BBC BASIC (Z80) for CP/M-80

User Manual

Converted from bbcbasic.co.uk/bbcbasic/mancpm on 18 March 2025
Doc v 1.00 based on BBC BASIC v5

BBC BASIC v5 is actively maintained and may be updated past what is described in this printed manual. See http://www.rtrussell.co.uk/bbcbasic/mancpm/ for the latest version.

(c) 2025 Doug Mounter & Richard Russell
(c) 2025 Richard Deane : conversion of HTML to word/pdf files

This manual is prepared for “Crown Quarto” page size to be bookcase friendly and to have sufficient page width for the text, tables and diagrams. This page size is supported by Lulu for printing around the world.

Table of Contents

1. Introduction 17
Before You Start 17
System Requirements 17
Configuration 17
Running BBC BASIC (Z80) 18
2. General Information 19
Introduction 19
Line numbers 19
Statement Separators 19
Expression Priority 20
Order of Evaluation 20
Examples 20
Variables 20
Specification 20
Numeric Variables 21
String Variables 23
Arrays 23
Array arithmetic 23
Initialising arrays 25
Program flow control 25
Introduction 25
Program structure limitations 26
Leaving program loops 26
Local arrays 28
Indirection 29
Introduction 29

The ? operator 30
The ! operator 31
The | operator 31
The $ operator 31
Use as binary (dyadic) operators 32
Power of Indirection Operators 33
The ^ operator 33
Operators and Special Symbols 34
Keywords 38
Debugging 39
Error handling 41
Default error handling 41
Reasons for trapping errors 41
Error trapping commands 42
Error reporting 46
Error trapping examples 46
Procedures and functions 49
Introduction 49
Names 51
Function and procedure definitions 51
Extending the language 52
Passing parameters 53
Local variables 55
Passing arrays to functions and procedures 56
Using parameters for output 57
Returning arrays from procedures and functions 59
3. Assembler 60

Introduction 60
Instruction mnemonics 60
Statements 60
Labels 60
Comments 60
Constants 61
Reserving Memory 62
The Program Counter 62
Using DIM to Reserve Memory 62
Moving HIMEM to Reserve Memory 63
Length of Reserved Memory 63
Initial Setting of the Program Counter 64
The Assembly Process 64
OPT 64
Assembly at a Different Address 65
OPT Summary 65
How the Assembler Works 66
Saving and Loading Machine Code Programs 67
Conditional Assembly and Macros 68
Introduction 68
Conditional Assembly 68
Macros 69
4. Statements and Functions 71
Introduction 71
Syntax 71
ABS 73
ACS 74

ADVAL 75
AND 75
ASC 77
ASN 79
ATN 79
AUTO 80
BGET# 81
BPUT# 83
BY 84
CALL 84
    Parameter Table 85
    Parameter Formats 86
CASE 87
CHAIN 88
CHR$ 89
CIRCLE 90
CLEAR 91
CLOSE# 91
CLG 92
CLS 93
COLOUR (COLOR) 93
COS 93
COUNT 94
DATA 95
DEF 96
DEG 97
DELETE 98

DIM ... 99
  Dimensioning arrays ... 99
  Reserving an area of memory ... 100
  Reserving a temporary area of memory ... 101
  DIM as a function ... 101
DIV ... 102
DRAW ... 103
EDIT ... 104
ELLIPSE ... 105
ELSE ... 106
END ... 107
ENDCASE ... 108
ENDIF ... 109
ENDPROC ... 109
ENDWHILE ... 109
ENVELOPE ... 110
EOF# ... 110
EOR ... 111
ERL ... 112
ERR ... 113
ERROR ... 113
EVAL ... 114
EXIT ... 115
EXP ... 117
EXT# ... 118
FALSE ... 118
FILL ... 120

FN 120
FOR 122
GCOL 124
GET/GET$ 124
    Reading from an I/O port 125
    Reading from a file 125
GOSUB 126
GOTO 127
HIMEM 128
IF 129
    Single-line IF statement 129
    Multi-line IF statement 130
INKEY/INKEY$ 131
INPUT 132
INPUT LINE 133
INPUT# 134
INSTR 134
INT 135
LEFT$ 136
LEN 138
LET 138
LINE 140
LIST 140
LIST IF 141
    Limitations 142
LISTO 143
    Bit Settings 143

LN 144
LOAD 145
LOCAL 146
LOG 147
LOMEM 148
MID$ 149
MOD 151
MODE 152
MOUSE 153
  MOUSE x,y,b 153
  MOUSE ON n 153
  MOUSE OFF 153
  MOUSE TO x,y 153
  MOUSE RECTANGLE l,b,w,h 153
  MOUSE RECTANGLE OFF 153
MOVE 154
NEW 154
NEXT 155
NOT 156
OF 157
OFF 157
OLD 158
ON 158
  Limitations 159
  Enabling the text cursor 160
ON ERROR 160
ON ERROR LOCAL 161

OPENIN 162
OPENOUT 163
OPENUP 164
OPT 165
- Code Assembled Starting at P% 165
- Code Assembled Starting at O% 165

OR 166
ORIGIN 167
OSCLI 168
OTHERWISE 168
PAGE 169
PI 170
PLOT 170
POINT 171
POS 171
PRINT 172
- General Information 172
- Print Format Control 174
- Examples 177

PRINT# 180
PROC 181
PTR# 182
PUT 183
QUIT 183
RAD 183
READ 184
READ# 185

RECTANGLE 185
REM 186
RENUMBER 187
REPEAT 188
REPORT/REPORT$ 189
RESTORE 190
  RESTORE DATA 191
  RESTORE ERROR 191
  RESTORE LOCAL 191
RETURN 192
RIGHT$ 193
RND 194
RUN 195
SAVE 196
SGN 197
SIN 197
SOUND 198
SPC 198
SQR 199
STEP 199
STOP 200
STR$ 201
STRING$ 201
SUM and SUMLEN 202
  SUM 202
  SUMLEN 203
SWAP 203

SYS 204
TAB 204
TAN 205
THEN 206
TIME 207
TINT 208
TO 209
TOP 210
TRACE 211
TRUE 212
UNTIL 213
USR 214
VAL 214
VDU 215
VPOS 216
WAIT 217
WHEN 218
WHILE 218
WIDTH 219
5. BBC BASIC (Z80) Files 221
Introduction 221
The Structure of Files 221
Basics 221
Serial (Sequential) Files 221
Random Access Files 222
Indexed Files 222
Files in BBC BASIC (Z80) 223

Introduction 223
How Data is Read/Written 224
How Data is Stored 225
How Files are Referred To 225
File Buffering 225
File Commands 225
Introduction 225
Filenames 226
Organisation of Examples 226
Program File Manipulation 226
Data Files 230
Serial Files 236
Introduction 236
Character Data Files 236
Mixed Numeric/Character Data Files 238
Compatible Data Files 243
Random (Relative) FIles 247
Introduction 247
Designing the File 247
Ex 9 - Simple Random Access File 251
Ex 10 - Simple Random Access Database 253
Ex 11 - Random Access Inventory Program 256
Indexed Data Files 265
Deficiencies of Random Access Files 265
The Address Book Program 265
Ex 12 (the LAST) 267
The Binary Chop 276

Explanation 276
6. Operating System Interface 280
Introduction 280
Accessing Star Commands 281
Resident Star Commands 282
7. Table of ASCII Codes 286
8. Mathematical Functions 289
9. Error Messages and Codes 290
Summary 290
Trappable - Program 290
Trappable - Operating System 291
Untrappable - Error Code 0 291
Details 291
Accuracy lost 291
Bad arguments 291
Bad call 292
Bad command 292
Bad DIM 292
Bad EXIT 292
Bad FOR variable 292
Bad hex or binary 293
Bad name 293
Bad program 293
Bad string 293
Bad subscript 294
Bad use of array 294
Can't match FOR 294

Channel....... 294
Close error....... 294
DATA not LOCAL....... 294
DIM space....... 295
Directory full....... 295
Disk full....... 295
Division by zero....... 295
Escape....... 296
Exponent range....... 296
Failed at nnn....... 296
File exists....... 296
File not found....... 297
Line Space....... 297
Log range....... 297
Missing ,....... 297
Missing "....... 297
Missing)....... 297
Missing #....... 298
Missing ENDCASE....... 298
Missing ENDIF....... 298
Missing TO....... 298
Mistake....... 298
Multiple label....... 298
-ve root....... 299
No GOSUB....... 299
No Room....... 299
No such FN/PROC....... 299

No such line 299
No such variable 300
Not in a FN 300
Not in a FN/PROC 300
Not in a FOR looop 300
Not in a PROC 300
Not in a REPEAT loop 301
Not in a WHILE loop 301
Number too big 301
OF not last 301
ON ERROR not LOCAL 301
ON range 302
ON syntax 302
Out of DATA 302
Out of range 302
RENUMBER space 302
Silly 303
String too long 303
Syntax error 303
Too many open files 303
Type mismatch 303
Unknown error 304
10. Format of Program and Variables in Memory 305
Memory Map 305
The Memory Map 306
Memory Management 307
Limiting the Number of Variables 307

String Management 307
Program Storage in Memory 309
Line Length 309
Line Number 309
Statements 309
Line Terminator 309
Variable Storage in Memory 310
Integer Variables 310
Byte Variables 311
Real Variables 311
String Variables 315
Fixed Strings 316
Array storage 316

1. Introduction

Before You Start

This is a reference manual; it is not intended to teach you BASIC. It gives a summary of the BBC BASIC commands and functions plus some hints and tips on their use. A general knowledge of BASIC has been assumed.

File handling with BBC BASIC (Z80) is different in approach from most other versions of BASIC. Because of this, BBC BASIC (Z80) file handling has been covered in some detail.
BBC BASIC (Z80) has been designed to be as compatible as possible with the 6502 BBC BASIC resident in the BBC Microcomputer. The language syntax is not always completely identical to that of the 6502 version, but in most cases the Z80 version is more tolerant.

BBC BASIC (Z80) version 5 is substantially compatible with Acorn's ARM BASIC V, as resident in the Acorn Archimedes and RISC PC etc.

System Requirements

BBC BASIC (Z80) requires a Z80 processor. It is assumed that all the Z80 registers are available for use; BBC BASIC will **not** run on a system which reserves the *alternate registers* (AF', BC', DE', HL') for the use of interrupts.

The CP/M version of BBC BASIC (Z80) requires CP/M-80 version 2.2 or later; about 20 Kbytes of code space and about 1 Kbyte of data space are used by the interpreter. The remainder of the TPA is available for the user's program, data (heap) and stack.

Configuration

Few CP/M systems offer colour graphics and sound of the quality provided on the BBC Microcomputer or Acorn Archimedes, therefore the generic CP/M version of BBC BASIC (Z80) does not support these features.

Many CP/M systems do, however, support cursor addressing (**TAB(X,Y)**), clearing the screen (**CLS**), an elapsed-time clock (**TIME**) and the ability to read the keyboard with a maximum waiting time (**INKEY**). Because these features do not, in general, use a standardised software interface BBC BASIC (Z80) cannot come preconfigured to make use of them. Therefore the source code of a small patch program (**DIST.Z80**) is supplied, which can be edited and assembled to make these facilities available. The patch resides at address 100H and must not exceed 256 bytes in length.

The CP/M edition of BBC BASIC (Z80) supports line-editing using the cursor-control and other keys on the keyboard. However the codes generated by these keys are not standardised. The last 12 bytes of the patch area (1F4H to 1FFH) contain the width of the screen (in characters) followed by the control codes corresponding to the editing keys. As supplied, these are set to the codes generated by an ADM3a terminal.

Running BBC BASIC (Z80)

In the following examples, the system prompts and responses are shown in normal type and your entries are shown in bold type.

To run BBC BASIC (Z80), type at the command prompt:

```
>BBCBASIC<Enter>
```

(<Enter> means 'press the Enter key')

The system will reply:

```
BBC BASIC (Z80) Version 5.00
(C) Copyright R.T.Russell 2024
>
```

To exit BBC BASIC (Z80) type

```
>*BYE<Enter>
```

2. General Information

Introduction

Line numbers

Line numbers are optional in BBC BASIC. A line must be numbered if it is referenced by a GOTO, GOSUB or RESTORE statement but otherwise the line number may be omitted. It can also be useful to number lines when developing and debugging a program, since it makes it easier to EDIT lines and error messages may contain the number of the line in which the error occurs.

It is never necessary to use line numbers. Instead of GOSUB you should use named user-defined procedures or functions. In the case of RESTORE you can use the relative option of the statement. You can avoid GOTO by making use of structures like REPEAT ... UNTIL and WHILE ... ENDWHILE and using the techniques described under Program Flow Control.

The RENUMBER command can be used to add line numbers automatically. Permissible line numbers are in the range 1 to 65535.

Statement Separators

When it is necessary to write more than one statement on a line, the statements should be separated by a colon ':'. BBC BASIC (Z80) will tolerate the omission of the separator if this does not lead to ambiguity. It's safer to leave it in and the program is easier to read.
For example, the following will work.

```
10 FOR i=1 TO 5 PRINT i : NEXT
```

Expression Priority

Order of Evaluation

The various mathematical and logical operators have a priority order. The computer will evaluate an expression taking this priority order into account. Operators with the same priority will be evaluated from left to right. For example, in a line containing multiplication and subtraction, ALL the multiplications would be performed before any of the subtractions were carried out. The various operators are listed below in priority order.

```
variables functions () ! ? & unary+- NOT
^
* / MOD DIV
+ -
= <> <= >= > < << >> >>>
AND
EOR OR
```

Examples

The following are some examples of the way expression priority can be used. It often makes things easier for us humans to understand if you include the brackets whether the computer needs them or not.

```
IF A=2 AND B=3 THEN
IF ((A=2)AND(B=3))THEN

IF A=1 OR C=2 AND B=3 THEN
IF((A=1)OR((C=2)AND(B=3)))THEN

IF NOT(A=1 AND B=2) THEN
IF(NOT((A=1)AND(B=2)))THEN

N=A+B/C-D N=A+(B/C)-D
N=A/B+C/D N=(A/B)+(C/D)
```

Variables

Specification

Variable names may be of unlimited length and all characters are significant. Variable names must start with a letter or underscore, and can only contain the characters A..Z, a..z, 0..9, @, ` (CHR$96) and underscore; embedded keywords are allowed. Upper and lower case variables of the same name are different.
The following types of variable are allowed:

A	real numeric
A%	integer numeric
A&	byte numeric
A$	string

Numeric Variables

Real Variables

Real variables have a range of ±5.9E-39 to ±3.4E38 and numeric functions evaluate to 9 significant figure accuracy. Internally every real number is stored in 40 bits (5 bytes). The number is composed of a 4 byte mantissa and a single byte exponent. An explanation of how variables are stored is given at Annex D.

Integer Variables

Integer variables are stored in 32 bits and have a range of -2147483648 to +2147483647. It is not necessary to declare a variable as an integer for advantage to be taken of fast integer arithmetic. For example, FOR...NEXT loops execute at integer speed whether or not the control variable is an 'integer variable' (% type), so long as it has an integer value.

Byte variables

(BBC BASIC version 5 or later only)

Byte variables are stored in 8 bits and have a range of 0 to 255.

Static Variables

The variables A%..Z% are a special type of integer variable in that they are not cleared by the statements RUN, CHAIN and CLEAR. In addition A%, B%, C%, D%, E%, H% and L% have special uses in CALL and USR routines and P% and O% have a special meaning in the assembler (P% is the program counter and O% points to the code origin). The special variable @% controls numeric print formatting. The variables @%..Z% are called 'static', all other variables are called 'dynamic'.

Boolean Variables

Boolean variables can only take one of the two values TRUE or FALSE. Unfortunately, BBC BASIC does not have true boolean variables. However, it does allow numeric variables to be used for logical operations. The operands are converted to 4 byte integers (by truncation) before the logical operation is performed. For example:

PRINT NOT 1.5 -2	The argument, 1.5, is truncated to 1 and the logical inversion of this gives -2
PRINT NOT -1.5 0	The argument is truncated to -1 and the logical inversion of this gives 0

Two numeric functions, TRUE and FALSE, are provided. TRUE returns the value -1 and FALSE the value 0. These values allow the logical operators (NOT, AND, EOR and OR) to work properly. However, anything which is non-zero is considered to be TRUE. This can give rise to confusion, since +1 is considered to be TRUE and NOT(+1) is -2, which is also considered to be TRUE.

Numeric Accuracy

Numbers are stored in binary format. Integers and the mantissa of real numbers are stored in 32 bits. This gives a maximum accuracy of just over 9 decimal digits. It is possible to display up to 10 digits before switching to exponential (scientific) notation (PRINT and STR$). This is of little use when displaying real numbers because the accuracy of the last digit is suspect, but it does allow the full range of integers to be displayed. Numbers up to the maximum integer value may be entered as a decimal constant without any loss of accuracy. For instance, A%=2147483647 is equivalent to A%=&7FFFFFFF.

String Variables

String variables may contain up to 255 characters. An explanation of how variables are stored is given at the Annex entitled “Format of Program and Variables in Memory”.

Arrays

Arrays of integer, byte, real and string variables are allowed. All arrays must be dimensioned before use. Integers, reals and strings cannot be mixed in a multi-dimensional array; you have to use one array for each type of variable you need.

Array arithmetic

(BBC BASIC version 5 or later only)

A limited number of arithmetic and logical operations are available which operate on entire arrays rather than on single values. These are addition (+), subtraction (-), multiplication (*), division (/), logical OR, logical AND, logical exclusive-or (EOR), integer quotient (DIV), integer remainder (MOD) and the dot product (.). For example:

```
C%() = A%() * B%() + C%()
B() = A() - PI
A() = B() . C()
A() = B() * 2 / C() + 3 - A()
```

Array expressions are evaluated strictly left-to-right, so higher priority operators must come first and brackets cannot be used to override the order of execution:

```
C%() = C%() + B%() * A%() : REM not allowed
C%() = A%() * (B%() + C%()) : REM not allowed
```

All arrays must be DIMensioned before use, and (with the exception of the dot product) the number of elements in all the arrays within an expression must be the same; if not, the Type mismatch error will result. In the case of the dot product the rules are as follows:

- Both source arrays must be 1-dimensional or 2-dimensional.
- If the first array is 1-dimensional it is treated as a single row.
- If the second array is 1-dimensional it is treated as a single column.
- The number of columns in the first array must equal the number of rows in the second array.
- The total number of elements in the destination array must equal the product of the number of rows in the first array and the number of columns in the second array.
-

The SUM function returns the *sum* of all the elements of a numeric array, so the *mean value* can be found as follows (assuming a one-dimensional array):

```
mean = SUM(A()) / (DIM(A(),1) + 1)
```

SUM may also be used with string arrays, it *concatenates* all the elements to form a single string (with a maximum length of 255 characters); the SUMLEN function sums the lengths of all the elements:

```
whole$ = SUM(A$())
wholelen% = SUMLEN(A$())
```

The MOD function returns the *modulus* (square-root of the sum of the squares of all the elements) of a numeric array, so an array can be *normalised* as follows:

```
A() = A() / MOD(A())
```

MOD may also be used to calculate the RMS (Root Mean Square) value of a numeric array:

```
rms = MOD(array()) / SQR(DIM(array(),1) + 1)
```

Don't confuse this with the MOD *operator*, which returns the remainder after an integer division.

You can use the compound assignment operators with arrays (e.g. +=, −=,*= or /=) but there is an important restriction. If the array is an integer (or byte) array the expression on the right-hand-side of the operator is converted to an integer before the operation is carried out. So the statement:

```
A%() *= 2.5
```

will multiply all the elements of array **A%()** by **2**, ***not*** by 2.5.

Initialising arrays

(BBC BASIC version 5 or later only)

You can initialise the individual elements of an array in exactly the same way as you would normal variables. However you can also initialise the contents of an entire array in one operation:

```
A%() = 1, 2, 3, 4, 5
B$() = "Alpha", "Beta", "Gamma", "Delta"
C() = PI
```

If there are fewer values supplied than the number of elements in the array, the remaining elements are unaffected. However in the special case of a single value being supplied, *all* the elements of the array are initialised to that value. So in the last example all the elements of C() will be initialised to PI. There must be enough free space on the stack to contain a copy of the array.

Program flow control

Introduction

Whenever BBC BASIC comes across a FOR, REPEAT, WHILE, GOSUB, FN or PROC statement, it needs to remember where it is in the program so that it can loop back or return there when it encounters
a NEXT, UNTIL, ENDWHILE or RETURN statement or when it reaches the end of a function or procedure. These 'return addresses' tell BBC BASIC where it is in the structure of your program.

Every time BBC BASIC encounters a FOR, REPEAT, WHILE, GOSUB, FN or PROC statement it 'pushes' the return address onto a 'stack' and every time it encounters a NEXT, UNTIL, ENDWHILE or RETURN statement, or the end of a function or procedure, it 'pops' the latest return address off the stack and goes back there.

Apart from memory size, there is no limit to the level of nesting of FOR ... NEXT, REPEAT ... UNTIL, WHILE ... ENDWHILE and GOSUB ... RETURN op erations. The untrappable error message 'No room' will be issued if all the stack space is used up.

Program structure limitations

The use of a common stack has one disadvantage (if it is a disadvantage) in that it forces strict adherence to proper program structure. It is not good practice to exit from a FOR ... NEXT loop without passing through the NEXT statement: it makes the program more difficult to understand and the FOR address is left on the stack. Similarly, the loop or return address is left on the stack if a REPEAT ... UNTIL loop or a GOSUB ... RETURN structure is incorrectly exited. This means that if you leave a FOR..NEXT loop without executing the NEXT statement, and then subsequently encounter, for example, an ENDWHILE statement, BBC BASIC will report an error (in this case a 'Not in a WHILE loop' error). The example below would result in the error message 'Not in a REPEAT loop at line 460'.

```
400 REPEAT
410   INPUT ' "What number should I stop at", num
420   FOR i=1 TO 100
430     PRINT i;
440     IF i=num THEN 460
450   NEXT i
460 UNTIL num=-1
```

Leaving program loops

There are a number of ways to leave a program loop which do not conflict with the need to write tidy program structures. These are discussed below.

REPEAT ... UNTIL loops

One way to overcome the problem of exiting a FOR ... NEXT loop is to restructure it as a REPEAT ... UNTIL loop. The example below performs the same function as the previous example, but exits the structure properly. It has the additional advantage of more clearly showing the conditions which will cause the loop to be terminated (and does not require line numbers):

```
REPEAT
 INPUT ' "What number should I stop at", num
 i=0
 REPEAT
  i=i+1
  PRINT i;
 UNTIL i=100 OR i=num
UNTIL num=-1
```

Changing the loop variable

Another way of forcing a premature exit from a FOR ... NEXT loop is to set the loop variable to a value equal to the limit value. Alternatively, you could set the loop variable to a value greater than the limit (assuming a positive step), but in this case the value on exit would be different depending on why the loop was terminated (in some circumstances, this might be an advantage). The example below uses this method to exit from the loop:

```
REPEAT
 INPUT ' "What number should I stop at", num
 FOR i=1 TO 100
  PRINT i;
  IF i=num THEN
   i=500
  ELSE
   REM More program here if necessary
  ENDIF
 NEXT
UNTIL num=-1
```

Using the EXIT statement

(BBC BASIC version 5 or later only)

You can use the EXIT FOR statement to achieve the same effect as the first example, without requiring a GOTO:

```
REPEAT
 INPUT ' "What number should I stop at", num
 FOR i=1 TO 100
  PRINT i;
```

```
   IF i=num THEN EXIT FOR
   REM More program here if necessary
  NEXT i
UNTIL num=-1
```

Moving the loop into a procedure

A radically different approach is to move the loop into a user-defined procedure or function. This allows you to jump out of the loop directly:

```
DEF PROCloop(num)
LOCAL i
FOR i=1 TO 100
  PRINT i;
  IF i=num THEN ENDPROC
  REM More program here if necessary
NEXT i
ENDPROC
```

If you needed to know whether the loop had been terminated prematurely you could return a different value:

```
DEF FNloop(num)
LOCAL i
FOR i=1 TO 100
  PRINT i;
  IF i=num THEN =TRUE
  REM More program here if necessary
NEXT i
=FALSE
```

Local arrays

Since local variables are also stored on the processor's stack, you cannot use a FOR ... NEXT loop to make an array LOCAL. For example, the following program will give the error message 'Not in a FN or PROC':

```
DEF PROC_error_demo
FOR i=0 TO 10
  LOCAL data(i)
NEXT
ENDPROC
```

(BBC BASIC version 5 or later only)

Fortunately BBC BASIC allows you to make an entire array 'local' as follows:

```
DEF PROC_error_demo
LOCAL data()
DIM data(10)
ENDPROC
```

Note the use of an opening and closing bracket with nothing in between.

Indirection

Introduction

Many versions of BASIC allow access to the computer's memory with the PEEK function and the POKE statement. Such access, which is limited to one byte at a time, is sufficient for setting and reading screen locations or 'flags', but it is difficult to use for building more complicated data structures. The indirection operators provided in BBC BASIC (Z80) enable you to read and write to memory in a far more flexible way. They provide a simple equivalent of PEEK and POKE, but they come into their own when used to pass data between CHAINed programs, build complicated data structures or for use with machine code programs.

There are four indirection operators:

Name	**Symbol**	**Purpose**	**No. of Bytes Affected**
Query	?	Byte Indirection Operator	1
Pling	!	Word Indirection Operator	4
Pipe	|	Floating Point Indirection Operator	5
Dollar	$	String Indirection Operator	1 to 256

The ? operator

Byte Access

The query operator accesses individual bytes of memory. ?M means 'the contents of' memory location 'M'. The first two examples below write &23 to memory location &4FA2, the second two examples set 'number' to the contents of that memory location and the third two examples print the contents of that memory location.

```
  ?&4FA2=&23
```
or
```
  memory=&4FA2
  ?memory=&23
  number=?&4FA2
```
or
```
  memory=&4FA2
  number=?memory
  PRINT ?&4FA2
```
or
```
  memory=&4FA2
  PRINT ?memory
```

Thus, '?' provides a direct replacement for PEEK and POKE.

```
?A=B  is equivalent to  POKE A,B
B=?A  is equivalent to  B=PEEK(A)
```

Query as a Byte Variable

A byte variable, '?count' for instance, may be used as the control variable in a FOR...NEXT loop and only one byte of memory will be used.

```
DIM count% 0
FOR ?count%=0 TO 20
 - - -

 - - -
NEXT
```

The ! operator

The word indirection operator (!) works on 4 bytes (an integer word) of memory. Thus,

```
!M=&12345678
```

would load

```
&78 into address M
&56 into address M+1
&34 into address M+2
&12 into address M+3.
```

and

```
PRINT ~!M   (print !M in hex format)
```

would give

```
12345678
```

The | operator

(BBC BASIC version 5 or later only)

The pipe (|) indirection operator accesses a floating-point value, which occupies 5 bytes of memory. Thus,

```
|F% = PI
```

would load the floating-point representation of PI into addresses F% to F%+4.

The $ operator

The string indirection operator ($) writes a string followed by a carriage-return (&0D) into memory starting at the specified address. Do not confuse M$ with $M. The former is the familiar string variable whilst the latter means 'the string starting at memory location M'. For example,

```
$M="ABCDEF"
```

would load the ASCII characters A to F into addresses M to M+5 and &0D into address M+6, and

```
PRINT $M
```

would print

```
ABCDEF
```

Use as binary (dyadic) operators

All the examples so far have used only one operand with the byte and word indirection operators. Provided the left-hand operand is a variable (such as 'memory') and not a constant, '?' and '!' can also be used as binary operators. (In other words, they can be used with two operands.) For instance, M?3 means 'the contents of memory location M plus 3' and M!3 means 'the contents of the 4 bytes starting at M plus 3'. In the following example, the contents of memory location &4000 plus 5 (&4005) is first set to &50 and then printed.

```
memory=&4000
memory?5=&50
PRINT memory?5
```

Thus,

`A?I=B` is equivalent to `POKE A+I,B`
`B=A?I` is equivalent to `B=PEEK(A+I)`

The two examples below show how two operands can be used with the byte indirection operator (?) to examine the contents of memory. The first example displays the contents of 12 bytes of memory from location &4000. The second example displays the memory contents for a real numeric variable. (See the Annex entitled “Format of Program and Variables in Memory”.)

```
10 memory=&4000
20 FOR offset=0 TO 12
30   PRINT ~memory+offset, ~memory?offset
40 NEXT
```

Line 30 prints the memory address and the contents in hexadecimal format.

```
 10 NUMBER=0
 20 DIM A% -1
 30 REPEAT
 40   INPUT"NUMBER PLEASE "NUMBER
 50   PRINT "& ";
 60   FOR I%=2 TO 5
 70     NUM$=STR$~(A%?-I%)
 80     IF LEN(NUM$)=1 NUM$="0"+NUM$
 90     PRINT NUM$;" ";
100   NEXT
110   N%=A%?-1
120   NUM$=STR$~(N%)
130   IF LEN(NUM$)=1 NUM$="0"+NUM$
140   PRINT " & "+NUM$''
150 UNTIL NUMBER=0
```

See the Annex entitled “Format of Program and Variables In Memory” for an explanation of this program.

Power of Indirection Operators

Indirection operators can be used to create special data structures, and as such they are an extremely powerful feature. For example, a structure consisting of a 10 character string, an 8 bit number and a reference to a similar structure can be constructed.

If M is the address of the start of the structure then:

```
$M    is the string
M?11  is the 8 bit number
M!12  is the address of the related structure
```

Linked lists and tree structures can easily be created and manipulated in memory using this facility.

The ^ operator

(BBC BASIC version 5 or later only)

You can discover the memory address at which a variable is stored using the 'address of' operator **^**. Once you know its address you can access the value of a variable by means of the appropriate indirection operator:

```
A% = 1234
PRINT !^A%
```

This will work for all types of variable (integer, floating-point, string, array etc.) but in the case of a normal string variable the address returned is not that of the first character of the string but of the 4-byte *string descriptor* (see the CALL statement for details of string descriptors). Therefore the address of the string itself is !^string$ >>> 16.

In the case of an array the address returned by ^array() is that of a pointer to the array parameter block, therefore the address of the parameter block is !^array() AND &FFFF. To obtain the address of the array data you should specify the name of the first element, e.g. ^array(0).

Knowing the memory address of a variable may seem to be of little value but can be useful in special circumstances. For example In assembly language code you might want to copy the address of a BBC BASIC (integer) variable into one of the processor's registers:

```
ld hl,^variable%
```

Another use is to alter the byte-order of a variable, for example from *little-endian* to *big-endian*. The following code segment reverses the byte-order of the value stored in A%:

```
SWAP ?(^A%+0), ?(^A%+3)
SWAP ?(^A%+1), ?(^A%+2)
```

Operators and Special Symbols

The following list is a rather terse summary of the meaning of the various operators and special symbols used by BBC BASIC (Z80). It is provided for reference purposes; you will find more detailed explanations elsewhere in this manual.

?	A unary and dyadic operator giving 8-bit indirection.
!	A unary and dyadic operator giving 32-bit indirection.
"	A delimiting character in strings. Strings always have an even number of " in them. " may be introduced into a string by the escape convention "".
#	Precedes reference to a file channel number (and is not optional).
$	*As a suffix* on a variable name, indicates a string variable. *As a prefix* indicates a 'fixed string' e.g. $A="WOMBAT" Store WOMBAT at address A followed by CR.
%	*As a suffix* on a variable name, indicates an integer (signed 32-bit) variable. *As a prefix* ***(BBC BASIC version 5 or later only)*** indicates a binary constant e.g. %11101111.
&	*As a suffix* ***(BBC BASIC version 5 or later only)*** on a variable name, indicates a byte (unsigned 8-bit) variable. *As a prefix* Indicates a hexadecimal constant e.g. &EF.
'	A character which causes new lines in PRINT or INPUT.
( )	Objects in parentheses have highest priority.
=	'Becomes' for LET statement and FOR, 'result is' for FN, relation of equal to on integers, reals and strings.
==	***(BBC BASIC version 5 or later only)*** Relation of equal to (alternative to =).
-	Unary negation and dyadic subtraction on integers and reals.

*	Binary multiplication on integers and reals; statement indicating operating system command (*DIR, *OPT).
:	Multi-statement line statement delimiter.
;	Suppresses forthcoming action in PRINT. Comment delimiter in the assembler. Delimiter in VDU and INPUT.
+	Unary plus and dyadic addition on integers and reals; concatenation between strings.
,	Delimiter in lists.
.	Decimal point in real constants; abbreviation symbol on keyword entry; introduce label in assembler; ***(BBC BASIC version 5 or later only)*** array dot-product operator.
<	Relation of less than on integers, reals and strings.
<<	Left-shift operator (signed or unsigned, ***BBC BASIC version 5 or later only***).
>	Relation of greater than on integers, reals and strings.
>>	Right-shift operator (signed, ***BBC BASIC version 5 or later only***).
>>>	Right-shift operator (unsigned, ***BBC BASIC version 5 or later only***).
/	Binary division on integers and reals.
\	Alternative comment delimiter in the assembler.
<=	Relation of less than or equal on integers, reals and strings.
>=	Relation of greater than or equal on integers, reals and strings.
<>	Relation of not equal on integers, reals and strings.

[ ]	Delimiters for assembler statements. Statements between these delimiters may need to be assembled twice in order to resolve any forward references. The pseudo operation OPT (initially 3) controls errors and listing.
^	Binary operation of exponentation between integers and reals; ***(BBC BASIC version 5 or later only)*** the address of operator.
~	A character in the start of a print field indicating that the item is to be printed in hexadecimal. Also used with STR$ to cause conversion to a hexadecimal string. ***(BBC BASIC version 5 or later only)***:
|	A unary operator giving floating-point indirection. A delimiter in the VDU statement.
+=	Assignment with addition (A += B is equivalent to A = A + B)
−=	Assignment with subtraction (A −= B is equivalent to A = A − B)
*=	Assignment with multiplication (A *= B is equivalent to A = A * B)
/=	Assignment with division (A /= B is equivalent to A = A / B)
AND=	Assignment with AND (A AND= B is equivalent to A = A AND B)
DIV=	Assignment with DIV (A DIV= B is equivalent to A = A DIV B)
EOR=	Assignment with EOR (A EOR= B is equivalent to A = A EOR B)
MOD=	Assignment with MOD (A MOD= B is equivalent to A = A MOD B)
OR=	Assignment with OR (A OR= B is equivalent to A = A OR B)

Keywords

It is not always essential to separate keywords with spaces, for example **DEGASN1** is equivalent to **DEG ASN 1**: both the DEG and the ASN are recognized as keywords. However **ADEGASN1** is treated as a variable name: neither **DEG** nor **ASN** is recognized as a keyword.

In general variable names cannot start with a keyword, but there are some exceptions. For example the constant **PI** and the pseudo-variables **LOMEM**, **HIMEM**, **PAGE**, and **TIME** *can* form the first part of the name of a variable. Therefore PILE and TIMER are valid variable names although PI$ and TIME% are not.

37 out of the total of 138 keywords are allowed at the start of a variable name; they are shown in **bold** type in the table below. 23 keywords are only available in ***BBC BASIC version 5 or later***; they are indicated with an asterisk.

Keywords Available

ABS	ACS	ADVAL	AND	ASC	ASN
ATN	**BGET**	**BPUT**	**BY***	CALL	CASE*
CHAIN	CHR$	CIRCLE*	**CLEAR**	**CLG**	**CLOSE**
CLS	COLOR	COLOUR	COS	**COUNT**	DATA
DEF	DEG	DIM	DIV	DRAW	ELLIPSE*
ELSE	**END**	**ENDCASE***	**ENDIF***	**ENDPROC**	**ENDWHILE***
ENVELOPE	**EOF**	EOR	**ERL**	**ERR**	ERROR
EVAL	**EXIT***	EXP	**EXT**	**FALSE**	FILL*
FN	FOR	GCOL	GET	GET$	GOSUB
GOTO	**HIMEM**	IF	INKEY	INKEY$	INPUT
INSTALL*	INSTR(	INT	LEFT$(	LEN	LET
LINE	LN	LOCAL	LOG	**LOMEM**	MID$(
MOD	MODE	MOUSE*	MOVE	NEXT	NOT

OF*	**OFF**	ON	OPENIN	OPENOUT	OPENUP
OR	ORIGIN*	OSCLI	OTHERWISE*	**PAGE**	**PI**
PLOT	POINT(	**POS**	PRINT	PROC	**PTR**
PUT	**QUIT***	RAD	READ	RECTANGLE*	REM
REPEAT	**REPORT**	RESTORE	**RETURN**	RIGHT$(	**RND**
RUN	SGN	SIN	SOUND	SPC	SQR
STEP	**STOP**	STR$	STRING$(	SUM*	SWAP*
SYS*	TAB(	TAN	THEN	**TIME**	TINT*
TO	TRACE	**TRUE**	UNTIL	USR	VAL
VDU	**VPOS**	**WAIT***	WHEN*	WHILE*	WIDTH

The following immediate-mode commands are strictly speaking not keywords:

AUTO	DELETE	EDIT	LIST	LISTO	LOAD
NEW	OLD	RENUMBER	SAVE		

Debugging

"If debugging is the process of removing bugs, then programming must be the process of putting them in."

It is inevitable that programs will sometimes contain bugs. You may be able to write small programs which are error-free and work first time, but the larger your program is, the greater the likelihood that it will have errors. These fall into three categories:

- Syntax errors. This is where the code doesn't make sense to the BASIC interpreter and often results from a typing mistake. For example, you might type:

```
PRONT "Hello world!"
```

when what you meant was:

```
PRINT "Hello world!"
```

When the interpreter encounters the word **PRONT** it won't know what to do, and will report a Mistake error.

- Run-time errors. These are like syntax errors in that they trigger BASIC's error reporting mechanism, but whereas a syntax error will inevitably cause a failure as soon as the code is executed, run-time errors only occur when certain conditions are met. For example, the code :

```
answer = A / B
```

is syntactically correct, and generally won't result in an error, but if variable **B** is zero it will result in the Division by zero error. It could also result in the Number too big error if **A** is very large and **B** is very small.

- Other errors. Unfortunately, many programming errors do not trigger BASIC's error reporting mechanism, and therefore go undetected by the interpreter, but result in your program producing incorrect output or failing to respond correctly to input. These can include some typing errors, errors of logic (incorrect processes take place, or take place in the wrong sequence) and computational errors (the processes themselves do the wrong things). In each case BASIC is doing what you said, but what you said is either not what you meant or doesn't have the outcome you wanted.

  Errors of logic and computational errors are best avoided at the design stage, before you even start writing code, but this ideal situation is rarely achieved in practice.

Indentation

By default, looping and nesting constructs in your program are indented when LISTed. This allows you to quickly spot errors such as:

- Unpaired keywords, e.g. a FOR without a NEXT, or a REPEAT without an UNTIL.
- Multi-line IF statements with the THEN omitted.

Fast edit-run cycle

As BBC BASIC is an interpreted language (albeit a very fast one) it has the advantage of allowing you to edit and re-run a program *instantly*. After you have made a change to a program you don't have to wait for it to be re-compiled and re-linked before you can run it. Although trial-and-error debugging isn't to be encouraged, there are occasions when it is the most efficient way of solving a problem.

Immediate mode

If an untrapped error occurs at run time the program exits to *immediate mode* and displays an error message followed by the **>** prompt. At the prompt you can type any valid BASIC statement then press the Enter key, whereupon (as the name suggests) the statement will be executed immediately. This can be very useful in ascertaining the values of, or performing calculations on, variables; issuing 'star' commands or even calling procedures and functions.

Error handling

Default error handling

By default, if BBC BASIC detects an error while running a program it will immediately halt execution and report an error message and the line number (if any). This behaviour may often be adequate, but you have the option of *trapping* errors and writing your own code to handle them.

Reasons for trapping errors

There are a number of reasons why you might want to trap errors:

To make your program more 'friendly'

When you write a program for yourself, you know what you want it to do and you also know what it can't do. If, by accident, you try to make it do something which could give rise to an error, you accept the fact that BBC BASIC might terminate the program and return to immediate mode. However, when somebody else uses your program they are not blessed with your insight and they may find the program 'crashing out' without knowing what they have done wrong. Such programs are called 'fragile'. You can protect your user from much frustration if you display a more 'friendly' error report and fail 'gracefully'.

To ensure the error message is visible

The default error reporting writes the error message to the screen just like any other text output. Unfortunately there are several reasons why your program might not display a useful message, for example the text colours, font, size, position etc. are such that the message is invisible.

To allow cleanup operations to take place

There can be situations, particularly when calling Operating System functions, when aborting an operation 'mid stream' may leave the system in an unstable state; for example it might result in a *memory leak*. It could even, in unusual circumstances, crash BBC BASIC. In such cases your own error handler can carry out the necessary tidying-up operations before reporting the error and exiting gracefully.

To allow execution to continue

Although it is preferable to prevent errors occurring in the first place, there can be occasions when this is difficult or impossible to achieve, and it is better to allow the error to happen. In these circumstances error trapping allows the program to continue execution, possibly with some specific action being taken as a result.

Error trapping commands

The two main commands provided by BBC BASIC for error trapping and recovery are:

```
ON ERROR ....
```

and

```
ON ERROR LOCAL ....
```

ON ERROR

The ON ERROR command directs BBC BASIC to execute the statement(s) following ON ERROR when a trappable error occurs:

```
ON ERROR PRINT '"Oh No!":END
```

If an error was detected in a program after this line had been encountered, the message 'Oh No!' would be printed and the program would terminate. If, as in this example, the ON ERROR line contains the END statement or transfers control elsewhere (e.g. using GOTO) then the position of the line within the program is unimportant *so long as it is encountered before the error occurs*. If there is no transfer of control, execution following the error continues as usual on the succeeding line, so in this case the position of the ON ERROR line can matter.

As explained in the Program Flow Control sub-section, every time BBC BASIC encounters a FOR, REPEAT, GOSUB, FN, PROC or WHILE statement it 'pushes' the return address on to a 'stack' and every time it encounters a NEXT, UNTIL, RETURN, ENDWHILE statement or the end of a function or procedure it 'pops' the latest return address of the stack and goes back there. The program stack is where BBC BASIC records where it is within the structure of your program.

When an error is detected by BBC BASIC, the stack is cleared. Thus, you cannot just take any necessary action depending on the error and return to where you were because BBC BASIC no longer knows where you were.

If an error occurs within a procedure or function, the value of any parameters and LOCAL variables will be the last value they were set to within the procedure or function which gave rise to the error (the values, if any, they had before entry to the procedure or function are lost).

ON ERROR LOCAL

(BBC BASIC version 5 or later only)

The ON ERROR LOCAL command prevents BBC BASIC clearing the program stack. By using this command, you can trap errors within a FOR ... NEXT, REPEAT ... UNTIL or WHILE ... ENDWHILE loop or a subroutine, function or procedure without BBC BASIC losing its place within the program structure.

The following example program will continue after the inevitable 'Division by zero' error:

```
FOR n=-5 TO 5
  ok% = TRUE
  ON ERROR LOCAL PRINT "Infinity" : ok% = FALSE
  IF ok% PRINT "The reciprocal of ";n;" is ";1/n
NEXT n
```

This is, of course, a poor use of error trapping. You should test for n=0 rather than allow the error to occur. However, it does provide a simple demonstration of the action of ON ERROR LOCAL. Also, you should test ERR to ensure that the error was the one you expected rather than, for example, Escape (ERR is a function which returns the number of the last error; it is explained later in this sub-section).

After the loop terminates (when 'n' reaches 6) the previous error trapping state is restored. Alternatively, you can explicitly restore the previous state using RESTORE ERROR.

You can call a subroutine, function or procedure to 'process' the error, but it must return to the loop, subroutine, function or procedure where the error was trapped:

```
x=OPENOUT "TEST"
FOR i=-5 TO 5
  ok% = TRUE
  ON ERROR LOCAL PROCerr : ok% = FALSE
  IF ok% PRINT#x,1/i
NEXT
CLOSE#x
:
x=OPENIN "TEST"
REPEAT
  INPUT#x,i
  PRINT i
UNTIL EOF#x
CLOSE#x
END
:
DEF PROCerr
```

```
IF ERR <> 18 REPORT:END
PRINT#x,3.4E38
ENDPROC
```

The position of the ON ERROR LOCAL within a procedure or function can be important. Consider the following two examples:

```
DEF PROC1(A)
ON ERROR LOCAL REPORT : ENDPROC
LOCAL B
....
DEF PROC1(A)
LOCAL B
ON ERROR LOCAL REPORT : ENDPROC
....
```

In the case of the first example, if an error occurs within the procedure, the formal parameter **A** will be automatically restored to its original value, but the LOCAL variable **B** will not; it will retain whatever value it had when the error occurred. In the case of the second example, *both* the formal parameter **A** *and* the LOCAL variable **B** will be restored to the values they had before the procedure was called.

ON ERROR or ON ERROR LOCAL?

If you use **ON ERROR**, BBC BASIC clears the program stack. Once your program has decided what to do about the error, it must re-enter the program at a point that is not within a loop, subroutine, function or procedure. In practice, this generally means somewhere towards the start of the program. This is often undesirable as it makes it difficult to build well structured programs which include error handling. On the other hand, you will probably only need to write one or two program sections to deal with errors.

At first sight **ON ERROR LOCAL** seems a more attractive proposition since BBC BASIC remembers where it is within the program structure. The one disadvantage of **ON ERROR LOCAL** is that you will need to include an error handling section at every level of your program where you need to trap errors. Many of these sections of program could be identical.

You can mix the use of **ON ERROR** and **ON ERROR LOCAL** within your program. A single **ON ERROR** statement can act as a 'catch all' for unexpected errors, and one or more **ON ERROR LOCAL** statements can be used when your program is able to recover from predictable errors. **ON ERROR LOCAL** 'remembers' the current **ON ERROR** setting, and restores it when the loop, procedure or function containing the **ON ERROR LOCAL** command is finished, or when a **RESTORE ERROR** is executed.

Error reporting

There are three functions, ERR, ERL, REPORT$ and one statement, REPORT, which may be used to investigate and report on errors. Using these, you can trap out errors, check that you can deal with them and abort the program run if you cannot.

ERR

ERR returns the error number (see the section entitled Error messages and codes).

ERL

ERL returns the line number where the error occurred. If an error occurs in a procedure or function call, ERL will return the number of the calling line, not the number of the line in which the procedure/function is defined. If an error in a DATA statement causes a READ to fail, ERL will return the number of the line containing the DATA, not the number of the line containing the READ statement. If the line is not numbered, ERL returns zero.

REPORT$

(BBC BASIC version 5 or later only)

REPORT$ returns the error string associated with the last error which occurred.

REPORT

REPORT prints out the error string associated with the last error which occurred. It is equivalent to PRINT REPORT$;

Error trapping examples

The majority of the following error trapping examples are all very simple programs without nested loops, subroutines, functions or procedures. Consequently, they use ON ERROR for error trapping in preference to ON ERROR LOCAL.

The example below does not try to deal with errors, it just uses ERR and REPORT to tell the user about the error. Its only advantage over BBC BASIC's normal error handling is that it gives the error number; it would probably not be used in practice. As you can see from the second run, pressing <ESC> is treated as an error (number 17).

```
ON ERROR PROCerror
REPEAT
  INPUT "Type a number " num
  PRINT num," ",SQR(num)
  PRINT
UNTIL FALSE
:
DEF PROCerror
PRINT
PRINT "Error No ";ERR
REPORT
END
```

Example run:

```
RUN
Type a number 1
        1        1
Type a number -2
       -2
Error No 21
Negative root



RUN
Type a number <Esc>
Error No 17
Escape
```

The example below has been further expanded to include error trapping. The only 'predictable' error is that the user will try a negative number. Any other error is unacceptable, so it is reported and the program aborted. Consequently, when

<ESC> is used to abort the program, it is reported as an error. However, a further test for ERR=17 could be included so that the program would halt on ESCAPE without an error being reported.

```
ON ERROR PROCerror
REPEAT
 INPUT "Type a number " num
  PRINT num," ",SQR(num)
 PRINT
UNTIL FALSE
:
DEF PROCerror
PRINT
IF ERR=21 THEN PRINT "No negatives":ENDPROC
REPORT
END
```

This is a case where the placement of the ON ERROR statement is important. When PROCerror exits, execution continues after the ON ERROR statement, which in this case causes the program to restart from the beginning.

Example run:

```
RUN
Type a number 5
     5       2.23606798


Type a number 2
     2       1.41421356
Type a number -1
    -1
No negatives


Type a number 4
     4       2


Type a number <Esc>
Escape
```

The above example is very simple and was chosen for clarity. In practice, it would be better to test for a negative number before using SQR rather than trap the

'Negative root' error. A more realistic example is the evaluation of a user-supplied HEX number, where trapping 'Bad hex or binary' would be much easier than testing the input string beforehand.

```
ON ERROR PROCerror
REPEAT
  INPUT "Type a HEX number " Input$
  num=EVAL("&"+Input$)
  PRINT Input$,num
  PRINT
UNTIL FALSE
:
DEF PROCerror
PRINT
IF ERR=28 THEN PRINT "Not hex":ENDPROC
REPORT
END
```

The next example is similar to the previous one, but it uses the ON ERROR LOCAL command to trap the error.

```
REPEAT
  ON ERROR LOCAL PROCerror
  INPUT "Type a HEX number " Input$
  num=EVAL("&"+Input$)
  PRINT Input$,num
  PRINT
UNTIL FALSE
:
DEF PROCerror
PRINT
IF ERR=28 THEN PRINT "Not hex":ENDPROC
REPORT
END
```

Note that had ON ERROR (rather than ON ERROR LOCAL) been used in this case, an error would give rise to a **Not in a REPEAT loop** error at the UNTIL statement. This is because ON ERROR clears the program's stack.

Procedures and functions

Introduction

Procedures and functions are like subroutines in that they are 'bits' of program which perform a discrete function. Like subroutines, they can be performed (called) from several places in the program. However, they have two great advantages over subroutines: you can refer to them by name and the variables used within them can be made private to the procedure or function.

Arguably, the major advantage of procedures and functions is that they can be referred to by name. Consider the two similar program lines below.

```
IF name$="ZZ" THEN GOSUB 500 ELSE GOSUB 800
```

```
IF name$="ZZ" THEN PROC_end ELSE PROC_print
```

The first statement gives no indication of what the subroutines at lines 500 and 800 actually do. The second, however, tells you what to expect from the two procedures. This enhanced readability stems from the choice of meaningful names for the two procedures.

A function often carries out a number of actions, but it always produces a single result. For instance, the 'built in' function INT returns the integer part of its argument.

```
age=INT(months/12)
```

A procedure on the other hand, is specifically intended to carry out a number of actions, some of which may affect program variables, but it does not directly return a result.

Whilst BBC BASIC has a large number of pre-defined functions (INT and LEN for example) it is very useful to be able to define your own to do something special. Suppose you had written a function called FN_discount to calculate the discount price from the normal retail price. You could write something similar to the following example anywhere in your program where you wished this calculation to be carried out.

```
discount_price=FN_discount(retail_price)
```

It may seem hardly worth while defining a function to do something this simple. However, functions and procedures are not confined to single line definitions, and they are very useful for improving the structure and readability of your program.

Names

The names of procedures and functions MUST start with PROC or FN and, like variable names, they cannot contain spaces. This restriction can give rise to some pretty unreadable names. However, the underline character can be used to advantage. Consider the procedure and function names below and decide which is the easier to read.

```
PROCPRINTDETAILS    FNDISCOUNT

PROC_print_details   FN_discount
```

Function and procedure names may end with a '$'. However, this is not compulsory for functions which return strings.

Function and procedure definitions

Starting a definition

Function and procedure definitions are 'signalled' to BBC BASIC by preceding the function or procedure name with the keyword DEF. DEF must be at the beginning of the line. If the computer encounters DEF during execution of the program, the rest of the line is ignored. Consequently, you can put single line definitions anywhere in your program.

The function/procedure body

The 'body' of a procedure or function must not be executed directly - it must be performed (called) by another part of the program. Since BBC BASIC only skips the rest of the line when it encounters DEF, there is a danger that the remaining lines of a multi-line definition might be executed directly. You can avoid this by putting multi-line definitions at the end of the main program text after the END statement. Procedures and functions do not need to be declared before they are used and there is no speed advantage to be gained by placing them at the start of the program.

Ending a definition

The end of a procedure definition is indicated by the keyword ENDPROC. The end of a function definition is signalled by using a statement which starts with an equals (=) sign. The function returns the value of the expression to the right of the equals sign.

Single line functions/procedures

For single line definitions, the start and end are signalled on the same line. The first example below defines a function which returns the average of two numbers. The second defines a procedure which clears from the current cursor position to the end of the line (on a 40 column screen):

```
DEF FN_average(A,B) = (A+B)/2
DEF PROC_clear:PRINT SPC(40-POS);:ENDPROC
```

Extending the language

You can define a whole library of procedures and functions and include them in your programs. By doing this you can effectively extend the scope of the language. For instance, BBC BASIC does not have a 'clear to end of screen' command. The example below is a procedure to clear to the end of 'screen' (BASIC's output window) with an 80 column by 25 row display (e.g. MODE 3). The three variables used (i%, x%, and y%) are declared as LOCAL to the procedure (see later).

```
DEF PROC_clear_to_end
LOCAL i%,x%,y%
x%=POS:y%=VPOS
REM If not already on the last line, print lines of
REM spaces which will wrap around until the last line
IF y% < 24 FOR i%=y% TO 23:PRINT SPC(80);:NEXT
REM Print spaces to the end of the last line.
PRINT SPC(80-x%);
REM Return the cursor to its original position
PRINT TAB(x%,y%);
ENDPROC
```

Passing parameters

When you define a procedure or a function, you list the parameters to be passed to it in brackets. For instance, the discount example expected one parameter (the retail price) to be passed to it. You can write the definition to accept any number of parameters. For example, we may wish to pass both the retail price and the discount percentage. The function definition would then look something like this:

```
DEF FN_discnt(price,pcent)=price*(1-pcent/100)
```

In this case, to use the function we would need to pass two parameters.

```
retail_price=26.55
discount_price=FN_discount(retail_price,25)
```

or

```
price=26.55
discount=25
price=FN_discount(price,discount)
```

or

```
price=FN_discount(26.55,25)
```

Formal and actual parameters

The value of the first parameter in the call to the procedure or function is passed to the first variable named in the parameter list in the definition, the second to the second, and so on. This is termed 'passing by value' (however note that arrays and structures are 'passed by reference'). The parameters declared in the definition are called 'formal parameters' and the values passed in the statements which perform (call) the procedure or function are called 'actual parameters'. There must be as many actual parameters passed as there are formal parameters declared in the definition.

```
FOR I% = 1 TO 10
PROC_printit(I%) : REM I% is the Actual parameter
NEXT
END
```

```
DEF PROC_printit(num1) : REM num1 is the Formal parameter
PRINT num1
ENDPROC
```

The formal parameters must be variables, but the actual parameters may be variables, constants or expressions. When the actual parameter is a variable, it need not be (and usually won't be) the same as the variable used as the formal parameter. Formal parameters are automatically made local to the procedure or function.

You can pass a mix of string and numeric parameters to the procedure or function, and a function can return either a string or numeric value, irrespective of the type of parameters passed to it. However, you must make sure that the parameter types match up. The first example below is correct; the second would give rise to an 'Incorrect arguments' error message and the third would cause a 'Type mismatch' error to be reported.

Correct

```
PROC_printit(1,"FRED",2)
END
:
DEF PROC_printit(num1,name$,num2)
PRINT num1,name$,num2
ENDPROC
```

Incorrect arguments error

```
PROC_printit(1,"FRED",2,4)
END
:
DEF PROC_printit(num1,name$,num2)
PRINT num1,name$,num2
ENDPROC
```

Type mismatch error

```
PROC_printit(1,"FRED","JIM")
END
:
DEF PROC_printit(num1,name$,num2)
PRINT num1,name$,num2
ENDPROC
```

Local variables

You can use the LOCAL statement to declare variables for use locally within individual procedures or functions (the formal parameters are also automatically local to the procedure or function declaring them). Changing the values of local variables has no effect on variables of the same name (if any) elsewhere in the program.
Declaring variables as LOCAL initialises them to zero (in the case of numeric variables) or null/empty (in the case of string variables). Declaring variables as PRIVATE causes them to retain their values from one call of the function/procedure to the next. If an array or structure is made LOCAL or PRIVATE it must be re-DIMensioned before use.

Variables which are not formal parameters nor declared as LOCAL are known to the whole program, including all the procedures and functions. Such variables are called **global**.

Recursive functions/procedures

Because the formal parameters which receive the passed parameters are local, all procedures and functions can be recursive. That is, they can call themselves. But for this feature, the short example program below would be difficult to code. It is the often-used example of a factorial number routine (the factorial of a number n is n * n−1 * n−2 * .... * 1. Factorial 6, for instance, is 6 * 5 * 4 * 3 * 2 * 1 = 720).

```
REPEAT
  INPUT "Enter an INTEGER less than 35 "num
UNTIL INT(num)=num AND num<35
fact=FN_fact_num(num)
PRINT num,fact
END
:
DEF FN_fact_num(n)
IF n=1 OR n=0 THEN =1
REM Return with 1 if n= 0 or 1
=n*FN_fact_num(n-1)
REM Else go round again
```

Since 'n' is the input variable to the function FN_fact_num, it is local to each and every use of the function. The function keeps calling itself until it returns the answer 1. It then works its way back through all the calls until it has completed the final multiplication, when it returns the answer. The limit of 35 on the input number prevents the answer being too big for the computer to handle.

Passing arrays to functions and procedures

(BBC BASIC version 5 or later only)

BBC BASIC allows you to pass an entire array (rather than just a single element) to a function or procedure. Unlike single values or strings, which are passed to a function or procedure **by value**, an array is passed **by reference**. That is, a pointer to the array contents is passed rather than the contents themselves. The consequence of this is that if the array contents are modified within the function or procedure, they remain modified on exit from the function or procedure. In the following example the procedure PROC_multiply multiplies all the values in a numeric array by a specified amount:

```
DIM arr(20)
FOR n%=0 TO 20 : arr(n%) = n% : NEXT
PROC_multiply(arr(), 2)
FOR n%=0 TO 20: PRINTarr(n%) : NEXT
END
:
DEF PROC_multiply(B(), M)
LOCAL n%
FOR n% = 0 TO DIM(B(),1)
  B(n%) = B(n%) * M
NEXT
ENDPROC
```

Note the use of DIM as a function to return the number of elements in the array.

The advantage of passing an array as a parameter, rather than accessing the 'global' array directly, is that the function or procedure doesn't need to know the *name* of the array. Ideally a FN/PROC shouldn't need to know the names of variables used in the *main program* from which it is called, and the main program shouldn't need to know the names of variables contained in a function or procedure it calls.

Using parameters for output

(BBC BASIC version 5 or later only)

Although function and procedure parameters are usually *inputs*, it is possible to specify that they can also be *outputs* by using the keyword RETURN in the definition:

```
DEF PROCcomplexsquare(RETURN r, RETURN i)
```

If a RETURNed parameter is modified within the function or procedure, it remains modified on exit. This is the case even if the *actual parameter* and the *formal parameter* have different names.

Suppose you want to write a function which returns a *complex number*. Since a complex number is represented by two numeric values (the *real part* and the *imaginary part*) a conventional user-defined function, which can return only a single value, is not suitable. Conventionally you would have to resort to using *global variables* or an array or structure to return the result, but by using RETURNed parameters this can be avoided.

The following example shows the use of a procedure to return the square of a complex number:

```
INPUT "Enter complex number (real, imaginary): " real, imag
PROCcomplexsquare(real, imag)
PRINT "The square of the number is ";real " + ";imag " i"
END
DEF PROCcomplexsquare(RETURN r, RETURN i)
LOCAL x,y
x = r : y = i
r = x^2 - y^2
i = 2 * x * y
ENDPROC
```

You can use a similar technique to return strings rather than numbers. The following example, rather pointlessly, takes two strings and mixes them up by exchanging alternate characters between them:

```
INPUT "Enter two strings, separated by a comma: " A$, B$
PROCmixupstrings(A$, B$)
PRINT "The mixed-up strings are " A$ " and " B$
END
DEF PROCmixupstrings(RETURN a$, RETURN b$)
LOCAL c$, I%
FOR I% = 1 TO LEN(a$) STEP 2
  c$ = MID$(a$,I%,1)
  MID$(a$,I%,1) = MID$(b$,I%,1)
  MID$(b$,I%,1) = c$
NEXT I%
ENDPROC
```

You can also use this facility to create a variable whose name is determined at run-time, something which is otherwise impossible to achieve:

```
INPUT "Enter a variable name: " name$
INPUT "Enter a numeric value for the variable: " value$
dummy% = EVAL("FNassign("+name$+","+value$+")")
PRINT "The variable "name$" has the value ";EVAL(name$)
END
DEF FNassign(RETURN n, v) : n = v : = 0
```

Returning arrays from procedures and functions

(BBC BASIC version 5 or later only)

It is possible to declare an array within a function or procedure and return that new array to the calling code. To do that you must pass an *undeclared* array as the parameter and use the RETURN keyword in the function or procedure definition:

```
PROCnewarray(alpha())
PROCnewarray(beta())
PROCnewarray(gamma())

DEF PROCnewarray(RETURN a())
DIM a(3, 4, 5)
...
ENDPROC
```

Note: You cannot use this technique to declare LOCAL arrays.

3. Assembler

Introduction

BBC BASIC (Z80) includes a Z80 assembler. This assembler is similar to the 6502 assembler on the BBC Micro and it is entered in the same way. That is, '[' enters assembler mode and ']' exits assembler mode.

Instruction mnemonics

All standard Zilog mnemonics are accepted: 'ADD', 'ADC' and 'SBC', must be followed by 'A' or 'HL'. For example **ADD A,C** is accepted but **ADD C** is not. However, the brackets around the port address in 'IN' and 'OUT' are optional; thus both **OUT (5),A** and **OUT 5,A** are accepted. The instruction 'IN F,(C)' is **not** accepted but the equivalent object code is produced from 'IN (HL),(C)'.

Statements

An assembly language statement consists of three elements; an optional label, an instruction and an operand. A comment may follow the operand field. The instruction following a label must be separated from it by at least one space. Similarly, the operand must also be separated from the instruction by a space. Statements are terminated by a colon (:) or end of line (<RET>).

Labels

Any BBC BASIC (Z80) numeric variable may be used as a label. These (external) labels are defined by an assignment (count=23 for instance). Internal labels are defined by preceding them with a full stop. When the assembler encounters such a label, a BASIC variable is created containing the current value of the Program Counter (P%). The Program Counter is described later.

In the example shown later under the heading The Assembly Process, two internal labels are defined and used. Labels have the same rules as standard BBC BASIC (Z80) variable names; they should start with a letter and not start with a keyword.

Comments

You can insert comments into assembly language programs by preceding them with a semi-colon (;) or a back-slash (\). In assembly language, a comment ends at the end of the statement. Thus, the following example will work (but it's a bit untidy).

```
[;start assembly language program
etc
LD A,B ;In-line comment:POP HL ;get start address
RET NZ ;Return if finished:JR loop ;else go back
etc
;end assembly language program:]
```

Constants

You can store constants within your assembly language program using the define byte (DEFB), define word (DEFW) and define message (DEFM) pseudo-operation commands.

Define Byte - DEFB

DEFB can be used to set one byte of memory to a particular value. For example,

```
.data DEFB 15
    DEFB 9
```

will set two consecutive bytes of memory to 15 and 9 (decimal). The address of the first byte will be stored in the variable 'data'.

Define Word - DEFW

DEFW can be used to set two bytes of memory to a particular value. The first byte is set to the least significant byte of the number and the second to the most significant byte. For example,

```
.data DEFW &90F
```

will have the same result as the Byte Constant example.

Define Message - DEFM

DEFM can be used to load a string of ASCII characters into memory. For example,

```
JR continue; jump round the data
.string DEFM "This is a test message"
DEFB &D
.continue; and continue the process
```

will load the string 'This is a test message' followed by a carriage-return into memory. The address of the start of the message is loaded into the variable 'string'. This is equivalent to the following program segment:

```
JR continue;        jump round the data
.string;    leave assembly and load the string
]
$P%="This is a test message" REM starting at P%
P%=P%+LEN($P%)+1 REM adjust P% to next free byte
[
OPT opt; reset OPT
.continue;          and continue the program
```

Reserving Memory

The Program Counter

Machine code instructions are assembled as if they were going to be placed in memory at the addresses specified by the program counter, P%. Their actual location in memory may be determined by O% depending on the OPTion specified (see below). You must make sure that P% (or O%) is pointing to a free area of memory before your program begins assembly. In addition, you need to reserve the area of memory that your machine code program will use so that it is not overwritten at run time. You can reserve memory by using a special version of the DIM statement or by changing HIMEM or LOMEM.

Using DIM to Reserve Memory

Using the special version of the DIM statement to reserve an area of memory is the simplest way for short programs which do not have to be located at a particular memory address. (See the keyword DIM for more details.) For example,

```
DIM code 20: REM Note the absence of brackets
```

will reserve 21 bytes of code (byte 0 to byte 20) and load the variable 'code' with the start address of the reserved area. You can then set P% (or O%) to the start of that area. The example below reserves an area of memory 100 bytes long and sets P% to the first byte of the reserved area.

```
DIM sort% 99
P%=sort%
```

Moving HIMEM to Reserve Memory

If you are going to use a machine code program in several of your BBC BASIC (Z80) programs, the simplest way is to assemble it once, save it using *SAVE and load it from each of your programs using *LOAD. For this to work, the machine code program must be loaded into the same address each time. The most convenient way to arrange this is to move HIMEM down by the length of the program and load the machine code program in to this protected area. Theoretically, you could raise LOMEM to provide a similar protected area below your BBC BASIC (Z80) program. However, altering LOMEM destroys ALL your dynamic variables and is riskier.

Length of Reserved Memory

You must reserve an area of memory which is sufficiently large for your machine code program before you assemble it, but you may have no real idea how long the program will be until after it is assembled. How then can you know how much memory to reserve? Unfortunately, the answer is that you can't. However, you can add to your program to find the length used and then change the memory reserved by the DIM statement to the correct amount.

In the example below, a large amount of memory is initially reserved. To begin with, a single pass is made through the assembly code and the length needed for the code is calculated (lines 100 to 120). After a CLEAR, the correct amount of memory is reserved (line 140) and a further two passes of the assembly code are performed as usual. Your program should not, of course, subsequently try to use variables set before the clear statement. If you use a similar structure to the example and place the program lines which initiate the assembly function at the start of your program, you can place your assembly code anywhere you like and still avoid this problem.

```
100 DIM free -1, code HIMEM-free-1000
110 PROC_ass(0)
120 L%=P%-code
130 CLEAR
140 DIM code L%
150 PROC_ass(0)
160 PROC_ass(2)

- - -
Put the rest of your program here.
- - -

1000 DEF PROC_ass(opt)
10010 P%=code
10020 [OPT opt
- - -
Assembler code program.
- - -

11000 ]
11010 ENDPROC
```

Initial Setting of the Program Counter

The program counters, P%, and O% are initialised to zero. Using the assembler without first setting P% (and O%) is liable to corrupt BBC BASIC (Z80)'s workspace, see the Annex entitled Format of Program and Variables in Memory.

The Assembly Process

OPT

The only assembly directive is OPT. As with the 6502 assembler, 'OPT' controls the way the assembler works, whether a listing is displayed and whether errors are reported. OPT should be followed by a number in the range 0 to 7. The way the assembler functions is controlled by the three bits of this number in the following manner.

Bit 0 – LSB

Bit 0 controls the listing. If it is set, a listing is displayed.

Bit 1

Bit 1 controls the error reporting. If it is set, errors are reported.

Bit 2

Bit 2 controls where the assembled code is placed. If bit 2 is set, code is placed in memory starting at the address specified by O%. However, the program counter (P%) is still used by the assembler for calculating the instruction addresses.

Assembly at a Different Address

In general, machine code will only run properly if it is in memory at the addresses for which it was assembled. Thus, at first glance, the option of assembling it in a different area of memory is of little use. However, using this facility, it is possible to build up a library of machine code utilities for use by a number of programs. The machine code can be assembled for a particular address by one program without any constraints as to its actual location in memory and saved using *SAVE. This code can then be loaded into its working location from a number of different programs using *LOAD.

OPT Summary

Code Assembled Starting at P%

The code is assembled using the program counter (P%) to calculate the instruction addresses and the code is also placed in memory at the address specified by the program counter.

OPT 0	reports no errors and gives no listing.
OPT 1	reports no errors, but gives a listing.
OPT 2	reports errors, but gives no listing.
OPT 3	reports errors and gives a listing.

Code Assembled Starting at O%

The code is assembled using the program counter (P%) to calculate the instruction addresses. However, the assembled code is placed in memory at the address specified by O%.

OPT 4	reports no errors and gives no listing.
OPT 5	reports no errors, but gives a listing.
OPT 6	reports errors, but gives no listing.
OPT 7	reports errors and gives a listing.

How the Assembler Works

The assembler works line by line through the machine code. When it finds a label declared it generates a BBC BASIC (Z80) variable with that name and loads it with the current value of the program counter (P%). This is fine all the while labels are declared before they are used. However, labels are often used for forward jumps and no variable with that name would exist when it was first encountered. When this happens, a 'No such variable' error occurs. If error reporting has not been disabled, this error is reported and BBC BASIC (Z80) returns to the direct mode in the normal way. If error reporting has been disabled (OPT 0, 1, 4 or 5), the current value of the program counter is used in place of the address which would have been found in the variable, and assembly continues. By the end of the assembly process the variable will exist (assuming the code is correct), but this is of little use since the assembler cannot 'back track' and correct the errors. However, if a second pass is made through the assembly code, all the labels will exist as variables and errors will not occur.

The example below shows the result of two passes through a (completely futile) demonstration program. Twelve bytes of memory are reserved for the program. (If the program was run, it would 'doom-loop' from line 50 to 70 and back again.) The program disables error reporting by using OPT 1.

```
10 DIM code 12
20 FOR opt=1 TO 3 STEP 2
30 P%=code
40 [OPT opt
50 .jim JR fred
60 DEFW &2345
70 .fred JR jim
80 ]
```

```
90 NEXT
```

This is the first pass through the assembly process (note that the 'JR fred' instruction jumps to itself):

```
3E7B          OPT opt
3E7B 18 FE    .jim JR fred
3E7D 45 23    DEFW &2345
3E7F 18 FA    .fred JR jim
```

This is the second pass through the assembly process (note that the 'JMP fred' instruction now jumps to the correct address):

```
3E7B          OPT opt
3E7B 18 02    .jim JR fred
3E7D 45 23    DEFW &2345
3E7F 18 FA    .fred JR jim
```

Generally, if labels have been used, you must make two passes through the assembly language code to resolve forward references. This can be done using a FOR...NEXT loop. Normally, the first pass should be with OPT 0 (or OPT 4) and the second pass with OPT 2 (OPT 6). If you want a listing, use OPT 3 (OPT7) for the second pass. During the first pass, a table of variables giving the address of the labels is built. Labels which have not yet been included in the table (forward references) will generate the address of the current opcode. The correct address will be generated during the second pass.

Saving and Loading Machine Code Programs

As mentioned earlier, you can use machine code routines in a number of BBC BASIC (Z80) programs by using *SAVE and *LOAD. The safest way to do this is to write a program which consists of only the machine code routines and enough BBC BASIC (Z80) to assemble them. They should be assembled 'out of the way' at the top of memory (each routine starting at a known address) and then *SAVEd. (Don't forget to move HIMEM down first.) The BBC BASIC (Z80) programs that use these routines should move HIMEM down to the same value before they *LOAD the assembly code routines into the address at which they were originally assembled. *SAVE and *LOAD are explained below.

*SAVE

Save an area of memory to a file. You MUST specify the start address (aaaa) and either the length of the area of memory (llll) or its end address+1 (bbbb).

```
*SAVE ufsp aaaa +llll
*SAVE ufsp aaaa bbbb
OSCLI "SAVE "+<st>+" "+STR$~(<n>)+"+"+STR$~(<n>)
*SAVE "WOMBAT" 8F00 +80
*SAVE "WOMBAT" 8F00 8F80
OSCLI "SAVE "+ufn$+" "+STR$~(add)+"+"+STR$~(len)
```

*LOAD

Load the specified file into memory at hexadecimal address 'aaaa'. The load address MUST always be specified. OSCLI may also be used to load a file. However, you must take care to provide the load address as a hexadecimal number in string format.

```
*LOAD ufsp aaaa
OSCLI "LOAD "+<str>+" "+STR$~<num>

*LOAD A:WOMBAT 8F00
OSCLI "LOAD "+f_name$+" "+STR$~(strt_address)
```

Conditional Assembly and Macros

Introduction

Most machine code assemblers provide conditional assembly and macro facilities. The assembler does not directly offer these facilities, but it is possible to implement them by using other features of BBC BASIC (Z80).

Conditional Assembly

You may wish to write a program which makes use of special facilities and which will be run on different types of computer. The majority of the assembly code will be the same, but some of it will be different. In the example below, different output routines are assembled depending on the value of 'flag'.

```
DIM code 200
FOR pass=0 TO 3 STEP 3
 [OPT pass
 .start    - - -
        - - - code - - -
        - - - :]
 :
 IF flag  [OPT  pass: - code for routine 1 -:]
 IF NOT flag [OPT pass: - code for routine 2 - :]
 :
 [OPT pass
 .more_code - - -
        - - - code - - -
        - - -:]
NEXT
```

Macros

Within any machine code program, it is often necessary to repeat a section of code a number of times and this can become quite tedious. You can avoid this repetition by defining a macro which you use every time you want to include the code. The example below uses a macro to pass a character to the operating system. Conditional assembly is used within the macro to select either the normal CP/M routine or one applicable to the Torch, depending on the value of op_flag.

It is possible to suppress the listing of the code in a macro by forcing bit 0 of OPT to zero for the duration of the macro code. This can most easily be done by ANDing the value passed to OPT with 6. This is illustrated in PROC_normal and PROC_torch in the example below.

```
DIM code 200
op_flag=TRUE
FOR pass=0 TO 3 STEP 3
 [OPT pass
 .start  - - -
       - - - code - - -
       - - -
:
 OPT FN_select(op_flag); Include code depending on op_flag
:
       - - -
```

```
        - - - code - - -
        - - -:]
NEXT
END
:
:
REM Include code depending on value of op_flag
:
DEF FN_select(op_flag)
IF op_flag PROC_torch ELSE PROC_normal
=pass
REM Return original value of OPT.  This is a
REM bit artificial, but necessary to insert
REM some BBC BASIC code in the assembly code.
:
DEF PROC_torch
[OPT pass AND 6
LD E,A
RST &30
DEFB 2
RET:]
ENDPROC
:
DEF PROC_normal
[OPT pass AND 6
PUSH BC
LD C,2
LD E,A
CALL 5
POP BC
RET:]
ENDPROC
```

The use of a function call to incorporate the code provides a neat way of incorporating the macro within the program and allows parameters to be passed to it. The function should return the original value of OPT.

4. Statements and Functions

Introduction

The commands and statements are listed alphabetically for ease of reference; they are not separated into two sections.

All statements, except INPUT, can also be used as direct commands.

Where appropriate, the abbreviated form is shown to the right of the statement. The associated keywords are listed at the end of each explanation.

If the lexical analyser tries to expand a line to more than 255 characters, a 'Line space' error will be reported.

Syntax

Abbreviated definitions for the commands and statements in BBC BASIC (Z80) are given at the end of the explanation for each keyword. Most of us have seen formal syntax diagrams and Backus-Naur Form (BNF) definitions for languages, and many of us have found them to be somewhat confusing. Consequently, we have attempted to produce something which, whilst being reasonably precise, is readable by the majority of BBC BASIC (Z80) users. To those amongst you who would have preferred 'the real thing' - we apologise.

Symbols

The following symbols have special meaning in the syntax definitions.

{ }	The enclosed item may be repeated zero or more times.
[ ]	The items enclosed are optional, they may occur zero or one time.
|	Indicates alternatives; one of which must be used.
<stmt>	Means a BBC BASIC (Z80) statement.

<var>	Means a numeric or string variable.
<exp>	Means an expression like PI*radius*height+2 or name$+"FRED"+CHR$(&0D). It can also be a single variable or constant like 23 or "FRED".
<l-num>	Means a line number in a BBC BASIC (Z80) program.
<k-num>	Means the number of one of the programmable keys.
<n-const>	Means a numeric constant like '26.4' or '256'.
<n-var>	Means a numeric variable like 'size' or 'weight'.
<numeric>	Means a <n-const> or a <n-var> or an expression combining them. For example: PI*radius+2.66
<s-const>	Means a string constant like "FRED".
<string>	Means an unquoted string of characters.
<s-var>	Means a string variable like 'address$'.
<str>	Means a <s-const> or a <s-var> or an expression combining them. For example: name$+add$+"Phone".
<t-cond>	Means a 'testable condition'. In other words, something which is either TRUE or FALSE. Since BBC BASIC does not have true Boolean variables, TRUE and FALSE are numeric (with a value of -1 and 0). Consequently, a <numeric> can be used anywhere a <t-cond> is specified.
<name>	Means a valid variable name.
<d:>	Means a drive name (A: to P:).
<afsp>	Means an ambiguous file specifier.
<ufsp>	Means an unambiguous file specifier.

<nchr>	Means a character valid for use in a name. 0 to 9, A to Z, a to z and underline.

ABS

A function giving the absolute value of its argument.

```
X = ABS(deficit)
length = ABS(X1-X2)
```

This function converts negative numbers into positive ones. It can be used to give the difference between two numbers without regard to the sign of the answer.

It is particularly useful when you want to know the difference between two values, but you don't know which is the larger. For instance, if X=6 and Y=10 then the following examples would give the same result.

```
difference = ABS(X-Y)
difference = ABS(Y-X)
```

You can use this function to check that a calculated answer is within certain limits of a specified value. For example, suppose you wanted to check that 'answer' was equal to 'ideal' plus or minus (up to) 0.5. One way would be:

```
IF answer>ideal-0.5 AND answer<ideal+0.5 THEN....
```

However, the following example would be a more elegant solution.

```
IF ABS(answer-ideal)<0.5 THEN....
```

Syntax

```
<n-var>=ABS(<numeric>)
```

Associated Keywords

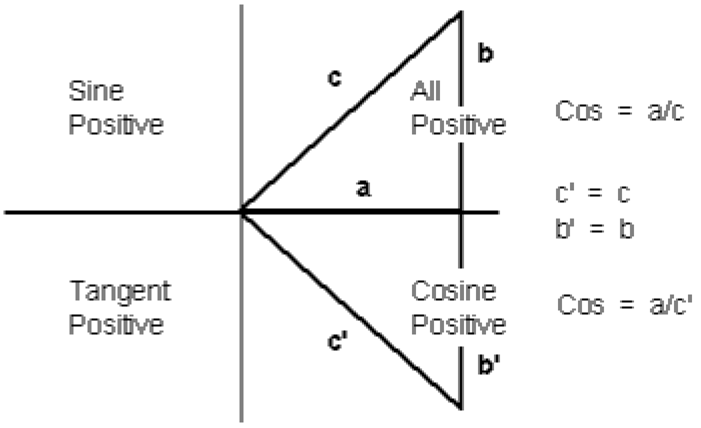
SGN

ACS

A function giving the arc cosine of its argument in radians. The permitted range of the argument is -1 to +1.

If you know the cosine of the angle, this function will tell you the angle (in radians). Unfortunately, you cannot do this with complete certainty because two angles within the range +/- PI (+/- 180 degrees) can have the same cosine. This means that one cosine has two associated angles.

The following diagram illustrates the problem:



Within the four quadrants, there are two angles which have the same cosine, two with the same sine and two with the same tangent. When you are working back from the cosine, sine or tangent you don't know which of the two possible angles is correct.

By convention, ACS gives a result in the top two quadrants (0 to PI - 0 to 180 degrees) and ASN and ATN in the right-hand two quadrants (-PI/2 to +PI/2 - -90 to + 90 degrees).

In the example below, 'radian_angle' becomes equal to the angle (in radians) whose cosine is 'y'.

```
radian_angle=ACS(y)
```

You can convert the answer to degrees by using the DEG function (or multiplying by 180/PI).

```
degree_angle=DEG(ACS(y))
```

Syntax

```
<n-var>=ACS(<numeric>)
```

Associated Keywords
ASN, ATN, SIN, COS, TAN, RAD, DEG

ADVAL

This function returns information about the analogue to digital converter channels or joystick (if present). The information returned depends upon the argument passed to ADVAL.

Not implemented in the generic CP/M version of BBC BASIC (Z80)
Syntax

```
<n-var>=ADVAL(<numeric>)
```

Associated Keywords
SOUND, INPUT, GET

AND **A.**

The operation of integer bitwise logical AND between two items. The 2 operands are internally converted to four byte integers before the AND operation.

```
answer=num1 AND num2
char=byte AND &7F

IF (num AND &F0)
```

```
test=(count=3 AND total=5)
```

You can use AND as a logical operator or as a 'bit-by-bit' (bitwise) operator. The operands can be boolean (logical) or numeric.

In the following example program segment, AND is used as a bitwise operator to remove the most significant bit of a byte read from a file before writing it to another file. This is useful for converting some word-processor files into standard ASCII format.

```
210 byte=BGET#infile AND &7F
220 BPUT#outfile,byte
```

Unfortunately, BBC BASIC does not have true boolean variables; it uses numeric variables and assigns the value 0 for FALSE and -1 for TRUE. This can lead to confusion at times. (See NOT for more details.) In the example below, the operands are boolean (logical). In other words, the result of the tests (IF) A=2 and (IF) B=3 is either TRUE or FALSE. The result of this example will be TRUE if A=2 and B=3.

```
answer=(A=2 AND B=3)
```

The brackets are not necessary, they have been included to make the example easier to follow.

The second example is similar to the first, but in the more familiar surroundings of an IF statement.

```
IF A=2 AND B=3 THEN 110
```

Or

```
answer= A=2 AND B=3
```
(without brackets this time)

```
IF answer THEN 110
```

The final example uses the AND in a similar fashion to the numeric operators (+, -, etc).

```
A=X AND 11
```

Suppose X was -20, the AND operation would be:

```
11111111 11111111 11111111 11101100
00000000 00000000 00000000 00001011
00000000 00000000 00000000 00001000  = 8
```

Syntax

```
<n-var>=<numeric> AND <numeric>
```

Associated Keywords
EOR, OR, FALSE, TRUE, NOT

ASC

A function returning the ASCII character value of the first character of the argument string. If the string is null then -1 will be returned.

A computer only understands numbers. In order to deal with characters, each character is assigned a code number. For example (in the ASCII code table) the character 'A' is given the code number 65 (decimal). A part of the computer generates special electronic signals which cause the characters to be displayed on the screen. The signals generated vary according to the code number.

You could use this function to convert ASCII codes to some other coding scheme.

ascii_code=ASC("H")	Result would be 72
X=ASC("HELLO")	Result would be 72
name$="FRED"	
ascii_code=ASC(name$)	Result would be 70
X=ASC"e"	Result would be 101
X=ASC(MID$(A$,position))	Result depends on A$ and position.

ASC is the complement of CHR$.

Syntax

```
<n-var>=ASC(<str>)
```

Associated Keywords
CHR$, STR$, VAL

ASN

A function giving the arc sine of its argument in radians. The permitted range of the argument is -1 to +1.

By convention, the result will be in the range -PI/2 to +PI/2 (-90 to +90 degrees).
If you know the sine of the angle, this function will tell you the angle (in radians). Unfortunately, you cannot do this with complete certainty because one sine has two associated angles. (See ACS for details.)

In the example below, 'radian_angle' becomes equal to the angle (in radians) whose sine is 'y'.

```
radian_angle=ASN(y)
```

You can convert the answer to degrees by using the DEG function. (The DEG function is equivalent to multiplying by 180/PI.) The example below is similar to the first one, but the angle is in degrees.

```
degree_angle=DEG(ASN(y))
```

Syntax

```
<n-var>=ASN(<numeric>)
```

Associated Keywords
ACS, ATN, SIN, COS, TAN, RAD, DEG

ATN

A function giving the arc tangent of its argument in radians. The permitted range of the argument is from - to + infinity.

By convention, the result will be in the range -PI/2 to +PI/2 (-90 to +90 degrees).

If you know the tangent of the angle, this function will tell you the angle (in radians).

As the magnitude of the argument (tangent) becomes very large (approaches + or - infinity) the accuracy diminishes.

In the example below, 'radian_angle' becomes equal to the angle (in radians) whose tangent is 'y'.

```
radian_angle=ATN(y)
```

You can convert the answer to degrees by using the DEG function. (The DEG function is equivalent to multiplying by 180/PI.) The example below is similar to the first one, but the angle is in degrees.

```
degree_angle=DEG(ATN(y))
```

Syntax

```
<n-var>=ATN(<numeric>)
```

Associated Keywords
ACS, ASN, SIN, COS, TAN, RAD, DEG

AUTO **AU.**

A command allowing the user to enter lines without first typing in the number of the line. The line numbers are preceded by the usual prompt (>).

You can use this command to tell the computer to type the line numbers automatically for you when you are entering a program (or part of a program).

If AUTO is used on its own, the line numbers will start at 10 and go up by 10 for each line. However, you can specify the start number and the value by which the line numbers will increment. The step size can be in the range 1 to 255.

You cannot use the AUTO command within a program or a multi-statement command line.

You can leave the AUTO mode by pressing the escape key.

AUTO	start_number,step_size

AUTO	offers line numbers 10, 20, 30 ...
AUTO 100	starts at 100 with step 10
AUTO 100,1	starts at 100 with step 1
AUTO ,2	starts at 10 with step 2

A hyphen is an acceptable alternative to a comma.

Syntax

```
AUTO [<n-const> [,<n-const>]]
```

Associated Keywords
None

BGET# B.#

A function which gets a byte from the file whose file handle is its argument. The file pointer is incremented after the byte has been read.

```
E=BGET#n
aux=BGET#3
```

You must normally have opened a file using OPENOUT, OPENIN or OPENUP before you use this statement. (See these keywords and the BBC BASIC (Z80) Files section for details.)

You can use BGET# to read single bytes from a file. This enables you to read back small integers which have been 'packed' into fewer than 5 bytes (see BPUT#). It is also very useful if you need to perform some conversion operation on a file. Each byte read is numeric, but you can use CHR$(BGET#n) to convert it to a string.

The input file in the example below is a text file produced by a word-processor.

Words to be underlined are 'bracketed' with ^S. The program produces an output file suitable for a printer which expects such words to be bracketed by ^Y. You could, of course, perform several such translations in one program.

```
 10 REM Open i/p and o/p files. End if error.
 20 infile=OPENIN "WSFILE.DOC"
 30 IF infile=0 THEN END
 40 outfile=OPENOUT "BROTH.DOC"
 50 IF outfile=0 THEN END
 60 :
 70 REM Process file, converting ^S to ^Y
 80 REPEAT
 90   temp=BGET#infile :REM Read byte
100    IF temp=&13 THEN temp=&19 :REM Convert ^S
110    BPUT#outfile,temp :REM Write byte
120 UNTIL temp=&1A :REM ^Z
130 CLOSE#0 :REM Close all files
140 END
```

To make the program more useful, it could ask for the names of the input and output files at 'run time':

```
 10INPUT "Enter name of INPUT file " infile$
 20 INPUT "Enter name of OUTPUT file " outfile$
 30 REM Open i/p and o/p files. End if error.
 40 infile=OPENIN(infile$)
 50 IF infile=0 THEN END
 60 outfile=OPENOUT(outfile$)
 70 IF outfile=0 THEN END
 80 :
 90 REM Process file, converting ^S to ^Y
100 REPEAT
110    temp=BGET#infile :REM Read byte
120    IF temp=&13 THEN temp=&19 :REM Convert ^S
130    BPUT#outfile,temp :REM Write byte
140 UNTIL temp=&1A :REM ^Z
150 CLOSE#0 :REM Close all files
160 END
```

Syntax

```
<n-var>=BGET#<numeric>
```

Associated Keywords

OPENIN, OPENUP, OPENOUT, CLOSE#, PRINT#, INPUT#, BGET#, EXT#, GET$#, PT R#, EOF#

BPUT# BP.#

A statement which writes a byte or string to the data file whose channel number is the first argument. If the second argument is numeric, its least significant byte is written to the file; the file pointer is incremented after the byte has been written.

```
BPUT#E,32
BPUT#staff_file,A/256
```

Before you use this statement you must normally have opened a file for output using OPENOUT or OPENUP (see these keywords and the Files section for details).

You can use this statement to write single bytes to a file. The number that is sent to the file is in the range 0 to 255. Real numbers are converted internally to integers and the top three bytes are 'masked off'. Each byte written is numeric, but you can use ASC(character$) to convert (the first character of) 'character$' to a number.

The example below is a program segment that 'packs' an integer number between 0 and 65535 (&FFFF) into two bytes, least significant byte first. The file must have already been opened for output and the channel number stored in 'fnum'. The integer variable number% contains the value to be written to the file.

```
BPUT#fnum,number% MOD 256
BPUT#fnum,number% DIV 256
```

(BBC BASIC version 5 or later only)

Alternatively BPUT# can be used to write a character string to a file. The difference between PRINT#file,string$ and BPUT#file,string$ is that PRINT# appends a carriage-return character (CHR$13) whereas BPUT# appends a line-feed character (CHR$10).

Adding a terminating semicolon (;) causes the contents of the string to be written to the file with nothing appended.

```
BPUT#chn,A$
BPUT#N,"a string";
```

Syntax

```
BPUT#<numeric>,<numeric>
BPUT#<numeric>,<string>[;]
```

Associated Keywords
OPENIN, OPENUP, OPENOUT, CLOSE#, PRINT#, INPUT#, BGET#, GET$#, EXT#, PT
R#, EOF#, GET$#

BY

(BBC BASIC version 5 or later only)

Used in conjunction with DRAW, FILL, MOVE or PLOT, indicates that the supplied X and Y coordinates are *relative* (offsets from the current point) rather than *absolute*.

BY is also used with the GET$#channel function to specify the number of bytes to be read from the file.
Syntax

```
DRAW BY <numeric>,<numeric>
FILL BY <numeric>,<numeric>
MOVE BY <numeric>,<numeric>
PLOT BY <numeric>,<numeric>
```

Associated Keywords
DRAW, FILL, GET$, MOVE, PLOT

CALL **CA.**

A statement to call a machine code subroutine.

```
CALL Muldiv,A,B,C,D
CALL &FFE3
CALL 12340,A$,M,J$
```

The processor's A, B, C, D, E, F, H and L registers are initialised to the least significant words of A%, B%, C%, D%, E%, F%, H% and L% respectively (see also USR).

Parameter Table

CALL sets up a table in RAM containing details of the parameters. The IX register is set to the address of this parameter table.

Variables included in the parameter list need not have been declared before the CALL statement.

The parameter types are:

Code No	**Parameter Type**	
0:	byte (8 bits)	eg ?A%
1:	byte (8 bits)	eg A&
4:	word (32 bits)	eg !A% or A%
5:	real (40 bits)	eg A
128:	fixed string	eg $A%
129:	movable string	eg A$

Number of parameters	1 byte (at IX)
Parameter type	1 byte (at IX+1)
Parameter address	2 bytes (at IX+2 IX+3 LSB first)
Parameter type	)repeated as often as necessary.
Parameter address	)

Except in the case of a movable string (normal string variable), the parameter address given is the absolute address at which the item is stored. In the case of movable strings (type 129), it is the address of a parameter block containing the current length, the maximum length and the start address of the string, in that order

Parameter Formats

Integer variables are stored in twos complement format with their least significant byte first.

Fixed strings are stored as the characters of the string followed by a carriage return (&0D).

Floating point variables are stored in binary floating point format with their least significant byte first. The fifth byte is the exponent. The mantissa is stored as a binary fraction in sign and magnitude format. Bit 7 of the most significant byte is the sign bit and, for the purposes of calculating the magnitude of the number, this bit is assumed to be set to one. The exponent is stored as a positive integer in excess 127 format. (To find the exponent subtract 127 from the value in the fifth byte.)

If the exponent of a floating point number is zero, the number is stored in integer format in the mantissa. If the exponent is not zero, then the variable has a floating point value. Thus, an integer can be stored in two different formats in a real variable. For example, 5 can be stored as

& 00 00 00 05 00   Integer 5

Or

& 20 00 00 00 82   (.5+.125) * 2^3 = 5

(the &20 becomes &A0 because the MSB is always assumed)

In the case of a movable string (normal string variable), the parameter address points to the 'string descriptor'. This descriptor gives the current length of the string, the number of bytes allocated to the string (the maximum length of the string) and the address of the start of the string (LSB first).

Syntax

```
CALL <numeric>{,<n-var>|<s-var>}
```

Associated Keywords

USR

CASE

(BBC BASIC version 5 or later only)

A statement which results in different actions depending on the value of a numeric or string variable. The value is compared with each of the alternatives given; if it matches then the appropriate statements are executed. If the value does not match any of the alternatives, the statements following the OTHERWISE statement (if any) are executed. CASE ... ENDCASE clauses can be nested.

WHEN and ENDCASE must be the first thing on the program line and OF must be the last thing on the line (it cannot even be followed by a REMark).

```
CASE toss% OF
 WHEN 0 : coin$ = "tails"
 WHEN 1 : coin$ = "heads"
 OTHERWISE coin$ = "cheat"
ENDCASE

CASE direction$ OF
 WHEN "left","LEFT" :
 PRINT "turn left"
 WHEN "right","RIGHT" :
 PRINT "Turn right"
 OTHERWISE
 PRINT "Straight on"
ENDCASE
```

Syntax

```
CASE <numeric>|<string> OF
 WHEN <numeric>|<string>{,<numeric>|<string>} : {<stmt>}
 {<stmt>}
 WHEN <numeric>|<string>{,<numeric>|<string>} : {<stmt>}
 {<stmt>}
 OTHERWISE {<stmt>}
 {<stmt>}
ENDCASE
```

Associated Keywords
ENDCASE, IF, OF, ON, OTHERWISE, WHEN

CHAIN **CH.**

A statement which loads and runs the program whose name is specified in the argument.

```
CHAIN "GAME1"
CHAIN A$
```

The program file must be in tokenised format.

All but the static variables @% to Z% are CLEARed.

CHAIN sets ON ERROR OFF before chaining the specified program.

RUN may be used as an alternative to CHAIN.

You can use CHAIN (or RUN) to link program modules together. This allows you to write modular programs which would, if written in one piece, be too large for the memory available.

Passing data between CHAINed programs can be a bit of a problem because COMMON variables cannot be declared and all but the static variables are cleared by CHAIN.

If you wish to pass large amounts of data between CHAINed programs, you should use a data file. However, if the amount of data to be passed is small and you do not wish to suffer the time penalty of using a data file, you can pass data to the CHAINed program by using the indirection operators to store them at known addresses. The safest way to do this is to move HIMEM down and store common data at the top of memory.

The following sample program segment moves HIMEM down 100 bytes and stores the input and output file names in the memory above HIMEM. There is, of course, still plenty of room for other data in this area.

```
100 HIMEM=HIMEM-100
110 $HIMEM=in_file$
120 $(HIMEM+13)=out_file$
130 CHAIN "NEXTPROG"
```

Syntax

```
CHAIN <str>
```

Associated Keywords
LOAD, RUN, SAVE

CHR$

A function which returns a string of length 1 containing the ASCII character specified by the least significant byte of the numeric argument.

```
A$=CHR$(72)
B$=CHR$(12)
C$=CHR$(A/200)
```

CHR$ generates an ASCII character (symbol, letter, number character, control character, etc) from the number given. The number specifies the position of the generated character in the ASCII table (See Annex A). For example:

```
char$=CHR$(65)
```

will set char$ equal to the character 'A'. You can use CHR$ to send a special character to the terminal or printer. (Generally, VDU is better for sending characters to the screen.) For example,

```
CHR$(7)
```

will generate the ASCII character ^G. So,

```
PRINT "ERROR"+CHR$(7)
```

will print the message 'ERROR' and sound the PC's 'bell'.

CHR$ is the complement of ASC.

Syntax

```
<s-var>=CHR$(<numeric>)
```

Associated Keywords
ASC, STR$, VAL, VDU

CIRCLE

(BBC BASIC version 5 or later only, not implemented in the generic CP/M edition)

A statement which draws a circle or disc (filled circle) in all screen modes except MODE 7. CIRCLE is followed by the X and Y coordinates of the centre of the circle and the radius. To draw a filled (solid) circle rather than an outline circle, use CIRCLE FILL.

The circle or disc is drawn in the current graphics foreground colour. This colour can be changed using the GCOL statement.

CIRCLE x,y,r is equivalent to MOVE x,y : PLOT 145,r,0
CIRCLE FILL x,y,r is equivalent to MOVE x,y : PLOT 153,r,0

```
CIRCLE 200,300,40
CIRCLE FILL 300,400,100
```

Syntax

```
CIRCLE [FILL] <numeric>,<numeric>,<numeric>
```

Associated Keywords
ELLIPSE, FILL, GCOL, MOVE, PLOT

CLEAR **CL.**

A statement which clears all the dynamically declared variables, including strings. CLEAR does not affect the static variables.

The CLEAR command tells BBC BASIC (Z80) to 'forget' about ALL the dynamic variables used so far. This includes strings and arrays, but the static variables (@% to Z%) are not altered.

You can use the indirection operators to store integers and strings at known addresses and these will not be affected by CLEAR. However, you will need to 'protect' the area of memory used. The easiest way to do this is to move HIMEM down. See CHAIN for an example.

Syntax

```
CLEAR
```

Associated Keywords
None

CLOSE# **CLO.#**

A statement used to close a data file. CLOSE #0 will close all data files.

```
CLOSE#file_num
CLOSE#0
```

You use CLOSE# to tell BBC BASIC (Z80) that you have completely finished with a data file for this phase of the program. Any data still in the file buffer is written to the file before the file is closed.

You can open and close a file several times within one program, but it is generally considered 'better form' not to close a file until you have finally finished with it. However, if you wish to CLEAR the variables, it is simpler if you close the data files first.

You should also close data files before chaining another program. CHAIN does not automatically close data files, but it does clear the variables in which the file handles were stored. You can still access the open file if you have used one of the static variables (A% to Z%) to store the file handle. Alternatively, you could reserve an area of memory (by moving HIMEM down for example) and use the byte indirection operator to store the file handle. (See the keyword CHAIN for more details.)

END or 'dropping off' the end of a program will also close all open data files. However, STOP does not close data files.

Syntax

```
CLOSE#<numeric>
```

Associated Keywords
OPENIN, OPENUP, OPENOUT, PRINT#, INPUT#, BPUT#, BGET#, GET$#, EXT#, PTR#, EOF#

CLG
A statement which clears the graphics area of the screen and sets it to the currently selected graphics background colour,

Not implemented in the generic CP/M version of BBC BASIC (Z80)

Syntax

```
CLG
```

Associated Keywords
CLS, GCOL

CLS

A statement which clears the text area of the screen and sets it to the currently selected text background colour. The text cursor is moved to the 'home' position (0,0) at the top left-hand corner of the text area.

Syntax

```
CLS
```

Associated Keywords
CLG, COLOUR

COLOUR (COLOR) C.

Sets the text foreground and background colours. If the parameter is less than 128, the colour of the text is set. If the number is 128 or greater, the colour of the background is set.

Not implemented in the generic CP/M version of BBC BASIC (Z80)

Syntax

```
COLOUR<numeric>
```

Associated Keywords
VDU, GCOL, MODE

COS

A function giving the cosine of its radian argument.

```
X=COS(angle)
```

This function returns the cosine of an angle. The angle must be expressed in radians, not degrees.

Whilst the computer is quite happy dealing with angles expressed in radians, you may prefer to express angles in degrees. You can use the RAD function to convert an angle from degrees to radians.

The example below sets Y to the cosine of the angle 'degree_angle' expressed in degrees.

```
Y=COS(RAD(degree_angle))
```

Syntax

```
<n-var>=COS(<numeric>)
```

Associated Keywords
SIN, TAN, ACS, ASN, ATN, DEG, RAD

COUNT COU.

A function returning the number of characters sent to the output stream (VDU or printer) since the last new line.

```
char_count=COUNT
```

Characters with an ASCII value of less than 13 (carriage return/new-line/enter) have no effect on COUNT.

Because control characters above 13 are included in COUNT, you cannot reliably use it to find the position of the cursor on the screen. If you need to know the cursor's horizontal position use the POS function.

Count is NOT set to zero if the output stream is changed using the *OPT command.

The example below prints strings from the string array 'words$'. The strings are printed on the same line until the line length exceeds 65. When the line length is in excess of 65, a new-line is printed.

```
 90 ....
100 PRINT
110 FOR i=1 TO 1000
120 PRINT words$(i);
130 IF COUNT>65 THEN PRINT
140 NEXT
150 ....
```

Syntax

```
<n-var>=COUNT
```

Associated Keywords
POS

DATA

D.

A program object which must precede all lists of data for use by the READ statement.

As for INPUT, string values may be quoted or unquoted. However, quotes need to be used if the string contains commas or leading spaces.

Numeric values may include calculation so long as there are no keywords.

Data items in the list should be separated by a comma.

```
DATA 10.7,2,HELLO," THIS IS A COMMA,",1/3,PRINT
DATA " This is a string with leading spaces."
```

You can use DATA in conjunction with READ to include data in your program which you may need to change from time to time, but which does not need to be different every time you run the program.

The following example program segment reads through a list of names looking for the name in 'name$'. If the name is found, the name and age are printed. If not, an error message is printed.

```
100 DATA FRED,17,BILL,21,ALLISON,21,NOEL,32
110 DATA JOAN,26,JOHN,19,WENDY,35,ZZZZ,0
120 REPEAT
130 READ list$,age
140 IF list$=name$ THEN PRINT name$,age
150 UNTIL list$=name$ OR list$="ZZZZ"
160 IF list$="ZZZZ" PRINT "Name not in list"
```

Syntax

```
DATA <s-const>|<n-const>{,<s-const>|<n-const>}
```

Associated Keywords
READ, RESTORE

DEF

A program object which must precede declaration of a user defined function (FN) or procedure (PROC). DEF must be used at the start of a program line.

If DEF is encountered during execution, the rest of the line is ignored. As a consequence, single line definitions can be put anywhere in the program.

Multi-line definitions must not be executed. The safest place to put multi-line definitions is at the end of the main program after the END statement.

There is no speed advantage to be gained by placing function or procedure definitions at the start of the program.

```
DEF FNMEAN ....
DEF PROCJIM ....
```

In order to make the text more readable the function or procedure name may start with an underline.

```
DEF FN_mean ....
DEF PROC_Jim$ ....
```

Function and procedure names may end with a '$'. This is not compulsory for functions which return strings.

A procedure definition is terminated by the statement ENDPROC.

A function definition is terminated by a statement which starts with an equals (=) sign. The function returns the value of the expression to the right of the equals sign.

For examples of function and procedure declarations, see FN and PROC. For a general explanation of functions and procedures, refer to the Procedures and Functions sub-section in the General Information section.

Syntax

```
DEF PROC<name>[([RETURN]<s-var>|<n-var>{,[RETURN]<s-var>|<n-var>})]
DEF FN<name>[([RETURN]<s-var>|<n-var>{,[RETURN]<s-var>|<n-var>})]
```

Associated Keywords

ENDPROC, FN, PROC, RETURN

DEG

A function which converts radians to degrees.

```
degree_angle=DEG(PI/2)
X=DEG(ATN(1))
```

You can use this function to convert an angle expressed in radians to degrees. One radian is approximately 57 degrees (actually 180/PI). PI/2 radians is 90 degrees and PI radians is 180 degrees.

Using DEG is equivalent to multiplying the radian value by 180/PI, but the result is calculated internally to a greater accuracy.

See ACS, ASN and ATN for further examples of the use of DEG.

Syntax

```
<n-var>=DEG(<numeric>)
```

Associated Keywords
RAD, SIN, COS, TAN, ACS, ASN, ATN, PI

DELETE **DEL.**

A command which deletes a group of lines from the program. Both start and end lines of the group will be deleted.

You can use DELETE to remove a number of lines from your program. To delete a single line, just type the line number followed by <Enter>.

The example below deletes all the lines between line 10 and 15 (inclusive).

```
DELETE 10,15
```

To delete up to a line from the beginning of the program, use 0 as the first line number. The following example deletes all the lines up to (and including) line 120.

```
DELETE 0,120
```

To delete from a given line to the end of the program, use 65535 as the last line number. To delete from line 2310 to the end of the program, type:

```
DELETE 2310,65535
```

A hyphen is an acceptable alternative to a comma.

Syntax

```
DELETE <n-const>,<n-const>
```

Associated Keywords
EDIT, LIST, OLD, NEW

DIM

DIM can be used as a statement or as a function. When used as a statement, DIM has two different forms: the first declares an array and the second reserves an area of memory for special applications. When used as a function, DIM returns the number of dimensions in an array or the size of a particular dimension.

Dimensioning arrays

The DIM statement is used to declare arrays. Arrays must be pre-declared before use and once declared their dimensions cannot be changed (with the exception of LOCAL arrays). Arrays may be integer numeric, byte numeric, real numeric or string and they may be multi-dimensional. However, you cannot mix integers, bytes, reals and/or strings in the same array.

```
DIM A(2),A&(2,3),A$(2,3,4),A%(3,4,5,6)
```

All elements in the array are initialised to zero (for numeric arrays) or to empty strings (for string arrays).

The subscript base is 0, so DIM X(12) defines an array of 13 elements whose subscript value can be from 0 to 12 inclusive.

Arrays are like lists or tables. A list of names is a single dimension array. In other words, there is only one column - the names. Its single dimension in a DIM statement would be the maximum number of names you expected in the table less 1.

If you wanted to describe the position of the pieces on a chess board you could use a two dimensional array. The two dimensions would represent the row (numbered 0 to 7) and the column (also numbered 0 to 7). The contents of each 'cell' of the array would indicate the presence (if any) of a piece and its value.

```
DIM chess_board(7,7)
```

Such an array would only represent the chess board at one moment of play. If you wanted to represent a series of board positions you would need to use a three dimensional array. The third dimension would represent the 'move number'. Each move would use about 320 bytes of memory, so you could record 40 moves in about 12.5k bytes.

```
DIM chess_game(7,7,40)
```

The 'DIM space' error will occur if you attempt to declare an array for which there is insufficient memory.

Reserving an area of memory

The DIM statement can be used to reserve an area of memory which the interpreter will not then use. The variable specified in the DIM statement is set to the start address of this memory area. This reserved area can then be used by the indirection operators, assembly language code, etc.

The example below reserves 68 bytes of memory from the *heap* and sets a% equal to the address of the first byte. Thus a%?0 to a%?67 are free for use by the program (68 bytes in all):

```
DIM a% 67
```

The amount of memory reserved is one byte greater than the value specified. DIM p% −1 is a special case; it reserves zero bytes of memory. This is of more use than you might think, since it tells you the limit of the dynamic variable allocation (the *heap*) so far. Thus,

```
DIM p% -1
PRINT HIMEM-p%
```

is the equivalent of PRINT HIMEM-END (or PRINT FREE(0) in some other dialects of BASIC).

Memory reserved in this way causes irreversible loss of *heap* space, therefore you must be very careful not to reserve the memory multiple times when once will suffice. This is a particular danger when DIM is used within a function or procedure.

Reserving a temporary area of memory

(BBC BASIC version 5 or later only)

If you want to reserve a temporary block of memory for use within a function or procedure you can use use the following construct:

```
DIM a% LOCAL 67
```

This reserves 68 bytes from the *stack* rather than from the *heap*, and the memory is automatically freed on exit from the function or procedure.

Note that the variable (a% in this case) is not automatically made LOCAL to the function or procedure. In most cases you will want to do that as well:

```
LOCAL a%
DIM a% LOCAL 67
```

The amount of memory reserved is one byte greater than the value specified. DIM p% LOCAL −1 is a special case; it reserves zero bytes of memory. This is of more use than you might think, since it tells you the current size of the *stack*:

```
DIM p% LOCAL -1
PRINT HIMEM-p%
```

DIM as a function

(BBC BASIC version 5 or later only)

You can use DIM as a function to discover the number of dimensions in an array or the size of an array dimension. The function DIM(array()) returns the number of dimensions in the array (note the use of opening and closing brackets with nothing in between). The function DIM (array(),n) returns the size of dimension 'n', where n is in the range 1 to the number of dimensions.

If an array is declared as follows:

```
DIM cuboid(4,5,7)
```

the function DIM(cuboid()) will return the value 3, the function DIM(cuboid(),1) will return the value 4, the function DIM(cuboid(),2) will return the value 5 and the function DIM(cuboid(),3) will return the value 7.

Discovering the size of an array is of particular interest when a whole array is passed as a parameter to a function or procedure. For example, if you write a general-purpose function to calculate the geometric mean of the values in a numeric array, the function must know the total number of elements in the array.

Syntax

```
DIM <n-var>|<s-var>(<numeric>{,<numeric>})
DIM <name>{<member>{,<member>}}
DIM <n-var> [LOCAL] <numeric>
<n-var> = DIM(<array()>[,<numeric>])
```

Associated Keywords
CLEAR, HIMEM, LOCAL, LOMEM

DIV

A binary operation giving the integer quotient of two items. The result is always an integer.

```
X=A DIV B
y=(top+bottom+1) DIV 2
```

You can use this function to give the 'whole number' part of the answer to a division.
For example,

```
21 DIV 4
```

would give 5 (with a 'remainder' of 1).

Whilst it is possible to use DIV with real numbers, it is really intended for use with integers. If you do use real numbers, BBC BASIC (Z80) converts them to integers by truncation before DIViding them.

Syntax

```
<n-var>=<numeric> DIV <numeric>
```

Associated Keywords
MOD

DRAW

A statement which draws a line on the screen in the graphics modes. The statement is followed by the absolute or relative X and Y coordinates of the end of the line.

Not implemented in the generic CP/M version of BBC BASIC (Z80)

Syntax

```
DRAW [BY] <numeric>,<numeric>
```

Associated Keywords
MODE, PLOT, MOVE, CLG, VDU, GCOL

EDIT

E.

(CP/M edition only, other versions may have a similar command, consult the relevant documentation)

A command to edit or concatenate and edit the specified program line(s). The specified lines (including their line numbers) are listed as a single line. By changing only the line number, you can also use EDIT to duplicate a line.

```
EDIT 230
EDIT 200,230
```

The codes shown are the default codes as supplied; they may be changed by editing the **DIST.Z80** patch program:

Ctrl/E	Move the cursor up one line
Ctrl/X	Move the cursor down one line
Ctrl/S	Move the cursor left one character
Ctrl/D	Move the cursor right one character
Ctrl/A	Move the cursor to the start of the line
Ctrl/F	Move the cursor to the end of the line
Del	Backspace and delete
Ctrl/G	Delete the character at the cursor
Ctrl/U	Clear line to the left of the cursor
Ctrl/T	Clear the line to the right of the cursor
Ctrl/V	Insert a space at the cursor position
Enter	Enter the line and exit the edit mode
Esc	Abort and leave the line unchanged

To abort the single line editor and leave the line unchanged, press <Esc>.

You can use the EDIT command to edit and join (concatenate) program lines. When you use it to join lines, remember to delete any unwanted ones. EDIT on its own will start at the beginning of the program and concatenate as many lines as it can. This process will stop when the concatenated line length exceeds 255.

Syntax

```
EDIT <l-num>
EDIT <l-num>,<l-num>
```

Associated Keywords
DELETE, LIST, OLD, NEW

ELLIPSE

(BBC BASIC version 5 or later only, not implemented in the generic CP/M edition)

A statement which draws an outline ellipse or filled ellipse, in all screen modes except MODE 7. ELLIPSE is followed by the X and Y coordinates of the centre of the ellipse, the horizontal radius and the vertical radius, in that order. To draw a filled ellipse, use ELLIPSE FILL.

Note: In *BBC BASIC (Z80)* this statement can draw only **axis-aligned** ellipses.
The ellipse is drawn in the current graphics foreground colour. This colour can be changed using the GCOL statement.

```
ELLIPSE x,y,a,b is equivalent to MOVE x,y : PLOT 0,a,0 : PLOT 193,0,b
ELLIPSE FILL x,y,a,b is equivalent to MOVE x,y : PLOT 0,a,0 : PLOT 201,0,b
ELLIPSE 200,300,40,50
ELLIPSE FILL 300,400,100,80
```

Syntax

```
ELLIPSE [FILL] <numeric>,<numeric>,<numeric>,<numeric>
```

Associated Keywords
CIRCLE, FILL, GCOL, MOVE, PLOT

ELSE EL.

A statement delimiter which provides an alternative course of action in IF...THEN, ON...GOSUB, ON...GOTO and ON...PROC statements.

In an IF statement, if the conditional expression evaluates to **zero**, the statements after ELSE will be executed. This makes the following work:

```
IF A=B THEN B=C ELSE B=D
IF A=B THEN B=C:PRINT"WWW" ELSE B=D:PRINT"QQQ"
IF A=B THEN B=C ELSE IF A=C THEN..............
```

In a multi statement line containing more than one IF, the statement(s) after the ELSE delimiter will be actioned if ANY of the tests fail. For instance, the example below would print the error message 'er$' if 'x' did not equal 3 OR if 'a' did not equal 'b'.

```
IF x=3 THEN IF a=b THEN PRINT a$ ELSE PRINT er$
```

If you want to 'nest' the tests, you should use a multi-line IF ... ENDIF statement. The following example would print "Bad" ONLY if x was equal to 3 AND 'a' was not equal to 'b'.

```
IF x=3 THEN
  IF a=b THEN PRINT a$ ELSE PRINT "Bad"
ENDIF
```

ELSE can be used to indicate an alternative course of action in a multi-line IF ... THEN ... ENDIF statement ***(BBC BASIC version 5 or later only)***:

```
IF x=3 THEN
  PRINT "x is 3"
ELSE
  PRINT "x is not 3"
ENDIF
```

In this case ELSE **must** be the first thing on the line. You can cascade multiple IF ... ENDIF statements to test a number of different alternatives:

```
IF x=3 THEN
 PRINT "x is 3"
ELSE IF x=4 THEN
  PRINT "x is 4"
 ELSE
  PRINT "x is neither 3 nor 4"
 ENDIF
ENDIF
```

but in this situation a CASE statement would be a better choice.

You can use ELSE with ON...GOSUB, ON...GOTO and ON...PROC statements to prevent an out of range control variable causing an 'ON range' error.

```
ON action GOTO 100, 200, 300 ELSE PRINT "Error"
ON number GOSUB 100,200,300 ELSE PRINT "Error"
ON value PROCa,PROCb,PROCc ELSE PRINT "Error"
```

Syntax

```
IF <t-cond> [THEN <stmt>{:<stmt>}] [ELSE <stmt>{:<stmt>}]
ELSE [<stmt>]
ON <n-var> GOTO <l-num>{,<l-num>} ELSE <stmt>
ON <n-var> GOSUB <l-num>{,<l-num>} ELSE <stmt>
ON <n-var> PROC<name>{,PROC<name>} ELSE <stmt>
```

Associated Keywords:
IF, THEN, ON, ENDIF

END

A statement causing the interpreter to return to immediate mode. There can be any number (>=0) of END statements anywhere in a program. END closes all open data files.

END tells BBC BASIC that it has reached the end of the program. You don't have to use END, just 'running out of program' will have the same effect, but it's a bit messy.
You should use END to stop BBC BASIC 'falling into' any procedure or function definitions at the end of your program.

(BBC BASIC version 5 or later only)

END can also be used as a function. It returns the address of the first byte above BASIC's dynamic variable area (the *heap*). These two statements are equivalent:

```
e% = END
DIM e% -1
```

Syntax

```
END
<n-var> = END
```

Associated Keywords

QUIT, STOP, CLOSE#

ENDCASE

(BBC BASIC version 5 or later only)

A keyword which terminates a CASE...ENDCASE clause. ENDCASE must be the first item on the program line.

```
CASE die% OF
  WHEN 1,2,3 : bet$ = "lose"
  WHEN 4,5,6 : bet$ = "win"
  OTHERWISE bet$ = "cheat"
ENDCASE
```

Syntax

```
ENDCASE
```

Associated Keywords
CASE, OF, OTHERWISE, WHEN

ENDIF

(BBC BASIC version 5 or later only)

A keyword which terminates a multi-line IF clause. ENDIF must be the first thing on the program line.

Syntax

```
ENDIF
```

Associated Keywords
ELSE, IF, THEN

ENDPROC

A statement denoting the end of a procedure.

All local variables and the dummy arguments are restored at ENDPROC and the program returns to the statement after the calling statement.

Syntax

```
ENDPROC
```

Associated Keywords
DEF, FN, PROC, LOCAL

ENDWHILE

(BBC BASIC version 5 or later only)

A keyword which terminates a WHILE... ENDWHILE clause. If the ENDWHILE statement is executed, BBC BASIC jumps to the matching WHILE statement. If the WHILE condition is not met, control is transferred to the statement *after* the matching ENDWHILE statement.

```
WHILE LEFT$(A$,1) = " " A$ = MID$(A$,2) : ENDWHILE
```

Syntax

```
ENDWHILE
```

Associated Keywords
REPEAT, UNTIL, WHILE

ENVELOPE

A statement which is used, in conjunction with the SOUND statement, to control the pitch of a sound whilst it is playing.

Not implemented in the generic CP/M version of BBC BASIC (Z80)

Syntax

```
ENVELOPE <numeric>, <numeric>, <numeric>, <numeric>, <numeric>, <numeric>, <numeric>, <numeric>,
<numeric>, <numeric>, <numeric>, <numeric>, <numeric>, <numeric>
```

Associated Keywords
SOUND

EOF#

A function which will return -1 (TRUE) if the data file whose file handle is the argument is at, or beyond, its end. In other words, when PTR# points beyond the current end of the file. When reading a serial file, EOF# would go true when the last byte of the file had been read.

EOF# is only true if PTR# is set beyond the last byte written to the file. It will NOT be true if an attempt has been made to read from an empty block of a sparse random access file. Because of this, it is difficult to tell which records of a random access file have had data written to them. These files need to be initialised and the unused records marked as empty.

Writing to a byte beyond the current end of file updates the file length immediately, whether the record is physically written to the device at that time or not. However, the file must be closed in order to ensure that all the data written to it is physically written to the device.

Syntax

```
<n-var>=EOF#(<numeric>)
```

Associated Keywords
OPENIN, OPENUP, OPENOUT, CLOSE#, PRINT#, INPUT#, BPUT#, EXT#, PTR#

EOR

The operation of bitwise integer logical exclusive-or between two items. The two operands are internally converted to 4 byte integers before the EOR operation. EOR will return a non-zero result if the two items are different.

```
X=B EOR 4
IF A=2 EOR B=3 THEN 110
```

You can use EOR as a logical operator or as a 'bit-by-bit' (bitwise) operator. The operands can be boolean (logical) or numeric.

Unfortunately, BBC BASIC does not have true boolean variables; it uses numeric variables and assigns the value 0 for FALSE and -1 for TRUE. This can lead to confusion at times. (See NOT for more details.)

In the example below, the operands are boolean (logical) and the result of the tests (IF) A=2 and (IF) B=3 is either TRUE or FALSE.

The result of this example will be FALSE if A=2 and B=3 or A<>2 and B<>3. In other words, the answer will only be TRUE if the results of the two tests are different.

```
answer=(A=2 EOR B=3)
```

The brackets are not necessary, they have been included to make the example easier to follow.

The last example uses EOR in a similar fashion to the numeric operators (+, -, etc).

```
A=X EOR 11
```

Suppose X was -20, the EOR operation would be:

```
11111111 11111111 11111111 11101100
00000000 00000000 00000000 00001011
11111111 11111111 11111111 11100111  = -25
```

Syntax

```
<n-var>=<numeric> EOR <numeric>
```

Associated Keywords
NOT, AND, OR

ERL

A function returning the line number of the line where the last error occurred.

```
X=ERL
```

If there was an error in a procedure call, the line number of the calling line would be returned, not the line number of the definition.

The number returned by ERL is the line number printed out when BBC BASIC (Z80) reports an error.

See the Error Handling sub-section for more information on error handling and correction.

Syntax

```
<n-var>=ERL
```

Associated Keywords
ON ERROR GOTO, ON ERROR OFF, REPORT, ERR

ERR

A function returning the error code number of the last error which occurred (see the Annex entitled Error Messages and Codes).

```
X=ERR
```

Once you have assumed responsibility for error handling using the ON ERROR statement, you can use this function to discover which error occurred.

See the Error Handling sub-section for more information on error handling and correction.

Syntax

```
<n-var>=ERR
```

Associated Keywords
ON ERROR GOTO, ON ERROR OFF, ERL, REPORT

ERROR

(BBC BASIC version 5 or later only)

A statement which can 'force' an error to occur. This can be useful for testing or for adding 'user-defined' errors. ERROR is followed by the error number and the error string:

```
200 ERROR 100,"Fault"
```

Unless errors are trapped using ON ERROR this will result in the message:

```
Fault at line 200
```

and ERR will be set to 100. User-defined errors should normally be given error numbers in the range 100 to 179 so that they don't conflict with the built-in error codes. However, if you specify an error number of zero BBC BASIC will treat it as a **fatal** error, i.e. it will not be trapped by the ON ERROR statement and will cause the program to end immediately.

Note that the error string is held in a temporary buffer, which will be overwritten if an INPUT, CALL or READ statement is executed, or if BBC BASIC returns to immediate mode. This means you should be careful when accessing the error string using REPORT or REPORT$; if necessary copy it into a conventional string variable first.

ERROR is also used in the ON ERROR, ON ERROR LOCAL and RESTORE ERROR statements.

Syntax

```
ERROR <numeric>,<string>
ON ERROR <stmt>{:<stmt>}
ON ERROR LOCAL <stmt>{:<stmt>}
ON ERROR OFF
RESTORE ERROR
```

Associated Keywords

ON ERROR, ON ERROR LOCAL, ERL, ERR, REPORT, RESTORE

EVAL **EV.**

A function which applies the interpreter's expression evaluation program to the characters held in the argument string.

```
X=EVAL("X^Q+Y^P")
X=EVAL"A$+B$"
X$=EVAL(A$)
```

In effect, you pass the string to BBC BASIC (Z80)'s evaluation program and say 'work this out'.

You can use this function to accept and evaluate an expression, such as a mathematical equation, whilst the program is running. You could, for instance, use it in a 'calculator' program to accept and evaluate the calculation you wished to perform.

Another use would be in a graph plotting program to accept the mathematical equation you wished to plot.

The example below is a 'bare bones' calculator program which evaluates the expression typed in by the user.

```
10 PRINT "This program evaluates the expression"
20 PRINT "you type in and prints the answer"
30 REPEAT
40   INPUT "Enter an expression" exp$
50   IF exp$<>"END" PRINT EVAL exp$
60 UNTIL exp$="END"
70 END
```

You can only use EVAL to work out functions (like SIN, COS, etc). It won't execute statements like MODE 0, PRINT, etc.

Syntax

```
<n-var>=EVAL(<str>)
<s-var>=EVAL(<str>)
```

Associated Keywords

STR$, VAL

EXIT **EX.**

(BBC BASIC version 5 or later only)

A statement which causes a premature exit from
a FOR...NEXT, REPEAT...UNTIL or WHILE...ENDWHILE loop.

EXIT FOR causes execution to continue after the matching NEXT statement, EXIT REPEAT causes execution to continue after the matching UNTIL statement and EXIT WHILE causes execution to continue after the matching ENDWHILE statement.

Typically you would use EXIT when a situation occurs which necessitates leaving the loop 'part way through':

```
FOR I% = start% TO finish%
  ...
  IF bool% THEN EXIT FOR
  ...
NEXT I%

REPEAT
  ...
  IF bool% THEN EXIT REPEAT
  ...
UNTIL condition%

WHILE condition%
  ...
  IF bool% THEN EXIT WHILE
  ...
ENDWHILE
```

In the case of EXIT FOR an optional loop variable can be specified, causing execution to continue after the NEXT statement which matches that variable:

```
FOR I% = start% TO finish%
  FOR J% = first% TO last%
    ...
    IF bool% THEN EXIT FOR I%
    ...
  NEXT
NEXT
REM Execution continues here
```

You can EXIT from a loop even when inside a nested loop of a different kind:

```
REPEAT
  FOR J% = first% TO last%
    ...
    IF bool% THEN EXIT REPEAT
    ...
  NEXT
UNTIL FALSE
REM Execution continues here
```

EXIT is not compatible with the use of non-standard FOR...NEXT constructs such as 'popping' an inner loop or listing multiple loop variables after NEXT. When EXIT is used, every FOR must have a matching NEXT.

Syntax

```
EXIT FOR [<n-var>]
EXIT REPEAT
EXIT WHILE
```

Associated Keywords

FOR, REPEAT, WHILE

EXP

A function returning 'e' to the power of the argument. The argument must be < 88.7228392. The 'natural' number, 'e', is approximately 2.71828183.

```
Y=EXP(Z)
```

This function can be used as the 'anti-log' of a natural logarithm. Logarithms are 'traditionally' used for multiplication (by adding the logarithms) and division (by subtracting the logarithms). For example,

```
10 log1=LN(2.5)
20 log2=LN(2)
30 log3=log1+log2
40 answer=EXP(log3)
```

```
50 PRINT answer
```

will calculate 2.5*2 by adding their natural logarithms and print the answer.

Syntax

```
<n-var>=EXP(<numeric>)
```

Associated Keywords
LN, LOG

EXT#

A function which returns the total length of the file whose file handle is its argument.

```
length=EXT#f_num
```

In the case of a sparse random-access file, the value returned is the complete file length from byte zero to the last byte written. This may be greater than the actual amount of data on the device, but it is the amount of device space allocated to the file by the Filing System.

The file must have been opened before EXT# can be used to find its length.

Syntax

```
<n-var>=EXT#(<numeric>)
```

Associated Keywords
OPENIN, OPENUP, OPENOUT, CLOSE#, PRINT#, INPUT#, BPUT#, BGET#, GET$#, PTR#, EOF#

FALSE **FA.**

A function returning the value zero.

```
10 flag=FALSE
20 ...
150 IF flag ...
```

BBC BASIC (Z80) does not have true Boolean variables. Instead, numeric variables are used and their value is interpreted in a 'logical' manner.

A value of zero is interpreted as FALSE and NOT FALSE (in other words, NOT 0) is interpreted as TRUE. In practice, any value other than zero is considered TRUE.

You can use FALSE in a REPEAT....UNTIL loop to make the loop repeat for ever. Consider the following example.

```
10 terminator=10
20 REPEAT
30 PRINT "An endless loop"
40 UNTIL terminator=0
```

Since 'terminator' will never be zero, the result of the test 'terminator=0' will always be FALSE. Thus, the following example has the same effect as the previous one.

```
10 REPEAT
20 PRINT "An endless loop"
30 UNTIL FALSE
```

Similarly, since FALSE=0, the following example will also have the same effect, but its meaning is less clear.

```
10 REPEAT
20 PRINT "An endless loop"
30 UNTIL 0
```

See the keyword AND for logical tests and their results.

Syntax

```
<n-var>=FALSE
```

Associated Keywords

TRUE, EOR, OR, AND, NOT

FILL

(BBC BASIC version 5 or later only, not implemented in the generic CP/M edition)

A statement which performs a 'flood fill' in the current graphics foreground colour, starting at the specified point and continuing until non-background-colour pixels are found. FILL is followed by the X and Y coordinates of the starting point; the optional qualifier BY indicates that the coordinates are *relative* (offsets from the current graphics cursor position) rather than *absolute*.

The operation may appear to fail if the graphics background colour is not available as a 'solid' colour, with the current display settings, and has therefore been approximated by 'dithering'.

```
FILL x,y is equivalent to PLOT 133,x,y
FILL BY x,y is equivalent to PLOT 129,x,y
```

FILL is also used with the CIRCLE, ELLIPSE and RECTANGLE statements to indicate that a filled (solid) shape should be drawn, or that a block move rather than a block copy is required.

```
FILL 400,400
```

Syntax

```
FILL <numeric>,<numeric>
FILL BY <numeric>,<numeric>
CIRCLE FILL <numeric>,<numeric>,<numeric>
ELLIPSE FILL <numeric>,<numeric>,<numeric>,<numeric>
RECTANGLE FILL <numeric>,<numeric>,<numeric>,<numeric> [TO <numeric>,<numeric>]
```

Associated Keywords
BY, CIRCLE, ELLIPSE, GCOL, RECTANGLE

FN

A keyword used at the start of all user declared functions. The first character of the function name can be an underline (or a number).

If there are spaces between the function name and the opening bracket of the parameter list (if any) they must be present both in the definition and the call. It's safer not to have spaces between the function name and the opening bracket.

A function may be defined with any number of parameters of any type, and may return (using =) a string or numeric result. It does not have to be defined before it is used.

A function definition is terminated by '=' used in the statement position.

The following examples show the '=' as part of a program line and at the start of a line. The first two examples are single line function definitions.

```
DEF FN_mean(Q1,Q2,Q3,Q4)=(Q1+Q2+Q3+Q4)/4

DEF FN_fact(N) IF N<2 =1 ELSE =N*FN_fact(N-1)

DEF FN_reverse(A$)
LOCAL B$,Z%
FOR Z%=1 TO LEN(A$)
  B$=MID$(A$,Z%,1)+B$
NEXT
=B$
```

Functions are re-entrant and the parameters (arguments) are passed by value.

You can write single line, multi statement functions so long as you have a colon after the definition statement.

The following function sets the print control variable to the parameter passed and returns a null string. It may be used in a PRINT command to change the print control variable (@%) within a print list.

```
DEF FN_pformat(N):@%=N:=""
```

Functions have to return an answer, but the value returned by this function is a null string. Consequently, its only effect is to change the print control variable. Thus the PRINT statement

```
PRINT FN_pformat(&90A) X FN_pformat(&2020A) Y
```

will print X in G9z10 format and Y in F2z10 format. See the keyword PRINT for print format details.

Syntax

```
<n-var>|<s-var>=FN<name>[(<exp>{,<exp>})]
DEF FN<name>[([RETURN]<n-var>|<s-var>{,[RETURN]<n-var>|<s-var>})]
```

Associated Keywords

ENDPROC, DEF, LOCAL

FOR

F.

A statement initialising a FOR...NEXT loop. The loop is executed at least once.

```
FOR temperature%=0 TO 9
FOR A(2,3,1)=9 TO 1 STEP -0.3
```

The FOR...NEXT loop is a way of repeating a section of program a set number of times. For example, the two programs below perform identically, but the second is easier to understand.

```
10 start=4: end=20: step=2
20 counter=start
30 PRINT counter," ",counter^2
40 counter=counter+step
50 IF counter<=end THEN 30
60 ...

10 start=4: end=20: step=2
20 FOR counter=start TO end STEP step
30   PRINT counter," ",counter^2
40 NEXT
50 ...
```

You can GOTO anywhere within one FOR...NEXT loop, but not outside it. This means you can't exit the loop with a GOTO. You can force a premature end to the loop by setting the control variable to a value equal to or greater than the end value (assuming a positive STEP).

```
110 FOR I=1 TO 20
120   X=A^I
130   IF X>1000 THEN I=20: GOTO 150
140   PRINT I,X
150 NEXT
```

It is not necessary to declare the loop variable as an integer type in order to take advantage of fast integer arithmetic. If it is an integer, then fast integer arithmetic is used automatically. See Annex E for an explanation of how BBC BASIC (Z80) recognises an integer value of a real variable.

Any numeric assignable item may be used as the control variable. In particular, a byte variable (?X) may act as the control variable and only one byte of memory will be used. See the Indirection sub-section for details of the indirection operators.

```
FOR ?X=0 TO 16: PRINT ~?X: NEXT
FOR !X=0 TO 16 STEP 4: PRINT ~!X: NEXT
```

Because a single stack is used, you cannot use a FOR...NEXT loop to set array elements to LOCAL in a procedure or function.

Syntax

```
FOR <n-var>=<numeric> TO <numeric> [STEP <numeric>]
```

Associated Keywords

TO, STEP, NEXT

GCOL

GC.

A statement which sets the graphics foreground or background logical colour to be used in all subsequent graphics operations.

Not implemented in the generic CP/M version of BBC BASIC (Z80)

Syntax

```
GCOL <numeric>,<numeric>
```

Associated Keywords

CLS, CLG, MODE, COLOUR, PLOT

GET/GET$

A function and compatible string function that reads the next character from the keyboard buffer (it waits for the character).

```
N=GET
N$=GET$
```

GET and GET$ wait for a 'key' (character) to be present in the keyboard buffer and then return the ASCII number of the key (see Annex A) or a string containing the character of the key. If there are any characters in the keyboard buffer when a GET is issued, then a character will be returned immediately. See the keyword INKEY for a way of emptying the keyboard buffer before issuing a GET.

GET and GET$ do not echo the pressed key to the screen. If you want to display the character for the pressed key, you must PRINT it.

You can use GET and GET$ whenever you want your program to wait for a reply before continuing. For example, you may wish to display several screens of instructions and allow the user to decide when he has read each screen.

```
REM First screen of instructions
```

```
CLS
PRINT .......
PRINT .......
PRINT "Press any key to continue ";
temp=GET
REM Second screen of instructions
CLS
PRINT ....... etc
```

Reading from an I/O port

GET can also be used to input data from an I/O port:

```
N=GET(X) :REM input from port X
```

Reading from a file

(BBC BASIC version 5 or later only)

GET$ can also be used to read a string from a file. GET$ differs from INPUT# and READ# in that it reads a string terminated by carriage-return (&0D), line-feed (&0A) or NUL (&00) - you cannot tell what the terminator actually was. GET$# is therefore more useful if the format of the file is uncertain.

```
A$=GET$#file_channel
```

GET$ may optionally be used with the qualifier **BY** (to specify how many bytes to read from the file, maximum 255) or **TO** (to specify the terminator character). These override the normal CR, LF or NUL terminators:

```
dest$=GET$#file_channel BY bytecount%
dest$=GET$#file_channel TO termchar%
```

The terminator character is specified as its ASCII code, for example to specify a **comma** terminator use **TO 44**, **TO &2C** or **TO ASC(",")**. If you add &100 to the value (e.g. **TO &12C**) the specified terminator will be recognised *in addition to* (rather than *instead of*) the normal CR and LF terminators.

If you add &8000 to the value (e.g. **TO &802C**) CR (carriage return) will be recognised as a terminator in addition to the specified character (in this example a comma). However LF will *not* be recognised as a terminator.

Syntax

```
<n-var> = GET
<s-var> = GET$
<n-var> = GET(<numeric>,<numeric>)
<s-var> = GET$(<numeric>,<numeric>)
<s-var> = GET$#<numeric>
<s-var> = GET$#<numeric> BY <numeric>
<s-var> = GET$#<numeric> TO <numeric>
```

Associated Keywords
INKEY, INKEY$, INPUT#, READ#, OPENIN, OPENOUT, OPENUP

GOSUB

A statement which calls a section of a program (which is a subroutine) at a specified line number. One subroutine may call another subroutine (or itself).

```
GOSUB 400
GOSUB (4*answer+6)
```

The only limit placed on the depth of nesting is the room available for the stack. You may calculate the line number. However, if you do, the program should not be RENUMBERed. A calculated value must be placed in brackets.

Very often you need to use the same group of program instructions at several different places within your program. It is tedious and wasteful to repeat this group of instructions every time you wish to use them. You can separate this group of instructions into a small sub-program. This sub-program is called a subroutine. The subroutine can be 'called' by the main program every time it is needed by using the GOSUB statement. At the end of the subroutine, the RETURN statement causes the program to return to the statement after the GOSUB statement.

Subroutines are similar to PROCedures, but they are called by line number not by name. This can make the program difficult to read because you have no idea what the subroutine does until you have followed it through. You will probably find that

PROCedures offer you all the facilities of subroutines and, by choosing their names carefully, you can make your programs much more readable.

Syntax

```
GOSUB <l-num>
GOSUB (<numeric>)
```

Associated Keywords
RETURN, ON, PROC

GOTO G.

A statement which transfers program control to a line with a specified or calculated line number.

```
GOTO 100
GOTO (X*10)
```

You may not GOTO a line which is outside the current FOR...NEXT, REPEAT...UNTIL or GOSUB loop.

If a calculated value is used, the program should not be RENUMBERed. A calculated value must be placed in brackets.

The GOTO statement makes BBC BASIC (Z80) jump to a specified line number rather than continuing with the next statement in the program.

You should use GOTO with care. Uninhibited use will make your programs almost impossible to understand (and hence, debug). If you use REPEAT....UNTIL and FOR....NEXT loops you will not need to use many GOTO statements.

Syntax

```
GOTO <l-num>
GOTO (<numeric>)
```

Associated Keywords
GOSUB, ON

HIMEM

A pseudo-variable which contains the address of the first byte that BBC BASIC (Z80) will not use.

HIMEM must not be changed within a subroutine, procedure, function, FOR...NEXT, REPEAT...UNTIL or GOSUB loop.

```
HIMEM=HIMEM-40
```

BBC BASIC (Z80) uses the computer's memory to store your program and the variables that your program uses. When BBC BASIC is first loaded and run it checks to find the highest memory address it can use. If this is in excess of &10000 bytes, HIMEM is set to &10000. Otherwise, HIMEM is set to the maximum available address.

If you want to use a machine code subroutine or store some data for use by a CHAINed program, you can move HIMEM down. This protects the area above HIMEM from being overwritten by BBC BASIC (Z80). See the Assembler section and the keyword CHAIN for details.

If you want to change HIMEM, you should do so early in your program. Once it has been changed it will stay at its new value until set to another value. Thus, if you wish to load a machine code subroutine for use by several programs, you only have to change HIMEM and load the subroutine once.

USE WITH CARE.

Syntax

```
HIMEM=<numeric>
<n-var>=HIMEM
```

Associated Keywords

LOMEM, PAGE, TOP

IF

A statement which tests a condition and controls the subsequent flow of the program depending on the result. It is part of the IF....THEN....ELSE structure. IF statements may be **single line** or **multi-line**.

Single-line IF statement

```
IF length=5 THEN 110
IF A<C OR A>D GOTO 110
IF A>C AND C>=D THEN GOTO 110 ELSE PRINT "BBC"
IF A>Q PRINT"IT IS GREATER":A=1:GOTO 120
```

The IF statement is the primary decision making statement. The testable condition (A<C, etc) is evaluated, and converted to an integer if necessary. If the value is **non-zero** the rest of the line (up to the ELSE clause if there is one) is executed. If the value is **zero**, the rest of the line is ignored, unless there is an ELSE clause in which case execution continues after the ELSE.

```
IF age<21 THEN PRINT "Under 21"

flag=age<21
IF flag THEN PRINT "Under 21"
```

The above examples will print "Under 21" if the value of 'age' is less than 21, and do nothing otherwise.

```
IF age<21 THEN PRINT "Under 21" ELSE PRINT "21 or over"

flag=age<21
IF flag THEN PRINT "Under 21" ELSE PRINT "21 or over"
```

The above examples will print "Under 21" if the value of 'age' is less than 21, and "21 or over" otherwise.

The keyword THEN is optional in most examples of the single-line IF statement. The exceptions are when THEN is followed immediately by a destination line number, by a pseudo-variable (e.g. TIME) or by an end-of-function (=). However, THEN is mandatory for the multi-line IF statement.

Multi-line IF statement

(BBC BASIC version 5 or later only)

```
IF length=5 THEN
  PRINT "length is 5"
ELSE
  PRINT "length is not 5"
ENDIF
```

A multi-line IF clause is begun by an IF statement which has the keyword THEN as the very last thing on the line (it cannot even be followed by a REMark). It is ended by an ENDIF statement, which must be the first thing on the line. If the integer numeric expression after IF is **non-zero**, the statements up to the ENDIF (or ELSE clause if present) are executed. If the expression evaluates to **zero** the statements after ELSE are executed, or if there is no ELSE clause execution continues after the ENDIF statement.

There may be any number of lines between the IF ... THEN statement and the ENDIF statement. Multi-line IF ... ENDIF clauses may be nested.

You can cascade multiple IF ... ENDIF clauses to test a number of different alternatives:

```
IF x=3 THEN
  PRINT "x is 3"
ELSE IF x=4 THEN
    PRINT "x is 4"
  ELSE IF x=5 THEN
      PRINT "x is 5"
    ELSE
      PRINT "x is not 3 or 4 or 5"
    ENDIF
  ENDIF
ENDIF
```

but in this situation a CASE statement would be a better choice.

Syntax

```
IF <t-cond> [THEN <stmt>{:<stmt>}] [ELSE <stmt>{:<stmt>}]
IF <t-cond> GOTO <l-num> [ELSE <l-num>]
IF <t-cond> THEN <l-num> [ELSE <l-num>]
IF <t-cond> THEN
```

Associated Keywords

THEN, ELSE, ENDIF

INKEY/INKEY$

A function and compatible string function which does a GET/GET$, waiting for a maximum of 'num' clock ticks of 10ms each. If no key is pressed in the time limit, INKEY will return -1 and INKEY$ will return a null string. The INKEY function will return the ASCII value of the key pressed.

```
key=INKEY(num)
N=INKEY(0)
N$=INKEY$(100)
```

Since INKEY and INKEY$ remove characters from the keyboard buffer, one character will be returned every time an INKEY is issued. A single INKEY will return the first character and leave the rest in the keyboard buffer.

You can use this function to wait for a specified time for a key to be pressed. A key can be pressed at any time before INKEY is used.

Pressed keys are stored in an input buffer. Since INKEY and INKEY$ get a character from the normal input stream, you may need to empty the input buffer before you use them. You can do this with the following program line.

```
REPEAT UNTIL INKEY(0)=-1
```

The number in brackets is the number of 'ticks' (one hundredths of a second) which BBC BASIC (Z80) will wait for a key to be pressed. After this time, BBC BASIC (Z80) will give up and return -1 or a null string. The number of 'ticks' may have any value between 0 and 32767.

Syntax

```
<n-var>=INKEY(<numeric>)
<s-var>=INKEY$(<numeric>)
```

Associated Keywords
GET, GET$

INPUT

A statement to input values from the console input channel (usually keyboard).

```
INPUT A,B,C,D$,"WHO ARE YOU",W$,"NAME"R$
```

If items are not immediately preceded by a printable prompt string (even if null) then a '?' will be printed as a prompt. If the variable is not separated from the prompt string by a comma, the '?' is not printed. In other words: no comma - no question mark.

Items A, B, C, D$ in the above example can have their answers returned on one to four lines, separate items being separated by commas. Extra items will be ignored.

Then WHO ARE YOU? is printed (the question mark comes from the comma) and W$ is input, then NAME is printed and R$ is input (no comma - no '? ').

When the <Enter> key is pressed to complete an entry, a new-line is generated. BBC BASIC has no facility for suppressing this new-line, but the TAB function can be used to reposition the cursor. For example,

```
INPUT TAB(0,5) "Name ? " N$,TAB(20,5) "Age ? " A
```

will position the cursor at column 0 of line 5 and print the prompt Name ?. After the name has been entered the cursor will be positioned at column 20 on the same line and Age ? will be printed. When the age has been entered the cursor will move to the next line.

The statement

```
INPUT A
```

is exactly equivalent to

```
INPUT A$: A=VAL(A$)
```

Leading spaces will be removed from the input line, but not trailing spaces. If the input string is not completely numeric, it will make the best it can of what it is given. If the first character is not numeric, 0 will be returned. Neither of these two cases will produce an error indication. Consequently, your program will not abort back to the command mode if a bad number is input. You may use the EVAL function to convert a string input to a numeric and report an error if the string is not a proper number or you can include your own validation checks.

```
INPUT A$
A=EVAL(A$)
```

Strings in quoted form are taken as they are, with a possible error occurring for a missing closing quote.

A semicolon following a prompt string is an acceptable alternative to a comma.

Syntax

```
INPUT [TAB(X[,Y])][SPC(<numeric>)]['][<s-const>[,|;]]
          <n-var>|<s-var>{,<n-var>|<s-var>}
```

Associated Keywords
INPUT LINE, INPUT#, GET, INKEY

INPUT LINE

A statement of identical syntax to INPUT which uses a new line for each item to be input. The item input is taken as is, including commas, quotes and leading spaces.

```
INPUT LINE A$
```

Syntax

```
INPUT LINE[TAB(X[,Y])][SPC(<numeric>)]['][<s-const>[,|;]]
            <s-var>{,<s-var>}
```

Associated Keywords

INPUT

INPUT#

A statement which reads data in internal format from a file and puts them in the specified variables. INPUT# is normally used with a file or device opened with OPENIN, OPENUP or OPENOUT.

```
INPUT #E,A,B,C,D$,E$,F$
INPUT #3,aux$
```

It is possible to read past the end-of-file without an error being reported. You should always include some form of check for the end of the file.

READ# can be used as an alternative to INPUT#.

See the Files section for more details and numerous examples of the use of INPUT#.

Syntax

```
INPUT #<numeric>,<n-var>|<s-var>{,<n-var>|<s-var>}
```

Associated Keywords

INPUT, OPENIN, OPENUP, OPENOUT, CLOSE#, PRINT#, BPUT#, BGET#, GET$#, EXT#, PTR#, EOF#

INSTR

A function which returns the position of a sub-string within a string, optionally starting the search at a specified place in the string. The leftmost character position is 1. If the sub-string is not found, 0 is returned.

The first string is searched for any occurrence of the second string.

There must not be any spaces between INSTR and the opening bracket.

```
X=INSTR(A$,B$)
position=INSTR(word$,guess$)
Y=INSTR(A$,B$,Z%) :REM START AT POSITION Z%
```

You can use this function for validation purposes. If you wished to test A$ to see if was one of the set 'FRED BERT JIM JOHN', you could use the following:

```
set$="FRED BERT JIM JOHN"
IF INSTR(set$,A$) PROC_valid ELSE PROC_invalid
```

The character used to separate the items in the set must be excluded from the characters possible in A$. One way to do this is to make the separator an unusual character, say CHR$(127).

```
z$=CHR$(127)
set$="FRED"+z$+"BERT"+z$+"JIM"+z$+"JOHN"
```

Syntax

```
<n-var>=INSTR(<str>,<str>[,<numeric>])
```

Associated Keywords
LEFT$, MID$, RIGHT$, LEN

INT

A function truncating a real number to the lower integer.

```
X=INT(Y)
```

```
INT(99.8)=99
INT(-12)  =-12
INT(-12.1)            =-13
```

This function converts a real number (one with a decimal part) to the nearest integer (whole number) less than the number supplied. Thus,

```
INT(14.56)
```

gives 14, whereas

```
INT(-14.5)
```

gives -15.

Syntax

```
<n-var>=INT<numeric>
```

Associated Keywords
None

LEFT$

A string function which returns the left 'num' characters of the string. If there are insufficient characters in the source string, all the characters are returned.

(BBC BASIC version 5 or later only)

LEFT$ may also be used to the left of an equals sign to change the leftmost part of a string whilst leaving the rest alone.

There must not be any spaces between LEFT$ and the opening bracket.

```
newstring$=LEFT$(A$,num)
A$=LEFT$(A$,2)
A$=LEFT$(RIGHT$(A$,3),2)
LEFT$(A$,3) = B$
For example,
name$="BBC BASIC"
```

```
FOR i=3 TO 9
  PRINT LEFT$(name$,i)
NEXT
END
```

would print

```
BBC
BBC
BBC B
BBC BA
BBC BAS
BBC BASI
BBC BASIC
```

LEFT$(A$,0) returns an empty string; LEFT$(A$,-1) returns the original string unmodified.

When using LEFT$ on the left of an equals sign, and the expression to the right of the equals sign evaluates to a string with *fewer* characters than the specified sub-string length, only that number of characters is changed. For example:

```
A$ = "BBC BASIC"
LEFT$(A$,3) = "ZZ"
```

will set A$ equal to "ZZC BASIC". Although the sub-string length is set to three, only two characters are actually modified since that is the length of the string "ZZ".

(BBC BASIC version 5 or later only)

LEFT$(A$) is shorthand for LEFT$(A$,LENA$-1), in other words it returns or sets all but the last character of A$.

Syntax

```
<s-var>=LEFT$(<string>[,<numeric>])
LEFT$(<s-var>[,<numeric>])=<string>
```

Associated Keywords
RIGHT$, MID$, LEN, INSTR

LEN

A function which returns the length of the argument string.

```
X=LEN"fred"
X=LENA$
X=LEN(A$+B$)
```

This function 'counts' the number of characters in a string. For example,

```
length=LEN("BBC BASIC (Z80)   ")
```

would set 'length' to 15 since the string consists of the 12 characters of BBC BASIC (Z80) followed by three spaces.

LEN is often used with a FOR....NEXT loop to 'work down' a string doing something with each letter in the string. For example, the following program looks at each character in a string and checks that it is a valid hexadecimal numeric character.

```
 10 valid$="0123456789ABCDEF"
 20 REPEAT
 30   INPUT "Type in a HEX number" hex$
 40   flag=TRUE
 50   FOR i=1 TO LEN(hex$)
 60     IF INSTR(valid$,MID$(hex$,i,1))=0 flag=FALSE
 80   NEXT
 90   IF NOT flag THEN PRINT "Bad HEX"
100 UNTIL flag
```

Syntax

```
<n-var>=LEN(<str>)
```

Associated Keywords

LEFT$, MID$, RIGHT$, INSTR

LET

LET is an optional assignment statement.

LET is not permitted in the assignment of the pseudo-
variables LOMEM, HIMEM, PAGE, PTR# and TIME.

LET was mandatory in early versions of BASIC. Its use emphasised that when we write

```
X=X+4
```

we don't mean to state that X equals X+4 - it can't be, but rather 'let X become equal to what it was plus 4':

```
LET X=X+4
```

Most modern versions of BASIC allow you to drop the 'LET' statement. However, if you are writing a program for a novice, the use of LET makes it more understandable.

Syntax

```
[LET] <var>=<exp>
```

Associated Keywords
None

LINE

(BBC BASIC version 5 or later only, not implemented in the generic CP/M edition)

A statement which draws a straight line between two points in the current graphics foreground colour. LINE is followed by the X and Y coordinates of the ends of the line.

```
LINE 100,200,300,400
```

This will draw a straight line from the point 100,200 to the point 300,400.
LINE x1,y1,x2,y2 is equivalent to MOVE x1,y1 : DRAW x2,y2

The keyword LINE is also used in the INPUT LINE statement. LINE INPUT is synonymous with INPUT LINE.

Syntax

```
LINE <numeric>,<numeric>,<numeric>,<numeric>
LINE INPUT [TAB(X[,Y])][SPC(<numeric>)]['][<s-const>[,|;]]
           <s-var>{,<s-var>}
```

Associated Keywords

DRAW, INPUT, MOVE, PLOT

LIST

L.

A command which causes lines of the current program to be listed out to the currently selected output stream (see *OPT) with the automatic formatting options specified by LISTO.

LIST	lists the entire program
LIST ,111	lists up to line 111
LIST 111,	lists from line 111 to the end
LIST 111,222	lists lines 111 to 222 inclusive
LIST 100	lists line 100 only

A hyphen is an acceptable alternative to a comma.

Escape will abort the listing.

LIST may be included within a program, but it will exit to the command mode on completion of the listing.

Syntax

```
LIST
LIST <n-const>
LIST <n-const>,
LIST ,<n-const>
LIST <n-const>,<n-const>
```

Associated Keywords
LIST IF, LISTO, OLD, NEW

LIST IF

A command which causes lines of the current program which contain the specified string to be listed to the currently selected output stream (see *OPT).

```
LIST IF *FX
LIST IF Please press <ENTER> to continue
```

You can specify the range of line numbers to be listed in a similar manner to LIST. For example,

```
LIST 100,2500 IF DEF
```

Will list all the lines between 100 and 2500 which contain the keyword 'DEF' Keywords are tokenised before the search begins. Consequently, you can use LIST IF to find lines with particular commands in them.

```
LIST IF PROC
LIST IF DEF
```

LIST IF is very useful for locating the lines in a program which define or use functions or procedures.

Limitations

Because keywords are tokenised wherever they occur in the command line, you cannot use LIST IF to search for a string (including a star command) which contains a keyword. For example, the following will not work:

```
LIST IF *LOAD
LIST IF DO YOU WANT TO PRINT THE RESULTS?
```

You cannot search for the 'left' form of those pseudo-variables which have two forms (PTR#=, PAGE=, TIME=, LOMEM=, HIMEM=) because the 'right' form is assumed when the name is tokenised. Consequently,

```
LIST IF TIME
```

will find line 20 but not line 10 in the following program segment

```
10 TIME=20
20 now=TIME
```

You cannot search for 'keywords' which are not tokenised in the context of the program. For example,

```
LIST IF LOAD
```

will not list lines containing

```
ZLOAD=1
PROCLOAD
FNLOAD
"LOAD"
REM LOAD

etc
```

because LOAD is not tokenised in any of these lines.

The internal format of line numbers (GOTO 1000, for example) may spuriously match a search string of three characters or less.

Syntax

```
LIST IF <string>
LIST <n-const> IF <string>
LIST <n-const>, IF <string>
LIST ,<n-const> IF <string>
LIST <n-const>,<n-const> IF <string>
```

Associated Keywords
LIST, OLD, NEW

LISTO

A command which controls the appearance of a LISTed program.

Bit Settings

Bit 0 (LSB)

If Bit 0 is set, a space will be printed between the line number and the remainder of the line (all leading spaces are stripped when the line is originally entered.)

Bit 1

If Bit 1 is set, two extra spaces will be printed out on lines between FOR and NEXT, between REPEAT and UNTIL and between WHILE and ENDWHILE. Two extra spaces will be printed for each depth of nesting.

The default setting of LISTO is 7; this will give a properly formatted listing. The indentation is done in the conventional manner for BASIC, in that the NEXT is aligned with the FOR, the REPEAT with the UNTIL and the ENDWHILE with the WHILE.

```
LISTO 7
```

will give

```
 10 A=20
 20 TEST$="FRED"
 30 FOR I=1 TO A
 40   Z=2^I
 50   PRINT I,Z
 60   REPEAT
 70     PRINT TEST$
 80     TEST$=LEFT$(TEST$,LEN(TEST$)-1)
 90   UNTIL LEN(TEST$)=0
100 NEXT
110 END
```

at the other extreme

```
LISTO 0
```

will give

```
 10A=20
 20TEST$="FRED"
 30FOR I=1 TO A
 40Z=2^I
 50PRINT I,Z
 60REPEAT
 70PRINT TEST$
 80TEST$=LEFT$(TEST$,LEN(TEST$)-1)
 90UNTIL LEN(TEST$)=0
100NEXT
110END
```

Syntax

```
LISTO <n-const>
```

Associated Keywords
LIST

LN

A function giving the natural logarithm of its argument.

```
X=LN(Y)
```

This function gives the logarithm to the base 'e' of its argument. The 'natural' number, 'e', is approximately 2.71828183.

Logarithms are 'traditionally' used for multiplication (by adding the logarithms) and division (by subtracting the logarithms). For example,

```
10 log1=LN(2.5)
20 log2=LN(2)
30 log3=log1+log2
40 answer=EXP(log3)
50 PRINT answer
```

will calculate 2.5*2 by adding their natural logarithms and print the answer.

Syntax

```
<n-var>=LN<numeric>
```

Associated Keywords
LOG, EXP

LOAD LO.

A command which loads a new program from a file and CLEARs the variables of the old program. The program file must be in 'internal' (tokenised) format.

```
LOAD "PROG1"
LOAD A$
```

File names must conform to the standard CP/M-80 format. However, if no extension is given, .BBC is assumed. If no drive and/or path are given, the current drive and/or path are assumed. See the Operating System Interface section for a more detailed description of valid file names.

You use LOAD to bring a program in a file into memory. The keyword LOAD should be followed by the name of the program file. If the program file is in the current directory, only the file name needs to be given. If the program is not in the current directory, its full drive, path and file name must be specified. For example:

```
LOAD "a:\bbcprogs\demo"
```

would load the program 'demo.bbc' from the directory 'bbcprogs' on drive a:.

Syntax

```
LOAD <str>
```

Associated Keywords
SAVE, CHAIN

LOCAL

A statement to declare variables for local use inside a function (FN) or procedure (PROC).

LOCAL saves the value(s) of the specified variable(s) on the stack, and initialises the variables to zero (in the case of numeric variables) or null (in the case of string variables). The original values are restored from the stack on exit from the function or procedure (i.e. at '=' or ENDPROC). The variables need not have been previously declared.

(BBC BASIC version 5 or later only)

An entire array may be made LOCAL, following which it is in an *undimensioned* state. Before the local array can be used within the function or procedure it must be dimensioned using a DIM statement. The new dimensions can be the same as or different from those of a global array with the same name. LOCAL arrays are allocated on the *stack*, and are freed when the function/procedure is exited.

```
LOCAL A$,X,Y%,items()
```

If a function or a procedure is used recursively, the LOCAL variables will be preserved at each level of recursion.

LOCAL DATA

(BBC BASIC version 5 or later only)

LOCAL DATA saves the current DATA pointer on the stack (but leaves its value unchanged). When used inside FOR...NEXT, REPEAT...UNTIL or WHILE...ENDWHILE loop, or inside a user-defined function, procedure or subroutine, the data pointer is automatically restored to its original value on exit. Otherwise it can be restored using the RESTORE DATA statement.

```
LOCAL DATA
RESTORE +1
DATA "Here", "is", "some", "local", "data"
READ A$, B$, C$, D$, E$
RESTORE DATA
```

LOCAL is also used in the ON ERROR LOCAL and DIM LOCAL statements.

Syntax

```
LOCAL <n-var>|<s-var>|<array()>{,<n-var>|<s-var>|<array()>}
LOCAL DATA
```

Associated Keywords

DEF, DIM, ENDPROC, FN, ON, PROC

LOG

A function giving the base-10 logarithm of its argument.

```
X = LOG(Y)
```

This function calculates the common (base 10) logarithm of its argument. Inverse logarithms (anti-logs) can be calculated by raising 10 to the power of the logarithm. For example, if `x=LOG(y)` then `y=10^x`.

Logarithms are 'traditionally' used for multiplication (by adding the logarithms) and division (by subtracting the logarithms). For example,

```
10 log1=LOG(2.5)
20 log2=LOG(2)
30 log3=log1+log2
40 answer=10^log3
50 PRINT answer
```

Syntax

```
<n-var>=LOG<numeric>
```

Associated Keywords
LN, EXP

LOMEM

A pseudo-variable which controls where in memory the dynamic data structures are to be placed. The default is TOP, the first free address after the end of the program.

```
LOMEM=LOMEM+100
PRINT ~LOMEM :REM The ~ makes it print in HEX.
```

Normally, dynamic variables are stored in memory immediately after your program. (See the Annex entitled Format of Program and Variables in Memory.) You can change the address where BBC BASIC (Z80) starts to store these variables by changing LOMEM.

USE WITH CARE. Changing LOMEM in the middle of a program causes BBC BASIC (Z80) to lose track of all the variables you are using.

Syntax

```
LOMEM=<numeric>
<n-var>=LOMEM
```

Associated Keywords
HIMEM, TOP, PAGE

MID$

A function which returns a part of a string starting at a specified position and with a given length. If there are insufficient characters in the string, or the length is not specified, then all the characters from the specified start position onwards are returned.

(BBC BASIC version 5 or later only)

MID$ may also be used to the left of an equals sign to change part of a string whilst leaving the rest alone.

```
C$ = MID$(A$,start_posn,num)
C$ = MID$(A$,Z)
MID$(A$,4,2) = B$
```

You can use this function to select any part of a string. For instance, if

```
name$="BBC BASIC for Z80"
```

then

```
part$=MID$(name$,5,3)
```

would set part$ to "BAS". If the third parameter is omitted or there are insufficient characters to the right of the specified position, MID$ returns the right hand part of the string starting at the specified position. Thus,

```
part$=MID$(name$,15)
```

would set part$ to "Z80".

For example,

```
name$="BBC BASIC for Z80"
FOR i=5 TO 15
  PRINT MID$(name$,i,11)
NEXT
```

would print

```
BASIC for Z
ASIC for Z8
SIC for Z80
IC for Z80
C for Z80
 for Z80
for Z80
or Z80
r Z80
 Z80
Z80
```

(BBC BASIC version 5 or later only)

When using MID$ on the left of an equals sign, and the expression to the right of the equals sign evaluates to a string with *fewer* characters than the specified sub-string length, only that number of characters is changed. For example:

```
A$ = "BBC BASIC"
MID$(A$,5,4) = "ZZ"
```

will set A$ equal to "BBC ZZSIC". Although the sub-string length is set to four, only two characters are actually modified since that is the length of the string "ZZ".

Syntax

```
<s-var>=MID$(<string>,<numeric>[,<numeric>])
MID$(<s-var>,<numeric>[,<numeric>])=<string>
```

Associated Keywords
LEFT$, RIGHT$, LEN, INSTR

MOD

An operator returning the signed remainder after an integer division. The result is always an integer.

```
X=A MOD B
```

MOD is defined such that,

```
A MOD B = A - ( A DIV B ) * B.
```

If you are doing integer division (DIV) of whole numbers it is often desirable to know the remainder (a 'teach children to divide' program for instance). For example, 23 divided by 3 is 7, remainder 2. Thus,

```
PRINT 23 DIV 3
PRINT 23 MOD 3
```

would print

```
     7
     2
```

You can use real numbers in these calculations, but they are truncated to their integer part before BBC BASIC calculates the result. Thus,

```
PRINT 23.1 DIV 3.9
PRINT 23.1 MOD 3.9
```

would give exactly the same results as the previous example.

(BBC BASIC version 5 or later only)

MOD can also be used as a function which operates on an entire numeric array. It returns the *modulus* (the square-root of the sum of the squares of all the elements) in the array. See the array arithmetic section.

Syntax

```
<n-var>=<numeric> MOD <numeric>
<n-var>MOD=<numeric>
<n-var>=MOD(<array()>)
```

Associated Keywords
DIV

MODE MO.

Not implemented in the generic CP/M version of BBC BASIC (Z80)

The MODE command sets the screen display mode. The screen is cleared and all the graphics and text parameters (colours, origin, etc) are reset to their default values.

(BBC BASIC version 5 or later only)

MODE can also be used as a function. It returns the current mode number (or −1 if no MODE statement has been executed, or a 'custom' mode is in use).

Syntax

```
MODE <numeric>
<n-var> = MODE
```

Associated Keywords
CLS, CLG

MOUSE

(BBC BASIC version 5 or later only, not implemented in the generic CP/M edition

A multi-purpose statement which controls, or returns information about, the mouse. The different variants of the statement are as follows (not all may be implemented):

MOUSE x,y,b

This returns the current position of the mouse pointer and the status of the mouse buttons; 'x', 'y' and 'b' are the names of numeric variables (which needn't have previously been defined). The variables 'x' and 'y' are set to the position of the mouse pointer in BBC BASIC graphics units and are affected by the current graphics origin.

The variable 'b' is set to a value depending on which mouse buttons (if any) are pressed: the value 1 indicates that the right button is pressed, 2 that the middle button (if any) is pressed and 4 that the left button is pressed. The values may be combined.

MOUSE ON n

This causes the mouse pointer to be displayed and determines the shape of the pointer depending on the value of 'n'. MOUSE ON is equivalent to MOUSE ON 0.

MOUSE OFF

This causes the mouse pointer to be hidden.

MOUSE TO x,y

This moves the mouse pointer to the specified coordinates, in BBC BASIC graphics units.

MOUSE RECTANGLE l,b,w,h

This confines the mouse pointer to remain within the specified bounding rectangle.

MOUSE RECTANGLE OFF

This clears the bounding rectangle so that the mouse pointer is no longer confined.

Syntax

```
MOUSE <n-var>,<n-var>,<n-var>
MOUSE ON [<numeric>]
MOUSE OFF
MOUSE TO <n-var>,<n-var>
MOUSE RECTANGLE <n-var>,<n-var>,<n-var>,<n-var>
MOUSE RECTANGLE OFF
```

Associated Keywords

ON, OFF

MOVE

A statement which moves the graphics cursor to an absolute or relative position without drawing a line.

Not implemented in the generic CP/M version of BBC BASIC (Z80)

Syntax

```
MOVE [BY] <numeric>,<numeric>
```

Associated Keywords

DRAW, MODE, GCOL, PLOT

NEW

A command which initialises the interpreter for a new program to be typed in. The old program may be recovered with the OLD command provided no new program lines have been typed in or deleted and no variables have been created.

```
NEW
```

This command effectively 'removes' a program from the computer's memory. In reality, the program is still there, but BBC BASIC (Z80) has been told to forget about it.

If you have made a mistake, you can recover your old program by typing OLD. However, this won't work if you have begun to enter a new program.

Syntax

```
NEW
```

Associated Keywords
OLD

NEXT N.

The statement delimiting FOR...NEXT loops. NEXT takes an optional control variable.

```
NEXT
NEXT J
```

If the control variable is present then FOR....NEXT loops may be 'popped' automatically in an attempt to match the correct FOR statement (this should not be necessary). If a matching FOR statement cannot be found, a 'Can't match FOR' error will be reported.

Leaving out the control variable will make the program run quicker, but this is not to be encouraged.

See the keyword FOR for more details about the structure of FOR....NEXT loops.

Syntax

```
NEXT [<n-var>{,<n-var>}]
```

Associated Keywords

FOR, TO, STEP

NOT

This is a high priority unary operator (the same priority as unary -). It causes a bit-by-bit binary inversion of the numeric to its right. The numeric may be a constant, a variable, or a mathematical or boolean expression. Expressions must be enclosed in brackets.

```
A=NOT 3
flag=NOT flag
flag=NOT(A=B)
```

NOT is most commonly used in an IF....THEN....ELSE statement to reverse the effect of the test.

```
IF NOT(rate>5 AND TIME<100) THEN .....
IF NOT flag THEN .....
```

BBC BASIC (Z80) does not have true boolean variables; it makes do with numeric variables. This can lead to confusion because the testable condition in an IF....THEN....ELSE statement is evaluated mathematically and can result in something other than -1 (TRUE) or 0 (FALSE).

When the test in an IF....THEN....ELSE is evaluated, FALSE=0 and anything else is considered to be TRUE. If you wish to use NOT to reverse the action of an IF statement it is important to ensure that the testable condition does actually evaluate to -1 for TRUE.

If the testable condition evaluates to 1, for example, the result of the test would be considered to be TRUE and the THEN part of the IF....THEN....ELSE statement would be carried out. However, using NOT in front of the testable condition would not reverse the action. NOT 1 evaluates to -2, which would also be considered to be TRUE.

Syntax

```
<n-var>=NOT<numeric>
```

Associated Keywords
None

OF

(BBC BASIC version 5 or later only)

A keyword which is part of the CASE... ENDCASE clause. OF follows the name of the CASE variable, and must be the last item on the program line (not even followed by a REMark).

```
CASE die% OF
  WHEN 1,2,3 : bet$ = "lose"
  WHEN 4,5,6 : bet$ = "win"
  OTHERWISE bet$ = "cheat"
ENDCASE
```

Syntax

```
CASE <var> OF
```

Associated Keywords
CASE, ENDCASE, OTHERWISE, WHEN

OFF

(BBC BASIC version 5 or later only)

A statement which hides the text cursor (which is normally a flashing underscore). The cursor can be switched on again with ON. OFF is equivalent to **VDU 23,1,0;0;0;0;**.

OFF is also used in the ON ERROR OFF statement.

Syntax

```
OFF
```

Associated Keywords
ON

OLD

A command which undoes the effect of NEW provided no lines have been typed in or deleted, and no variables have been created.

```
OLD
```

Syntax

```
OLD
```

Associated Keywords
NEW

ON

A statement controlling a multi-way switch. The line numbers in the list may be constants or calculated and the 'unwanted' ones are skipped without calculation. The ON statement is used in conjunction with four other key-words: GOTO, GOSUB, PROC and ERROR. (ON ERROR is explained separately.)

```
ON option GOTO 1000,2000,3000,4000
ON action GOSUB 100,3000,200,5000,30
ON choice PROC_add,PROC_find,PROC_delete
```

The ON statement alters the path through your program by transferring control to one of a selection of line numbers depending on the value of a variable. For example,

```
200 ON number GOTO 1000,2000,500,100
```

would send your program to line 1000 if 'number' was 1, to line 2000 if 'number' was 2, to line 500 if 'number' was 3 and to line 100 if 'number' was 4.

Exceptions may be trapped using the ELSE statement delimiter.

```
ON action GOTO 100,300,120 ELSE PRINT"Illegal"
```

If there is no statement after the ELSE, the program will 'drop through' to the following line if an exception occurs. In the two following examples, the program would drop through to the error handling part of the program if 'choice' or 'B-46' was less than one or more than 3.

```
ON choice PROC_add,PROC_find(a$),PROC_delete ELSE PRINT
"Illegal Choice - Try again"
```

```
ON B-46 GOSUB 100,200,(C/200) ELSE PRINT "ERROR"
```

You can use ON...GOTO, ON...GOSUB, and ON...PROC to execute the appropriate part of your program as the result of a menu selection. The following skeleton example offers a menu with three choices.

```
20 CLS
30 PRINT "SELECT THE ACTION YOU WISH TO TAKE"
40 PRINT "1 OPEN A NEW DATA FILE"
50 PRINT "2 ADD DATA TO THE FILE"
60 PRINT "3 CLOSE THE FILE AND END""
70 REPEAT
80   INPUT TAB(10,20)"WHAT DO YOU WANT ? "choice
90 UNTIL choice>0 AND choice<4
100 ON choice PROC_open,PROC_add,PROC_close ELSE
110 .....etc
```

Limitations

If a statement terminator (: or the token for ELSE) appears within the line, the interpreter assumes that the ON... statement is terminated. For example, you cannot pass a colon as a literal string parameter in an ON...PROC command. The program line

```
ON entry PROC_start,PROC_add(":"),PROC_end
```

would be interpreted as

```
ON entry PROC_start,PROC_add("
:"),PROC_end
```

and give rise to an interesting crop of error messages.

Syntax

```
ON <numeric> GOTO <l-num>{,<l-num>}
        [ELSE <stmt>{:<stmt>}]
ON <numeric> GOSUB <l-num>{,<l-num>}
        [ELSE <stmt>{:<stmt>}]
ON <numeric> PROC<name>[(<exp>{,<exp>})]
        {,PROC<name>[(<exp>{,<exp>})]}
        [ELSE <stmt>{:<stmt>}]
```

Enabling the text cursor

(BBC BASIC version 5 or later only)

ON can also be used on its own to enable the text cursor (caret). ON is equivalent to **VDU 23,1,1;0;0;0;**.

Associated Keywords
ON ERROR, ON ERROR LOCAL, GOTO, GOSUB, PROC, ON

ON ERROR

A statement controlling error trapping. If an ON ERROR statement has been encountered, BBC BASIC (Z80) will transfer control to it (without taking any reporting action) when an error is detected. This allows error reporting/recovery to be controlled by the program. However, the program control stack is still cleared when the error is detected and it is not possible to RETURN to the point where the error occurred.

ON ERROR OFF returns the control of error handling to BBC BASIC (Z80).

```
ON ERROR PRINT"Suicide":END
ON ERROR GOTO 100
ON ERROR OFF
```

For example, the ON ERROR statement can be used to trap out the escape key to prevent a program being terminated at the wrong time by its accidental use.

```
50 ON ERROR IF ERR=17 THEN 70
60 PRINT:REPORT:PRINT " at line ";ERL:END
70 : etc.
```

Error handling is explained more fully in the General Information section.

Syntax

```
ON ERROR <stmt>{:<stmt>}
ON ERROR OFF
```

Associated Keywords
ON, GOTO, GOSUB, PROC

ON ERROR LOCAL

(BBC BASIC version 5 or later only)

A statement controlling error trapping. If an ON ERROR LOCAL statement has been encountered, BBC BASIC will transfer control to it (without taking any reporting action) when an error is detected. This allows error reporting/recovery to be controlled by the program.

Unlike the ON ERROR statement, ON ERROR LOCAL prevents BBC BASIC clearing the program stack. By using this statement, you can trap errors within a FOR ... NEXT, REPEAT ... UNTIL or WHILE ... ENDWHILE, loop or a subroutine, function or procedure without BBC BASIC losing its place within the program structure.

ON ERROR OFF returns the control of error handling to BBC BASIC.

```
ON ERROR LOCAL PRINT"Suicide":END
ON ERROR LOCAL ENDPROC
ON ERROR OFF
```

The following example program will continue after the inevitable '*Division by zero*' error.

```
FOR n=-5 TO 5
  ok% = TRUE
  ON ERROR LOCAL PRINT "Infinity" : ok% = FALSE
  IF ok% PRINT "The reciprocal of ";n;" is ";1/n
NEXT n
```

If ON ERROR LOCAL is used within a procedure, function, or loop structure then the previous error-trapping status (which might be an earlier ON ERROR LOCAL) is automatically restored on exit from the structure. However, ON ERROR LOCAL can be used anywhere within a BBC BASIC program, not just in these structures. In this case you must explicitly restore the previous error-trapping status using RESTORE ERROR.

ON ERROR LOCAL OFF temporarily disables error trapping; the default error reporting mechanism will be used until the previous error-trapping status is restored.

Syntax

```
ON ERROR LOCAL <stmt>{:<stmt>}
ON ERROR LOCAL OFF
ON ERROR OFF
```

Associated Keywords
ON, ON ERROR, GOTO, GOSUB, PROC, RESTORE

OPENIN OP.

A function which opens a file for reading and returns the file handle of the file. This number must be used in subsequent references to the file with BGET#, GET$#, INPUT#, EXT#, PTR#, EOF# or CLOSE#.

A returned value of zero signifies that the specified file was not found on the storage device.

```
X=OPENIN "jim"
X=OPENIN A$
X=OPENIN (A$)
X=OPENIN ("FILE1")
```

The example below reads data from file into an array. If the data file does not exist, an error message is printed and the program ends.

```
10 DIM posn(10),name$(10)
20 fnum=OPENIN "TOPTEN"
30 IF fnum=0 THEN PRINT "No TOPTEN data": END
40 FOR i=1 TO 10
50   INPUT#fnum,posn(i),name$(i)
60 NEXT
70 CLOSE#fnum
```

Syntax

```
<n-var>=OPENIN(<str>)
```

Associated Keywords
OPENOUT, OPENUP, CLOSE#, PTR#, PRINT#, INPUT#, BGET#, GET$#, BPUT#, EOF #

OPENOUT

A function which opens a file for writing and returns the file handle of the file. This number must be used in subsequent references to the file with BPUT#, PRINT#, EXT#, PTR# or CLOSE#. If the specified file does not exist it is created. If the specified file already exists it is truncated to zero length.

A returned value of zero indicates that the specified file could not be created.

```
X=OPENOUT(A$)
X=OPENOUT("DATAFILE")
X=OPENOUT("LPT1")
```

You can also read from a file which has been opened using OPENOUT. This is of little use until you have written some data to it. However, once you have done so, you can move around the file using PTR# and read back previously written data.

Data is not written to the file at the time it is opened. Consequently, it is possible to successfully open a file on a full device. Under these circumstances, a 'Disk full' error would be reported when you tried to write data to the file for the first time.

The example below writes the contents of two arrays (tables) to a file called 'TOPTEN.BBC'.

```
10 A=OPENOUT "TOPTEN"
20 FOR Z=1 TO 10
30   PRINT#A,N(Z),N$(Z)
40 NEXT
50 CLOSE#A
60 END
```

Syntax

```
<n-var>=OPENOUT(<str>)
```

Associated Keywords
OPENIN, OPENUP, CLOSE#, PTR#, PRINT#, INPUT#, BGET#, BPUT#, GET$#, EOF#

OPENUP

A function which opens a data file for update (reading and writing) and returns the file handle of the file. This number must be used in subsequent references to the file with BGET#, BPUT#, GET$#, INPUT#, PRINT#, EXT#, PTR#, EOF# or CLOSE#.

A returned value of zero signifies that the specified file was not found on the device.

```
X=OPENUP "jim"
X=OPENUP A$
X=OPENUP (A$)
X=OPENUP ("FILE1")
```

See the random file examples (F-RAND?) in the BBC BASIC Files section for examples of the use of OPENUP.

Syntax

<n-var>=OPENUP(<str>)

Associated Keywords
OPENIN, OPENOUT, CLOSE#, PTR#, PRINT#, INPUT#, BGET#, BPUT#, GET$#, EOF#

OPT

An assembler pseudo operation controlling the method of assembly. (See the Assembler section for more details.) OPT is followed by an expression with the following meanings:

Code Assembled Starting at P%

Value	Action	
0	assembler errors suppressed;	no listing.
1	assembler errors suppressed;	listing.
2	assembler errors reported;	no listing.
3	assembler errors reported;	listing (default).

Code Assembled Starting at O%

Value	Action	
4	assembler errors suppressed;	no listing.
5	assembler errors suppressed;	listing.
6	assembler errors reported;	no listing.
7	assembler errors reported;	listing.

The possible assembler errors are:

Out of range - error code 40.
No such variable - error code 26.

Syntax

```
OPT <numeric>
```

Associated Keywords
None

OR

The operation of bitwise integer logical OR between two items. The two operands are internally converted to 4 byte integers before the OR operation.

```
IF A=2 OR B=3 THEN 110
X=B OR 4
```

You can leave out the space between OR and a preceding constant, but it makes your programs difficult to read.

You can use OR as a logical operator or as a 'bit-by-bit' (bitwise) operator. The operands can be boolean (logical) or numeric.

Unfortunately, BBC BASIC does not have true boolean variables; it uses numeric variables and assigns the value 0 for FALSE and -1 for TRUE. This can lead to confusion at times. (See NOT for more details.)

In the example below, the operands are boolean (logical). In other words, the result of the tests (IF) A=2 and (IF) B=3 is either TRUE or FALSE. The result of this example will be TRUE if A=2 or B=3.

```
answer=(A=2 OR B=3)
```

The brackets are not necessary, they have been included to make the example easier to follow.

The previous example uses the OR in a similar fashion to the numeric operators (+, -, etc).

Suppose X was -20 in the following example,

```
A=X OR 11
```

the OR operation would be:

```
11111111 11111111 11111111 11101100
00000000 00000000 00000000 00001011
11111111 11111111 11111111 11101111  = -17
```

Syntax

```
<n-var>=<numeric> OR <numeric>
```

Associated Keywords
AND, EOR, NOT

ORIGIN

(BBC BASIC version 5 or later only, not implemented in the generic CP/M edition)

A statement which sets the graphics origin. ORIGIN is followed by the X and Y coordinates of the new origin; subsequent graphics commands operate with respect to these coordinates. The coordinates must be in the range −32768 to +32767.

```
ORIGIN x,y is equivalent to VDU 29,x;y;
ORIGIN 640,512
```

Syntax

```
ORIGIN <numeric>,<numeric>
```

Associated Keywords
MODE, PLOT, VDU

OSCLI

This command allows a string expression to be passed to the operating system. It overcomes the problems caused by the exclusion of variables in the star (*) commands. Using this statement, you can, for instance, erase and rename files whose names you only know at run-time.

```
command$="ERA PHONE.DTA"
OSCLI command$

command$="REN ADDRESS.DTA=NAME.DTA"
OSCLI command$
```

See the Operating System Interface section for more details.

Syntax

```
OSCLI <str>
```

Associated Keywords
All operating system (*) commands.

OTHERWISE

(BBC BASIC version 5 or later only)

A keyword which is an optional part of the CASE ... ENDCASE clause. OTHERWISE precedes the statement(s) which should be executed if the CASE variable matches none of the values specified in a WHEN statement. OTHERWISE must be the first item on the program line.

```
CASE die% OF
  WHEN 1,2,3 : bet$ = "lose"
  WHEN 4,5,6 : bet$ = "win"
  OTHERWISE bet$ = "cheat"
ENDCASE
```

Syntax

```
OTHERWISE [<stmt>]{:<stmt>}
```

Associated Keywords

CASE, ENDCASE, OF, WHEN

PAGE **PA.**

A pseudo-variable controlling the starting address of the current user program area. It addresses the area where a program is (or will be) stored.

```
PAGE=&3100
PRINT ~PAGE
PAGE=TOP+&100: REM Move to start of next page.
```

PAGE is automatically initialised by BBC BASIC (Z80) to the address of the lowest available page in RAM, but you may change it.

If you make PAGE less than its original value or greater than the original value of HIMEM, you will get a 'Bad program' error when you try to enter a program line and you may well crash BBC BASIC (Z80).

If you make PAGE greater than HIMEM, a 'No room' error will occur if the program exits to command level.

With care, several programs can be left around in RAM without the need for saving them.

USE WITH CARE.

Syntax

```
PAGE=<numeric>
<n-var>=PAGE
```

Associated Keywords
TOP, LOMEM, HIMEM

PI

A function returning 3.14159265.

```
X=PI
```

You can use PI to calculate the circumference and area of a circle. The example below calculates the circumference and area of a circle of a given radius.

```
10 CLS
20 INPUT "What is the radius of the circle ",rad
30 PRINT "The circumference is: ";2*PI*rad
40 PRINT "The area is: ";PI*rad*rad
50 END
```

PI can also be used to convert degrees to radians and radians to degrees.

```
radians=PI/180*degrees
degrees=180/PI*radians
```

However, BBC BASIC (Z80) has two functions (RAD and DEG) which perform these conversions to a higher accuracy.

Syntax

```
<n-var>=PI
```

Associated Keywords

RAD, DEG

PLOT

PL.

PLOT is a multi-purpose drawing statement. Three numbers follow the PLOT statement: the first specifies the type of point, line, triangle, circle etc. to be drawn; the second and third give the X and Y coordinates to be used.

Not implemented in the generic CP/M version of BBC BASIC (Z80)

Syntax

```
PLOT [BY] <numeric>,<numeric>,<numeric>
```

Associated Keywords
MODE, CIRCLE, CLG, MOVE, DRAW, ELLIPSE, LINE, ORIGIN, RECTANGLE, POINT, VDU, GCOL

POINT

A function which returns a number giving the logical colour of the screen at the coordinates specified. If the point is outside the graphics viewport, then -1 is returned.

Not implemented in the generic CP/M version of BBC BASIC (Z80)

Syntax

```
<n-var>=POINT(<numeric>,<numeric>)
```

Associated Keywords
PLOT, DRAW, MOVE, GCOL

POS

A function returning the horizontal position of the cursor on the screen. The left hand column is 0 and the right hand column is one less than the width of the display.

```
X=POS
```

COUNT will tell you the print head position of the printer. It is an uncertain indicator of the horizontal position of the cursor on the screen. (See the keyword COUNT for details.)

See VPOS for an example of the use of POS and VPOS.

Syntax

```
<n-var>=POS
```

Associated Keywords
COUNT, TAB, VPOS

PRINT P.

A statement which prints characters to the screen or printer.

General Information

The items following PRINT are called the print list. The print list may contain a sequence of string or numeric literals or variables. The spacing between the items printed will vary depending on the punctuation used. If the print list does not end with a semi-colon, a new-line will be printed after all the items in the print list.

In the examples which follow, commas have been printed instead of spaces to help you count.

The screen is divided into zones (initially) 10 characters wide. By default, numeric quantities are printed right justified in the print zone and strings are printed just as they are (with no leading spaces). Numeric quantities can be printed left justified by preceding them with a semi-colon. In the examples the zone width is indicated as z10, z4 etc.

```
           z10
           012345678901234567890123456789
PRINT 23.162      ,,,,23.162
PRINT "HELLO"     HELLO
PRINT ;23.162     23.162
```

Initially numeric items are printed in decimal. If a tilde (~) is encountered in the print list, the numeric items which follow it are printed in hexadecimal. If a comma or a semi-colon is encountered further down the print list, the format reverts to decimal.

```
          z10
          012345678901234567890123456789
PRINT ~10 58,58    ,,,,,,,,,A,,,,,,,,3A,,,,,,,,58
```

A comma (,) causes the cursor to TAB to the beginning of the next print zone unless the cursor is already at the start of a print zone. A semi-colon causes the next and following items to be printed on the same line immediately after the previous item. This 'no-gap' printing continues until a comma (or the end of the print list) is encountered. An apostrophe (') will force a new line. TAB(X) and TAB(Y,Z) can also be used at any position in the print line to position the cursor.

```
          z10
          012345678901234567890123456789
PRINT "HELLO",24.2  HELLO    ,,,,,,24.2
PRINT "HELLO";24.2  HELLO24.2
PRINT ;2 5 4.3,2   254.3    ,,,,,,,,,2
PRINT "HELLO"'2.45  HELLO
          ,,,,,,2.45
```

Unlike most other versions of BASIC, a comma at the end of the print list will not suppress the new line and advance the cursor to the next zone. If you wish to split a line over two or more PRINT statements, end the previous print list with a semicolon and start the following list with a comma or end the line with a comma followed by a semicolon.

```
          z10
          012345678901234567890123456789
PRINT "HELLO" 12;   HELLO,,,,,,,,12,,,,,,,,,,23.67
PRINT ,23.67
```

or

```
PRINT "HELLO" 12,;
PRINT 23.67
```

Printing a string followed by a numeric effectively moves the start of the print zones towards the right by the length of the string. This displacement continues until a comma is encountered.

```
          z10
```

```
             012345678901234567890123456789
PRINT "HELLO"12 34  HELLO,,,,,,,,12,,,,,,,,34
PRINT "HELLO"12,34  HELLO,,,,,,,,12     ,,,,,,,,34
```

Print Format Control

Although PRINT USING is not implemented in BBC BASIC, similar control over the print format can be obtained. The overall width of the print zones and print field, the number of figures or decimal places and the print format may be controlled by setting the print variable, @%, to the appropriate value. The print variable (@%) comprises 4 bytes and each byte controls one aspect of the print format. @% can be set equal to a decimal integer, but it is easier to use hexadecimal, since each byte can then be considered separately.

```
@%=&SSNNPPWW
```

Byte	Range	Default	Purpose
SS	00-01	00	STR$ Format Control
NN	00-02	00	Format Selection
PP	??-??	09	Number of Digits Printed
WW	00-0F	0A(10)	Zone and Print Field Width

STR$ Format Control – SS

Byte 3 effects the format of the string generated by the STR$ function. If Byte 3 is 1 the string will be generated according to the format set by @%, otherwise the G9 format will be used.

Format Selection – NN

Byte 2 selects the general format as follows:

00 General Format (G).
01 Exponential Format (E).
02 Fixed Format (F).

G Format	Numbers that are integers are printed as such. Numbers in the range 0.1 to 1 will be printed as such. Numbers less than 0.1 will be printed in E format. Numbers greater than the range set by Byte 1 will be printed in E format. In which case, the number of digits printed will still be controlled by Byte 1, but according to the E format rules.

The earlier examples were all printed in G9 format.

E Format	Numbers are printed in the scientific (engineering) notation.
F Format	Numbers are printed with a fixed number of decimal places.

Number of Digits - PP

Byte 1 controls the number of digits printed in the selected format. The number is rounded (NOT truncated) to this size before it is printed. If Byte 1 is set outside the range allowed for by the selected format, it is taken as 9. The effect of Byte 1 differs slightly with the various formats.

Format	Range	Control Function
G	01-0A	The maximum number of digits which can be printed, excluding the decimal point, before changing to the E format.

```
                01234567890123456789
&030A - G3z10
(00'00'03'0A)
PRINT 1000.31     ,,,,,,,1E3
PRINT 1016.31     ,,,,1.02E3
PRINT 10.56      ,,,,,,10.6
```

Format	Range	Control Function
E	01-FF	The total number of digits to be printed excluding the decimal

point and the digits after the E. Three characters or spaces are always printed after the E. If the number of significant figures called for is greater than 10, then trailing zeros will be printed.

```
01030A - E3z10
(00'01'03'0A)
              01234567890123456789
PRINT 10.56         ,,1.06E1

&010F0A - E15z10
(00'01'0F'0A)
              01234567890123456789
PRINT 10.56         1.05600000000000E1
```

F 00-0A The number of digits to be printed after the decimal point.

```
&02020A - F2z10
(00'02'02'0A)
              01234567890123456789
PRINT 10.56         ,,,,,10.56
PRINT 100.5864      ,,,,100.59
PRINT .64862        ,,,,,,0.65
```

Zone Width – WW

Byte 0 sets the width of the print zones and field.

```
&020208 - F2z8
(00'00'02'08)
```

followed by

```
&020206 - F2z6
(00'02'02'06)
```

```
              01234567890123456789
PRINT 10.2,3.8     ,,,10.20,,,,3.80
PRINT 10.2,3.8     ,10.20,,3.80
```

Changing the Print Control Variable

It is possible to change the print control variable (@%) within a print list by using the function:

```
DEF FN_pformat(N):@%=N:=""
```

Functions have to return an answer, but the value returned by this function is a null string. Consequently, its only effect is to change the print control variable. Thus the PRINT statement

```
PRINT FN_pformat(&90A) x FN_pformat(&2020A) y
```

will print x in G9z10 format and y in F2z10 format.

Examples

```
G9z10              G2z10
&00090A            &00020A
012345678901234    012345678901234
1111.11111         ,,,,,1.1E3
13.7174211         ,,,,,,,,14
,1.5241579         ,,,,,,,1.5
1.88167642E-2      ,,,,1.9E-2
2.09975158E-3      ,,,,2.1E-3

F2z10              E2z10
&02020A            &0102A
012345678901234    012345678901234
,,,1111.11         ,,,1.1E3
,,,,,13.72         ,,,1.4E1
,,,,,,1.52         ,,,1.5E0
,,,,,,0.02         ,,,1.9E-2
,,,,,,0.00         ,,,2.1E-3
```

The results obtained by running the following example program show the effect of changing the zone width. The results for zone widths of 5 and 10 (&0A) illustrate what happens when the zone width is too small for the number to be printed properly. The example also illustrates what happens when the number is too large for the chosen precision.

```
 10 test=7.8123
 20 FOR i=5 TO 25 STEP 5
 30   PRINT
 40   @%=&020200+i
 50   PRINT "@%=&000";~@%
 60   PRINT STRING$(3,"0123456789")
 70   FOR j=1 TO 10
 80     PRINT test^j
 90   NEXT
100   PRINT '
110 NEXT
120 @%=&90A
```

```
&00020205
012345678901234567890123456789
 7.81
61.03
476.80
3724.91
29100.11
227338.75
1776038.54
13874945.89
1.083952398E8
8.46816132E8

&0002020A
012345678901234567890123456789
      7.81
     61.03
    476.80
   3724.91
  29100.11
 227338.75
1776038.54
13874945.89
1.083952398E8
8.46816132E8

&0002020F
012345678901234567890123456789
           7.81
          61.03
```

```
         476.80
        3724.91
       29100.11
      227338.75
     1776038.54
    13874945.89
  1.083952398E8
   8.46816132E8

&00020214
012345678901234567890123456789
                7.81
               61.03
              476.80
             3724.91
            29100.11
           227338.75
          1776038.54
         13874945.89
       1.083952398E8
        8.46816132E8

&00020219
012345678901234567890123456789
                     7.81
                    61.03
                   476.80
                  3724.91
                 29100.11
                227338.75
               1776038.54
              13874945.89
            1.083952398E8
             8.46816132E8
```

Syntax

```
PRINT {[TAB(<numeric>[,<numeric>])][SPC(<numeric>]
    ['][,][;][~][<str>|<numeric>]}
```

Associated Keywords

PRINT#, TAB, POS, STR$, WIDTH, INPUT, VDU

PRINT# P.#

A statement which writes the internal form of a value out to a data file.

```
PRINT#E,A,B,C,D$,E$,F$
PRINT#4,prn$
```

The format of the variables as written to the file differs from the format used on the BBC Micro. All numeric values are written as five bytes of binary real data (see the Annex entitled Format of Program and Variables in Memory). Strings are written as the bytes in the string (in the correct order) plus a carriage return.

Before you use this statement, you must normally have opened a file or device using OPENOUT or OPENUP.

You can use PRINT# to write data (numbers and strings) to a data file in the 'standard' manner. If you wish to 'pack' your data in a different way, you should use BPUT#. You can use PRINT# and BPUT# together to mix or modify the data format. For example, if you wish to write a 'compatible' text file, you could PRINT# the string and BPUT# a line-feed. This would write the string followed by a carriage-return and a line-feed to the file.

Remember, with BBC BASIC (Z80) the format of the file is completely under your control.

Syntax

```
PRINT#<numeric>{,<numeric>|<str>}
```

Associated Keywords

PRINT, OPENUP, OPENOUT, CLOSE#, INPUT#, BPUT#, BGET#, GET$, EXT#, PTR#, E OF#

PROC

A keyword used at the start of all user declared procedures. The first character of a procedure name can be an underline (or a number).

If there are spaces between the procedure name and the opening bracket of the parameter list (if any) they must be present both in the definition and the call. It's safer not to have spaces between the procedure name and the opening bracket.

A procedure may be defined with any number of parameters of any type.

A procedure definition is terminated by ENDPROC.

A procedure does not have to be declared before it is called.

Procedures are re-entrant and the parameters (arguments) are passed by value.

```
 10 INPUT"Number of discs "F
 20 PROC_hanoi(F,1,2,3)
 30 END
 40 :
 50 DEF PROC_hanoi(A,B,C,D)
 60 IF A=0 THEN ENDPROC
 70 PROC_hanoi(A-1,B,D,C)
 80 PRINT"Move disk ";A" from ";B" to ";C
 90 PROC_hanoi(A-1,D,C,B)
100 ENDPROC
```

See the Procedures and Functions sub-section for more details.

Syntax

```
PROC<name>[(<exp>{,<exp>})]
```

Associated Keywords

DEF, ENDPROC, LOCAL

PTR#

A pseudo-variable allowing the random-access pointer of the file whose file handle is its argument to be read and changed.

```
PTR#F=PTR#F+5 :REM Move pointer to next number
PTR#F=recordnumber*recordlength
```

Reading or writing (using BGET#, BPUT#, GET$, INPUT# or PRINT#) takes place at the current position of the pointer. The pointer is automatically updated following a read or write operation.

You can use PTR# to select which item in a file is to be read or written to next. In a random file (see the section on BBC BASIC (Z80) Files) you use PTR# to select the record you wish to read or write.

If you wish to move about in a file using PTR# you will need to know the precise format of the data in the file.

A file opened with OPENUP may be extended by setting its pointer to its end (PTR#fnum = EXT#fnum) and then writing to it. If you do this, you must remember to CLOSE the file when you have finished with it in order to update the directory entry.

By using PTR# you have complete control over where you read and write data in a file. This is simple concept, but it may initially be difficult to grasp its many ramifications. The BBC BASIC (Z80) Files section has a number of examples of the use of PTR#.

Syntax

```
PTR#<numeric>=<numeric>
<n-var>=PTR#<numeric>
```

Associated Keywords

OPENIN, OPENUP, OPENOUT, CLOSE#, PRINT#, INPUT#, BPUT#, BGET#, GET$, EX T#, EOF#

PUT

A statement to output data to an output port.

```
PUT A,N :REM output N to port A.
```

This instruction gives direct access from BBC BASIC (Z80) to the computer's I/O hardware. Typically, you can use it to directly access I/O ports.

Syntax

```
PUT <numeric>,<numeric>
```

Associated Keywords
GET

QUIT

A statement which causes the BBC BASIC interpreter to terminate. QUIT has the same effect as *BYE.

Syntax

```
QUIT [<numeric>]
```

Associated Keywords
END, STOP

RAD

A function which converts degrees to radians.

```
X=RAD(Y)
X=SIN RAD(90)
```

Unlike humans, BBC BASIC (Z80) wants angles expressed in radians. You can use this function to convert an angle expressed in degrees to radians before using one of the angle functions (SIN, COS, etc).

Using RAD is equivalent to multiplying the degree value by PI/180, but the result is calculated internally to a greater accuracy.

See COS, SIN and TAN for further examples of the use of RAD.

Syntax

```
<n-var>=RAD<numeric>
```

Associated Keywords
DEG

READ

A statement which will assign to variables values read from the DATA statements in the program. Strings must be enclosed in double quotes if they have leading spaces or contain commas.

```
READ C,D,A$
```

In many of your programs, you will want to use data values which do not change frequently. Because these values are subject to some degree of change, you won't want to use constants. On the other hand, you won't want to input them every time you run the program either. You can get the best of both worlds by declaring these values in DATA statements at the beginning or end of your program and READing them into variables in your program.

A typical use for DATA and READ is a name and address list. The addresses won't change very often, but when they do you can easily amend the appropriate DATA statement.

See DATA for more details and an example of the use of DATA and READ.

Syntax

```
READ <n-var>|<s-var>{<n-var>|<s-var>}
```

Associated Keywords
DATA, RESTORE

READ#

A statement which is synonymous with INPUT#. READ# and INPUT# have identical effects; READ# is implemented in the interests of improved compatibility with other dialects of BASIC.

Syntax

```
READ #<numeric>,<n-var>|<s-var>{,<n-var>|<s-var>}
```

Associated Keywords
READ, INPUT#, OPENIN, OPENUP, OPENOUT, CLOSE#, PRINT#, BPUT#, BGET#, EXT#, PTR#, EOF#, GET$#

RECTANGLE

(BBC BASIC version 5 or later only, not implemented in the generic CP/M edition)

A statement which draws an outline rectangle or a filled rectangle. RECTANGLE is followed by the X and Y coordinates of the bottom left-hand corner of the rectangle then the width and height of the rectangle. To draw a square you can omit the fourth parameter; the height is then assumed to be the same as the width. To draw a filled (solid) rectangle or square use RECTANGLE FILL.

The rectangle is drawn in the current graphics foreground colour. This colour can be changed using the GCOL statement.

RECTANGLE can also be used to copy or move a rectangular area of the screen to a different place. In this case RECTANGLE is followed by the X and Y coordinates of the bottom left-hand corner of the source rectangle, the width and height of the rectangle, the keyword TO and finally the X and Y coordinates of the bottom left-hand corner of the destination rectangle. To *move* the rectangle rather than *copy* it, use RECTANGLE FILL. To *swap* two rectangular areas use RECTANGLE SWAP.

```
RECTANGLE xpos,ypos,width,height
RECTANGLE FILL 400,500,200
RECTANGLE 100,100,200,200 TO 300,300
RECTANGLE FILL srcx,srcy,dx,dy TO dstx,dsty
RECTANGLE SWAP x1,y1,dx,dy TO x2,y2
```

RECTANGLE is also used in the MOUSE RECTANGLE statement.

Syntax

```
RECTANGLE [FILL] <numeric>,<numeric>,<numeric>[,<numeric>] [TO <numeric>,<numeric>]
RECTANGLE SWAP <numeric>,<numeric>,<numeric>[,<numeric>] TO <numeric>,<numeric>
```

Associated Keywords
FILL, MOUSE, SWAP, TO

REM

A statement that causes the rest of the line to be ignored thereby allowing comments to be included in a program.

You can use the REM statement to put remarks and comments in to your program to help you remember what the various bits of your program do. BBC BASIC (Z80) completely ignores anything on the line following a REM statement.

You will be able to get away without including any REMarks in simple programs. However, if you go back to a lengthy program after a couple of months you will find it very difficult to understand if you have not included any REMs.

If you include nothing else, include the name of the program, the date you last revised it and a revision number at the start of your program.

```
10 REM WSCONVERT REV 2.30
20 REM 5 AUG 84
30 REM Converts 'hard' carriage-returns to 'soft'
40 REM ones in preparation for use with WS.
```

Syntax

REM any text

Associated Keywords
None

RENUMBER **REN.**

A command which will renumber the lines and correct the cross references inside a program.

The options are as for AUTO, but increments of greater than 255 are allowed.

You can specify both the new first number (n1) and/or the step size (s). The default for both the first number and the step size is 10. The two parameters should be separated by a comma or a hyphen.

```
RENUMBER
RENUMBER n1
RENUMBER n1,s
RENUMBER ,s
```

For example:

```
RENUMBER            First number 10, step 10
RENUMBER 1000       First number 1000, step 10
RENUMBER 1000-5     First number 1000, step 5
RENUMBER ,5         First number 10, step 5
RENUMBER -5         First number 10, step 5
```

RENUMBER produces error messages when a cross reference fails. The line number containing the failed cross-reference is renumbered and the line number in the error report is the new line number.

If you renumber a line containing an ON GOTO/GOSUB statement which has a calculated line number, RENUMBER will correctly cope with line numbers before the calculated line number. However, line numbers after the calculated line number will not be changed.

In the following example, the first two line numbers would be renumbered correctly, but the last two would be left unchanged.

```
ON action GOSUB 100,200,(type*10+300),400,500
```

Syntax

```
RENUMBER [<n-const>[,<n-const>]]
```

Associated Keywords
LIST

REPEAT REP.

A statement which is the starting point of a REPEAT...UNTIL loop. A single REPEAT may have more than one UNTIL, but this is bad practice.

The purpose of a REPEAT...UNTIL loop is to make BBC BASIC (Z80) repeat a set number of instructions until some condition is satisfied.

```
REPEAT UNTIL GET=13 :REM wait for CR

X=0
REPEAT
  X=X+10
  PRINT "What do you think of it so far?"
UNTIL X>45
```

You must not exit a REPEAT...UNTIL loop with a GOTO. If you jump out of a loop with a GOTO (How could you!!!) you should jump back in. If you must jump out of the loop, you should use UNTIL TRUE to 'pop' the stack. For (a ghastly) example:

```
 10 i=1
 20 REPEAT: REM Print 1 to 100 unless
 30   I=I+1: REM interrupted by the
 40   PRINT i: REM space bar being pressed
 50   x=INKEY(0): REM Get a key
 60   IF x=32 THEN 110:REM exit if <SPACE>
 70 UNTIL i=100
 80 PRINT "****"
 90 END
100 :
110 UNTIL TRUE: REM Pop the stack
120 PRINT "Forced exit":REM Carry on with program
130 FOR j=1000 TO 1005
140   PRINT j
150 NEXT
160 END
```

See the keyword UNTIL for ways of using REPEAT...UNTIL loops to replace unconditional GOTOs for program looping.

See the sub-section on Program Flow Control in the General Information section for more details on the working of the program stack.

Syntax

REPEAT

Associated Keywords

UNTIL

REPORT/REPORT$

A statement (REPORT) which outputs, and a string function (REPORT$) which returns, the error string associated with the last error which occurred.

You can use these keywords within your own error handling routines to print out or process an error message for those errors you are not able to cope with.

The example below is an error handling routine designed to clear the screen before reporting the error:

```
ON ERROR MODE 3 : REPORT : PRINT " at line ";ERL : END
```

See the sub-section on Error Handling and the keywords ERR, ERL and ON ERROR for more details.

Syntax

```
REPORT
<s-var> = REPORT$
```

Associated Keywords
ERR, ERROR, ERL, ON ERROR

RESTORE **RES.**

A statement which moves the data pointer. RESTORE can be used at any time in a program to set the line from where READ reads the next DATA item.

RESTORE on its own resets the data pointer to the first data item in the program.

RESTORE followed by a line number sets the data pointer to the first item of data in the specified line (or the next line containing a DATA statement if the specified line does not contain data). This optional parameter for RESTORE can specify a calculated line number.

(BBC BASIC version 5 or later only)

RESTORE followed by a plus sign (+) and a positive numeric value sets the data pointer to the first item of data in the line *offset from the line containing the RESTORE statement* by the specified number of lines (or the next DATA statement if the specified line does not contain any data).

```
RESTORE
RESTORE 100
RESTORE +2
RESTORE (10*A+20)
RESTORE (mydata)
```

You can use RESTORE to reset the data pointer to the start of your data in order to re-use it. alternatively, you can have several DATA lists in your program and use RESTORE to set the data pointer to the appropriate list.

RESTORE DATA

(BBC BASIC version 5 or later only)

RESTORE DATA causes the DATA pointer saved by a previous LOCAL DATA statement to be restored from the stack.

RESTORE ERROR

(BBC BASIC version 5 or later only)

RESTORE ERROR causes the error-trapping status saved by a previous ON ERROR LOCAL statement to be restored from the stack.

RESTORE LOCAL

(BBC BASIC version 5 or later only)

RESTORE LOCAL (which can be used only inside a user-defined procedure or function) restores the values of formal parameters and LOCAL variables to those they had before the procedure/function was called. RESTORE LOCAL also performs the actions of RESTORE ERROR and RESTORE DATA. This is primarily intended for use within an ON ERROR LOCAL error handler.

Syntax

```
RESTORE [<l-num>]
RESTORE [(<numeric>)]
RESTORE +<numeric>
RESTORE DATA
RESTORE ERROR
RESTORE LOCAL
```

Associated Keywords

READ, DATA, ON ERROR LOCAL

RETURN

R.

A statement causing a RETURN to the statement after the most
recent GOSUB statement.

You use RETURN at the end of a subroutine to make BBC BASIC (Z80) return to the place in your program which originally 'called' the subroutine.
You may have more than one return statement in a subroutine, but it is preferable to have only one entry point and one exit point (RETURN).

Try to structure your program so you don't leave a subroutine with a GOTO. If you do, you should always return to the subroutine and exit via the RETURN statement. If you insist on using GOTO all over the place, you will end up confusing yourself and maybe confusing BBC BASIC (Z80) as well. The sub-section on Program Flow Control explains why.

Syntax

```
RETURN
```

(BBC BASIC version 5 or later only)

RETURN is also used in a function or procedure DEFinition to specify that a parameter should be passed by reference rather than by value. When used with a whole array parameter, RETURN allows the array to be *declared* (using DIM) within the function or procedure.

Associated Keywords
GOSUB, ON GOSUB

RIGHT$

A string function which returns the right 'num' characters of the string. If there are insufficient characters in the string then all are returned.

(BBC BASIC version 5 or later only)

RIGHT$ may also be used to the left of an equals sign to change the end of a string whilst leaving the rest alone.

There must not be any spaces between the RIGHT$ and the opening bracket.

```
A$ = RIGHT$(A$,num)
A$ = RIGHT$(A$,2)
A$ = RIGHT$(LEFT$(A$,3),2)
RIGHT$(A$,3) = B$
```

For example,

```
name$="BBC BASIC for Z80"
FOR i=3 TO 13
  PRINT RIGHT$(name$,i)
NEXT
END
```

would print

```
Z80
 Z80
r Z80
or Z80
for Z80
 for Z80
C for Z80
IC for Z80
SIC for Z80
ASIC for Z80
BASIC for Z80
```

When using RIGHT$ on the left of an equals sign, and the expression to the right of the equals sign evaluates to a string with *fewer* characters than the specified sub-string length, only that number of characters is changed. For example:

```
A$ = "BBC BASIC"
RIGHT$(A$,4) = "ZZ"
```

will set A$ equal to "BBC BASZZ". Although the sub-string length is set to four, only the last two characters are actually modified since that is the length of the string "ZZ".

(BBC BASIC version 5 or later only)

RIGHT$(A$) is shorthand for RIGHT$(A$,1), in other words it returns or sets the last character of A$.

Syntax

```
<s-var>=RIGHT$(<string>[,<numeric>])
RIGHT$(<s-var>[,<numeric>])=<string>
```

Associated Keywords
LEFT$, MID$, INSTR

RND

A function which returns a random number. The type and range of the number returned depends upon the optional parameter.

RND	returns a random integer 1 - &FFFFFFFF.
RND(n)	returns an integer in the range 1 to n (n>1).
RND(1)	returns a real number in the range 0.0 to .99999999.

If n is negative the pseudo random sequence generator is set to a number based on n and n is returned.
If n is 0 the last random number is returned in RND(1) format.

```
X=RND(1)
X%=RND
N=RND(6)
```

The random number generator is initialised by RUN (or CHAIN). Consequently, RND will return zero until the RUN (or CHAIN) command is first issued.

Syntax

```
<n-var>=RND[(<numeric>)]
```

Associated Keywords

None

RUN

Start execution of the program.

```
RUN
```

RUN is a statement and so programs can re-execute themselves.

All variables except @% to Z% are CLEARed by RUN.

If you want to start a program without clearing all the variables, you can use the statement

```
GOTO nnnn
```

where nnnn is number of the line at which you wish execution of the program to start.
RUN "filename" can be used as an alternative to CHAIN "filename".

Syntax

```
RUN
```

Associated Keywords
NEW, OLD, LIST, CHAIN

SAVE SA.

A statement which saves the current program area to a file, in internal (tokenised) format.

```
SAVE "FRED.BBC"
SAVE A$
```

You use SAVE to save your program for use at a later time. The program must be given a name (file-specifier) and this name becomes the name of the file in which your program is saved.

The name (file-specifier) must follow the normal CP/M-80 naming conventions. See the Operating System Interface section for a description of a file-specifier (name). Unless a different extension is specified in the file-specifier, .BBC is automatically used. Thus,

```
SAVE "FRED"
```

would save the current program to a file called FRED.BBC in the current directory. If you want to exclude the extension, include the full-stop in the file name, but don't follow it with anything. For example, to save a program to a file called 'FRED', type,

```
SAVE "FRED."
```

Syntax

```
SAVE <str>
```

Associated Keywords
LOAD, CHAIN

SGN

A function returning -1 for negative argument, 0 for zero argument and +1 for positive argument.

```
X=SGN(Y)
result=SGN(answer)
```

You can use this function to determine whether a number is positive, negative or zero.

SGN returns:

```
 +1 for positive numbers
   0 for zero
  -1 for negative numbers
```

Syntax

```
<n-var>=SGN(<numeric>)
```

Associated Keywords
ABS

SIN

A function giving the sine of its radian argument.

```
X=SIN(Y)
```

This function returns the sine of an angle. The angle must be expressed in radians, not degrees.

Whilst the computer is quite happy dealing with angles expressed in radians, you may prefer to express angles in degrees. You can use the RAD function to convert an angle from degrees to radians.

The example below sets Y to the sine of the angle 'degree_angle' expressed in degrees.

```
Y=SIN(RAD(degree_angle))
```

Syntax

```
<n-var>=SIN(<numeric>)
```

Associated Keywords
COS, TAN, ACS, ASN, ATN, DEG, RAD

SOUND

A statement which generates sounds.

Not implemented in the generic CP/M version of BBC BASIC (Z80)

Syntax

```
SOUND <numeric>,<numeric>,<numeric>,<numeric>
```

Associated Keywords
ENVELOPE

SPC

A statement which prints a number of spaces to the screen (or currently selected console output stream). The argument specifies the number of spaces (up to 255) to be printed.

SPC can only be used within a PRINT or INPUT statement.

```
PRINT DATE;SPC(6);SALARY
```

```
INPUT SPC(10) "What is your name ",name$
```

Syntax

```
PRINT SPC(<numeric>)
INPUT SPC(<numeric>)
```

Associated Keywords
TAB, PRINT, INPUT

SQR

A function returning the square root of its argument.

```
X=SQR(Y)
```

If you attempt to calculate the square root of a negative number, a '-ve root' error will occur. You could use error trapping to recover from this error, but it is better to check that the argument is not negative before using the SQR function.

Syntax

```
<n-var>=SQR(<numeric>)
```

Associated Keywords
None

STEP S.

Part of the FOR statement, this optional section specifies step sizes other than 1.

```
FOR i=1 TO 20 STEP 5
```

The step may be positive or negative. STEP is optional; if it is omitted, a step size of +1 is assumed.

You can use this optional part of the FOR...TO...STEP...NEXT structure to specify the amount by which the FOR...NEXT loop control variable is changed each time round the loop. In the example below, the loop control variable, 'cost' starts as 20, ends at 5 and is changed by -5 each time round the loop.

```
10 FOR cost=20 TO 5 STEP -5
20   PRINT cost,cost*1.15
30 NEXT
```

Syntax

```
FOR <n-var>=<numeric> TO <numeric> [STEP <numeric>]
```

Associated Keywords
FOR, TO, NEXT

STOP

Syntactically identical to END, STOP also prints a message to the effect that the program has stopped.

You can use STOP at various places in your program to aid debugging. If your program is going wrong, you can place STOP commands at various points to see the path taken by your program. (TRACE is generally a more useful aid to tracing a program's flow unless you are using formatted screen displays.)

Once your program has STOPped you can investigate the values of the variables to find out why things happened the way they did.

STOP DOES NOT CLOSE DATA FILES. If you use STOP to exit a program for debugging, CLOSE all the data files before RUNning the program again. If you don't you will get some most peculiar error messages.

Syntax

```
STOP
```

Associated Keywords
END

STR$

A string function which returns the string form of the numeric argument as it would have been printed.

If the most significant byte of @% is not zero, STR$ uses the current @% description when generating the string. If it is zero (the initial value) then the G9 format (see PRINT) is used.

If STR$ is followed by ~ (tilde) then a hexadecimal conversion is carried out.

```
A$=STR$(PI)
B$=STR$~(100) :REM B$ will be "64"
```

The opposite function to STR$ is performed by the VAL function.

Syntax

```
<s-var>=STR$[~](<numeric>)
```

Associated Keywords
VAL, PRINT

STRING$

A function returning N concatenations of a string.

```
A$=STRING$(N,"hello")
B$=STRING$(10,"-*-")
C$=STRING$(Z%,S$)
```

You can use this function to print repeated copies of a string. It is useful for printing headings or underlinings. The last example for PRINT uses the STRING$ function to print the column numbers across the page. For example,

```
PRINT STRING$(4,"-=*=-")
```

would print

```
-=*=--=*=--=*=--=*=-
```

and

```
PRINT STRING$(3,"0123456789")
```

would print

```
012345678901234567890123456789
```

Syntax

```
<s-var>=STRING$(<numeric>,<str>)
```

Associated Keywords
None

SUM and SUMLEN

(BBC BASIC version 5 or later only)

SUM

A function which returns the sum of all the elements of a numeric array (floating-point, integer or byte) or the concatenation of all the strings in a string array.

```
DIM marks%(10)
marks%() = 0,1,2,3,4,5,6,7,8,9,10
PRINT SUM(marks%())
```

The above program segment will print the value 55, being the sum of the eleven values in the array marks%(10).

If the array has more than one dimension, all the elements in all the dimensions are summed:

```
DIM grid(7,7)
grid() = PI
PRINT SUM(grid())
```

This will print the value 201.06193, being PI * 64 (64 being the number of elements in the 8 x 8 array).

SUMLEN

A function which returns the total length of all the strings in a string array. The difference between SUMLEN(array$()) and LENSUM(array$()) is that the former will work correctly whatever the total length of the strings, whereas the latter may result in the String too long error.

Syntax

```
<n-var>=SUM(<n-array()>)
<s-var>=SUM(<s-array()>)
<n-var>=SUMLEN(<s-array()>)
```

Associated Keywords

DIM

SWAP

(BBC BASIC version 5 or later only)

A statement which swaps (exchanges) the contents of two variables or two arrays. The variables or arrays must be of the **same type**, for example two integer numeric variables, or two string arrays. If the types differ, a Type mismatch error will result. If two arrays are swapped, their dimensions are also swapped.

```
SWAP a,b
SWAP Coat$,Hat$
SWAP A$(),B$()
```

Syntax

```
SWAP <n-var>,<n-var>
SWAP <s-var>,<s-var>
SWAP <array()>,<array()>
```

Associated Keywords
DIM, RECTANGLE

SYS

(BBC BASIC version 5 or later only, if implemented)

A statement which calls an Operating System API (Application Program Interface) function or a function in a Shared Object (e.g. a DLL). SYS is followed by the function's address or the function's name as a (case sensitive) string plus, optionally, a list of comma-separated numeric or string parameters to be passed to the function. A 32-bit integer value may be returned from the function by adding TO followed by a numeric variable name.

```
SYS "FunctionName", parameter, list
SYS "FunctionName", parameter, list TO result%
```

Syntax

```
SYS <numeric>|<string> {,<numeric>|<string>} [TO <n-var>]
<n-var>=SYS(<string>)
```

Associated Keywords
CALL

TAB

A keyword available in PRINT or INPUT.

There are two versions of TAB: TAB(X) and TAB(X,Y) and they are effectively two different keywords.

TAB(X) is a printer orientated statement. The number of printable characters since the last new-line (COUNT) is compared with X. If X is equal or greater than COUNT, sufficient spaces to make them equal are printed. These spaces will overwrite any characters which may already be on the screen. If X is less than COUNT, a new-line will be printed first.

TAB(X,Y) is a screen orientated statement. It will move the cursor on the screen to character cell X,Y (column X, row Y) if possible. No characters are overwritten and COUNT is NOT updated. Consequently, a TAB(X,Y) followed by a TAB(X) will give unpredictable (at first glance) results.

The leftmost column is column 0 and the top of the screen is row 0.

```
PRINT TAB(10);A$
PRINT TAB(X,Y);B$
```

Syntax

```
PRINT TAB(<numeric>[,<numeric>])
INPUT TAB(<numeric>[,<numeric>])
```

Associated Keywords

POS, VPOS, PRINT, INPUT

TAN T.

A function giving the tangent of its radian argument.

```
X = TAN(Y)
```

This function returns the tangent of an angle. The angle must be expressed in radians, not degrees.

Whilst the computer is quite happy dealing with angles expressed in radians, you may prefer to express angles in degrees. You can use the RAD function to convert an angle from degrees to radians.

The example below sets Y to the tangent of the angle 'degree_angle' expressed in degrees.

```
Y=TAN(RAD(degree_angle))
```

Syntax

```
<n-var>=TAN<numeric>
```

Associated Keywords

COS, SIN, ACS, ATN, ASN, DEG, RAD

THEN TH.

An optional part of the IF... THEN ... ELSE statement. It introduces the action to be taken if the testable condition evaluates to a non-zero value.

```
IF A=B THEN 3000
IF A=B THEN PRINT "Equal" ELSE PRINT "Help"
IF A=B THEN
  PRINT "Equal"
ELSE
  PRINT "Help"
ENDIF
```

You need to use THEN if it is:

- Followed directly by a line number:

```
IF a=b THEN 320
```

- Followed by a 'star' (*) command:

```
IF a=b THEN *DIR
```

- Followed by an assignment of a pseudo-variable:

```
IF a=b THEN TIME=0
```

- Introducing a multi-line IF ... ENDIF clause ***(BBC BASIC version 5 or later only)***:

```
IF a=b THEN
  PRINT "Equals"
ENDIF
```

or you wish to exit from a function as a result of the test:

```
DEF FN_test(a,b,num)
IF (a>b) THEN =num
=num/256
```

When THEN introduces a multi-line IF ... ENDIF statement it must be the very last thing on the line, not even followed by a REMark.
THEN may be followed immediately by a semicolon (;) to aid translation of the ELSEIF keyword available in some other BASIC dialects, for example:

```
IF condition1 THEN
  PROCone
ELSEIF condition2 THEN;
  PROCtwo
ELSEIF condition3 THEN;
  PROCthree
ENDIF
```

This facility is intended for use only by automatic translators, rather than in new programs.

Syntax

```
IF <t-cond> THEN <stmt>{:<stmt>} [ELSE <stmt>{:<stmt>}]
IF <t-cond> THEN
ELSEIF <t-cond> THEN;
```

Associated Keywords

IF, ELSE, ENDIF

TIME TI.

A pseudo-variable which reads and sets the elapsed time clock. This functionality depends on BBC BASIC (Z80) being configured using the **DIST.Z80** patch program.

```
X=TIME
TIME=100
```

The following example is a simple program to provide a 24 hour clock. Lines 20 to 40 get the correct time, lines 50 and 60 calculate the number of centi-seconds and set TIME, and lines 110 to 130 convert the value in TIME to hours, minutes and seconds. Line 90 stops the time being printed unless it has changed by at least one second.

```
 10 CLS
 20 INPUT "HOURS ",H
 30 INPUT "MINUTES ",M
 40 INPUT "SECONDS ",S
 50 PRINT "PUSH ANY KEY TO SET THE TIME ";:X=GET
 60 TIME=((H*60+M)*60+S)*100
 70 T=0
 80 REPEAT
 90   IF TIME DIV 100=T DIV 100 THEN 150
100   T=TIME
110   S=(T DIV 100) MOD 60
120   M=(T DIV 6000) MOD 60
130   H=(T DIV 360000) MOD 24
140   PRINT TAB(0,23) H;":";M;":";S;
150 UNTIL FALSE
```

Syntax

```
TIME=<numeric>
<n-var>=TIME
```

Associated Keywords

TINT

(BBC BASIC version 5 or later only, not implemented in the generic CP/M edition)

A keyword which sets and/or returns an extended colour value. On systems which use a **colour palette** TINT may optionally be used to specify a colour in terms of its RGB (red, green and blue) components. For more details see the documentation of your specific system.

Associated Keywords
PLOT, DRAW, MOVE, GCOL, POINT

TO

The part of the FOR ... TO ... STEP statement which introduces the terminating value for the loop. When the loop control variable exceeds the value following 'TO' the loop is terminated.

For example,

```
10 FOR i=1 TO 5 STEP 1.5
20   PRINT i
30   NEXT
40 PRINT "**********"
50 PRINT i
```

will print

```
         1
       2.5
         4
**********
       5.5
```

Irrespective of the initial value of the loop control variable and the specified terminating value, the loop will execute at least once. For example,

```
10 FOR i= 20 TO 10
20   PRINT i
30 NEXT
```

will print

```
        20
```

Syntax

```
FOR <n-var>=<numeric> TO <numeric> [STEP <numeric>]
```

(BBC BASIC version 5 or later only)

TO is also used with the GET$#channel function to specify an optional terminator character.

Associated Keywords
FOR, GET$#, NEXT, STEP

TOP

A function which returns the value of the first free location after the end of the current program.

The length of your program is given by TOP-PAGE.

```
PRINT TOP-PAGE
```

Syntax

```
<n-var>=TOP
```

Associated Keywords
PAGE, HIMEM, LOMEM

TRACE TR.

TRACE ON causes the interpreter to print executed line numbers when it encounters them.

TRACE X sets a limit on the size of line numbers which will be printed out. Only those line numbers less than X will appear. If you are careful and place all your subroutines at the end of the main program, you can display the main structure of the program without cluttering up the trace with the subroutines.

TRACE OFF turns trace off. TRACE is also turned off if an error is reported or you press <Esc>.

Line numbers are printed as the line is entered. For example,

```
10 FOR Z=0 TO 2:Q=Q*Z:NEXT
20 END
```

would trace as

```
[10] [20] >_
```

Whereas

```
10 FOR Z=0 TO 2
20   Q=Q*Z:NEXT
30 END
```

would trace as

```
[10] [20] [20] [20] [30] >_
```

And

```
10 FOR Z=0 TO 3
20 Q=Q*Z
30 NEXT
40 END
```

would trace as

```
[10] [20] [30] [20] [30] [20] [30] [40] >_
```

Syntax

```
TRACE ON|OFF|<l-num>
TRACE ON|OFF|(<numeric>)
```

Associated Keywords
None

TRUE

A function returning the value -1.

```
 10 flag=FALSE
....
100 IF answer$=correct$ flag=TRUE
....
150 IF flag PROC_got_it_right ELSE PROC_wrong
```

BBC BASIC (Z80) does not have true Boolean variables. Instead, numeric variables are used and their value is interpreted in a 'logical' manner. A value of 0 is interpreted as false and NOT FALSE (in other words, NOT 0 (= -1)) is interpreted as TRUE.

In practice, any value other than zero is considered TRUE. This can lead to confusion; see the keyword NOT for details.

See the Variables sub-section for more details on Boolean variables and the keyword AND for logical tests and their results.

Syntax

```
<n-var>=TRUE
```

Associated Keywords

FALSE

UNTIL

U.

The part of the REPEAT ... UNTIL structure which signifies its end.

You can use a REPEAT...UNTIL loop to repeat a set of program instructions until some condition is met.

If the condition associated with the UNTIL statement is never met, the loop will execute for ever. (At least, until <Esc> is pressed or some other error occurs.)

The following example will continually ask for a number and print its square. The only way to stop it is by pressing <Esc> or forcing a 'Too big' error.

```
10 z=1
20 REPEAT
30   INPUT "Enter a number " num
40   PRINT "The square of ";num;" is ";num*num
50 UNTIL z=0
```

Since the result of the test z=0 is ALWAYS FALSE, we can replace z=0 with FALSE. The program now becomes:

```
20 REPEAT
30   INPUT "Enter a number " num
40   PRINT "The square of ";num;" is ";num*num
50 UNTIL FALSE
```

This is a much neater way of unconditionally looping than using a GOTO statement. The program executes at least as fast and the section of program within the loop is highlighted by the indentation.

See the keyword REPEAT for more details on REPEAT...UNTIL loops. See the Variables sub-section for more details on Boolean variables and the keyword AND for logical tests and their results.

Syntax

```
UNTIL <t-cond>
```

Associated Keywords
REPEAT

USR

A function which allows a machine code routine to return a value directly.

USR calls the machine code subroutine whose start address is its argument. The processor's A, B, C, D, E, F, H and L registers are initialised to the least significant words of A%, B%, C%, D%, E%, F%, H% and L% respectively (see also CALL).

USR provides you with a way of calling a machine code routine which is designed to return one integer value. Parameters are passed via the processor's registers and the machine code routine returns a 32-bit integer result composed of the processor's HLH'L' registers.

```
X=USR(lift_down)
```

Unlike CALL, USR returns a result. Consequently, you must assign the result to a variable. It may help your understanding if you look upon CALL as the machine code equivalent to a PROCedure and USR as the equivalent to Function.

Syntax

```
<n-var>=USR(<numeric>)
```

Associated Keywords
CALL

VAL

A function which converts a character string representing a number into numeric form.

```
X=VAL(A$)
```

VAL makes the best sense it can of its argument. If the argument starts with numeric characters (with or without a preceding sign), VAL will work from left to right until it meets a non numeric character. It will then 'give up' and return what it has got so far. If it can't make any sense of its argument, it returns zero.

For example,

```
PRINT VAL("-123.45.67ABC")
```

would print

```
-123.45
```

And

```
PRINT VAL("A+123.45")
```

would print

```
0
```

VAL will NOT work with hexadecimal numbers. You must use EVAL to convert hexadecimal number strings.

Syntax

```
<n-var>=VAL(<str>)
```

Associated Keywords
STR$, EVAL

VDU **V.**

A statement which takes a list of numeric arguments and sends their least-significant bytes as characters to the current 'output stream' (see *OPT).

A 16-bit value can be sent if the value is followed by a '**;**'. It is sent as a pair of characters, least significant byte first.

It is very important that the correct number of bytes be sent, as appropriate for the particular VDU command. If too few are sent, the 'remaining' bytes will be taken from subsequent VDU or PRINT statements, sometimes with (at first sight) extremely strange results. In ***BBC BASIC version 5 or later only*** this problem can be circumvented by terminating the VDU statement with the '**|**' character which ensures that sufficient bytes are sent.

```
VDU 8,8 :REM cursor left two places.
VDU &0A0D;&0A0D; :REM CRLF twice
VDU 23,1,0| : REM disable cursor
```

The bytes sent using the VDU statement do not contribute to the value of COUNT, but may well change POS and VPOS.

You can use VDU to send characters direct to the current output stream without having to use a PRINT statement. It offers a convenient way of sending a number of control characters to the console or printer.

Syntax

```
VDU <numeric>{,|;<numeric>}[;]
```

Associated Keywords
CHR$

VPOS

A function returning the vertical cursor position. The top of the screen is line 0.

```
Y=VPOS
```

You can use VPOS in conjunction with POS to return to the present position on the screen after printing a message somewhere else. The example below is a procedure for printing a 'status' message at line 23. The cursor is returned to its previous position after the message has been printed.

```
1000 DEF PROC_message(message$)
1010 LOCAL x,y
1020 x=POS
1030 y=VPOS
1040 PRINT TAB(0,23) CHR$(7);message$;
1050 PRINT TAB(x,y);
1060 ENDPROC
```

Syntax

```
<n-var>=VPOS
```

Associated Keywords

POS

WAIT

(BBC BASIC version 5 or later only, not implemented in the generic CP/M edition)

A statement causing the program to pause for the specified number of centiseconds (hundredths of a second), or to wait for a Vertical Sync (retrace) event.

```
WAIT 100
```

This causes the program to pause for approximately one second.

WAIT 0 optionally results in only a short pause (typically about 1 millisecond), but does give other programs an opportunity to run; it can be useful in preventing your program consuming 100% of the processor's time.

WAIT without a parameter optionally waits for a Vertical Sync (retrace) event, if this is possible.

Syntax

```
WAIT [<numeric>]
```

Associated Keywords
INKEY, TIME

WHEN

(BBC BASIC version 5 or later only)

A keyword which is part of the CASE... ENDCASE clause. WHEN precedes the value (or values) against which the CASE variable is tested. WHEN must be the first item on the program line.

```
CASE die% OF
 WHEN 1,2,3 : bet$ = "lose"
 WHEN 4,5,6 : bet$ = "win"
 OTHERWISE bet$ = "cheat"
ENDCASE
```

Syntax

```
WHEN <numeric>{,<numeric>} : {<stmt>}
WHEN <string>{,<string>} : {<stmt>}
```

Associated Keywords
CASE, OF, OTHERWISE

WHILE

(BBC BASIC version 5 or later only)

A statement which is the starting point of a WHILE...ENDWHILE loop.

The purpose of a WHILE...ENDWHILE loop is to make BBC BASIC repeat a set number of instructions while some condition is satisfied. The difference between a WHILE...ENDWHILE loop and a REPEAT...UNTIL loop is that the instructions within a REPEAT...UNTIL loop are always executed *at least once* (the test for completion is performed at the *end* of the loop) whereas the instructions within a WHILE...ENDWHILE loop are not executed at all if the condition is initially FALSE (the test is performed at the *start* of the loop).

```
WHILE LEFT$(A$,1)=" "
  A$=MID$(A$,2) : REM Remove leading spaces
ENDWHILE
```

You must not exit a WHILE...ENDWHILE loop with a GOTO; you can force a premature end to the loop with the EXIT WHILE statement.

WHILE...ENDWHILE loops may be nested.

Syntax

```
WHILE <t-cond>
```

Associated Keywords
ENDWHILE, EXIT, REPEAT

WIDTH W.

A statement controlling output overall line width.

```
WIDTH 80
```

If the specified width is zero (the initial value) the interpreter will not attempt to control the overall line width.

WIDTH n will cause the interpreter to force a new line after n MOD 256 characters have been printed; it also affects output to the printer.

(BBC BASIC version 5 or later only)

WIDTH can also be used as a function. It returns the current line width.

Syntax

```
WIDTH <numeric>
<n-var> = WIDTH
```

Associated Keywords

COUNT

5. BBC BASIC (Z80) Files

Introduction

These notes start with some basic information on files, and then go on to discuss program file manipulation, simple serial files, random files and, finally, indexed files. The commands and functions used are explained and followed by examples.

If you are new to BBC BASIC (Z80), or you are having trouble with files you might find these notes useful. Some of the concepts and procedures described are quite complicated and require an understanding of file structures. If you have trouble understanding these parts, don't worry. Try the examples and write some programs for yourself and then go back and read the notes again. As you become more comfortable with the fundamentals, the complicated bits become easier.

The Structure of Files

If you understand the way files work, skip the next two paragraphs. If you understand random and indexed files, skip the following two paragraphs as well.

Basics

Many people are confused by the jargon that is often used to describe the process of storing and retrieving information. This is unfortunate, because the individual elements are very simple, and the most complicated procedures are only a collection of simple bits and pieces.

All computers can store and retrieve information from a non-volatile medium. (In other words, you don't lose the information when the power gets turned off.) Audio cassettes were used for small microcomputers, diskettes for medium sized systems and magnetic tape and large disks for big machines. In order to be able to find the information you want, the information has to be organised in some way. All the information on one general subject is gathered together into a FILE. Within the file, information on individual items is grouped together into RECORDS.

Serial (Sequential) Files

Look upon the storage device or directory as a drawer in a filing cabinet. The drawer is full of folders called FILES and each file holds a number of enclosures called RECORDS. Sometimes the files are in order in the drawer, sometimes not. If you want a particular file, you start at the beginning of the drawer and search your way through until you find the file you want. Then you search your way through the records in the file until you find the record you want.

This is very similar to the way a cassette is searched for a particular file. You put the cassette in the recorder, type in the name of the file you want and push play. You then go and make a cup of tea whilst the computer reads through all the files until it comes to the one you want. Because the cassette is read serially from start to end, it's very difficult to do it any other way.

There are a number of ways to overcome this problem. We will consider the two simplest; random access (or relative) files and indexed files.

Random Access Files

The easiest way to find the record you want is to identify each record with a number, like an account number. You can then ask for, say, the 23rd record. This is similar to turning to page 23 in the account book. This works very well at first. Every time you get a new customer you start a new page. Most of the pages have a lot of empty space, but you must have the same amount of space available for each account, otherwise your numbering system won't work. So, even at the start, there are a lot of gaps.

What happens when you close an account? You can't tear out the page because that would upset the numbering system. All you can do is draw a line through it - in effect, turn it into a blank page. Before long, quite a number of pages will be 'blank' and a growing proportion of your book is wasted.

With other forms of 'numbering', say by the letters of the alphabet, you could still not guarantee to fill all the pages. You would have to provide room for the Zs, but you may never get one. When you started entering data, most of the pages would be blank and the book would only gradually fill up.

Indexed Files

Suppose we want to hold our address book on the computer. We need a number of records each holding the name, address, telephone number, etc of one person. In our address book, we have one or two pages per letter of the alphabet and a number of 'slots' on each page. With this arrangement, the names are in alphabetical order of their first letter. This is very similar to the way the accounts book was organised except that we don't know the page number for each name.

If we had an index at the front of the book we could scan the index for the name and then turn to the appropriate page. We would still be wasting a lot of space because some names, addresses etc are longer than others and our 'slots' must be large enough to hold the longest.

Suppose we numbered all the character positions in the book and we could easily move to any character position. We could write all the names, addresses, etc, one after the other and our index would tell us the character position for the start of each name and address. There would be no wasted space and we would still be able to turn directly to the required name.

What would happen when we wanted to cancel an entry? We would just delete the name from the index. The entry would stay in its original place in the book, but we would never refer to it. Similarly, if someone changed their address, we would just write the name and the new address immediately after the last entry in the book and change the start position in the index. Every couple of years we would rewrite the address book, leaving out those items not referenced in the index and up-date the index (or write another one).

This is not a practical way to run a paper and pencil address book because it's not possible to turn directly to the 3423rd character in a book, and the saving in space would not be worth the tedium involved. However, with BBC BASIC you can turn to a particular character in a file and the tedium only takes a couple of seconds, so it's well worth doing.

Files in BBC BASIC (Z80)

Introduction

Conventional serial file procedures are little different from file procedures for cassette based computers. With serial files the records need only be as large as the data to be stored and there are no empty records. (The data item FRED only occupies 4 bytes whereas ERMINTRUDE occupies 10 bytes.) Consequently serial files are the most space efficient way to hold data on a disk (or any other storage media).

Serial files cannot be used to access particular records from within the file quickly and easily. In order to do this with the minimum access time, random access files are necessary. However, a random file generally occupies more space than a serial file holding the same amount of data because the records must be a fixed length and some of the records will be empty.

Most versions of BASIC only offer serial and random files, but because of the way that files are handled by BBC BASIC, it is possible to construct indexed, and even linked, files as well. Indexed files take a little longer to access than random files and it is necessary to have a separate index file, but they are generally the best space/speed compromise for files holding a large amount of data.

How Data is Read/Written

As far as the programmer is concerned, data can be written to and read from a file a data item or a character (byte) at a time. In fact, there is a buffer between the program and the Operating System, but this need only concern you when you are organising your program for maximum storage efficiency.
Because of the character by character action of the write/read process, it is possible (in fact, necessary) to keep track of your position within the file. BBC BASIC does this for you automatically and provides a pointer PTR (a pseudo-variable) which holds the position of the NEXT character (byte) to be written/read. Every time a character is written/read PTR is incremented by 1, but it is possible to set PTR to any number you like. This ability to 'jump around' the file enables you to construct both random (relative) and indexed files.

BBC BASIC provides the facility for completely free-format binary data files. Any file which can be read by the computer, from any source and in any data format, can be processed using the BGET, BPUT and PTR functions.

How Data is Stored

Numeric Data

In order to make the most efficient use of storage space and to preserve accuracy, numerics are stored in a data file in binary format, not as strings of characters. To prevent confusion when numerics are being read from a file, both integers and reals occupy 5 bytes (40 bits). If they were stored as character strings they could occupy up to 10 bytes. For compatibility with other BASICs, you can store numerics as strings by using the STR$ function.

How Strings are Stored

Strings are stored in a data file as the ASCII bytes of the string followed by a carriage-return. If you need a line feed as well, it's no problem to add it using the Byte-Put function BPUT#. Similarly, extraneous characters included in files produced by other programs can be read and, if necessary, discarded using BGET#.

How Files are Referred To

We refer to a file by its name. Unfortunately, this is too complicated for the Operating System. Consequently, the only time BBC BASIC refers to a file by its name is when it opens the file. From then on, it refers to the file by the number it allocated to it when it was opened. This number is called the 'file handle'.

File Buffering

Logically, BBC BASIC (Z80) transfers data to and from files one byte at a time. sometimes the Filing System does not handle single byte data transfer directly so BBC BASIC (Z80) buffers the data into blocks; this is transparent to the user.

File Commands

Introduction

The commands and statements used in file manipulation are described below. They are not in alphabetical order, but in the order you are likely to want to use them. Whilst these notes repeat much of the material covered in the Statements and Functions section, additional information has been added and they are presented in a more readable order.

Filenames

Please refer to your Operating System documentation for a full explanation of drive, directory and file names. The explanation below is only intended as a brief reference guide.

The CP/M operating system allows a composite file name in the following format: DRIVENAME:FILENAME.EXTension

The drivename is a single letter followed by a colon and denotes the drive on which the file will be found or created.

The file name can be up to 8 characters long, and the extension up to three characters. Whenever a file name without an extension is given, BBC BASIC (Z80) will append .BBC as the default extension.

Organisation of Examples

Simple examples are given throughout this section with the explanation of the various commands. The following sections contain examples of complete programs for serial files, random files and, finally, indexed files. If you have problems understanding the action of any of the commands you may find the examples helpful. The best way to learn is to do - so have a go.

Program File Manipulation

SAVE

Save the current program to a file, in internal (tokenised) format. The filename can be a variable or a string.

```
SAVE filename
```

```
SAVE "FRED"
```

```
A$="COMPOUND"
SAVE A$
```

The first example will save the program to a file named FRED.BBC. The second will save COMPOUND.BBC.

You can specify a drivename as well as the file name. The following example will save the current program to a file called TEST.BBC on drive D:

```
SAVE "D:TEST"
```

LOAD

Load the program 'filename' into the program area. The old program is deleted (as if a NEW command had been given prior to the LOAD) and all the dynamic variables are cleared. The program must be in tokenised format. If no filename extension is given, .BBC is assumed.

```
LOAD filename
```

```
LOAD "FRED"
```

```
A$="HEATING"
LOAD A$
```

As with SAVE, you can specify a drive name. The example below loads the program saved previously as an example of the SAVE command.

```
LOAD "D:TEST"
```

CHAIN

LOAD and RUN the program 'filename'. All the dynamic variables are cleared. The program must be in tokenised format.

```
CHAIN filename
```

```
CHAIN "GAME1"
```

```
A$="PART2"
CHAIN A$
```

As with SAVE and LOAD, you can specify a drive name.

MERGE

There is no MERGE command, however there are two ways of merging BBC BASIC (Z80) programs.

Using MERGE.BBC

MERGE.BBC is a BBC BASIC (Z80) program which combines two program files into a third program file. It asks you for the names of the two input files and the name of the output file. If the same line number exists in both files, the program line from the second file will be included in the output file immediately after the line from the first file (the number of both lines will be the same). This may confuse you, but it won't confuse your computer; providing the program is still syntactically correct, it will run. If you want to clean up the mess, renumber the resulting program and delete the lines you don't want.

Using *LOAD

You can also use *LOAD to perform a quick (and somewhat 'dirty') merge of two files. If you don't want to get disconcerting results, you should ensure that the second program uses larger line numbers than the first program.
Load the first program (with the lower line numbers) in the normal way. Then, find out the top address of the program less 3 by typing

```
PRINT ~TOP-3<Enter>
```

This will print the address in hex (nnnn) at which the first byte of the second program file must be loaded. Finally, load the second program by typing

```
*LOAD "PROG2" nnnn<Enter>
OLD<Enter>
```

**ERA*

Delete the file 'filename'. Since variables are not allowed as arguments to * commands, the filename must be a constant.

```
*ERA filename
```

```
*ERA FRED
*ERA PHONE.DTA
```

To delete a file whose name is known only at run-time, use the OSCLI command. It's a bit clumsy, but a lot better than the original specification for BBC BASIC allowed. This time all of the command, including the ERA, must be supplied as the argument for the OSCLI command. You can use OSCLI for erasing a file whose name is a constant, but you must include all of the command line - in quotes this time.

```
fname$="FRED"
OSCLI "ERA "+fname$

fname$="PHONE.DTA"
command$="ERA "
OSCLI command$+fname$

OSCLI "ERA FRED"
```

You can include a drive name in both the *ERA and *OSCLI command formats.

You should not attempt to erase an open file.

*REN

Rename 'file1' to be called 'file2'. The syntax is similar to the normal CP/M command except that the extension defaults to .BBC.

```
*REN file2=file1

*REN FRED2=FRED1

*REN PHONE.DTA=PHONE
```

Once again, if you want to rename files whose names are only known at run-time, you must use the OSCLI command.

```
fname1$="FRED1"
fname2$="FRED2"
OSCLI "REN "+fname2$+"="+fname1$
```

You should not attempt to rename an open file.

*DIR *.

List the directory. The default drive is the currently logged drive and the default extension is .BBC. The format is the same as the normal CP/M command.

*DIR	List *.BBC files on the current drive.
.B:.DTA	List *.DTA files on drive B.

Data Files

Introduction

The statements and functions used for data files are:

OPENIN
OPENUP
OPENOUT
EXT#
PTR#
INPUT# BGET#
PRINT# BPUT#
CLOSE# END
EOF#

Opening Files

You cannot use a file until you have told the system it exists. In order to do this you must OPEN the file for use. Other versions of BASIC allow you to choose the file number. In order to improve efficiency, BBC BASIC (Z80) chooses the number for you.

When you open the file, a file handle (an integer number) is returned by the interpreter and you will need to store it for future use. (The open commands are, in fact, functions which open the appropriate file and return its file handle.)

You use the file handle for all subsequent access to the file. (With the exception of the STAR commands outlined previously.)

If the system has been unable to open the file, the handle returned will be 0. This will occur if you try to open a non-existent file in the input mode (OPENIN or OPENUP).

File Opening Functions

The three functions which open files are OPENIN, OPENUP and OPENOUT. OPENOUT should be used to create new files, or overwrite old ones. OPENIN should be used for input only and OPENUP should be used for input/output.

OPENOUT

Open the file 'filename' for output and return the file handle allocated. The use of OPENOUT destroys the contents of the file if it previously existed. (The directory is updated with the length of the new file you have just written when you close the file.)

```
OPENOUT filename
file_num=OPENOUT "PHONENUMS"
```

You always need to store the file handle because it must be used for all the other file commands and functions. If you choose a variable with the same name as the file, you will make programs which use a number of files easier to understand.

```
phonenums=OPENOUT "PHONENUMS"
opfile=OPENOUT opfile$
```

On a networked system, OPENOUT opens the file in 'compatibility' mode and the file is not available to any other user. If you wish to create a new file which can be read, concurrently by other users, you should open it with OPENOUT, immediately close it and re-open it with OPENUP.

OPENIN

Open the file 'filename' for input only. Unlike the Z80 version of BBC BASIC, you cannot write to a file opened with OPENIN.

```
OPENIN filename
address=OPENIN "ADDRESS"
check_file=OPENIN check_file$
```

You will be unable to open for input (file handle returned = 0) if the file does not already exist.

OPENUP

Open the file 'filename' for update (input or output) without destroying the contents of the file. The file may be read from or written to. When the file is closed, the directory is updated to show the maximum used length of the file. None of the previously written data is lost unless it has been overwritten. Consequently, you would use OPENUP for reading serial and random files, adding to the end of serial files or writing to random files.

```
OPENUP filename
address=OPENUP "ADDRESS"
check_file=OPENUP check_file$
```

You will be unable to open for update (file handle returned = 0) if the file does not already exist.

On a networked system, OPENUP opens a file in the 'read-write, deny write' mode. A file may be opened **once** with OPENUP and any number of times by any number of users with OPENIN.

CLOSE#

Close the file opened as 'fnum'. CLOSE#0, END or 'dropping off the end' of a program will close all files.

```
CLOSE#fnum
```

When a file is closed its file buffer (if it has one) will be flushed to CP/M before the file is closed.

INPUT#

Read from the file opened as 'fnum' into the variable 'var'. Several variables can be read using the same INPUT# statement.

```
INPUT#fnum,var
data=OPENIN "DATA"
:
INPUT#data,name$,age,height,sex$
:
:
```

READ# can be used as an alternative to INPUT#

PRINT#

Write the variable 'var' to the file opened as 'fnum'. Several variables can be written using the same PRINT# statement.

```
PRINT#fnum,var
```

String variables are written as the character bytes in the string plus a carriage-return. Numeric variables are written as 5 bytes of binary data.

```
data=OPENOUT "DATA"
:
:
PRINT#data,name$,age,height,sex$
:
:
```

EXT#

Return the total length of the file opened as 'fnum'.

```
EXT#fnum
```

In the case of a sparse random-access file the value returned is the length of the file to the last byte actually written to the file. Although much of the file may well be unused, writing this 'last byte' reserves physical space for a file of this length. Thus it is possible to write a single byte to a file and get a 'Disk full' error.

PTR#

A pseudo-variable which points to the position within the file from where the next byte to be read will be taken or where the next byte to be written will be put.

```
PTR#fnum
```

When the file is OPENED, PTR# is set to zero. However, you can set PTR# to any value you like. (Even beyond the end of the file - so take care).

Reading or writing, using INPUT# and PRINT#, (and BGET# and BPUT# - explained later), takes place at the current position of the pointer. The pointer is automatically updated following a read or write operation.

A file opened with OPENUP may be extended by setting PTR# to its end (PTR# = EXT#) and then writing the new data to it. You must remember to CLOSE such a file in order to update its directory entry with its new length. A couple of examples of this are included in the sections on serial and indexed files.

Using a 'PTR#fnum=' statement will flush the appropriate BBC BASIC (Z80) file buffer to CP/M.

EOF#

A function which will return -1 (TRUE) if the data file whose file handle is the argument is at (or beyond) its end. In other words, when PTR# points beyond the current end of the file.

```
eof=EOF#fnum
```

Attempting to read beyond the current end of file will not give rise to an error. Either zero or a null string will be returned depending on the type of variable read.

EOF# is only really of use when dealing with serial (sequential) files. It indicates that PTR# is greater than the recorded length of the file (found by using EXT#).

When reading a serial file, EOF# would go true when the last byte of the file had been read.

EOF# is only true if PTR# is set beyond the last byte written to in the file. It will NOT be true if an attempt has been made to read from an empty area of a sparse random access file. Reading from an empty area of a sparse file will return garbage. Because of this, it is difficult to tell which records of an uninitialised random access file have had data written to them and which are empty. These files need to be initialised and the unused records marked as empty.

Writing to a byte beyond the current end of file updates the file length immediately, whether the record is physically written to the storage device at that time or not.

BGET#

A function which reads a byte of data from the file opened as 'fnum', from the position pointed to by PTR#fnum. PTR#fnum is incremented by 1 following the read. A positive integer between 0 and 255 is returned (as you might expect). This can be converted into a string variable using the CHR$ function.

```
BGET#fnum
```

```
byte=BGET#fnum
char$=CHR$(byte)
```

or, more expediently

```
char$=CHR$(BGET#fnum)
```

BPUT#

Write the least significant byte of the variable 'var' to the file opened as 'fnum', at the position pointed to by PTR#fnum. PTR#fnum is incremented by 1 following the write.

```
BPUT#fnum,var
```

```
BPUT#fnum,&1B
BPUT#fnum,house_num
BPUT#fnum,ASC "E"
```

Serial Files

Introduction

The section on serial files is split into three parts. The first deals with character data files. These are the simplest type of files to use and the examples are correspondingly short. The second part looks at mixed numeric/character data files. The final part describes conversion between BBC BASIC (Z80) format files and the file formats required/produced by other systems.

Character Data Files

The first three examples are programs to write data in character format to a serial file and to read the data back. All the data is in character format and, since the files will not be read by other versions of BASIC, no extra control characters have been added.
You may notice that we have cheated a little in that a procedure is called to close the files and end the program without returning. This saves using a GOTO, but leaves the return address on the stack. However, ending a program clears the stack and no harm is done. You should not use this sort of trick anywhere else in a program. If you do you will quickly use up memory.

Ex 1 - Writing Serial Character Data

```
 10 REM F-WSER1
 20 :
 30 REM WRITING TO A SERIAL CHARACTER DATA FILE
 40 :
 50 REM This program opens a data file and writes
 60 REM serial character data to it.  The use of
 70 REM OPENOUT ensures that, even if the file
 80 REM existed before, it is cleared before
 90 REM being written to.
100 :
110 phonenos=OPENOUT "PHONENOS"
120 PRINT "File Name PHONENOS Opened as Handle ";phonenos
130 PRINT
140 REPEAT
150   INPUT "Name ? " name$
160   IF name$="" THEN PROC_end
170   INPUT "Phone Number ? " phone$
180   PRINT
```

```
190   PRINT#phonenos,name$,phone$
200 UNTIL FALSE
210 :
220 DEF PROC_end
230 CLOSE#phonenos
240 END
```

Ex 2 - Reading Serial Character Data

```
 10 REM F-RSER1
 20 :
 30 REM EXAMPLE OF READING A SERIAL CHARACTER FILE
 40 :
 50 REM This program opens a previously written
 60 REM serial file and reads it.
 70 :
 80 :
 90 phonenos=OPENIN "PHONENOS"
100 PRINT "File Name PHONENOS Opened as Handle ";phonenos
110 PRINT
120 REPEAT
130   INPUT#phonenos,name$,phone$
140   PRINT name$,phone$
150 UNTIL EOF#phonenos
160 :
170 CLOSE#phonenos
180 END
```

Ex 3 - Writing 'AT END' of Character Files

The next example extends the write program from Example 1. This new program opens the file, sets PTR# to the end (line 380) and then adds data to it. A procedure is used to open the file. This has the advantage of making the program more understandable by putting the detailed 'open at end' coding out of the main flow of the program.

```
 10 REM F-WESER1
 20 :
 30 REM EXAMPLE OF WRITING TO THE END OF A SERIAL DATA FILE
 40 :
 50 REM The program opens a file and sets PTR
 60 REM to the end before writing data to it.
 70 :
```

```
 80 REM A function is used to open the file.
 90 :
100 :
110 phonenos=FN_openend("PHONENOS")
120 PRINT "File Name PHONENOS Opened as Handle ";phonenos
130 PRINT
140 REPEAT
150   INPUT "Name ? " name$
160   IF name$="" THEN PROC_end
170   INPUT "Phone Number ? " phone$
180   PRINT
190   PRINT#phonenos,name$,phone$
200 UNTIL FALSE
210 :
220 DEF PROC_end
230 CLOSE#phonenos
240 END
250 :
260 :
270 REM Open the file 'AT END'.
280 :
290 REM If the file does not already exist, it
300 REM is created with OPENOUT.  PTR# is left
310 REM at zero and the handle is returned.  If
320 REM the file exists, PTR# is set to the end
330 REM and the file handle returned.
340 DEF FN_openend(name$)
350 LOCAL fnum
360 fnum=OPENUP(name$)
370 IF fnum=0 THEN fnum=OPENOUT(name$): =fnum
380 PTR#fnum=EXT#fnum
390 =fnum
```

Mixed Numeric/Character Data Files

The second three examples are also programs which write data to a file and read it back, but this time the data is mixed. They are simply extensions of the previous examples which illustrate the handling of mixed data.

Ex 4 - Writing a Mixed Data File

```
10 REM F-WSER2
20 :
30 REM EXAMPLE OF WRITING TO A MIXED NUMERIC/CHAR DATA FILE
40 :
50 REM This program opens a file and writes
```

```
 60 REM numeric and char data to it.  The use
 70 REM of OPENOUT ensures that, even if the
 80 REM file exists, it is cleared before
 90 REM being written to.  Functions
100 REM are used to accept and validate
110 REM the data before writing it to the file.
120 :
130 :
140 stats=OPENOUT("STATS")
150 PRINT "File Name STATS Opened as Handle ";stats
160 PRINT
170 REPEAT
180   name$=FN_name
190   IF name$="" THEN PROC_end
200   age=FN_age
210   height=FN_height
220   sex$=FN_sex
230   PRINT
240   PRINT#stats,name$,age,height,sex$
250 UNTIL FALSE
260 :
270 DEF PROC_end
280 PRINT "The file is ";EXT#stats;" bytes long"
290 CLOSE#stats
300 END
310 :
320 :
330 REM Accept a name from the keyboard and make
340 REM sure it consists only of spaces and
350 REM upper or lower case characters. Leading
360 REM spaces are ignored on input.
370 :
380 DEF FN_name
390 LOCAL name$,FLAG,n
400 REPEAT
410   FLAG=TRUE
420   INPUT "Name ? " name$
430   IF name$="" THEN 490
440   FOR I=1 TO LEN(name$)
450     n=ASC(MID$(name$,I,1))
460     IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123) THEN FLAG=FALSE
470   NEXT
480   IF NOT FLAG THEN PRINT "No funny characters please !!!"
490 UNTIL FLAG
500 =name$
510 :
520 :
530 REM Accept the age from the keyboard and
```

```
540 REM round to one place of decimals.  Ages
550 REM of 0 or less, or 150 or more are
560 REM considere dto be in error.
570 DEF FN_age
580 LOCAL age
590 REPEAT
600   INPUT "What age ? " age
610   IF age<=0 OR age >=150 THEN PRINT "No impossible ages please !!!"
620 UNTIL age>0 AND age<150
630 =INT(age*10+.5)/10
640 :
650 :
660 REM Accept the height in centimetres from
670 REM the keyboard and round to an integer.
680 REM Heights of 50 or less and 230 or more
690 REM are considered to be in error.
700 DEF FN_height
710 LOCAL height
720 REPEAT
730   INPUT "Height in centimetres ? " height
740   IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
750 UNTIL height>50 AND height<230
760 =INT(height+.5)
770 :
780 :
790 REM Accept the sex from the keyboard.  Only
800 REM words beginning with upper or lower case
810 REM M or F are OK.  The returned string is
820 REM truncated to 1 character.
830 DEF FN_sex
840 LOCAL sex$,FLAG
850 REPEAT
860   FLAG=TRUE
870   INPUT "Male or Female - M or F ? " sex$
880   IF sex$<>"" THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
890   IF sex$<>"M" AND sex$<>"F" THEN FLAG=FALSE
900   IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
910 UNTIL FLAG
920 =sex$
```

Ex 5 - Reading a Mixed Data File

```
10 REM F-RSER2
20 :
30 REM EXAMPLE OF READING FROM A MIXED NUMERIC/CHAR DATA FILE
40 :
50 REM This program opens a file and reads
60 REM numeric and character data from it.
```

```
 70 :
 80 :
 90 stats=OPENIN("STATS")
100 PRINT "File Name STATS Opened as Handle ";stats
110 PRINT
120 REPEAT
130   INPUT#stats,name$,age,height,sex$
140   PRINT "Name ";name$
150   PRINT "Age ";age
160   PRINT "Height in centimetres ";height
170   IF sex$="M" THEN PRINT "Male" ELSE PRINT "Female"
180   PRINT
190 UNTIL EOF#stats
200 :
210 CLOSE#stats
220 END
```

Ex 6 - Writing 'AT END' of Mixed Files

This example is similar to Example 3, but for a mixed data file.

```
 10 REM F-WESER2
 20 :
 30 REM EXAMPLE OF WRITING AT THE END OF A
 40 REM MIXED NUMERIC/CHAR DATA FILE
 50 :
 60 REM This program opens a file, sets PTR
 70 REM to its end and then writes numeric and
 80 REM character data to it.
 90 :
100 REM Functions are used to accept and
110 REM validate the data before writing it to
120 REM the file.
130 :
140 stats=FN_open("STATS")
150 PRINT "File Name STATS Opened as Handle ";stats
160 PRINT
170 REPEAT
180   name$=FN_name
190   IF name$="" THEN PROC_end
200   age=FN_age
210   height=FN_height
220   sex$=FN_sex
230   PRINT
240   PRINT#stats,name$,age,height,sex$
250 UNTIL FALSE
260 :
```

```
270 DEF PROC_end
280 PRINT "The file is ";EXT#stats;" bytes long"
290 CLOSE#stats
300 END
310 :
320 :
330 REM Open the file.  If it exists, set PTR#
340 REM to vthe end and return the handle.  If
350 REM it does not exist, open it, leave PTR#
360 REM as it is and return the file handle.
370 DEF FN_open(name$)
380 LOCAL fnum
390 fnum=OPENUP(name$)
400 IF fnum=0 THEN fnum=OPENOUT(name$): =fnum
410 PTR#fnum=EXT#fnum
420 =fnum
430 :
440 :
450 REM Accept a name from the keyboard and make
460 REM sure it consists of spaces and upper or
470 REM lower case characters.  Leading spaces
480 REM are automatically ignored on input.
490 DEF FN_name
500 LOCAL name$,FLAG,n
510 REPEAT
520   FLAG=TRUE
530   INPUT "Name ? " name$
540   IF name$="" THEN 600
550   FOR I=1 TO LEN(name$)
560     n=ASC(MID$(name$,I,1))
570     IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123) THEN FLAG=FALSE
580  NEXT
590   IF NOT FLAG THEN PRINT "No funny characters please !!!"
600 UNTIL FLAG
610 =name$
620 :
630 :
640 REM Accept the age from the keyboard and
650 REM round to one place of decimals. Ages of
660 REM 0 or less or 150 or more are in error.
670 :
680 DEF FN_age
690 LOCAL age
700 REPEAT
710   INPUT "What age ? " age
720   IF age<=0 OR age >=150 THEN PRINT "No impossible ages please !!!"
730 UNTIL age>0 AND age<150
740 =INT(age*10+.5)/10
```

```
 750 :
 760 :
 770 REM Accept the height in centimetres from
 780 REM the keyboard and round to an integer.
 790 REM Heights of 50 or less or 230 or more
 800 REM are in error.
 810 DEF FN_height
 820 LOCAL height
 830 REPEAT
 840   INPUT "Height in centimetres ? " height
 850   IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
 860 UNTIL height>50 AND height<230
 870 =INT(height+.5)
 880 :
 890 :
 900 REM Accept the sex from the keyboard.  Only
 910 REM words beginning with upper or lower
 920 REM case M or F are valid.  The returned
 930 REM string is truncated to 1 character.
 940 DEF FN_sex
 950 LOCAL sex$,FLAG
 960 REPEAT
 970   FLAG=TRUE
 980   INPUT "Male or Female - M or F ? " sex$
 990   IF sex$<>"" THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
1000   IF sex$<>"M" AND sex$<>"F" THEN FLAG=FALSE
1010   IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
1020 UNTIL FLAG
1030 =sex$
```

Compatible Data Files

The next example tackles the problem of writing files which will be compatible with other versions of BASIC. The most common format for serial files is as follows:

- Data is written to the file as ASCII characters.
- Data items are separated by commas.
- Records are terminated by the two characters CR and LF.
- The file is terminated by a Control Z (&1A).

The example program accepts data from the keyboard and writes it to a file in the above format.

Ex 7 - Writing a Compatible Data File

```
 10 REM F-WSTD
 20 :
 30 REM EXAMPLE OF WRITING A COMPATIBLE FILE
 40 :
 50 REM This program opens a file and writes
 60 REM numeric and character data to it in a
 70 REM compatible format.  Numerics are changed
 80 REM to strings before they are written and
 90 REM the data items are separated by commas.
100 REM Each record is terminated by CR LF and
110 REM the file is terminated by a Control Z.
120 :
130 REM Functions are used to accept and
140 REM validate the data before writing it to
150 REM the file.
160 :
170 record$=STRING$(100," "): REM Reserve room for the longest
180 name$=STRING$(20," "): REM record necessary.
190 : REM It saves on string space.
200 compat=OPENOUT("COMPAT")
210 PRINT "File Name COMPAT Opened as Handle ";compat
220 PRINT
230 REPEAT
240   name$=FN_name
250   IF name$="" THEN PROC_end
260   age=FN_age
270   height=FN_height
280   sex$=FN_sex
290   PRINT
300   record$=name$+","+STR$(age)+","+STR$(height)+","+sex$
310   PRINT#compat,record$
320   BPUT#compat,&0A
330 UNTIL FALSE
340 :
350 DEF PROC_end
360 BPUT#compat,&1A
370 CLOSE#compat
380 END
390 :
400 :
410 REM Accept a name from the keyboard and make
420 REM sure it consists only of spaces and
430 REM upper or lower case characters. Leading
440 REM spaces are ignored on input.
450 :
460 DEF FN_name
```

```
470 LOCAL name$,FLAG,n
480 REPEAT
490   FLAG=TRUE
500   INPUT "Name ? " name$
510   IF name$="" THEN 570
520   FOR I=1 TO LEN(name$)
530     n=ASC(MID$(name$,I,1))
540     IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123) THEN FLAG=TRUE
550   NEXT
560   IF NOT FLAG THEN PRINT "No funny characters please !!!"
570 UNTIL FLAG
580 =name$
590 :
600 :
610 REM Accept the age from the keyboard and
620 REM round to one place of decimals.  Ages
630 REM of 0 or less or 150 or more are
640 REM considered to be in error.
650 DEF FN_age
660 LOCAL age
670 REPEAT
680   INPUT "What age ? " age
690   IF age<=0 OR age >=150 THEN PRINT "No impossible ages please !!!"
700 UNTIL age>0 AND age<150
710 =INT(age*10+.5)/10
720 :
730 :
740 REM Accept the height in centimetres from
750 REM the keyboard and round to an integer.
760 REM Heights of 50 or less and 230 or more
770 REM are considered to be in error.
780 DEF FN_height
790 LOCAL height
800 REPEAT
810   INPUT "Height in centimetres ? " height
820   IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
830 UNTIL height>50 AND height<230
840 =INT(height+.5)
850 :
860 :
870 REM Accept the sex from the keyboard. Only
880 REM words beginning with upper or lower
890 REM case M or F are valid.  The returned
900 REM string is truncated to 1 character.
910 DEF FN_sex
920 LOCAL sex$,FLAG
930 REPEAT
940   FLAG=TRUE
```

```
 950  INPUT "Male or Female - M or F ? " sex$
 960  IF sex$<>"" THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
 970  IF sex$<>"M" AND sex$<>"F" THEN FLAG=FALSE
 980  IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
 990 UNTIL FLAG
1000 =sex$
```

Ex 8 - Reading a Compatible Data File

The last example in this section reads a file written in the above format and strips off the extraneous characters. The file is read character by character and the appropriate action taken. This is a simple example of how BBC BASIC (Z80) can be used to manipulate any CP/M-80 file by processing it on a character-by-character basis.

```
 10 REM F-RSTD
 20 :
 30 REM EXAMPLE OF READING A COMPATIBLE FILE
 40 :
 50 REM This program opens a data file and reads
 60 REM numeric and character data from it.  The
 70 REM data is read a byte at a time and the
 80 REM appropriate action taken depending on
 90 REM whether it is a character, a comma, or
100 REM a control char.
110 compat=OPENUP("COMPAT")
120 PRINT "File Name COMPAT Opened as Handle ";compat
130 PRINT
140 REPEAT
150  name$=FN_read
160  PRINT "Name ";name$
170  age=VAL(FN_read)
180  PRINT "Age ";age
190  height=VAL(FN_read)
200  PRINT "Height in centimetres ";height
210  sex$=FN_read
220  IF sex$="M" THEN PRINT "Male" ELSE PRINT "Female"
230  PRINT
240 UNTIL FALSE
250 :
260 :
270 REM Read a data item from the file.  Treat
280 REM commas and CRs as data item terminators
290 REM and Control Z as the file terminator.
300 REM Since we are not interested in reading a
310 REM record at a time, the record terminator
```

```
320 REM CR LF is of no special interest to us.
330 REM We use the CR, along with commas, as a
332 REM data item separator and discard the LF.
334 :
340 DEF FN_read
350 LOCAL data$,byte$,byte
360 data$=""
370 REPEAT
380   byte=BGET#compat
390   IF byte=&1A OR EOF#compat THEN CLOSE#compat: END
400   IF NOT(byte=&0A OR byte=&0D OR byte=&2C) THEN data$=data$+CHR$(byte)
410 UNTIL byte=&0D OR byte=&2C
420 =data$
```

Random (Relative) FIles

Introduction

There are three example random file programs. The first is very simple, but it demonstrates the principle of random access files. The second expands the first into quite a useful database program. The final example is an inventory program. Although it does not provide application dependent features, it would serve as it stands and it is sufficiently well structured to be expanded without too many problems.

Designing the File

Unlike other versions of BASIC, there is no formalised record structure in BBC BASIC. A file is considered to be a continuous stream of bytes (characters) and you can directly access any byte of the file. This approach has many advantages, but most files are logically considered as a sequence of records (some of which may be empty). How then do we create this structure and access our logical records?

Record Structure

Creating the structure is quite simple. You need to decide what information you want to hold and the order in which you want store it. In the first example, for instance, we have two items of information (fields) per logical record; the name and the remarks. The name can be a maximum of 30 characters long and the remarks a maximum of 50 characters. So our logical record has two fields, one 30 characters long and the other 50 characters long. When the name string is written to the file it will be terminated by a CR - and so will the remarks string. So each record will be a maximum of 82 characters long.

We haven't finished yet, however. We need to be able to tell whether any one record is 'live' or empty (or deleted). To do this we need an extra byte at the start of each record which we set to one value for 'empty' and another for 'live'. In all the examples we use 0 to indicate 'empty' and NOT 0 to indicate 'live'. We are writing character data to the file so we could use the first byte of the name string as the indicator because the lowest ASCII code we will be storing is 32 (space). You can't do this for mixed data files because this byte could hold a data value of zero. Because of this, we have chosen to use an additional byte for the indicator in all the examples.

Our logical record thus consists of:

1	indicator byte
31	bytes for the name
51	bytes for the remarks

Thus the maximum amount of data in each record is 83 bytes. Because we cannot tell in advance how big each record needs to be (and we may want to change it later), we must assume that ALL the records will be this length. Since most of the records will be smaller than this, we are going to waste quite a lot of space in our random access file, but this is the penalty we pay for convenience and comparative simplicity.

When we write the data to the file, we could insist that each field was treated as a fixed length field by packing each string out with spaces to make it the 'correct' length. This would force each field to start at its 'proper' byte within the record. We don't need to do this, however, because we aren't going to randomly access the fields within the record; we know the order of the fields within the record and we are going to read them sequentially into appropriately named variables. We can write the fields to the file with each field following on immediately behind the previous one. All the 'spare' room is now left at the end of the record and not split up at the end of each field.

Accessing The Records

In order to access any particular record, you need to set ***PTR#*** to the first byte of that record. Remember, you can't tell BBC BASIC (Z80) that you want 'record 5', because it knows nothing of your file and record structure. You need to calculate the position of the first byte of 'record 5' and set PTR# to this value.

To start with, let's call the first record on the file 'record zero', the second record 'record 1', the third record 'record 2', etc. The first byte of 'record zero' is at byte zero on the file. The first byte of 'record 1' is at byte 83 on the file. The first byte of 'record 2' is at byte 166 (2*83) on the file. And so on. So, the start point of any record can be calculated by:

```
first_byte= 83*record_number
```

Now, we need to set PTR# to the position of this byte in order to access the record. If the record number was held in 'recno' and the file handle in 'fnum', we could do this directly by:

```
PTR#fnum=83*recno
```

However, we may want to do this in several places in the program so it would be better to define and use a function to set PTR# as illustrated below.

```
190 ...
200 PTR#fnum=FN_ptr(recno)
210 ...
etc

900 DEF FN_ptr(record)=83*record
```

Whilst the computer is quite happy with the first record being 'record zero', us mere humans find it a little confusing. What we need is to be able to call the first record 'record 1', etc. We could do this without altering the function which calculates the start position of each record, but we would waste the space allocated to 'record 0' since we would never use it. We want to call it 'record 1' and the program wants to call it 'record 0'. We can change the function to cater for this. If we subtract 1 from the record number before we multiply it by the record length, we will get the result we want. Record 1 will start at byte zero, record 2 will start at byte 83, etc. Our function now looks like this:

```
DEF FN_ptr(record)=83*(record-1)
```

In our example so far we have used a record length of 83. If we replace this with a variable 'rec_len' we have a general function which we can use to calculate the start position of any record in the file in any program. (You will need to set rec_len to the appropriate value at the start of the program.) The function now becomes:

```
DEF FN_ptr(record)=rec_len*(record-1)
```

We use this function (or something very similar to it) in the following three example programs using random access files.

Ex 9 - Simple Random Access File

```
 10 REM F-RAND1
 20 :
 30 REM VERY SIMPLE RANDOM ACCESS PROGRAM
 40 :
 50 REM This program maintains a random access
 60 REM file of names and remarks.  There is
 70 REM room for a maximum of 20 entries. Each
 80 REM name can be up to a max of 30 chars
 90 REM long and each remark up to 50 chars.
100 REM The first byte of the record is set non
110 REM zero (in fact &FF) if there is a record
120 REM present.  This gives a maximum record
130 REM length of 1+31+51=83. (Including CRs)
140 :
150 bell$=CHR$(7)
160 temp$=STRING$(50," ")
170 maxrec=20
180 rec_len=83
190 ans$=""
200 CLS
210 WIDTH 0
220 fnum=OPENUP "RANDONE"
230 IF fnum=0 fnum=FN_setup("RANDONE")
240 REPEAT
250   REPEAT
260     INPUT '"Enter record number: "ans$
270     IF ans$="0" CLOSE#fnum:CLS:END
280     IF ans$="" record=record+1 ELSE record=VAL(ans$)
290     IF record<1 OR record>maxrec PRINT bell$;
300   UNTIL record>0 AND record<=maxrec
310   PTR#fnum=FN_ptr(record)
320   PROC_display
330   INPUT '"Do you wish to change this record" ,ans$
340   PTR#fnum=FN_ptr(record)
350   IF FN_test(ans$) PROC_modify
360 UNTIL FALSE
370 END
380 :
390 :
400 DEF FN_test(A$) =LEFT$(A$,1)="Y" OR LEFT$(A$,1)="y"
410 :
420 :
430 DEF FN_ptr(record)=rec_len*(record-1)
440 REM This makes record 1 start at PTR# = 0
450 :
460 :
```

```
470 DEF PROC_display
480 PRINT '"Record number ";record'
490 flag=BGET#fnum
500 IF flag=0 PROC_clear:ENDPROC
510 INPUT#fnum,name$,remark$
520 PRINT name$;" ";remark$ '
530 ENDPROC
540 :
550 :
560 DEF PROC_clear
570 PRINT "Record empty"
580 name$=""
590 remark$=""
600 ENDPROC
610 :
620 :
630 DEF PROC_modify
640 PRINT '"(Enter <Enter> for no change or DELETE to delete)"'
650 INPUT "Name ",temp$
660 temp$=LEFT$(temp$,30)
670 IF temp$<>"" name$=temp$
680 INPUT "Remark ",temp$
690 temp$=LEFT$(temp$,50)
700 IF temp$<>"" remark$=temp$
710 INPUT '"Confirm update record",ans$
720 IF NOT FN_test(ans$) ENDPROC
730 IF name$="DELETE" BPUT#fnum,0:ENDPROC
740 BPUT#fnum,255
750 PRINT#fnum,name$,remark$
760 ENDPROC
770 :
780 :
790 DEF FN_setup(fname$)
800 PRINT "Setting up the database file"
810 fnum=OPENOUT(fname$)
820 FOR record=1 TO maxrec
830   PTR#fnum=FN_ptr(record)
840   BPUT#fnum,0
850 NEXT
860 =fnum
```

Ex 10 - Simple Random Access Database

The second program in this sub-section expands the previous program into a simple, but quite versatile, database program. A setup procedure has been added which allows you to specify the file name. If it is a new file, you are then allowed to specify the number of records and the number, name and size of the fields you wish to use. This information is stored at the start of the file. If the file already exists this data is read from the records at the beginning of the file. The function for calculating the start position of each record is modified to take into account the room used at the front of the file to store information about the database.

```
10 REM F-RAN
20 REM SIMPLE DATABASE PROGRAM
30 REM Written by R T Russell Jan 1983
40 REM Mod for BBC BASIC (Z80): D Mounter Dec 1985
50 :
60 REM This is a simple database program.  You
70 REM are asked for the name of the file you
 80 REM wish to use.  If the file does not
 90 REM already exist, you are asked to enter
100 REM the number and format of the records.
110 REM If the file does already exist, the file
120 REM specification is read from the file.
130 :
140 @%=&90A
150 bell$=CHR$(7)
160 CLS
170 WIDTH 0
180 INPUT '"Enter the filename of the data file: "filename$
190 fnum=OPENUP(filename$)
200 IF fnum=0 fnum=FN_setup(filename$) ELSE PROC_readgen
210 PRINT
220 :
230 REPEAT
240   REPEAT
250     INPUT '"Enter record number: "ans$
260     IF ans$="0" CLOSE#fnum:CLS:END
270     IF ans$="" record=record+1 ELSE record=VAL(ans$)
280     IF record<1 OR record>maxrec PRINT bell$;
290   UNTIL record>0 AND record<=maxrec
300   PTR#fnum=FN_ptr(record)
310   PROC_display
320   INPUT '"Do you wish to change this record" ,ans$
330   PTR#fnum=FN_ptr(record)
340   IF FN_test(ans$) PROC_modify
```

```
350 UNTIL FALSE
360 END
370 :
380 :
390 DEF FN_test(A$) =LEFT$(A$,1)="Y" OR LEFT$(A$,1)="y"
400 :
410 :
420 DEF FN_ptr(record)=base+rec_len*(record-1)
430 :
440 :
450 DEF FN_setup(filename$)
460 PRINT "New file."
470 fnum=OPENOUT(filename$)
480 REPEAT
490   INPUT "Enter the number of records (max 1000): "maxrec
500 UNTIL maxrec>0 AND maxrec<1001
510 REPEAT
520   INPUT "Enter number of fields per record (max 20): "fields
530 UNTIL fields>0 AND fields<21
540 DIM title$(fields),size(fields),A$(fields)
550 FOR field=1 TO fields
560   PRINT '"Enter title of field number ";field;": ";
570   INPUT ""title$(field)
580   PRINT
590     REPEAT
600     INPUT "Max size of field (characters)",size(field)
610   UNTIL size(field)>0 AND size(field)<256
620 NEXT field
630 rec_len=1
640 PRINT#fnum,maxrec,fields
650 FOR field=1 TO fields
660   PRINT#fnum,title$(field),size(field)
670   rec_len=rec_len+size(field)+1
680 NEXT field
690 base=PTR#fnum
700 :
710 FOR record=1 TO maxrec
720   PTR#fnum=FN_ptr(record)
730   BPUT#fnum,0
740 NEXT
750 =fnum
760 :
770 :
780 DEF PROC_readgen
790 rec_len=1
800 INPUT#fnum,maxrec,fields
810 DIM title$(fields),size(fields),A$(fields)
820 FOR field=1 TO fields
```

```
830  INPUT#fnum,title$(field),size(field)
 840  rec_len=rec_len+size(field)+1
 850 NEXT field
 860 base=PTR#fnum
 870 ENDPROC
 880 :
 890 :
 900 DEF PROC_display
 910 PRINT '"Record number ";record'
 920 flag=BGET#fnum
 930 IF flag=0 PROC_clear:ENDPROC
 940 FOR field=1 TO fields
 950  INPUT#fnum,A$(field)
 960  PRINT title$(field);" ";A$(field)
 970 NEXT field
 980 ENDPROC
 990 :
1000 :
1010 DEF PROC_clear
1020 FOR field=1 TO fields
1030  A$(field)=""
1040 NEXT
1050 ENDPROC
1060 :
1070 :
1080 DEF PROC_modify
1090 PRINT '"(Enter <Enter> for no change)"'
1100 FOR field=1 TO fields
1110  REPEAT
1120   PRINT title$(field);" ";
1130   INPUT LINE ""A$
1140   IF A$="" PRINT TAB(POS,VPOS-1)title$(field);" ";A$(field)
1150   REM TAB(POS,VPOS-1) moves the cursor up 1 line
1160  UNTIL LEN(A$)<=size(field)
1170  IF A$<>"" A$(field)=A$
1180 NEXT field
1190 INPUT '"Confirm update record",ans$
1200 IF NOT FN_test(ans$) ENDPROC
1210 IF A$(1)="DELETE" BPUT#fnum,0:ENDPROC
1220 BPUT#fnum,255
1230 FOR field=1 TO fields
1240  PRINT#fnum,A$(field)
1250 NEXT field
1260 ENDPROC
```

Ex 11 - Random Access Inventory Program

The final example in this sub-section is a full-blown inventory program. Rather than go through all its aspects at the start, they are discussed at the appropriate point in the listing. (These comments do not have line numbers and are not, of course, part of the program.)

```
 10 REM F-RAND
 20 :
 30 REM Written by Doug Mounter   Jan 1982
 40 REM Modified for BBC BASIC (Z80) Dec 1985
 50 :
 60 REM EXAMPLE OF A RANDOM ACCESS FILE
 70 :
 80 REM This is a simple inventory program.  It
 90 REM uses the item's part number as the key
 92 REM and stores:
100 REM  The item description - char max len 30
110 REM  The quantity in stock - numeric
120 REM  The re-order level - numeric
130 REM  The unit price - numeric
140 REM In addition, the first byte of the rec
150 REM is used as a valid data flag.  Set to 0
160 REM if empty, D if the record has been
170 REM deleted or V if the record is valid.
180 REM This gives a MAX record len of 47 bytes
190 REM (Don't forget the CR after the string)
200 :
210 PROC_initialise
220 inventry=FN_open("INVENTRY"
```

The following section of code is the command loop. You are offered a choice of functions until you eventually select function 0. The more traditional ***ON GOSUB*** statement has been used for menu selection processing. The newer ***ON PROC*** statement is illustrated in the indexed file example which follows. There are some forward jumps within procedures, etc to overcome the lack of a multi line ***IF*** statement. It would have been possible to have used further procedures, but the whole thing would have become rather laboured.

```
230 REPEAT
240   CLS
250   PRINT TAB(5,3);"If you want to:-"'
260   PRINT TAB(10);"End This Session";TAB(55);"Type 0"
```

```
270  PRINT TAB(10);"Amend or Create an Entry";TAB(55);"Type 1"
280  PRINT TAB(10);"Disp Inventory for One Part";TAB(55);"Type 2"
290  PRINT TAB(10);"Alter Stock  of One Part";TAB(55);"Type 3"
300  PRINT TAB(10);"Disp Items to Reorder";TAB(55);"Type 4"
310  PRINT TAB(10);"Recover a Deleted Item";TAB(55);"Type 5"
320  PRINT TAB(10);"List Deleted Items";TAB(55);"Type 6"
330  PRINT TAB(10);"Set Up a New Inventory";TAB(55);"Type 9"
340  REPEAT
350   PRINT TAB(5,15);bell$;
360   PRINT "Please enter selection (0 to 6 or 9) ";
370   function$=GET$
380  UNTIL function$>"/" AND function$<"8" OR function$="9"
390  function=VAL(function$)
400  ON function GOSUB 500,670,810,1100,1350,1540,1770,1790,1840 ELSE
410 UNTIL function=0
420 CLS
430 PRINT "Inventory File Closed" ''
440 CLOSE#inventry
450 END
460 :
470 :
480 REM AMEND/CREATE AN ENTRY
```

This is the data entry function. You can delete or amend an entry or enter a new one. Have a look at the definition of FN_getrec for an explanation of the ***ASC***"V" in its parameters.

```
490 :
500 REPEAT
510  CLS
520  PRINT "AMEND/CREATE"
530  partno=FN_getpartno
540  flag=FN_getrec(partno,ASC"V")
550  PROC_display(flag)
560  PRINT'"Do you wish to ";
570  IF flag PRINT "change this entry ? "; ELSE PRINT "enter data ? ";
580  IF GET$<>"N" flag=FN_amend(partno):PROC_cteos
590  PROC_write(partno,flag,type)
600  PRINT bell$;"Do you wish to amend/create another record ? ";
610 UNTIL GET$="N"
620 RETURN
630 :
640 :
650 REM DISPLAY AN ENTRY
```

This subroutine allows you to look at a record without the ability to change or delete it.

```
660 :
670 REPEAT
680   CLS
690   PRINT "DISPLAY"
700   partno=FN_getpartno
710   flag=FN_getrec(partno,ASC"V")
720   PROC_display(flag)
730   PRINT '
740   PRINT "Do you wish to view another part?";
750 UNTIL GET$="N"
760 RETURN
770 :
780 :
790 REM CHANGE THE STOCK LEVEL FOR ONE PART
```

The purpose of this subroutine is to allow you to update the stock level without having to amend the rest of the record.

```
800 :
810 REPEAT
820   CLS
830   PRINT "CHANGE STOCK"
840   partno=FN_getpartno
850   flag=FN_getrec(partno,ASC"V")
860   REPEAT
870     PROC_display(flag)
880     PROC_cteos
890     REPEAT
900       PRINT TAB(0,12);:PROC_cteol
910       INPUT "What is the change ? " temp$
920       change=VAL(temp$)
930     UNTIL INT(change)=change AND stock+change>=0
940     IF temp$="" flag=FALSE:GOTO 1000
950     stock=stock+change
960     PROC_display(flag)
970     PRINT'"Is this correct ? ";
980     temp$=GET$
990 :
1000   UNTIL NOT flag OR temp$="Y"
1010   PROC_write(partno,flag,ASC"V")
1020   PRINT return$;bell$;
1030   PRINT "Do you want any more updates ? ";
1040 UNTIL GET$="N"
```

```
1050 RETURN
1060 :
1070 :
1080 REM DISPLAY ITEMS BELOW REORDER LEVEL
```

This subroutine goes through the file in stock number order and lists all those items where the current stock is below the reorder level. You can interrupt the process at any time by pushing a key.

```
1090 :
1100 partno=1
1110 REPEAT
1120   CLS
1130   PRINT "ITEMS BELOW REORDER LEVEL"'
1140   line_count=2
1150   REPEAT
1160     flag=FN_getrec(partno,ASC"V")
1170     IF NOT(flag AND stock<reord) THEN 1230
1180     PRINT "Part Number ";partno
1190     PRINT desc$;" Stock ";stock;" Reorder Level ";reord
1200     PRINT
1210     line_count=line_count+3
1220 :
1230     partno=partno+1
1240     temp$=INKEY$(0)
1250   UNTIL partno>maxpartno OR line_count>20 OR temp$<>""
1260   PRINT TAB(0,23);bell$;"Push any key to continue or E to end ";
1270   temp$=GET$
1280 UNTIL partno>maxpartno OR temp$="E"
1290 partno=0
1300 RETURN
1310 :
1320 :
1330 REM RECOVER A DELETED ENTRY
```

Deleted entries are not actually removed from the file, just marked as deleted. This subroutine makes it possible for you to correct the mistake you made by deleting data you really wanted. If you have never used this type of program seriously, you won't believe how useful this is.

```
1340 :
1350 REPEAT
1360   CLS
1370   PRINT "RECOVER DELETED RECORDS"
1380   partno=FN_getpartno
```

```
1390  flag=FN_getrec(partno,ASC"D")
1400  PROC_display(flag)
1410  PRINT
1420  IF NOT flag THEN 1470
1430  PRINT "If you wish to recover this entry type Y ";
1440  temp$=GET$
1450  IF temp$="Y"PROC_write(partno,flag,ASC"V")
1460 :
1470  PRINT return$;bell$;"Do you wish to recover another record ? ";
1480 UNTIL GET$="N"
1490 RETURN
1500 :
1510 :
1520 REM LIST DELETED ENTRIES
```

This subroutine lists all the deleted entries so you can check you really don't want the data.

```
1530 :
1540 partno=1
1550 REPEAT
1560  CLS
1570  PRINT "DELETED ITEMS"'
1580  line_count=2
1590  REPEAT
1600   flag=FN_getrec(partno,ASC"D")
1610   IF NOT flag THEN 1660
1620   PRINT "Part Number ";partno
1630   PRINT "Description ";desc$'
1640   line_count=line_count+3
1650 :
1660   partno=partno+1
1670   temp$=INKEY$(0)
1680  UNTIL partno>maxpartno OR line_count>20 OR temp$<>""
1690  PRINT TAB(0,23);bell$;"Push any key to continue or E to end ";
1700 UNTIL partno>maxpartno OR GET$="E"
1710 partno=0
1720 RETURN
1730 :
1740 :
1750 REM DUMMY RETURNS FOR INVALID FUNCTION NUMs
1760 :
1770 RETURN
1780 :
1790 RETURN
1800 :
1810 :
```

```
1820 REM REINITIALISE THE INVENTORY DATA FILE
1830 :
1840 CLS
1850 PRINT TAB(0,3);bell$;"Are you sure you want to set up a new inventory?"
1860 PRINT "You will DESTROY ALL THE DATA YOU HAVE ACCUMULATED so far."
1890 PRINT "If you are SURE you want to do it, enter YES"
1900 PRINT "If you want to start a new inventory file, enter NEW"
1910 INPUT "Otherwise, just hit return ",temp$
1920 IF temp$="YES" PROC_setup(inventry)
1930 IF temp$="NEW" function=0
1940 RETURN
1950 :
1960 :
1970 REM INITIALISE ALL THE VARIOUS PRESETS ETC
```

This is where all the variables that you usually write as ***CHR$***(#) go. Then you can find them if you want to change them.

```
1980 :
1990 DEF PROC_initialise
2010 bell$=CHR$(7)
2020 return$=CHR$(13)
2030 rec_length=47
2040 partno=0
```

If you initially set strings to the maximum length you will ever use, you will save prevent the generation of 'garbage'.

```
2050 desc$=STRING$(30," ")
2060 temp$=STRING$(40," ")
2070 WIDTH 0
2130 REM OPEN FILE AND RETURN THE FILE HANDLE
2140 :
2150 REM If the file already exists, the largest permitted
2160 REM part number is read into maxpartno.
2170 REM If it is a new file, the file is
2180 REM initialised and the largest part
2190 REM number is written as the first record.
2200 :
2210 DEF FN_open(name$)
2220 fnum=OPENUP(name$)
2230 IF fnum>0 INPUT#fnum,maxpartno: =fnum
2240 fnum=OPENOUT(name$)
2250 CLS
It's a new file, so we won't go through the warning bit.
2260 PROC_setup(fnum)
```

```
2270 =fnum
2280 :
2290 REM SET UP THE FILE
2300 :
2310 REM Ask for maximum part number required,
2320 REM write it as the first record and then
2330 REM write 0 in to first byte of each rec.
2340 :
2350 DEF PROC_setup(fnum)
2360 REPEAT
2370   PRINT TAB(0,12);bell$;:PROC_cteos
2380   INPUT "What is the highest part number required (Max 5000)",maxpartno
2390 UNTIL maxpartno>0 AND maxpartno<5000 AND INT(maxpartno)=maxpartno
2400 PTR#fnum=0
2410 PRINT#fnum,maxpartno
2420 FOR partno=1 TO maxpartno
2430   PTR#fnum=FN_ptr(partno)
2440   BPUT#fnum,0
2450 NEXT
2460 partno=0
2470 ENDPROC
2480 :
2490 :
2500 REM GET AND RETURN THE REQUIRED PART NUMBER
```

Ask for the required part number. If a null is entered, make the next part number one more than the last.

```
2510 :
2520 DEF FN_getpartno
2530 REPEAT
2540   PRINT TAB(0,5);bell$;:PROC_cteos
2550   PRINT "Enter a Part Number Between 1 and ";maxpartno '
2560   IF partno=maxpartno THEN 2590
2570   PRINT "The Next Part Number is ";partno+1;
2580   PRINT " Just hit RETURN to get this"'
2590 :
2600   INPUT "What is the Part Number You Want ", partno$
2610   IF partno$<>"" partno=VAL(partno$):GOTO 2630
2620   IF partno=maxpartno partno=0 ELSE partno=partno+1
2630 :
2640   PRINT TAB(35,9);partno;:PROC_cteol
2650 UNTIL partno>0 AND partno<maxpartno+1 AND INT(partno)=partno
2660 =partno
2670 :
2680 :
2690 REM GET THE RECORD FOR THE PART NUMBER
```

```
2700 :
2710 REM Return TRUE if the record exists and
2720 REM FALSE if not  If the record does not
2730 REM exist, load desc$ with "No Record" The
2740 REM remainder of the record is set to 0.
2742 :
2750 DEF FN_getrec(partno,type)
2760 stock=0
2770 reord=0
2780 price=0
2790 PTR#inventry=FN_ptr(partno)
2800 test=BGET#inventry
2810 IF test=0 desc$="No Record": =FALSE
2820 IF test=type THEN 2850
2830 IF type=86 desc$="Record Deleted" ELSE desc$="Record Exists"
2840 =FALSE
2850 :
2860 INPUT#inventry,desc$
2870 INPUT#inventry,stock,reord,price
2880 =TRUE
2890 :
2900 :
2910 REM CALCULATE THE VALUE OF PTR FOR THIS REC
```

Part numbers run from 1 up. The record for part number 1 starts at byte 5 of the file. The start position could have been calculated as (part-no -1) *record_length + 5. The expression below works out to the same thing, but it executes quicker.

```
2920 :
2930 DEF FN_ptr(partno)=partno*rec_length+5-rec_length
2940 :
2950 :
2960 REM AMEND THE RECORD
```

This function amends the record as required and returns with flag=***TRUE*** if any amendment has taken place. It also sets the record type indicator (valid deleted or no record) to ASC"V" or ASC"D" as appropriate.

```
2970 :
2980 DEF FN_amend(partno)
2990 PRINT return$;:PROC_cteol:PRINT TAB(0,4);
3000 PRINT "Please Complete the Details for Part Number ";partno
3010 PRINT "Just hit Return to leave the entry as it is"'
3020 flag=FALSE
3030 type=ASC"V"
```

```
3040 INPUT "Description - Max 30 Chars " temp$
3050 IF temp$="DELETE" type=ASC"D": =TRUE
3060 temp$=LEFT$(temp$,30)
3070 IF temp$<>"" desc$=temp$:flag=TRUE
3080 IF desc$="No Record" OR desc$="Record Deleted" =FALSE
3090 INPUT "Current Stock Level " temp$
3100 IF temp$<>"" stock=VAL(temp$):flag=TRUE
3110 INPUT "Reorder Level " temp$
3120 IF temp$<>"" reord=VAL(temp$):flag=TRUE
3130 INPUT "Unit Price " temp$
3140 IF temp$<>"" price=VAL(temp$):flag=TRUE
3150 =flag
3160 :
3170 :
3180 REM WRITE THE RECORD
```

Write the record to the file if necessary (flag=TRUE)

```
3190 :
3200 DEF PROC_write(partno,flag,type)
3210 IF NOT flag ENDPROC
3220 PTR#inventry=FN_ptr(partno)
3230 BPUT#inventry,type
3240 PRINT#inventry,desc$,stock,reord,price
3250 ENDPROC
3260 :
3270 :
3280 REM DISPLAY THE RECORD DETAILS
```

Print the record details to the screen. If the record is not of the required type (V or D) or it does not exist, stop after printing the description. The description holds "Record Exists" or "Record Deleted" or valid data as set by FN_getrec.

```
3290 :
3300 DEF PROC_display(flag)
3310 PRINT TAB(0,5);:PROC_cteos
3320 PRINT "Part Number ";partno'
3330 PRINT "Description ";desc$
3340 IF NOT flag ENDPROC
3350 PRINT "Current Stock Level ";stock
3360 PRINT "Reorder Level ";reord
3370 PRINT "Unit Price ";price
3380 ENDPROC
3390 :
3400 :
```

The two following procedures rely on the screen width being 80 characters:

```
3410 REM There are no 'native' clear to end of
3420 REM line/screen vdu procedures.  The
3430 REM following two procedures clear to the
3440 REM end of the line/screen.
3450 DEF PROC_cteol
3460 LOCAL x,y
3470 x=POS:y=VPOS
3480 IF y=31 PRINT SPC(79-x); ELSE PRINT SPC(80-x);
3490 PRINT TAB(x,y);
3500 ENDPROC
3510 :
3520 :
3530 DEF PROC_cteos
3540 LOCAL I,x,y
3550 x=POS:y=VPOS
3560 IF y<31 FOR I=y TO 30:PRINT SPC(80);:NEXT
3570 PRINT SPC(79-x);TAB(x,y);
3580 ENDPROC
```

Indexed Data Files

Deficiencies of Random Access Files

As you will see if you dump a random file, a lot of space is wasted. This is because all the records must be allocated the same amount of space, otherwise you could not calculate where the record started. For large data files, over 50% of the space can be wasted. Under these circumstances it is possible to save space by using two files, one as an index to the other. In order for this to work efficiently, you must have complete control over the file pointer. Not many versions of BASIC allow this control, but it is quite simple with BBC BASIC.

The Address Book Program

The final program is an example of an indexed file. It is a computer implementation of the address book discussed way back at the beginning of these notes. Two files are used, one as an index to the other. Both are serial and no space is wasted between records.

File Organisation

The files are organised as shown below:

NAME.NDX (index file)

maxrec	length	index$(1)	index(1)	index$(2)	index(2)	etc.→
5 bytes	5 bytes	1 to 31 bytes	5 bytes	1 to 31 bytes	5 bytes	

Where index(n) points to a record in the data file as follows:

ADDRESS.DTA (data file)

Phone Num	Address 1	Address 2	Address 3	Address 4	Post Code
1 to 31 bytes	1 to 31 bytes	1 to 31 bytes	1 to 31 bytes	1 to 31 bytes	1 to 31 bytes

maxrec	Is the maximum number of records permitted in this file. The practical limit is governed by the amount of memory available for the index arrays which are held in memory.
length	Is the number of entries in the index.
index(n)	Is the value of PTR#datanum just prior to the first byte of the data for this entry being written to it. In the Random File examples this value was calculated, and it increased by a constant amount for every record.

Program Organisation

The example looks horribly long and complicated. However the actual file handling bits are quite simple. The rest is, as usual, required for tidy input and output of data. The meat of the program is in the procedures and functions for putting and deleting index entries and finding the right place in the index. The latter uses a routine called a 'binary chop' (you could get arrested for that). This looks simple, and it is - when it works. If you are interested there is a flow chart and a brief explanation of how it works at the end of these notes. For the faithful, just use it. It takes considerably less time than any other method to search an ordered list.

The Index

The index is read into memory at the start and written back at the end. In memory, it consists of two arrays called index$() and index(). Oh that we could have mixed type arrays!

Ex 12 (the LAST)

```
 10 REM F-INDEX
 20 REM EXAMPLE OF AN INDEXED FILE
 30 :
 40 REM Written by Doug Mounter - Feb 1982
 50 REM Modified for BBC BASIC (Z80) - Dec 1985
 60 :
 70 REM This is a simple address book filing
 80 REM system.  It will accept names, telephone
 90 REM numbers and addresses and store them in a
100 REM file called ADDRESS.DTA.  The index is in
110 REM name order and is kept in a file called
120 REM NAME.NDX.  All the fields are character
122 REM and the maximum length  of any field
124 REM is 30.
130 :
140 PROC_initialise
150 PROC_open_files
160 ON ERROR IF ERR<>17 PRINT:REPORT:PRINT" At line ";ERL:END
170 REPEAT
180   CLS
190   PRINT TAB(5,3);"If you want to:-" '
200   PRINT TAB(10);"End This Session";TAB(55);"Type 0"
210   PRINT TAB(10);"Enter Data";TAB(55);"Type 1"
220   PRINT TAB(10);"Search For/Delete an Entry";TAB(55);"Type 2"
230   PRINT TAB(10);"List in Alphabetical Order";TAB(55);"Type 3"
240   PRINT TAB(10);"Reorg data File and Index";TAB(55);"Type 4";
250   REPEAT
260     PRINT TAB(5,11);
270     PRINT "Please enter the appropriate number (0 to 4) ";
280     function$=GET$
290     PRINT return$;:PROC_cteol
300   UNTIL function$>"/" AND function$<"5"
310   function=VAL(function$)
320   PRINT TAB(54,function+5);"<====<<";
330   ON function PROC_enter,PROC_search,PROC_list,PROC_reorg,ELSE
340 UNTIL function=0
350 CLS
360 PROC_close_files
370 *ESC ON
```

```
380 PRINT "Address Book Files Closed"'
390 END
400 :
410 :
420 REM ENTER DATA
430 :
440 DEF PROC_enter
450 flag=TRUE
460 temp$=""
470 i=1
480 REPEAT
490   REPEAT
500     IF temp$="N" PROC_message("Data NOT Accepted")
510     PROC_get_data
520     IF length=maxrec OR data$(1)="" flag=FALSE:GOTO 590
530     IF data$(1)="+" OR data$(1)="-" PROC_message("Bad Data"):GOTO 590
540     i=FN_find_place(0,data$(1))
550     IF i>0 PROC_message("Duplicate Record")
560     PRINT '"Is this data correct ? ";
570     temp$=FN_yesno
580     :
590   UNTIL NOT flag OR temp$<>"N"
600   PROC_cteos
610   IF NOT flag THEN 670
620   PROC_put_index(i,data$(1),PTR#datanum)
630   FOR i=2 TO 7
640     PRINT#datanum,data$(i)
650   NEXT
660   :
670 UNTIL NOT flag
680 ENDPROC
690 :
700 :
710 REM SEARCH FOR AN ENTRY
720 :
730 DEF PROC_search
740 i=0
750 REPEAT
760   PRINT TAB(0,11);:PROC_cteol
770   INPUT "What name do you want to look for ",name$
780   IF name$="" THEN 800
790   IF name$<>""IF name$="DELETE" PROC_delete(i) ELSE i=FN_display(i,name$)
800 UNTIL name$=""
810 ENDPROC
820 :
830 :
840 REM LIST IN ALPHABETICAL ORDER
850 :
```

```
860 DEF PROC_list
870 entry=1
880 REPEAT
890   CLS
900   line_count=0
910   REPEAT
920     PRINT TAB(0,line_count);
930     PROC_read_data(entry)
940     PROC_print_data
950     entry=entry+1
960     line_count=line_count+8
970     temp$=INKEY$(0)
980   UNTIL entry>length OR line_count>16 OR temp$<>""
990   PROC_message("Push any key to continue or E to end ")
1000 UNTIL entry>length OR GET$="E"
1010 ENDPROC
1020 :
1030 :
1040 REM REORGANISE THE DATA FILE AND INDEX
1050 :
1060 DEF PROC_reorg
1070 entry=1
1080 PRINT TAB(0,13);"Reorganising the Data File" '
1090 newdata=OPENOUT"ADDRESS.BAK"
1100 REPEAT
1110   PROC_read_data(entry)
1120   index(entry)=PTR#newdata
1130   FOR i=2 TO 7
1140     PRINT#newdata,data$(i)
1150   NEXT
1160   entry=entry+1
1170 UNTIL entry>length
1180 CLOSE#newdata
```

The time taken to rename a file can be considerable.

```
1190 PRINT "Re-naming the Data File" '
1200 *REN ADDRESS.$$$=ADDRESS.BAK
1210 PRINT "*";
1220 *REN ADDRESS.BAK=ADDRESS.DTA
1230 PRINT "*";
1240 *REN ADDRESS.DTA=ADDRESS.$$$
1250 PRINT "*";
1260 datanum=OPENUP "ADDRESS.DTA"
1270 ENDPROC
1280 :
1290 :
```

```
1300 REM INITIALISE VARIABLES AND ARRAYS
1310 :
1320 DEF PROC_initialise
1340 *ESC OFF
1350 esc$=CHR$(27)
1360 bell$=CHR$(7)
1370 return$=CHR$(13)
1380 maxrec=100
1390 :
1400 REM The maximum record number, maxrec, is
1402 REM read in
1410 REM PROC_read_index if the file already exists.
1420 :
1430 DIM message$(7)
1440 FOR i=1 TO 7
1450   READ message$(i)
1460 NEXT
1470 DATA Name,Phone Number,Address,-- " --,-- "--,-- " --,Post Code
1480 :
1490 DIM data$(7)
1500 FOR i=1 TO 7
1510   data$(i)=STRING$(30," ")
1520 NEXT
1530 temp$=STRING$(255," ")
1540 temp$=""
1550 :
1610 REM OPEN THE FILES
1620 :
1630 DEF PROC_open_files
1640 indexnum=OPENUP"NAME.NDX"
1650 datanum=OPENUP"ADDRESS.DTA"
1660 IF indexnum=0 OR datanum=0 PROC_setup ELSE PROC_read_index
1670 PTR#datanum=EXT#datanum
1680 ENDPROC
1690 :
1700 :
1710 REM SET UP NEW INDEX AND DATA FILES
1720 :
1730 DEF PROC_setup
1740 CLS
1750 PRINT TAB(0,13);"Setting Up Address Book"
1760 indexnum=OPENOUT"NAME.NDX"
1770 datanum=OPENOUT"ADDRESS.DTA"
1780 length=0
1790 PRINT#indexnum,maxrec,length
1800 CLOSE#indexnum
1810 DIM index$(maxrec+1),index(maxrec+1)
1820 index$(0)=""
```

```
1830 index(0)=0
1840 index$(1)=CHR$(&FF)
1850 index(1)=0
1860 ENDPROC
1870 :
1880 :
1890 REM READ INDEX AND LENGTH OF DATA FILE
1900 :
1910 DEF PROC_read_index
1920 CLS
1930 INPUT#indexnum,maxrec,length
1940 DIM index$(maxrec+1), index(maxrec+1)
1950 index$(0)=""
1960 index(0)=0
1970 FOR i=1 TO length
1980   INPUT#indexnum,index$(i),index(i)
1990 NEXT
2000 CLOSE#indexnum
2010 index$(length+1)=CHR$(&FF)
2020 index(length+1)=0
2030 ENDPROC
2040 :
2050 :
2060 REM WRITE INDEX AND CLOSE FILES
2070 :
2080 DEF PROC_close_files
2090 indexnum=OPENOUT"NAME.NDX"
2100 PRINT#indexnum,maxrec,length
2110 FOR i=1 TO length
2120   PRINT#indexnum,index$(i),index(i)
2130 NEXT
2140 CLOSE#0
2150 ENDPROC
2160 :
2170 :
2180 REM WRITE A MESSAGE AT LINE 23
2190 :
2200 DEF PROC_message(line$)
2210 LOCAL x,y
2220 x=POS
2230 y=VPOS
2240 PRINT TAB(0,23);:PROC_cteol:PRINT bell$;line$;
2250 PRINT TAB(x,y);
2260 ENDPROC
2270 :
2280 :
2290 REM GET A Y/N ANSWER
2300 :
```

```
2310 DEF FN_yesno
2320 LOCAL temp$
2330 temp$=GET$
2340 IF temp$="y" OR temp$="Y" ="Y"
2350 IF temp$="n" OR temp$="N" ="N"
2360 =""
2370 :
2380 :
2390 REM CLEAR 9 LINES FROM PRESENT POSITION
```

This procedure makes use of the machine code routine at the end of the program. It works in a similar fashion to the clear-to-end-of-line and clear-to-end-of-screen procedures defined towards the end of the program.

```
2400 :
2410 DEF PROC_clear9
2420 LOCAL x,y,i
2430 PRINT return$;
2440 A%=&A20:B%=0:C%=720:D%=0
2450 CALL int10
2460 ENDPROC
2470 :
2480 :
2490 REM GET INPUT DATA - LIMIT TO 30 CHAR
2500 :
2510 DEF PROC_get_data
2520 LOCAL i
2530 PRINT TAB(0,13);
2540 PROC_clear9
2550 IF length=maxrec PROC_message("Add Book Full")
2560 FOR i=1 TO 7
2570   PRINT TAB(10);message$(i);TAB(25);
2580   INPUT temp$
2590   data$(i)=LEFT$(temp$,30)
2600   IF data$(1)="" i=7
2610 NEXT
2620 ENDPROC
2630 :
2640 :
2650 REM FIND AND DISPLAY THE REQUESTED DATA
2660 :
2670 DEF FN_display(i,name$)
2680 PRINT TAB(0,12);:PROC_cteos
2690 i=FN_find_place(i,name$)
2700 IF i<0 PROC_message("Name Not Known - Next Highest Given")
2710 PROC_read_data(i)
2720 PRINT
```

```
2730 PROC_print_data
2740 =i
2750 :
2760 :
2770 REM DELETE THE ENTRY FROM THE INDEX
2780 :
```

Move everything below the entry you want deleted up one and subtract 1 from the length

```
2790 DEF PROC_delete(i)
2800 INPUT "Are you SURE ",temp$
2810 PRINT TAB(0,VPOS-1);:PROC_cteos
2820 IF temp$<>"YES" ENDPROC
2830 IF i<0 i=-i
2840 FOR i=i TO length
2850   index$(i)=index$(i+1)
2860   index(i)=index(i+1)
2870 NEXT
2880 length=length-1
2890 ENDPROC
2900 :
2910 :
2920 REM READ DATA FOR ENTRY i
```

Get the start of the position of the start of the data record for entry 'i' in the index and read it into the buffer array data$(). Save the current value of the data file pointer on entry and restore it before leaving.

```
2930 :
2940 DEF PROC_read_data(i)
2950 PTRdata=PTR#datanum
2960 IF i<0 i=-i
2970 PTR#datanum=index(i)
2980 data$(1)=index$(i)
2990 FOR i=2 TO 7
3000   INPUT#datanum,data$(i)
3010 NEXT
3020 PTR#datanum=PTRdata
3030 ENDPROC
3040 :
3050 :
3060 REM PRINT data$() ON VDU
3070 :
3080 DEF PROC_print_data
```

```
3090 LOCAL i
3100 FOR i=1 TO 7
3110   IF data$(i)<>"" PRINT TAB(10);message$(i);TAB(25);data$(i)
3120   IF data$(1)=CHR$(&FF) i=7
3130 NEXT
3140 ENDPROC
3150 :
3160 :
3170 REM PUT A NEW ENTRY IN INDEX AT POSITION i
```

Move all the directory entries from position i onwards down the index. (In fact you have to start at the end and work back.) Slot the new entry in in the gap made at position i and add 1 to the length.

```
3180 :
3190 DEF PROC_put_index(i,entry$,ptr)
3200 LOCAL j
3210 IF i<0 i=-i
3220 FOR j=length+1 TO i STEP -1
3230   index$(j+1)=index$(j)
3240   index(j+1)=index(j)
3250 NEXT
3260 index$(i)=entry$
3270 index(i)=ptr
3280 length=length+1
3290 ENDPROC
3300 :
3310 :
3320 REM FIND ENTRY IN INDEX OR PLACE TO PUT IT
```

This function looks in the index for the string entry$. If it finds it it returns with i set to its position in the index. If not, i is set to minus the position of the next highest string. (In other words, the position you wish to put the a new entry.) Thus if a part of the index looked like:

```
(34)    BERT
(35)    FRED
(36)    JOHN
```

and you entered with FRED, it would return 35. However if you entered with GEORGE, it would return -36.

The function consists of two parts. The first looks at the entry$ to see if it should just up or down the entry number by 1, taking account of wrap-around at the start and end of the index. The second part is the binary chop advertised with such telling wit in the introduction to indexed files. Since we enter this function with the entry pointer i set to its previous value, we must cater for a negative value.

```
3330 :
3340 DEF FN_find_place(i,entry$)
3350 LOCAL top,bottom
3360 IF i<0 i=-i
3370 IF entry$="+" AND i<length =i+1
3380 IF entry$="+" AND i=length =1
3390 IF entry$="-" AND i>1 =i-1
3400 IF entry$="-" AND i<2 =length
```

Here, at last, **T H E B I N A R Y C H O P**

```
3410 top=length+1
3420 bottom=0
3430 i=(top+1) DIV 2
3440 IF entry$<>index$(i) i=FN_search(entry$)
3450 REPEAT
3460   IF entry$=index$(i-1) i=i-1
```

This bit moves the pointer up the index to the first of any duplicate entries.

```
3470 UNTIL entry$<>index$(i-1)
3480 IF entry$=index$(i) =i ELSE =-i
3490 :
3500 :
3510 REM DO THE SEARCHING FOR FN_find_place
3520 :
3530 DEF FN_search(entry$)
3540 REPEAT
3550   IF entry$>index$(i) bottom=i ELSE top=i
3560   i=(top+bottom+1) DIV 2: REM round
3570 UNTIL entry$=index$(i) OR top=bottom+1
3580 =i
3590 :
3600 :
```

The two following procedures rely on the screen width being 80 characters:

```
3410 REM There are no 'native' clear to end of
3420 REM line/screen vdu procedures.  The
3430 REM following two procedures clear to the
3440 REM end of the line/screen.
3450 DEF PROC_cteol
3460 LOCAL x,y
3470 x=POS:y=VPOS
3480 IF y=31 PRINT SPC(79-x); ELSE PRINT SPC(80-x);
3490 PRINT TAB(x,y);
3500 ENDPROC
3510 :
3520 :
3530 DEF PROC_cteos
3540 LOCAL I,x,y
3550 x=POS:y=VPOS
3560 IF y<31 FOR I=y TO 30:PRINT SPC(80);:NEXT
3570 PRINT SPC(79-x);TAB(x,y);
3580 ENDPROC
```

Well, that's it. Apart from the following notes on the binary chop you have read it all.

The Binary Chop

Explanation

The quickest way to find an entry in an ORDERED list is not to search through it from start to end, but to continue splitting the list in two until you reach the entry you are looking for. You begin by setting one pointer to the bottom of the list, another to the top, and a third to mid-way between bottom and top. Then you compare the entry pointed to by this third pointer with the number you are searching for. If your number is bigger you make the bottom equal the pointer, if not make the top equal to it. Then you repeat the process.

Let's try searching the list of numbers below for the number 14.

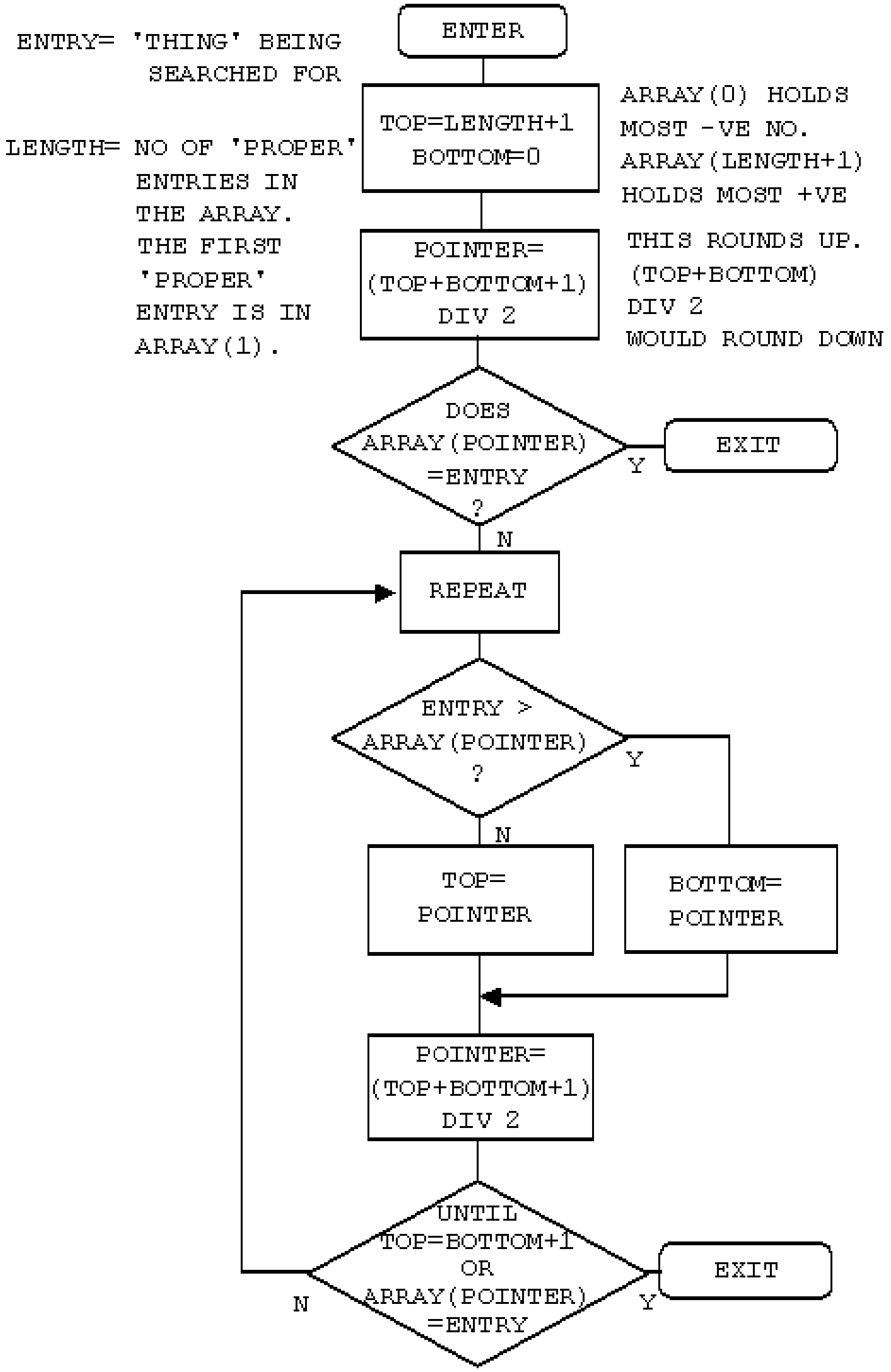
```
bottom>  (1)  3     Set bottom to the lowest position
         (2)  6     in the list, and top to the highest.
         (3)  8     Set the pointer to
         (4)  14    (top+bottom)/2. Is that entry 14?
```

pointer>	(5)	19	No it's more, so set top to the
	(6)	23	current value of pointer and
	(7)	34	repeat the process.
	(8)	45	
top>	(9)	61	
bottom>	(1)	3	Set the pointer to (top+bottom)/2.
	(2)	6	Is that entry 14? No it's less, so
pointer>	(3)	8	set bottom to the current value of
	(4)	14	pointer and try again.
top>	(5)	19	
	(6)	23	
	(7)	34	
	(8)	45	
	(9)	61	
	(1)	3	Set the pointer to (top+bottom)/2.
	(2)	6	Is that entry 14? Yes, so exit with
bottom>	(3)	8	the pointer set to the position in
pointer>	(4)	14	the list of the number you are
top>	(5)	19	looking for.
	(6)	23	
	(7)	34	
	(8)	45	
	(9)	61	

As you can imagine, things are not always as simple as this carefully chosen example. You have to cater for the number not being there, and for the list being empty. There are a number of ways of doing this, but the easiest is to add two numbers of your choice to the list. Make the first entry the most negative number the computer can hold, and the last entry the most positive. This will prevent you ever trying to search outside the list. Preventing a perpetual loop when the number you want is not in the list is quite simple, just exit when 'top' is equal to 'bottom'+1. If you have not found the number by then, it's not in the list.

You can use this routine to add numbers to the list in order. If you can't find the number, you exit with the position it should go in the list. Just move all the numbers under it down one slot and put the new number in. This works just as well when the list is empty except for your two 'end markers'.

Have a look at the flow chart below and work through a couple of dry runs with a short list of numbers. You may think that it's not worth doing it this way and that a 'linear search' would be as quick. Try it with a list of 100 numbers. It should take you no more than 7 goes to find the number. The AVERAGE number of comparisons required for a linear search would be 50.


ENTRY= 'THING' BEING SEARCHED FOR
LENGTH= NO OF 'PROPER' ENTRIES IN THE ARRAY. THE FIRST 'PROPER' ENTRY IS IN ARRAY(1).
ENTER
TOP=LENGTH+1
BOTTOM=0
ARRAY(0) HOLDS MOST -VE NO. ARRAY(LENGTH+1) HOLDS MOST +VE
POINTER= (TOP+BOTTOM+1) DIV 2
THIS ROUNDS UP. (TOP+BOTTOM) DIV 2 WOULD ROUND DOWN
DOES ARRAY(POINTER) =ENTRY ?
Y
EXIT
N
REPEAT
ENTRY > ARRAY(POINTER) ?
Y
N
TOP= POINTER
BOTTOM= POINTER
POINTER= (TOP+BOTTOM+1) DIV 2
UNTIL TOP=BOTTOM+1 OR ARRAY(POINTER) =ENTRY
N
Y
EXIT

6. Operating System Interface

Introduction

As with the BBC microcomputer, the star (*) commands provide access to the Operating System.

File Specifiers

File specifiers must comply with the standard Operating System conventions, for example in the case of CP/M:

```
[drive:]filename.extension
```

drive:	The single letter name (A to P) of the drive where the file will be found. The colon is mandatory. If the drive name is omitted, the currently logged-on drive is assumed.
filename	The name of the file. The length of the name must not exceed 8 characters.
extension	The optional extension of the file. If an extension is used it must be separated from the filename by a full-stop. If the extension is omitted, .BBC is assumed.

Drives A: to P: are accepted in file specifications. Filenames in star commands may optionally be enclosed in quotes; unmatched quotes will cause a 'Bad string' error.

?	Allow any single character in this position. If this is used as the last character in the name, a null character will be accepted.
*	Allow any character (including a null) from the position of the '*' to the end of the name or extension.

Symbols

The following symbols and abbreviations are used as part of the explanation of the operating system commands.

{}	The enclosed items may be repeated 0 or more times.
[]	The items enclosed are optional, they may occur zero or one time.
num	A numeric constant.
str	A string constant.
<num>	A numeric variable.
<str>	A string variable.
afsp	An ambiguous file specifier.
ufsp	An unambiguous file specifier.
d:	A drive name.
dir	A directory name.

Accessing Star Commands

The star commands may be accessed directly or via the OSCLI statement. The two examples below both access the BYE command.

```
*BYE
```

```
OSCLI("BYE")
```

Syntax

A star command must be the last (or only) command on a program line and its argument may not be a variable. If you need to use one of these commands with a variable as the argument, use the OSCLI statement. Examples of the use of the OSCLI statement are given below in the Resident Star Commands sub-section.

Case Conversion

Typically star commands and their associated qualifiers may be entered in lower-case to upper-case. For example, *era filename is equivalent to *ERA filename. This is in keeping with the BBC Micro's machine operating system (MOS).

Special Characters

Control characters, lower-case characters, DEL and quotation marks may be incorporated in filenames by using the 'escape' character '|'.

|A	gives ^A.
|a	gives lower-case A.
|?	gives Del.
|"	gives the quote marks ".
||	gives the escape character |.
|!	sts bit 7 of the following character.

Resident Star Commands

(CP/M edition only, other versions may have similar commands, consult the relevant documentation)

**BYE*

Return to CP/M.

**CPM*

Return to CP/M (same as *BYE).

**DIR*

List the directory. The syntax is similar to the CP/M DIR command except that the extension defaults to .BBC if it is omitted.

```
*DIR [afsp]
```

*DIR	List all .BBC files in the directory.
DIR B:.*	List all files on drive B.
..*	List all the files in the current directory.

*DRIVE

Select the drive to be used as the default drive for subsequent file operations. The colon is a mandatory part of the drive name.

```
*DRIVE d:
```

```
*DRIVE A:
```

*ERA

Delete the specified file. The syntax is similar to the CP/M-80 ERA command except that the extension defaults to .BBC if it is omitted.

```
*ERA ufsp
```

```
*ERA GAME1.DTA
```

This command will delete only one file at a time; wild cards are not permitted in the file specifier.

OSCLI may also be used to delete files.

```
OSCLI "ERA "+<str>
```

```
OSCLI "ERA "+name$
```

```
OSCLI "ERA GAME1.DTA"
```

*LOAD

Load the specified file into memory at hexadecimal address 'aaaa'. The load address MUST always be specified.

```
*LOAD ufsp aaaa
```

```
*LOAD A:WOMBAT 8F00
```

OSCLI may also be used to load a file. However, you must take care to provide the load address as a hexadecimal number in string format.

```
OSCLI "LOAD "+<str>+" "+STR$~<num>

OSCLI "LOAD "+file_name$+" "+STR$~(start_address)
```

*OPT

Select the destination 'output stream' for console output. The default is OPT 0.

*OPT 0	**Console output.** The output is sent to the screen.
*OPT 1	**Auxiliary Output.** The output is sent to the auxiliary output.
*OPT 2	**Printer Output.** The output is sent to the list device.

All characters sent to the console are vectored with *OPT.

```
10 *OPT 2
20 PRINT "THIS WILL APPEAR ON THE PRINTER"
30 *OPT 0
40 PRINT "THIS WILL APPEAR ON THE SCREEN"
```

*REN

Rename a file. The syntax is similar to the CP/M REN command except that the extension defaults to .BBC if it is omitted. OSCLI can also be used to rename a file.

```
*RENAME ufspnew=ufspold
*REN ufspnew=ufspold
OSCLI "REN "+<str>+"="+<str>

*REN NEWFILE=OLDFILE

OSCLI "REN "file_name$+".BAK="+filename$+".BBC"
```

*RESET

Reset the Filing System.

*SAVE

Save an area of memory to a file. If the extension is omitted, .BBC is assumed. You MUST specify the start address (aaaa) and either the length of the area of memory (llll) or its end address+1 (bbbb). There is no 'load address' or 'execute address'.

```
*SAVE ufsp aaaa +llll
*SAVE ufsp aaaa bbbb

*SAVE "WOMBAT" 8F00 +80
*SAVE "WOMBAT" 8F00 8F80
```

OSCLI can also be used to save a file.

```
OSCLI "SAVE "+<str>+" "+STR$~(<num>)+" "+STR$(<num>)

OSCLI "SAVE "+ufn$+" "+STR$~(add)+"+"+STR$~(len)
```

*TYPE

Type the specified file on the screen. This command is similar in action to the CP/M-80 TYPE command except that the extension .BBC is assumed if it is omitted.

```
*TYPE ufsp

*TYPE ADDRESS.LST
```

7. Table of ASCII Codes

Annex A to BBC BASIC (Z80)

Binary	Hex	Dec	Char	
00000000	00	0	^@	NUL
00000001	01	1	^A	SOH Start of Heading
00000010	02	2	^B	STX Start of Text
00000011	03	3	^C	ETX End of Text
00000100	04	4	^D	EOT End of Transmit
00000101	05	5	^E	ENQ Enquiry
00000110	06	6	^F	ACK Acknowledge
00000111	07	7	^G	BEL Bell - Audible Signal
00001000	08	8	^H	BS Back Space
00001001	09	9	^I	HT Horizontal Tab
00001010	0A	10	^J	LF Line Feed
00001011	0B	11	^K	VT Vertical Tab
00001100	0C	12	^L	FF Form Feed
00001101	0D	13	^M	CR Carriage Return
00001110	0E	14	^N	SO Shift Out
00001111	0F	15	^O	SI Shift In
00010000	10	16	^P	DLE Data Link Escape
00010001	11	17	^Q	DC1 X On
00010010	12	18	^R	DC2 Aux On
00010011	13	19	^S	DC3 X Off
00010100	14	20	^T	DC4 Aux Off
00010101	15	21	^U	NAK Negative Acknowledge
00010110	16	22	^V	SYN Synchronous File
00010111	17	23	^W	ETB End of Transmitted Block
00011000	18	24	^X	CAN Cancel
00011001	19	25	^Y	EM End of Medium
00011010	1A	26	^Z	SUB Substitute
00011011	1B	27	^[	ESC Escape
00011100	1C	28	^\	FS File Separator
00011101	1D	29	^]	GS Group Separator
00011110	1E	30	^^	RS Record Separator
00011111	1F	31	^_	US Unit Separator

Binary	Hex	Dec	Char
00100000	20	32	Space
00100001	21	33	!
00100010	22	34	"
00100011	23	35	#
00100100	24	36	$
00100101	25	37	%
00100110	26	38	&
00100111	27	39	'
00101000	28	40	(
00101001	29	41	)
00101010	2A	42	*
00101011	2B	43	+
00101100	2C	44	,
00101101	2D	45	-
00101110	2E	46	.
00101111	2F	47	/
00110000	30	48	0
00110001	31	49	1
00110010	32	50	2
00110011	33	51	3
00110100	34	52	4
00110101	35	53	5
00110110	36	54	6
00110111	37	55	7
00111000	38	56	8
00111001	39	57	9
00111010	3A	58	:
00111011	3B	59	;
00111100	3C	60	<
00111101	3D	61	=
00111110	3E	62	>
00111111	3F	63	?

Binary	Hex	Dec	Char	Binary	Hex	Dec	Char
01000000	40	64	@	01100000	60	96	`
01000001	41	65	A	01100001	61	97	a
01000010	42	66	B	01100010	62	98	b
01000011	43	67	C	01100011	63	99	c
01000100	44	68	D	01100100	64	100	d
01000101	45	69	E	01100101	65	101	e
01000110	46	70	F	01100110	66	102	f
01000111	47	71	G	01100111	67	103	g
01001000	48	72	H	01101000	68	104	h
01001001	49	73	I	01101001	69	105	i
01001010	4A	74	J	01101010	6A	106	j
01001011	4B	75	K	01101011	6B	107	k
01001100	4C	76	L	01101100	6C	108	l
01001101	4D	77	M	01101101	6D	109	m
01001110	4E	78	N	01101110	6E	110	n
01001111	4F	79	O	01101111	6F	111	o
01010000	50	80	P	01110000	70	112	p
01010001	51	81	Q	01110001	71	113	q
01010010	52	82	R	01110010	72	114	r
01010011	53	83	S	01110011	73	115	s
01010100	54	84	T	01110100	74	116	t
01010101	55	85	U	01110101	75	117	u
01010110	56	86	V	01110110	76	118	v
01010111	57	87	W	01110111	77	119	w
01011000	58	88	X	01111000	78	120	x
01011001	59	89	Y	01111001	79	121	y
01011010	5A	90	Z	01111010	7A	122	z
01011011	5B	91	[	01111011	7B	123	{
01011100	5C	92	\	01111100	7C	124	|
01011101	5D	93	]	01111101	7D	125	}
01011110	5E	94	^	01111110	7E	126	~
01011111	5F	95	_	01111111	7F	127	DEL Delete

8. Mathematical Functions

Annex B to BBC BASIC (Z80)

BBC BASIC (Z80) has more intrinsic mathematical functions than many other versions of BASIC. Those that are not provided may be calculated as shown below.

Function	**Calculation**
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
Inverse SECANT	ARCSEC(X)=ACS(1/X)
Inverse COSECANT	ARCCSC(X)=ASN(1/X)
Inverse COTANGENT	ARCCOT(X)=ATN(1/X) =PI/2-ATN(X)
Hyperbolic SINE	SINH(X)=(EXP(X)-EXP(-X))/2
Hyperbolic COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
Hyperbolic TANGENT	TANH(X)=EXP(-X)/(EXP(X)+EXP(-X))*2+1
Hyperbolic SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
Hyperbolic COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
Hyperbolic COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
Inverse Hyperbolic SIN	ARCSINH(X)=LN(X+SQR(X*X+1))
Inverse Hyperbolic COSINE	ARCCOSH(X)=LN(X+SQR(X*X-1))
Inverse Hyperbolic TANGENT	ARCTANH(X)=LN((1+X)/(1-X))/2
Inverse Hyperbolic SECANT	ARCSECH(X)=LN((SQR(-X*X+1)+1)/X)
Inverse Hyperbolic COSECANT	ARCCSCH(X)=LN((SGN(X)*SQR(X*X+1)+1)/X
Inverse Hyperbolic COTANGENT	ARCCOTH(X)=LN((X+1)/(X-1))/2
LOGn(X)	LOGn(X)=LN(X)/LN(n) =LOG(X)/LOG(n)

9. Error Messages and Codes

Annex C to BBC BASIC (Z80)

Summary

Trappable - Program

No	Error	No	Error
1	Out of range	2	*
3	Multiple label	4	Mistake
5	Missing ,	6	Type mismatch
7	Not in a FN	8	*
9	Missing "	10	Bad DIM
11	DIM space	12	Not in a FN or PROC
13	Not in a PROC	14	Bad use of array
15	Bad subscript	16	Syntax error
17	Escape	18	Division by zero
19	String too long	20	Number too big
21	-ve root	22	Log range
23	Accuracy lost	24	Exponent range
25	*	26	No such variable
27	Missing)	28	Bad hex or binary
29	No such FN/PROC	30	Bad call
31	Bad arguments	32	Not in a FOR loop
33	Can't match FOR	34	Bad FOR variable
35	*	36	Missing TO
37	*	38	No GOSUB
39	ON syntax	40	ON range
41	No such line	42	Out of DATA
43	Not in a REPEAT loop	44	Bad EXIT
45	Missing #	46	Not in a WHILE loop
47	Missing ENDCASE	48	OF not last
49	Missing ENDIF	50	*
51	*	52	*
53	ON ERROR not LOCAL	54	DATA not LOCAL

* Not applicable to BBC BASIC (Z80)

Trappable - Operating System

No	Error	No	Error
190	Directory full	192	Too many open files
196	File exists	198	Disk full
200	Close error	204	Bad name
214	File not found	222	Channel
253	Bad string	254	Bad command
255	Unknown error		

Untrappable - Error Code 0

No room	RENUMBER space
Silly	LINE space
Bad program	Failed at nnn

Strictly speaking 'Bad program' does not have an error code. It leaves ERR and ERL unchanged.

Details

BBC BASIC (Z80)'s error messages and codes are briefly explained below in alphabetical order.

Accuracy lost 23

Before BBC BASIC (Z80) calculates trigonometric functions (sin, cos, etc) of very large angles the angles are reduced to +/- PI radians. The larger the angle, the greater the inaccuracy of the reduction, and hence the result. When this inaccuracy becomes unacceptable, BBC BASIC (Z80) will issue an 'Accuracy lost' error message.

Bad arguments 31

This error indicates that too many or too few arguments have been passed to a procedure or function or an invalid formal parameter has been used. See the sub-section on Procedures and Functions.

Bad call 30

This error indicates that a procedure or function has been incorrectly called, for example if there is a space after the PROC or FN.

```
PROC test
```

Bad command 254

This error occurs when a command name is not recognized as a valid BBC BASIC (Z80) command. Star commands which are unknown to BBC BASIC (Z80) are passed to the Operating System. If the command is not recognised, an untrappable 'Bad command or file name' error occurs.

Bad DIM 10

Arrays must be positively dimensioned. In other words, the numbers within the brackets must not be negative. This error would be produced by the following example.
DIM table(20,-10)

Bad EXIT 44

(BBC BASIC version 5 or later only)

An EXIT statement was encountered when not inside a loop of the specified kind:

```
REPEAT
  EXIT WHILE
UNTIL FALSE
```

Bad FOR variable 34

The variable in a FOR...NEXT loop must be a numeric variable. If you use a constant or a string variable this error message will be generated. For example, the following statements are not legal.

```
20 FOR name$=1 TO 20

20 FOR 10=1 TO 20
```

Bad hex or binary **28**

Hexadecimal numbers can only include the characters 0 to 9 and A to F; similarly binary numbers can only contain the digits 0 and 1. If you try to form a hexadecimal or binary number with other characters this error will occur. For example:

```
n = &OF
n = %21
```

(in the first example a letter O was used where the figure 0 was intended).

Bad name **204**

This error is generated if a path name exceeds 64 characters in length.

Bad program

From time to time BBC BASIC (Z80) checks to see that the program in memory is of the correct format (See Annex E). If it is unable to follow the program from the start to the 'program end marker' it will report this untrappable error. The error can be caused by a read error, by only loading part of the program or by overwriting part of the program in some way. (Machine code programmers beware.) Without a full understanding of how a program is stored in memory, there is little you can do to recover a bad program.

Bad string **253**

File names in 'star' commands may optionally be enclosed in quotes. This error will occur if the quotes are unmatched. The following example would give rise to this error.

```
*SAVE "GRAPHS
```

Bad subscript 15

If you try to access an element of an array less than zero or greater than the size of the array you will generate this error. Both lines 20 and 30 of the following example would give rise to this error message.

```
10 DIM test(10)
20 test(-4)=20
30 test(30)=10
```

Bad use of array 14

This error occurs when BBC BASIC (Z80) thinks it should be accessing an array, but does not know which one.

Can't match FOR 33

BBC BASIC (Z80) has been unable to find a FOR statement corresponding to the NEXT statement.

Channel 222

This error is generated by the filing system. It occurs if you try to use a channel which has not been opened, possibly because you are using the wrong channel number.

Close error 200

This error will occur if the file(s) specified cannot be closed for any reason.

DATA not LOCAL 54

(BBC BASIC version 5 or later only)

This error is generated if BBC BASIC encounters a RESTORE DATA statement but is unable to match it with a corresponding LOCAL DATA statement. This can happen as a result of *jumping out of a loop* in the meantime.

Note that when LOCAL DATA is used inside a FOR... NEXT, REPEAT... UNTIL or WHILE... ENDWHILE loop, or inside a user-defined function, procedure or subroutine, the data pointer is automatically restored on exit (there is no need to use RESTORE DATA).

DIM space 11

This error will be generated if:

- There is insufficient room for an array when you try to dimension it.
- An attempt has been made to reserve a negative amount of memory. For example,

```
DIM A% -2
```

Directory full 190

This error will occur if an attempt is made to create more files than the device or directory has capacity for.

Disk full 198

This error will occur if there is insufficient room on the storage device for the data/program being written.

Division by zero 18

Mathematically, dividing by zero gives an infinitely large answer. The computer is unable to understand the concept of infinity (it's not alone) and this error is generated. If there is any possibility that the divisor might be zero, you should test for this condition before carrying out the division. For example:

```
200 IF divisor=0 THEN PROC_error ELSE...
```

Escape 17

This error is generated by pressing the <Esc> key. You can trap this, and other errors, by using the ON ERROR GOTO statement.

Exponent range 24

The EXP function is unable to cope with powers greater than 88. If you try to use a larger power, this error will be generated.

Failed at nnn

During renumbering, BBC BASIC (Z80) tries to resolve all line numbers referred to by GOTO and GOSUB statements. Should it fail, it will generate a 'Failed at nnn' error, where nnn is the RENUMBERED line which contains the unresolved reference.

The following example:

```
100 REM Demonstration renumber fail program
110 GOTO 250
120 END
```

would renumber as:

```
10 REM Demonstration renumber fail program
20 GOTO 250
30 END
```

and generate the error message 'Failed at 20'.

File exists 196

This error will be generated if you try to rename a file and a file with the new name already exists.

File not found 214

This error will occur if you try to LOAD, *LOAD or CHAIN a file which does not exist.

Line Space

A program line is too long to be represented in BBC BASIC (Z80)'s internal format.

Log range 22

Logarithms for zero and negative numbers do not exist. This error message will be generated if you try to calculate the log of zero or a negative number or raise a negative number to a non-integer power.

Missing , 5

This error message is generated if BBC BASIC (Z80) was unable to find a comma where one was expected. The following example would give rise to this error.

```
20 PRINT TAB(10 5)
```

Missing " 9

This error message is generated if BBC BASIC (Z80) was unable to find a double-quote where one was expected. The following example would give rise to this error.

```
10 name$="Douglas
```

Missing) 27

This error message is generated if BBC BASIC (Z80) was unable to find a closing bracket where one was expected. The following example would give rise to this error.

```
10 PRINT SQR(num
```

Missing # 45

This error will occur if BBC BASIC (Z80) is unable to find a hash symbol (a pound symbol on some computers) where one was expected. The following example would cause this error.

```
CLOSE 7
```

Missing ENDCASE 47

(BBC BASIC version 5 or later only)

This error is generated if BBC BASIC cannot locate an ENDCASE statement to terminate the current CASE statement.

Missing ENDIF 49

(BBC BASIC version 5 or later only)

This error is generated if BBC BASIC cannot locate an ENDIF statement to terminate the current multi-line IF ... THEN statement.

Missing TO 36

This error message will be generated if BBC BASIC (Z80) encounters a FOR...NEXT loop with the TO part missing.

Mistake 4

This error will be generated if BBC BASIC (Z80) is unable to make any sense at all of the input line.

Multiple label 3

This error is generated by the assembler if, during the first pass of an assembly (OPT 0, 1, 4, 5, 8, 9, 12 or 13), a label is found to have an existing non-zero value. This is most likely to be caused by the same label being used twice (or more times) in the program.

-ve root 21

This error message will occur if BBC BASIC (Z80) attempted to calculate the square root of a negative number. It is possible for this error to occur with ASN and ACS as well as SQR

```
 90 num=-20
100 root=SQR(num)
```

No GOSUB 38

This error message will be generated if BBC BASIC (Z80) finds a RETURN statement without first encountering a GOSUB statement. (See the sub-section on Program Flow Control.)

No Room

This untrappable error indicates that all the computer's available memory was used up whilst a program was running. This error may occur as a result of numerous assignments to string variables, as in a string sort. See the explanation of String Variables and Garbage in the Variables sub-section for details.

No such FN/PROC 29

When BBC BASIC (Z80) encounters a name beginning with FN or PROC it expects to be able to find a corresponding function or procedure definition. This error will occur if such a definition does not exist.

No such line 41

This error will occur if BBC BASIC (Z80) tries
to GOTO, GOSUB, TRACE or RESTORE to a non-existent line number.

No such variable 26

Variables are brought into existence by assigning a value to them or making them LOCAL in a function or procedure definition. This error message will be generated if you try to use a variable on the right-hand side of an assignment or access it in a PRINT statement before it has been created. As shown below, you can create variables very simply.

```
10 count=0
20 name$=""
```

Not in a FN 7

If BBC BASIC (Z80) encounters an end of function without calling a function definition, this error message will be issued. If you forget to put multi-line function definitions out of harm's way at the end of the program you are very likely to get this error message. (See the sub-section on Procedures and Functions.)

Not in a FN/PROC 12

If you try to define a variable as LOCAL outside a procedure or function, this error message will be generated. If you forget to put multi-line function definitions out of harm's way at the end of the program you are very likely to get this error message. (See the sub-section on Procedures and Functions.)

Not in a FOR looop 32

This error message indicates that BBC BASIC (Z80) has found a NEXT statement without first encountering a FOR statement.

Not in a PROC 13

If BBC BASIC (Z80) encounters an ENDPROC without performing (calling) a procedure definition, this error message will be issued. If you forget to put multi-line procedure definitions out of harm's way at the end of the program you are very likely to get this error message. (See the sub-section on Procedures and Functions.)

Not in a REPEAT loop 43

This error message indicates that BBC BASIC (Z80) has found an UNTIL statement without first encountering a REPEAT statement.

Not in a WHILE loop 46

(BBC BASIC version 5 or later only)

This error message indicates that BBC BASIC (Z80) has found an ENDWHILE statement without first encountering a WHILE statement.

Number too big 20

This error will occur if a number is entered or calculated which is too big for BBC BASIC (Z80) to cope with.

OF not last 48

(BBC BASIC version 5 or later only)

This error is generated if the keyword OF is not the last thing on the line in a CASE statement. OF cannot even be followed by a REMark.

ON ERROR not LOCAL 53

(BBC BASIC version 5 or later only)

This error is generated if BBC BASIC encounters a RESTORE ERROR statement but is unable to match it with a corresponding ON ERROR LOCAL statement. This can be caused by having 'jumped out of a loop' in the meantime (see the sub-section on Program Flow Control for more details).

ON range 40

This error will be generated if, in a simple ON GOTO/GOSUB/PROC statement, the control variable was less than 1 or greater than the number of entries in the ON list. These exceptions can be trapped in ON GOTO/GOSUB/PROC statements by using the ELSE option. The first example below will generate an 'ON range' error, whilst the second is correct.

```
10 num=4
20 ON num GOTO 100,200,300
```

```
10 num=4
20 ON num GOTO 100,200,300 ELSE 1000
```

ON syntax 39

This error will be reported if the ON...GOTO statement was misformed. For example, the following statement is not legal. (Refer to the keyword ON for details of legal statements.)

```
20 ON x TIME=0
```

Out of DATA 42

If your program tried to read more items of data than there were in the data list, this error will be generated. You can use RESTORE to return the data pointer to the first data statement (or to a particular line with a data statement) if you wish.

Out of range 1

This assembly language error will be reported if you tried to perform a relative jump of more than +127 or -128 bytes or you used a 16 bit port address when only an 8 bit address is allowed.

RENUMBER space

When BBC BASIC RENUMBERs a program it has to build a cross-reference table of line numbers. If there is insufficient memory to hold this table, the 'RENUMBER space' error results. In this case you can still renumber the program using the RENUMBER.COM utility program supplied.

Silly

This error message will be issued if you try to renumber a program or enter AUTO with a step size of 0. AUTO with a step size of more than 255 will work, but it will be evaluated MOD 256.

String too long 19

You will get this error message if your program tries to generate a string which is longer than 255 characters.

Syntax error 16

A command was terminated incorrectly. In other words, the first part of the command was recognized, but the rest of it was meaningless or incomplete. Unlike Mistake, BBC BASIC (Z80) was able to recognise the start of the command.

Too many open files 192

This error will occur if you try to open more than seven files at any one time.

Type mismatch 6

This error indicates that a number was encountered when a string was expected and vice-versa. Don't forget that this can occur if the actual parameters and the formal parameters for a function or procedure do not correspond. (See sub-section on Procedures and Functions for details of parameter passing to functions and procedures.)

Unknown error 255

Error number 255 is reserved for errors with an unknown cause, such as those thrown by the Operating System (such an error may be reported as 'CP/M Error').

10. Format of Program and Variables in Memory

Annex D to BBC BASIC (Z80)

Memory Map

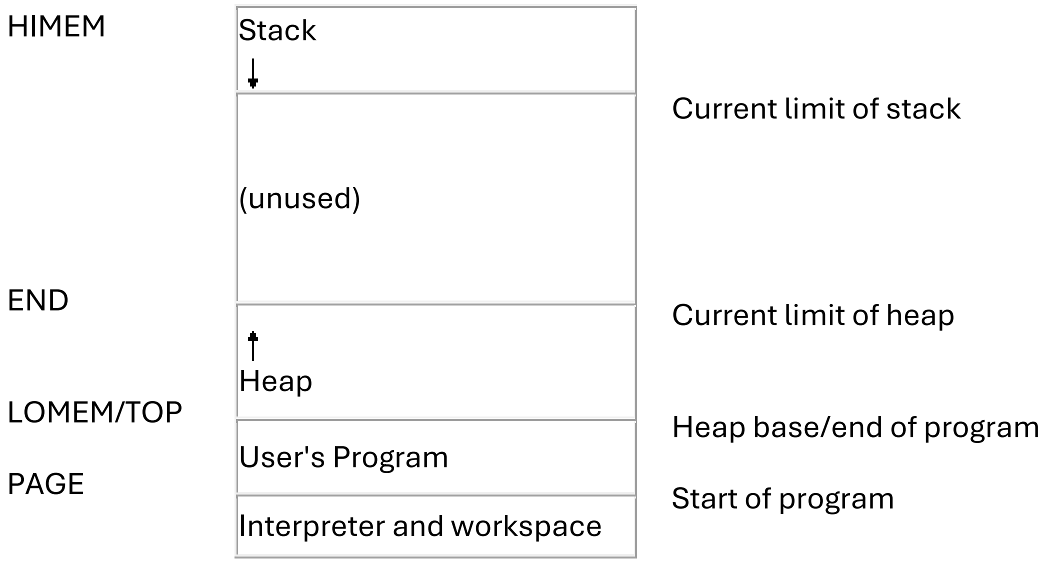
The BBC BASIC (Z80) version 5 interpreter requires 16 Kbytes of code space. The interpreter's internal variables occupy approximately 1 Kbyte of memory
By default, your program will start on the page boundary immediately following the interpreter's data area and the 'dynamic data structures' will immediately follow your program. The total group of the dynamic data structures is called the 'heap'. The base of the program control stack is located at HIMEM.

As your program runs, the heap expands upwards towards the stack and the stack expands downwards towards the heap. If the two should meet, you get a 'No room' error. Fortunately, there is a limit to the amount by which the stack and the heap expand.

In general, the heap only expands whilst new variables are being declared. However, bad management of string variables can also cause the heap to expand.

In addition to running your program, the stack is also used 'internally' by the BBC BASIC (Z80) interpreter. Its size fluctuates but, in general, it expands every time you increase the depth of nesting of your program structure and every time you increase the number of local variables in use.

The Memory Map



The function of HIMEM, END, LOMEM, TOP and PAGE are briefly discussed below. You will find more complete definitions elsewhere in this manual. You can directly set HIMEM, LOMEM and PAGE. However, for most of your programs you won't need to alter any of them. You will probably only need to change HIMEM if you want to put some machine code sub-routines at the top of memory.

HIMEM	The first address at the top of memory which is not available for use by BBC BASIC (Z80). The base of the program stack is set at HIMEM. (The first 'thing' stored on the stack goes at HIMEM-1.)
END	***(BBC BASIC version 5 or later only)*** The first unused location above the heap. The current size of the heap is given by: PRINT END-LOMEM
LOMEM	The start address for the heap. The first of the dynamic data structures starts at LOMEM.
TOP	The first free location after the end of your program. Unless you have set LOMEM yourself, LOMEM=TOP. You cannot directly set TOP. It alters as you enter your program. The current length of your program is given by: PRINT TOP-PAGE
PAGE	The address of the start of your program. You can place several programs in memory and switch between them by using PAGE. Don't forget to control LOMEM as well. If you don't, the heap for one program might overwrite another program.

Memory Management

There is little you can do to control the growth of the stack. However, with care, you can control the growth of the heap. You can do this by limiting the number of variables you use and by good string variable management.

Limiting the Number of Variables

Each new variable occupies room on the heap. Restricting the length of the names of variables and limiting the number of variables used will limit the size of the heap. However, of the techniques available to you, this is the least rewarding. In addition, it leads to incomprehensible programs because your variable names become meaningless. You should keep this technique in the back of your mind whilst you are programming, but only apply it rigorously if you are really stuck for space.

String Management

Garbage Generation

Unlike numeric variables, string variables do not have a fixed length. When you create a string variable it is added to the heap and sufficient memory is allocated for the initial value of the string. If you subsequently assign a longer string to the variable there will be insufficient room for it in its original position and the string will have to be relocated with its new value at the top of the heap. The initial area will then become 'dead' and the heap will have grown by the new length of the string. The areas of 'dead' memory are called garbage. As more and more re-assignments take place, the heap grows and eventually there is no more room. Thus, it is possible to run out of room for variables even though there should be enough space.

Memory Allocation for String Variables

You can overcome the problem of 'garbage' by reserving enough memory for the longest string you will ever put into a variable before you use it. You do this simply by assigning a string of spaces to the variable. If your program needs to find an empty string the first time it is used, you can subsequently assign a null string to it.

The same technique can be used for string arrays. The example below sets up a single dimensional string array with room for 20 characters in each entry and then empties it.

```
10 DIM names$(10)
20 FOR i=0 TO 10
30   name$(i)=STRING$(20," ")
40 NEXT
50 stop$=""
60 FOR i=0 TO 10
70   name$(i)=""
80 NEXT
```

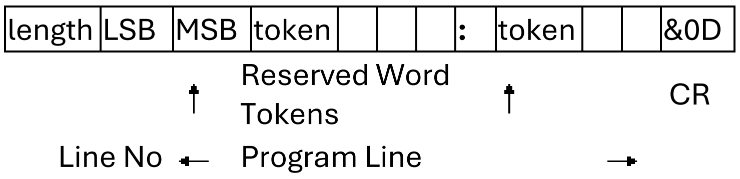
Assigning a null string to stop$ prevents the space for the last entry in the array being recovered when it is emptied.

Although it will rarely be necessary to do this in practice, an exception is in the case of LOCAL string arrays when you should initialise the array with strings of the maximum length to avoid a memory leak each time the PROC/FN is called:

```
LOCAL a$()
DIM a$(size%)
a$() = STRING$(max%, "a")
a$() = ""
```

Program Storage in Memory

The program is stored in memory in the format shown below. The first program line commences at PAGE.



Line Length

The line length includes the line length byte. The address of the start of the next line is found by adding the line length to the address of the start of the current line. The end of the program is indicated by a line length of zero and a line number of &FFFF.

Line Number

The line number is stored in two bytes, LSB first. The end of the program is indicated by a line number of &FFFF and a line length of zero.

Statements

With the exception of the symbols '*', '=' and '[' and the optional reserved word LET, each statement in the line commences with the appropriate reserved word token. Reserved words are tokenised wherever they occur. A token is indicated by bit 7 of the byte being set. Statements within a line are separated by colons.

Line Terminator

Each program line is terminated by a carriage-return (&0D).

Variable Storage in Memory

Variables are held within memory as linked lists (chains). The first variable in each chain is accessed via an index which is maintained by BBC BASIC (Z80). There is an entry in the index for each of the characters permitted as the first letter of a variable name. Each entry in the index has a word (two bytes) address field which points to the first variable in the linked list with a name starting with its associated character. If there are no variables with this character as the first character in the name, the pointer word is zero. The first word of all variables holds the address of the next variable in the chain. The address word of the last variable in the chain is zero. All addresses are held in the standard Z80 format - LSB first.

The first variable created for each starting character is accessed via the index and subsequently created variables are accessed via the index and the chain. Consequently, there is some speed advantage to be gained by arranging for all your variables to start with a different character. Unfortunately, this can lead to some pretty unreadable names and programs.

Integer Variables

Integers are held in 32-bit two's complement format. They occupy 4 bytes, with the LSB first. Bit 7 of the MSB is the sign bit. To make up the complete variable, the address word, the name and a separator (zero) byte are added to the number. The format of the memory occupied by an integer variable called 'NUMBER%' is shown below. Note that since the first character of the name is found via the index, it is not stored with the variable.

LSB	MSB		U	M	B	E	R	%	&00	LSB			MSB

↑ ↑ ← Rest of Name → ← Value →

Address of next variable
starting with the same letter

The smallest amount of space is taken up by a variable with a single letter name. The static integer variables, which are not included in the variable chains, use the names A% to Z%. Thus, the only single character names available for dynamic integer variables are a% to z% plus _% and ` % (CHR$(96)). As shown below, integer variables with these names will occupy 8 bytes.

LSB	MSB	%	&00	LSB			MSB

↑ ↑ ← Value →

Address of next variable
starting with the same letter

Byte Variables

(BBC BASIC version 5 or later only)

Byte variables are 8-bit unsigned numbers in the range 0-255. The format of the memory occupied by a byte variable called 'BYTE&' is shown below. Note that since the first character of the name is found via the index, it is not stored with the variable.

LSB	MSB	Y	T	E	&	&00	Value

↑ ↑ ← Rest of Name →

Address of next variable
starting with the same letter

Real Variables

Real numbers are held in binary floating point format. The mantissa is held as a 4 byte binary fraction in sign and magnitude format. Bit 7 of the MSB of the mantissa is the sign bit. When working out the value of the mantissa, this bit is assumed to be 1 (a decimal value of 0.5). The exponent is held as a single byte in 'excess 127' format. In other words, if the actual exponent is zero, the value of stored in the exponent byte is 127. To make up the complete variable, the address word, the name and a separator (zero) byte are added to the number. The format of the memory occupied by a real variable called 'NUMBER' is shown below.

LSB	MSB	U	M	B	E	R	&00	LSB			MSB	EXP

↑ ↑ ← Rest of Name → ← Mantissa → ↑

Address of next variable starting with the same letter — Exponent

As with integer variables, variables with single character names occupy the least memory. (However, the names A to Z are available for dynamic real variables.) Whilst a real variable requires an extra byte to store the number, the '%' character is not needed in the name. Thus, integer and real variables with the same name occupy the same amount of memory. However, this does not hold for arrays, since the name is only stored once.

In the following examples, the bytes are shown in the more human-readable manner with the MSB on the left.

The value 5.5 would be stored as shown below.

Mantissa				**Exponent**
.0011 0000	0000 0000	0000 0000	0000 0000	1000 0010
↑ Sign Bit				
&30	00	00	00	&82

Because the sign bit is assumed to be 1, this would become:

Mantissa				**Exponent**
.1011 0000	0000 0000	0000 0000	0000 0000	1000 0010
&B0	00	00	00	&82

The equivalent in decimal is:

(0.5+0.125+0.0625) * 2^(130-127)
= 0.6875 * 2^3
= 0.6875 * 8
= 5.5

BBC BASIC (Z80) stores integer values in real variables in a special way which allows the faster integer arithmetic routines to be used if appropriate. The presence of an integer value in a real variable is indicated by the stored exponent being zero. Thus, if the stored exponent is zero, the real variable is being used to hold an integer and the 4 byte mantissa holds the number in normal integer format.

Depending on how it is put there, an integer value can be stored in a real variable in one of two ways. For example,

```
number=5
```

will set the exponent to zero and store the integer &00 00 00 05 in the mantissa. On the other hand,

```
number=5.0
```

will set the exponent to &82 and the mantissa to &20 00 00 00.

The two ways of storing an integer value are illustrated in the following four examples.

```
Example 1
 number=5              & 00   00   00   00   05   Integer 5
Example 2
 number=5.0            & 82   20   00   00   00   Real 5.0
This is treated as
                       & 82   A0   00   00   00
=                      (0.5+0.125)*2^(130-127)
=                      0.625*8
=                      5
```

because the sign bit is assumed to be 1.

Example 3

```
 number=-5             & 00   FF   FF   FF   FB
```

The 2's complement gives

```
                       & 00   00   00   00   05   Integer -5
```

Example 4

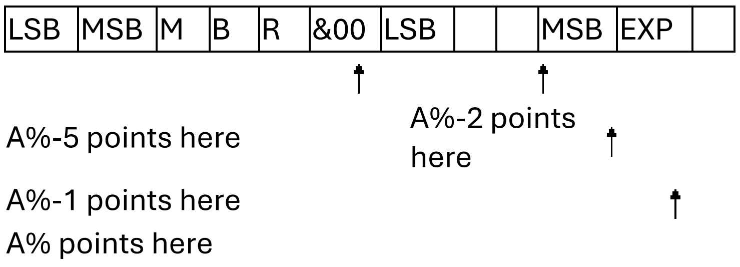
```
 number=-5.0           & 82   A0   00   00   00   Real -5.0
```

(The sign bit is already 1)

```
=                       (0.5+0.125)*2^(130-127)
=                       0.625*8
Magnitude =             5
```

If all this seems a little complicated, try using the program on the next page to accept a number from the keyboard and display the way it is stored in memory. The program displays the 4 bytes of the mantissa in 'human readable order' followed by the exponent byte. Look at what happens when you input first 5 and then 5.0 and you will see how this corresponds to the explanation given above. Then try -5 and -5.0 and then some other numbers. (The program is an example of the use of the byte indirection operator. See the Indirection section for details.)

The layout of the variable 'NMBR' in memory is shown below.



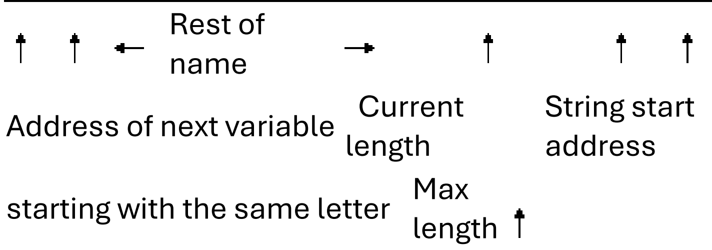
```
10 NUMBER=0
 20 DIM A% -1
 30 REPEAT
 40   INPUT"NUMBER PLEASE "NUMBER
 50   PRINT "& ";
 60   :
 70   REM Step through mantissa from MSB to LSB
 80   FOR I%=2 TO 5
 90     REM Look at value at address A%-I%
100      NUM$=STR$~(A%?-I%)
110      IF LEN(NUM$)=1 NUM$="0"+NUM$
120      PRINT NUM$;" ";
130   NEXT
140   :
150   REM Look at exponent at address A%-1
160   N%=A%?-1
170   NUM$=STR$~(N%)
180   IF LEN(NUM$)=1 NUM$="0"+NUM$
```

```
190  PRINT " & "+NUM$"
200 UNTIL NUMBER=0
```

String Variables

String variables are stored as the string of characters. Since the current length of the string is stored in memory an explicit terminator for the string in unnecessary. As with numeric variables, the first word of the complete variable is the address of the next variable starting with the same character. However, since BBC BASIC (Z80) needs information about the length of the string and the address in memory where the it starts, the overheads for a string are more than for a numeric. The format of a string variable called 'NAME$' is shown below.

LSB	MSB	A	M	E	$	&00	len	max	LSB	MSB



When a string variable is first created in memory, the characters of the string follow immediately after the two bytes containing the start address of the string and the current and maximum lengths are the same. While the current length of the string does not exceed its length when created, the characters of the string will follow the address bytes. When the string variable is set to a string which is longer than its original length, there will be insufficient room in the original position for the characters of the string. When this happens, the string will be placed on the top of the heap and the new start address will be loaded into the two address bytes. The original string space will remain, but it will be unusable. This unusable string space is called 'garbage'. See the Memory Management sub-section for ways to avoid creating garbage.

Because the original length and the current length of the string are each stored in a single byte in memory, the maximum length of a string held in a string variable is 255 characters.

Fixed Strings

You can place a string starting at a given location in memory using the indirection operator '$'. For example,

```
$&8000="This is a string"
```

would place &54 (T) at address &8000, &68 (h) at address &8001, etc. Because the string is placed at a predetermined location in memory it is called a 'fixed' string.

Fixed strings are not included in the variable chains and they do not have the overheads associated with a string variable. However, since the length of the string is not stored, an explicit terminator (&0D) is used. Consequently, in the above example, byte &8010 would be set to &0D.

Array storage

The format of an array in memory consists of three parts:

- A heap entry containing the name of the array (excluding the first character) and a pointer to the array descriptor (header).
- An array descriptor containing the number of dimensions and the size of each dimension.
- The array data (integer, byte, variant numeric or string).

The format of the array heap entry is very similar to that of an integer variable, except that instead of a 32-bit data value it consists of a 16-bit pointer (link) to the array descriptor. The heap entry of an array called ARRAY%() would be stored as follows:

LS	MS	R	R	A	Y	%	(	&00	LS	MS

↑ ↑ ← Rest of Name → Array descriptor address

Link to next variable starting with the same letter

Note that the stored name includes the left bracket (parenthesis) and it is this which identifies the heap entry as an array.

The array descriptor is of variable length, and consists of a single byte containing the number of dimensions (suffices) and two bytes for each of the dimensions, containing the size of that dimension (number of rows, columns etc). The array descriptor for the array ARRAY%(10,20) would be stored in memory as follows:

2	11	0	21	0

↑ ← No. of rows → ← No. of cols →

Number of dimensions

Note that the size of each dimension is equal to **one greater than** the suffix specified in the DIM statement, since the index can take any value from zero to the specified maximum suffix.

The array data follows immediately after the array descriptor, the number of elements being equal to the product of the sizes of each dimension. For example in the case of ARRAY%(10,20) the data consists of 11*21 = 231 values. Each data value consists of one byte (in the case of a byte array), four bytes (in the case of a 32-bit integer numeric array or string array) or five bytes (in the case of a 40-bit variant numeric array).