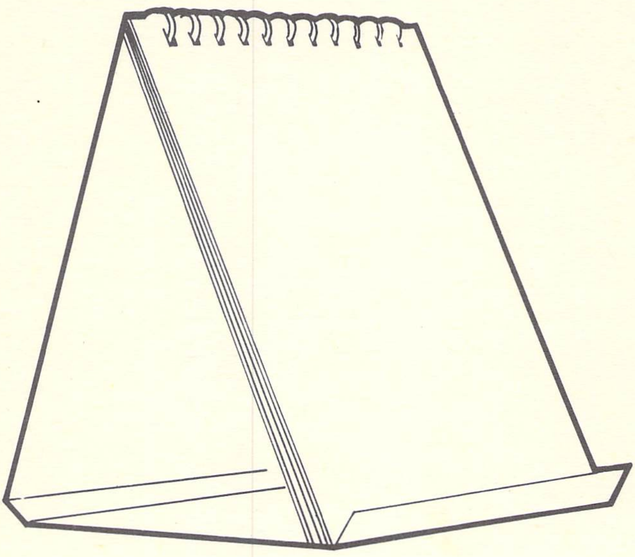
```
    Y = USR(4)
```

there is an immediate jump to locations 785 and 786, and these locations point to the start of a machine code routine. The value in parentheses is used by the routine to which the jump is made. The parameter is placed in FPA #1 and the routine picks up that value.

VAL(S$) We have already met VAL in many guises (see GOTO, GOSUB, INPUT, amongst others). VAL takes a string expression and tries to convert it into a number.
The null string is set to zero, and the conversion continues until the first non-numeric character (except E). VAL uses the CHRGET routine at locations 115 to 138 (PREG page 313). Most other routines to interpret characters use CHRGET, and this is why there are so many family problems.

VERIFY S$, F%, D% This command verifies the accuracy of the saving of a stored portion of memory.
In most respects VERIFY behaves like file input/output commands (which see), and actually uses a very similar routine. The main difference is that the program is read but the bytes input are compared with the contents of memory. If they do not agree, ST is altered, and—more useful—?VERIFY ERROR is printed.

WAIT X, I0%, I1% This is a mostly pointless command. The C64 waits until the result of NOT(PEEK(X) OR NOT(I1%)) AND I0% does not equal zero. If I1% is not present then the check is against PEEK(X) and I0%.



The diagram shows, how to arrange the Pocket Guide in an upright position.

Bend at score mark indicated by arrow.

Pocket Guide

Commodore 64

Boris Allan

Programming Pocket Guides

Commodore 64

ISBN 0 273 02125 7

Pitman

Pitman

ISBN 0 273 02125 7

Commodore 64