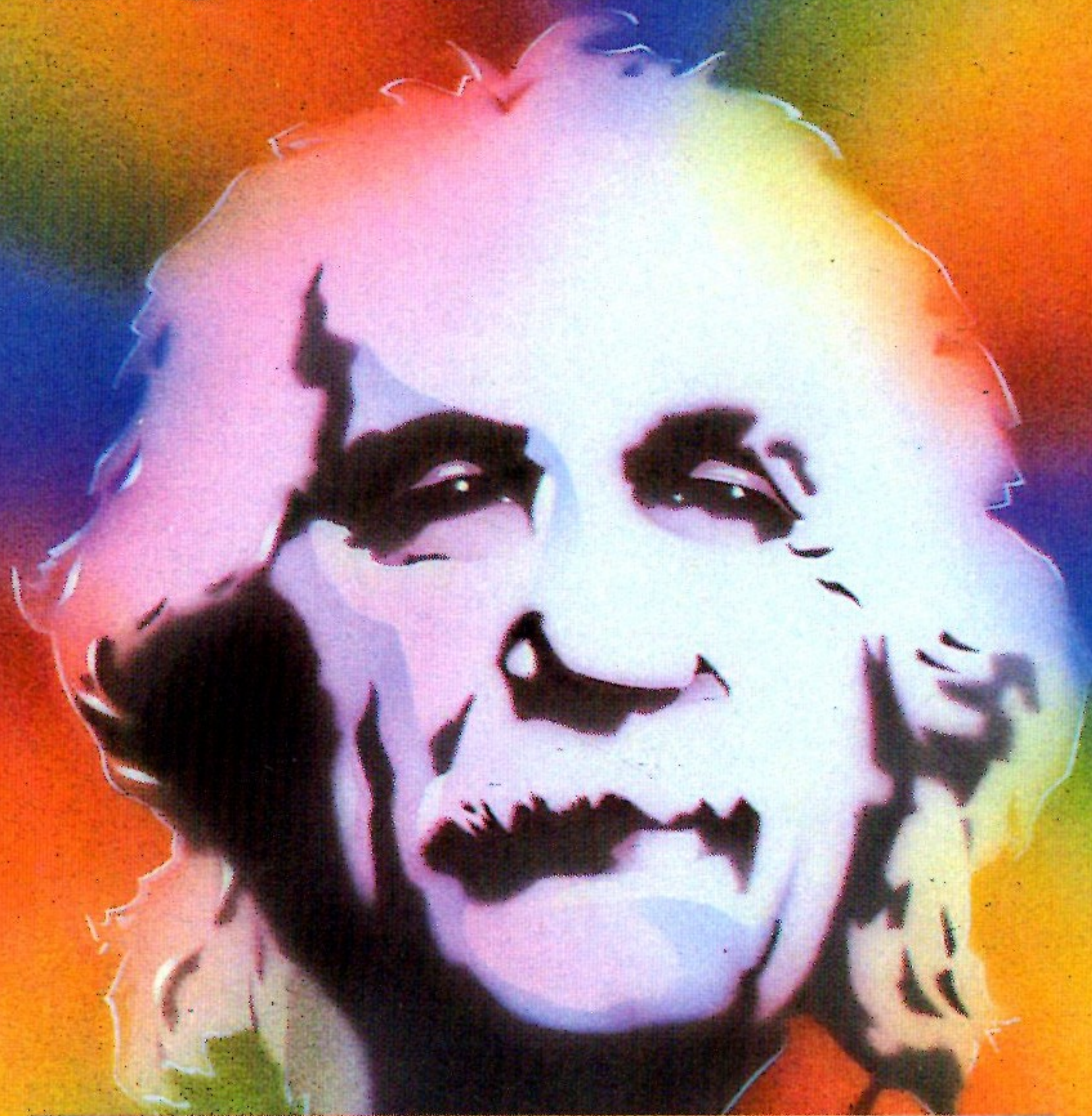


M-TECH SOFT®

BBC BASIC (Z80) Reference Manual



TATUNG
Einstein
COLOUR MICRO COMPUTER

EINSTEIN LANGUAGE MASTER/BBCBASIC(Z80) DISK

The following files are supplied on the BBCBASIC(Z80) distribution disk.

READ.ME
BBCBASIC.COM
HELP.COM
HELP.HLP
BBCBASIC.HLP
CONVERT.COM
UNLIST.COM
RUN.COM
MERGE.BBC
ANIMAL.BBC
ANIMALS.DAT
SORT.BBC
SORTREAL.BBC
SETTIME.BBC
F-?????.BBC

READ.ME A brief introduction for all of you who, like me, insist on trying your programs without reading the instructions.

BBCBASIC.COM The Einstein version of BBCBASIC(Z80).

HELP.COM A public-domain utility for managing and displaying files of information on various subjects (*.HLP files). To run, type:

HELP <RET> for general information on HELP.
HELP BBCBASIC <RET> for more information on BBCBASIC(Z80).

BBCBASIC.HLP The source of the helpful information displayed by HELP

CONVERT.COM A program to convert internal format (tokenised) program files to ASCII files, and vice versa. UNLISTED, SYS and R/O programs cannot be converted. Type CONVERT <RET> for a brief description.

UNLIST.COM	A program to modify BBCBASIC(Z80) programs so they cannot be listed. Type UNLIST <RET> for a brief description.
RUN.COM	When you crash out of any program, typing RUN <RET> will get you back into BBCBASIC(Z80) at the start (100H). Unless it has been corrupted, typing OLD <RET> will then recover your program.
MERGE.BBC	A BBCBASIC(Z80) program which will merge program files. See the notes on BBCBASIC(Z80) DISK I/O for details. UNLISTED programs cannot be merged. MERGE.BBC is provided in the UNLISTED format.
ANIMAL.BBC	An example of a learning program written in BBCBASIC.
ANIMALS.DAT	The data file for ANIMAL.BBC
SORT.BBC	A program demonstrating the use of an assembly language routine for high speed string sorting.
SORTREAL.BBC	A program demonstrating the use of an assembly language routine for the high speed sorting of an array of real numbers.
SETTIME.BBC	A program to set the Einstein's real-time clock.
F-???.BBC	The series of example programs listed in the BBCBASIC(Z80) DISK I/O notes.

CONTENTS

REFERENCE SECTION

INTRODUCTIONHints and Tips for BeginnersGENERAL INFORMATION ABOUT BBCBASIC(Z80)

Control Codes and Functions

Expression Priority

Line Numbers

Statement Separators

Variables

Specification

Types Allowed

Real Variables

Integer Variables

Special (Static) Variables

Boolean Variables

Strings

Arrays

String Variables and Garbage

Garbage Generation

Memory Allocation for String Variables

Numeric Arrays

1
2
5
5
6
6
7
9
9
9
9
9
10
10
10
11
11
11
12

<u>Program Flow Control</u>	13
Loop Operation Errors	13
Program Structure Limitations	13
Leaving Program Loops	14
Local Variables	16
Stack Pointer Control	16
<u>Indirection</u>	17
Introduction	17
Query	18
Exclamation	19
Dollar	20
Use of Binary Operators	21
Power of Indirection Operators	22
<u>Operators and Special Symbols</u>	23
<u>Keywords</u>	25
<u>Error Handling</u>	27
Limitations	31
<u>Procedures and Functions</u>	33
Introduction	33
Names	34
Defining Functions and Procedures	35
Passing Parameters	36
Local (Private) Variables	38

<u>ASSEMBLER</u>	41
<u>Introduction</u>	41
<u>Mnemonics</u>	41
<u>Assembler Statements</u>	41
<u>Labels</u>	42
<u>Comments</u>	42
<u>Program Counter</u>	43
<u>Assembler Listing Control</u>	44
OPT Summary	45
<u>Byte, Word and String Constants</u>	45
<u>The Assembly Process</u>	46
<u>Length of Reserved Memory Area</u>	48
<u>Conditional Assembly and Macros</u>	49
<u>Saving and Loading Machine Code Programs</u>	52
<u>Assembler Limitations</u>	53
<u>Reading the Screen</u>	53
<u>STATEMENTS AND FUNCTIONS</u>	55
ABS	57
ACS	58
ADVAL	59
AND	61
ASC	62
ASN	63
ATN	64
AUTO	65
BGET#	66
BPUT#	67
CALL	68
CHAIN	71
CHR\$	72
CLEAR	73
CLOSE#	74
CLG	75

CLS	76
COLOUR	77
COS	79
COUNT	80
DATA	81
DEF	82
DEG	83
DELETE	84
DIM	85
DIV	87
DRAW	88
EDIT	89
ELSE	91
END	92
ENDPROC	93
ENVELOPE	94
EOF#	96
EOR	97
ERL	98
ERR	99
EVAL	100
EXP	101
EXT#	102
FALSE	103
FN	104
FOR	105
GCOL	107
GET	108
GOSUB	109
GOTO	110
HIMEM	111
IF	112
INKEY	113
INKEY\$	115
INPUT	116
INPUT LINE	118
INPUT#	119
INSTR	120
INT	121
LEFT\$	122
LEN	123
LET	124
LIST	125
LISTO	126
LN	128
LOAD	129
LOCAL	130
LOG	131
LOMEM	132
MID\$	133
MOD	134

MODE	135
MOVE	136
NEW	137
NEXT	138
NOT	139
OLD	140
ON	141
ON ERROR	142
OPENIN	143
OPENOUT	144
OPENUP	145
OPT	146
OR	147
OSCLI	148
PAGE	149
PI	150
PLOT	151
POINT	155
POS	156
PRINT	157
PRINT#	165
PROC	166
PTR#	167
PUT	168
RAD	169
READ	170
REM	171
RENUMBER	172
REPEAT	173
REPORT	174
RESTORE	175
RETURN	176
RIGHT\$	177
RND	178
RUN	179
SAVE	180
SGN	181
SIN	182
SOUND	183
SPC	187
SQR	188
STEP	189
STOP	190
STR\$	191
STRING\$	192
TAB	193
TAN	194
THEN	195
TIME	196
TIMES	196
TO	197

TOP	198
TRACE	199
TRUE	200
UNTIL	201
USR	202
VAL	203
VDU	204
VPOS	205
WIDTH	206

OPERATING SYSTEM COMMANDS

Introduction	207
Syntax	207
Case Conversion	207
Special Characters	207
File Specifiers	208
Symbols	208
*BYE	209
*CHAR	209
*DIR	209
*DISP	209
*DOS	209
*DRIVE	210
*ERA	210
*EXEC	210
*KEY	211
*LOAD	211
*LOCK	211
*MAG	211
*MOS	212
*OPT	212
*PSW	212
*REN	212
*RESET	213
*SAVE	213
*SPOOL	213
*SPRITE	214
*TYPE	214
*UNLOCK	214

BBCBASIC(Z80) DISK FILES

<u>INTRODUCTION</u>	215
<u>THE STRUCTURE OF FILES</u>	217
<u>Basics</u>	217
Serial (Sequential) Files	217
Random Access Files	218
Indexed Files	219
<u>Files in BBCBASIC(Z80)</u>	220
Introduction	220
HowDataisRead/Written	221
How Numeric Data is Stored	221
Numeric Data	221
How Strings are Stored	222
The Limitations of CP/M	222
BBCBASIC(Z80) File Format	222
Overcoming the Limitations of CP/M	223
Examples	223
<u>DISK FILE COMMANDS</u>	224
Introduction	224
Conventions	224
Organization of Examples	224
<u>Program File Manipulation</u>	224
SAVE	225
LOAD	225
CHAIN	226
MERGE	227
*ERA	228
*REN	229

*DIR	229
UNLIST	230
<u>Disk Data Files</u>	231
Introduction	231
Opening Files	231
OPENOUT	232
OPENIN	233
OPENUP	233
INPUT#	234
PRINT#	235
CLOSE#	236
EOF#	237
PTR#	238
BGET#	239
BPUT#	239
EXT#	240
<u>SERIAL FILES</u>	241
<u>Character Data Files</u>	241
Example 1 - Writing Serial Character Data	242
Example 2 - Reading Serial Character Data	243
Example 3 - Writing ~AT END~ of Character Files	243
<u>Mixed Numeric/Character Data Files</u>	245
Example 4 - Writing a Mixed Data File	245
Example 5 - Reading a Mixed Data File	248
Example 6 - Writing ~AT END~ of Mixed Files	249
<u>Compatible Data Files</u>	252
Example 7 - Writing a Compatible Data File	252
Example 8 - Reading a Compatible Data File	254
<u>RANDOM (RELATIVE) FILES</u>	257
Example 1 - Simple Random Access File	257
Example 2 - Simple Random Access Database	258
Random File Initialisation	262
Example 3 - Random Access Inventory Program	262

<u>INDEXED DATA FILES</u>	275
Deficiencies of Random Files	275
The Address Book Program	275
File organization	275
Program Organization	276
The Index	276
Example the Last - Indexed File Address Book	277
<u>THE BINARY CHOP</u>	289
<u>Explanation</u>	289
<u>Binary Chop Flow Chart</u>	291
<u>ANNEXES</u>	
ASCII Codes	Annex A
Mathematical Functions	Annex B
Error Codes and Messages	Annex C
Syntax	Annex D
Format of Program and Variables in Memory	Annex E

EINSTEIN BBCBASIC (Z80)

INTRODUCTION

1. This is a reference manual, it is not intended to teach you BASIC. It gives a summary of the commands and functions plus some hints and tips on their use. It also describes the minor differences between the Acorn 6502 and Z80 versions of the BASIC. A general knowledge of BASIC has been assumed. If you are a newcomer to programming you should read one of the many books on BBCBASIC.
2. However, not a lot has been written on file handling with BBCBASIC(Z80), so an explanation of BBCBASIC(Z80) file handling has been included.
3. Please read the Einstein Introduction and DOS/MOS manuals before you try to use your computer seriously. We have included some (very) basic hints for beginners later in this section and these should be sufficient to get you going. However, if you are going to make the best use of your computer, you need to understand what you are doing. Understanding does not come easily, but if you study the documentation and try things out for yourself you will be well rewarded.
4. The Einstein version of BBCBASIC(Z80) is supplied on a 3" compact floppy disk. It runs under the Tatung/Xtal DOS (Disk Operating System). The name of the file is BBCBASIC.COM.
5. Einstein BBCBASIC(Z80) is fully configured for the Einstein and all the statements and functions specified for BBCBASIC are available. It has been designed to be as compatible as possible with the 6502 version resident in the BBC Micro. However, the difference in the hardware, particularly in the area of

graphics and sound, makes full compatibility impossible and programs which explore the full capabilities of the BBC Micro in these areas will not run on the Einstein without modification. In addition, any machine code routines will need to be converted from 6502 to Z80 assembler code. The language syntax is not always completely identical to that of the 6502 version, but in most cases the Einstein version is more tolerant (For example, both a comma and a semi-colon are accepted after the prompt string in an INPUT statement).

6. To run BBCBASIC, place your BBCBASIC(Z80) working disk in drive 0: bring up the DOS as described in the Introduction to the Einstein DOS/MOS Introduction, and type the following:

```
A>BBCBASIC<ENTER>
```

After a couple of seconds, the system will reply:

```
Einstein BBCBASIC Version 2.31
(C) Copyright R.T.Russell 1984
>
```

7. Alternatively a filename may be given after BBCBASIC, in which case the system proceeds as if a CHAIN "filename" command had been typed after initialisation. A default extension of .BBC is used if none is supplied. This feature allows BASIC programs to be executed in batch mode.

HINTS AND TIPS FOR BEGINNERS

8. Within the space available, we can't go into details of what an operating system is, how programs run, etc. What we can do is to go through the things you need to know to get something working on your computer. Your Einstein User Guide gives an explanation of what the Disk Operating System (DOS) does, what the 'built in' commands do, how to load and run a program, etc - please read it.

9. The Einstein's <ENTER> key is in the same place (and serves the same purpose) as the key labeled <RETURN> on many computers. The Einstein also uses the divide sign (÷) in place of the more usual tilde (~). Throughout this manual <ENTER> and <RETURN>, and ~ and ÷ are interchangeable.

10. Before you start, are you sure that your computer is working? Good! Plug it in and turn it on. The message illustrated in your Introduction to the Einstein Colour Micro Computer should appear on the screen. If you don't get the illustrated message, your computer may not be working correctly. Please ask your dealer to check it (and, if necessary, explain why the display looks different).

11. Put your BBCBASIC(Z80) master disk in Drive 0: and press both the CTRL and BREAK keys at the same time. The DOS will load and the following message (or something very like it) should appear on the screen.

```
*** EINSTEIN ***
```

```
TATUNG/XtalDOS 1.##
```

```
(c)1983 1984
```

```
0:■
```

12. Look up 'Backup' under 'Utilities' in the Einstein DOS/MOS Introduction. This will tell you how to duplicate your master disk. Duplicate your BBCBASIC(Z80) master disk and put it away. Do NOT use your master disk for day-to-day work.

13. Put the working copy of the BBCBASIC(Z80) disk in drive 0: and type

```
DIR<ENTER>    <ENTER> means 'the enter key' NOT the
                characters <, E, N, T, E, R, and >.
```

The names of the files on the disk will be listed on the screen.

14. There are many types of file on the disk. Some of them are text files (they contain words). To read a text file type

```
DISP filename<ENTER>
```


15. The text file you should read now is called READ.ME. So, type

DISP READ.ME<ENTER>

and the file will be listed on the screen. It will flash past so fast that you will have trouble reading it. To stop the screen 'scrolling', push the <CTRL> and 'S' keys together. To start it again, press <CTRL> and 'S' again. Alternatively, you can use the break key to pause the scrolling.

16. Now, to get BBCBASIC(Z80) up and running, follow the instructions in paragraph 7 (the BBCBASIC disk is in drive 0: isn't it?). The 'prompt' has changed to '>' and this indicates that BBCBASIC(Z80) has taken control from the operating system.

17. If you now want to read any of the text files, you should type

*DISP filename<ENTER>

(See the section on Operating System Commands for more details.)

18. Now, at last, the time has come to run a BBCBASIC(Z80) program. Type

LOAD "F-INDEX"<ENTER>

followed by

RUN<ENTER>

and you will be able to 'play' with the address book program. If you want to look at the program before running it, type

LIST<ENTER>

instead of RUN. You will be able to control the scrolling as described earlier.

19. The rest is up to you. Have fun.

GENERAL INFORMATION ABOUT BBCBASIC(Z80)

1. Control Codes and Functions.

a. Generation. Control codes are generated by holding down the <CTRL> key and pressing another key. Thus 'control A' is generated by holding the <CTRL> key down and pressing <A>. Since phrases like 'control A' are a little clumsy, we have used the generally accepted convention of preceding the character with the '^' symbol to indicate it is a control code. Thus 'control C' is shown as ^C, etc.

b. What Happens to Control Codes? Unlike the BBC Micro, all control codes other than those which have a control or edit function are filtered out and have no effect. This is compatible with most other DOS (and CP/M) programs, but it may cause you minor inconvenience. If you wish to send control codes to the VDU whilst in the direct mode, you can do so using the VDU command. For instance VDU 26 will send ^Z to the VDU.

c. Control Functions.

Escape (^[]) - Program/Command/Edit Abort.

^A - Dump screen to printer.

^P - Toggle printer.

^S - Stop and start output to output stream (see *OPT).

If escape is sent using VDU 27, it is ignored.

d. Editing Functions.

← Backspace the cursor one character position.

→ Advance the cursor one character position.

↑ Move the cursor to the line start.

↓ Move the cursor to the line end.

INS Insert a space at the cursor position.

CTRL F Delete the character at the cursor position.

DEL Backspace and delete.

CTRL U Delete from the cursor to the line end.

CTRL X Delete from left of the cursor to the line start.

Expression Priority

2. 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.

- (1) variables functions () ! ? & unary + - NOT
- (2) ^
- (3) * / MOD DIV
- (4) + -
- (5) = <> <= >= > <
- (6) AND
- (7) EOR OR

3. 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)

Line Numbers

4. Line numbers up to 65535 are allowed. If line 65535 does not exist, then GOTO 65535 is equivalent to END. Line number 0 is not permitted.

Statement Separators

5. When it is necessary to write more than one statement on a line, the statements may be separated by a colon ':'. BBCBASIC(Z80) will tolerate the omission of the separator if this does not lead to ambiguity. It's safer to leave it in.

VARIABLES

1. Specification. Variable names may be of unlimited length and all characters are significant. Variable names must start with a letter. They can only contain the characters A..Z, a..z, 0..9 and underline. Embedded keywords are allowed. Upper and lower case variables of the same name are different.

2. Types Allowed. The following types of variable are allowed:

A	real numeric
A%	integer numeric
A\$	string

3. Real Variables. Real variables have a range of $\pm 5.9 \times 10^{-39}$ to $\pm 3.4 \times 10^{38}$ 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 E.

4. Integer Variables. Integer variables are stored in 32 bits and have a range of ± 2147483647 to ± 2147483648 . 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.

5. Special (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%, F%, H% and L% (plus X% and Y% on the Torch) 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'.

6. Boolean Variables. Boolean variables can only take one of the 2 values TRUE or FALSE. Unfortunately, BBCBASIC 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      The argument, 1.5, is truncated to 1
      -2             and the logical inversion of this
>_                  gives -2.

>PRINT NOT -1.5     The argument is truncated to -1 and
      0              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.

7. Strings. String variables may contain up to 255 characters. An explanation of how variables are stored is given at Annex E.

8. Arrays. Arrays of integer, 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.

String Variables and Garbage

9. Garbage Generation. Unlike numeric variables, string variables do not have a fixed length. When you create a string variable, the memory used is sufficient for the initial value of the string. If you subsequently assign a longer string to the variable there will be insufficient room for it and the string will have to occupy a different area in memory. The initial area will then become 'dead'. These areas of 'dead' memory are called garbage. As more and more re-assignments take place, the area of memory used for the variables grows and eventually there is no more room. Several versions of BASIC have automatic 'garbage collection' routines which tidy up the variable memory space when this occurs. Unfortunately, this can take several seconds and can be embarrassing if your program is time conscious. BBCBASIC does not incorporate 'garbage collection' routines and it is possible to run out of room for variables even though there should be enough space.

10. 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 ready for use.

```
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.

Numeric Accuracy.

11. 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.

PROGRAM FLOW CONTROL

1. Unlike the BBC Micro, which has separate stacks for FOR...NEXT, REPEAT...UNTIL and GOSUB...RETURN operations, BBCBASIC(Z80) uses a single control stack (the processor's hardware stack) for all looping and nesting operations. The main effects of this difference are discussed below.

2. Loop Operation Errors. Apart from memory size, there is no limit to the level of nesting of FOR...NEXT, REPEAT...UNTIL and GOSUB...RETURN operations. The untrappable error message 'No room' will be issued if all the stack space is used up. Because a single stack is used the following error messages do not exist.

Too many FORs
Too many REPEATs
Too many GOSUBs

3. Program Structure Limitations. The use of a common stack has one disadvantage (if it is a disadvantage) in that it forces stricter 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, a RETURN statement, BBCBASIC(Z80) will report an error. (In this case, a 'No GOSUB at line nnnn' error.) The example below would result in the error message 'No PROC at line 500'.


```

400 - - -
410 INPUT "What number should I stop at", num
420 PROC_error_demo
430 END
440 :
450 DEF PROC_error_demo
460 FOR i=1 TO 100
470   PRINT i;
480   IF i=num THEN 500
490 NEXT i
500 ENDPROC

```

BBCBASIC(Z80) is a little unusual in detecting this error, but it is always risky. It usually results in an inconsistent program structure and an unexpected 'Too many FORs/REPEATs/GOSUBs' error when the control stack overflows.

4. 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.

a. REPEAT...UNTIL Loops. The simplest 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.

```

400 - - -
410 INPUT "What number should I stop at", num
420 PROC_error_demo
430 END
440 :
450 DEF PROC_error_demo
460 i=0
470 REPEAT
480   i=i+1
490   PRINT i;
500 UNTIL i=100 OR i=num
510 ENDPROC

```

b. Changing the Loop Variable. A simple way of forcing an exit from a FOR...NEXT loop is to set the loop variable to a value equal to the limit value and then GOTO to the NEXT statement. 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 may be an advantage.) The example below uses this method to exit from the loop. Notice, however, that the conditions which cause the loop to terminate are less clear since they do not appear together.

```

400 - - -
410 INPUT "What number should I stop at", num
420 PROC_error_demo
430 END
440 :
450 DEF PROC_error_demo
460 FOR i=1 TO 100
470   PRINT i;
480   IF i=num THEN i=500: GOTO 510
490   ....
500   More program here if necessary
510 NEXT
520 ENDPROC

```

c. Popping the Inner Variable. A less satisfactory way of exiting a FOR...NEXT loop is to enclose the loop in a dummy outer loop and rely on BBCBASIC(Z80)'s ability to 'pop' inner control variables off the stack until they match. If you use this method you MUST include the variable name in the NEXT statement. This method, which is demonstrated below, is very artificial and the conditions which cause the loop to terminate are unclear.


```

400 - - -
410 INPUT "What number should I stop at", num
420 PROC_error_demo
430 END
440 :
450 DEF PROC_error_demo
460 FOR dummy=1 TO 1 :REM Loop once only
470   FOR i=1 TO 100
480     PRINT i;
490     IF i=num THEN 530 :REM Jump to outer NEXT
500     - - -
510     More program here if necessary
520   NEXT i
530 NEXT dummy
540 ENDPROC

```

5. Local Variables. 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 the error message 'Not LOCAL at line 400'.

```

380 DEF PROC_error_demo
390 FOR i=1 TO 10
400   LOCAL data(i)
410 NEXT
420 ENDPROC

```

You can overcome this by fabricating the loop using an IF...THEN statement as shown below. This is probably the only occasion when the use of a single stack promotes poor program structure.

```

380 DEF PROC_error_demo
390 i=1
400 LOCAL data(i)
410 i=i+1
420 IF i<11 THEN 400
430 ENDPROC

```

6. Stack Pointer Control. The stack is initialised to begin at HIMEM and, because of this, you cannot change the value of HIMEM when there is anything on the stack. As a result, you cannot change HIMEM from within a procedure, function, subroutine, FOR...NEXT loop or REPEAT...UNTIL loop.

INDIRECTIONIntroduction.

1. Most versions of BASIC allow access to the computer's memory with the PEEK function and the POKE command. 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 BBCBASIC(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.

2. There are 3 indirection operators:

Name	Purpose	No of Bytes Affected
Query '?'	Byte Indirection Operator	1
Exclamation '!'	Word Indirection Operator	4
Dollar '\$'	String Indirection Operator	1 to 256

Query.

3. ?M means 'the contents of' memory location 'M'. The first 2 examples below write &23 to memory location &4FA2, the second 2 examples set 'number' to the contents of that memory location and the third 2 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
```

4. 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)
```

5. 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 A% 0
FOR ?A%=0 TO 20
  - - -
  - - -
NEXT
```

Exclamation.

6. The query (?) indirection operator works on one byte of memory. 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
and &12 into address M+3.
```

and

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

would give

```
12345678
```


Dollar.

7. 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 Operators.

8. 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 2 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)
```

9. The 2 examples below show how 2 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 annex F.)

```
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 Annex E for an explanation of this program.

10. 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.

OPERATORS AND SPECIAL SYMBOLS

- ? A unary and binary operator giving 8 bit indirection:
- ! A unary and binary 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).
- \$ A character indicating that the object has something to do with a string. The syntax \$<expression> may be used to position a string anywhere in memory, overriding the interpreter's space allocation. As a suffix on a variable name it indicates a string variable.
- \$A="WOMBAT" :REM store WOMBAT at address A followed by CR.
- % A suffix on a variable name indicating an integer variable.
- & Precedes hexadecimal constants e.g. &EF
- ' A character which causes newlines 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.
- Unary negation and binary 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 binary 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.
- < Relation of less than on integers, reals and strings.
- > Relation of greater than on integers, reals and strings.
- / 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 exponentiation between integers and reals.
- ~ A character in the start of a print field indicating that the item is to be printed in hexadecimal.

KEYWORDS

1. Keywords are recognized before anything else (e.g. both DEG and ASN in DEGASN are recognized, but neither in ADEGASN). Consequently, you don't have to type a space between a keyword and a variable, but please avoid this disgusting habit. Pseudo variables (PI, LOMEM, HIMEM, PAGE, TIME, etc.) act as variables in that if PI is a (pseudo-) variable then it does not affect PILE (or if A is a variable, then AB can be). However, PI%, PI\$ etc. are not allowed. Since variables named in lower case will never be confused with keywords, many programmers use upper case only for keywords.

2. The following list of BBCBASIC's 123 keywords includes those applicable to graphics capable computers (bracketed). Ninety-three of the keywords are not allowed in upper case at the start of a variable name (anything may be used in lower case). These keywords are underlined.

3. Because most computers running CP/M lack a generally defined graphics capability, none of the graphics and sound keywords (bracketed) are available in the general purpose version of BBCBASIC(Z80).

4. However, the version supplied for the Torch computers and disk-packs and the Wren computer implements ALL the keywords listed below, and the versions provided with some other computers have a subset of the graphics commands.

5. Because of different hardware configurations not all computers perform identical functions for some of the graphics and sound keywords. Please refer to the information provided with your computer for a list of the graphics and sound commands which are applicable and the precise detail of their function.

KEYWORDS AVAILABLE

<u>ABS</u>	<u>ACS</u>	<u>(ADVAL)</u>	<u>AND</u>	<u>ASC</u>
<u>ASN</u>	<u>ATN</u>	<u>AUTO</u>	<u>BGET</u>	<u>BPUT</u>
<u>CALL</u>	<u>CHAIN</u>	<u>CHR\$</u>	<u>CLEAR</u>	<u>(CLG)</u>
<u>CLOSE</u>	<u>CLS</u>	<u>(COLOUR)</u>	<u>(COLOR)</u>	<u>COS</u>
<u>COUNT</u>	<u>DATA</u>	<u>DEF</u>	<u>DEG</u>	<u>DELETE</u>
<u>DIM</u>	<u>DIV</u>	<u>(DRAW)</u>	<u>ELSE</u>	<u>END</u>
<u>ENDPROC</u>	<u>(ENVELOPE)</u>	<u>EOF</u>	<u>EOR</u>	<u>ERL</u>
<u>ERR</u>	<u>ERROR</u>	<u>EVAL</u>	<u>EXP</u>	<u>EXT</u>
<u>FALSE</u>	<u>FN</u>	<u>FOR</u>	<u>(GCOL)</u>	<u>GET</u>
<u>GET\$</u>	<u>GOSUB</u>	<u>GOTO</u>	<u>HIMEM</u>	<u>IF</u>
<u>INKEY</u>	<u>INKEY\$</u>	<u>INPUT</u>	<u>INSTR(</u>	<u>INT</u>
<u>LEFT\$(</u>	<u>LEN</u>	<u>LET</u>	<u>LINE</u>	<u>LIST</u>
<u>LN</u>	<u>LOAD</u>	<u>LOCAL</u>	<u>LOG</u>	<u>LOMEM</u>
<u>MID\$(</u>	<u>MOD</u>	<u>(MODE)</u>	<u>(MOVE)</u>	<u>NEW</u>
<u>NEXT</u>	<u>NOT</u>	<u>OFF</u>	<u>OLD</u>	<u>ON</u>
<u>OPENIN</u>	<u>OPENOUT</u>	<u>OPENUP</u>	<u>OR</u>	<u>OSCLI</u>
<u>PAGE</u>	<u>PI</u>	<u>(PLOT)</u>	<u>(POINT(</u>	<u>POS</u>
<u>PRINT</u>	<u>PROC</u>	<u>PTR</u>	<u>PUT</u>	<u>RAD</u>
<u>READ</u>	<u>REM</u>	<u>RENUMBER</u>	<u>REPEAT</u>	<u>REPORT</u>
<u>RESTORE</u>	<u>RETURN</u>	<u>RIGHT\$(</u>	<u>RND</u>	<u>RUN</u>
<u>SAVE</u>	<u>SGN</u>	<u>SIN</u>	<u>(SOUND)</u>	<u>SPC</u>
<u>SQR</u>	<u>STEP</u>	<u>STOP</u>	<u>STR\$</u>	<u>STRING\$(</u>
<u>TAB(</u>	<u>TAN</u>	<u>THEN</u>	<u>TIME</u>	<u>TO</u>
<u>TRACE</u>	<u>TRUE</u>	<u>UNTIL</u>	<u>USR</u>	<u>VAL</u>
<u>VDU</u>	<u>VPOS</u>	<u>WIDTH</u>		

ERROR HANDLING

1. Once you have written your program and removed all the syntax errors, you might think that your program is error free. Unfortunately life is not so simple, you have only passed the first hurdle. There are 2 kinds of errors which you could still encounter; errors of logic and run-time errors. Errors of logic are where BBCBASIC(Z80) understands exactly what you said, but what you said is not what you meant. Run-time errors are where something occurs during the running of the program which BBCBASIC(Z80) is unable to cope with. For example,

answer=A/B

is quite correct and it will work for all values of A. But if B is zero, the answer is 'infinity'. BBCBASIC(Z80) has no way of dealing with 'infinity' and it will report a 'Division by zero' error.

2. There is no way that BBCBASIC(Z80) can trap errors of logic, since it has no way of understanding what you really meant it to do. However, you can generally predict which of the run-time errors are likely to occur and include a special 'error handling' routine in your program to recover from them.

3. Why would you want to take over responsibility for handling run-time errors? When BBCBASIC(Z80) detects a run-time error, it reports it and RETURNS TO THE COMMAND MODE. 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 BBCBASIC(Z80) might terminate the program and return to the command mode. However, when somebody else uses your program they are not blessed with your insight and they may find the program 'crashing out' to the command mode without knowing what they have done wrong. Such programs are called 'fragile'. You can protect your user from much frustration if you predict what these problems are likely to be and include an error handling routine. In the example below, a '-ve root' error would occur if the number input was negative and BBCBASIC(Z80) would return to the command mode.


```

10 REPEAT
20  INPUT "Type in a number " num
30  PRINT num," ",SQR(num)
40  PRINT
50 UNTIL FALSE      :REM A way of looping until the ESCAPE
60 :                REM key is pressed

```

Example run:

```

>RUN
Type in a number 5
      5          2.23606798

Type in a number 23
      23         4.79583152

Type in a number 2
      2          1.41421356

Type in a number -2
      -2
-ve root at line 30

```

4. The main command provided by BBCBASIC(Z80) for error trapping and recovery is:

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

In addition, there are 2 functions, ERR and ERL, 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.

a. ERR returns the error number (see Annex C).

b. 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 READ statement, not the number of the line containing the DATA.

c. REPORT prints out the error string associated with the last error which occurred.

5. The example below does not try to deal with errors, it just uses ERR, ERL and REPORT to tell the user about the error. Its only advantage over BBCBASIC(Z80)'s normal error handling is that it gives the error number and it would probably not be used in practice. As you can see from the second run, pressing <ESCAPE> is treated as an error (number 17).

```

5 ON ERROR GOTO 100
10 REPEAT
20  INPUT "Type a number " num
30  PRINT num," ",SQR(num)
40  PRINT
50 UNTIL FALSE
60 :
70 :
100 PRINT
110 PRINT "Error No ";ERR
120 REPORT:PRINT " at line ";ERL
130 END

```

Example run:

```

>RUN
Type in a number 1
      1          1

Type in a number -2
      -2
Error No 21
-ve root at line 30

>RUN
Type in a number <ESCAPE>
Error No 17
Escape at line 20

```

6. 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 <ESCAPE> 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.

```

5 ON ERROR GOTO 100
10 REPEAT
20   INPUT "Type a number " num
30   PRINT num, " ", SQR(num)
40   PRINT
50 UNTIL FALSE
60 :
70 :
100 PRINT
110 IF ERR=21 THEN PRINT "No negative numbers":GOTO 10
120 REPORT:PRINT " at line ";ERL
130 END

```

Example run:

```

>RUN
Type a number 5
      5      2.23606798

Type a number 2
      2      1.41421356

Type a number -1
      -1
No negative numbers
Type a number 4
      4      2

Type a number <ESCAPE>
Escape at line 20

```

7. The above example is very simple and was chosen for clarity. In practise, it would be better to test for a negative number before using SQR rather than trap the '-ve root' error. A more realistic example is the evaluation of a user-supplied HEX number, where trapping 'Bad hex' would be much easier than testing the input string beforehand.

```

10 ON ERROR GOTO 100
20 REPEAT
30   INPUT "Type a HEX number " input$
40   num=EVAL("&"+input$)
50   PRINT input$,num
60   PRINT
70 UNTIL FALSE
80 :
90 :
100 PRINT
110 IF ERR=28 THEN PRINT "Not a hexadecimal number":GOTO 20
120 REPORT:PRINT " at line ";ERL
130 END

```

Limitations

7. Apart from the obvious one of trying to out-guess the user, there is one problem with error trapping. When an error is detected, the stack is cleared down. Consequently, you cannot return back to a FOR...NEXT or REPEAT...UNTIL loop or to a procedure, function or subroutine. In addition, if an error occurred within a procedure or function, the value of any PRIVATE variables will be the last value they were set to within the procedure or function which gave rise to the error. Thus, error recovery cannot be as complete as might be desirable.

PROCEDURES AND FUNCTIONSIntroduction

1. Procedures and functions are similar to 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 2 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.

2. Arguably, the major advantage of procedures and functions is that they can be referred to by name. Consider the 2 similar program lines below.

```
100 IF name$="ZZZZ" THEN GOSUB 1000 ELSE GOSUB 2000
```

```
100 IF name$="ZZZZ" THEN PROC_end ELSE PROC_print_details
```

The first statement gives no indication of what the subroutines at 1000 and 2000 actually do. The second, however, tells you what to expect from the 2 procedures. This enhanced readability stems from the choice of meaningful names for the 2 procedures.

3. 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.

4. Whilst BBCBASIC(Z80) 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

5. The names of procedures and functions MUST start with PROC or FN and, like variable names, they cannot contain spaces. (A space tells BBCBASIC(Z80) that it has reached the end of the word.) This restriction can give rise to some pretty unreadable names. However, the underline character (which displays as a long hyphen in the TELETXT character set) 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.

Defining Functions and Procedures

6. Functions and procedure definitions are 'signalled' to BBCBASIC(Z80) 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.

7. The 'body' of a procedure or function must not be executed directly - they must be performed (called) by another part of the program. Since BBCBASIC(Z80) 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.

8. 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.

9. 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 2 numbers. The second defines a procedure which clears from the current cursor position to the end of line on a 40 column screen.

```
500 DEF FN_average(n1,n2)=(n1+n2)/2
```

↑
| Expression to right of = sign
↑
Signals the end of the definition

```
120 DEF PROC_clear:PRINT SPC(40-POS);:ENDPROC
```


10. 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, BBCBASIC(Z80) does not have a 'clear to end of screen' command. Some computers will perform this function on receipt of a sequence of control characters and in this case you can use VDU or CHR\$ to send the appropriate codes. However, many computers (the Torch amongst them) do not have this facility and a procedure to clear to the end of the screen would be useful. The example below is a procedure to clear to the end of screen on a computer with an 80 by 24 display. In order to prevent the display from scrolling, you must not write to the last column of the last row. The 3 variables used (i, x, and y) are declared as LOCAL to the procedure (see later).

```
100 DEF PROC_clear_to_end
110 LOCAL i,x,y
120 x=POS:y=VPOS
130 REM If not last line, print lines of spaces which
140 REM will wrap around and end up on last line.
150 IF y<23 THEN FOR i=y TO 22:PRINT SPC(80);:NEXT
160 REM Print spaces to end-1 of last line.
170 PRINT SPC(79-x);
180 PRINT TAB(x,y);
190 ENDPROC
```

Passing Parameters

11. 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_discount(price,percent)=price*(1-percent/100)
```

In this case, to use the function we would need to pass 2 parameters.

```
90 ....
100 retail_price=26.55
110 discount_price=FN_discount(retail_price,25)
120 ....
or
90 ....
100 price=26.55
110 discount=25
120 price=FN_discount(price,discount)
130 ....
or
90 ....
100 price=FN_discount(26.55,25)
110 ....
```

12. The value of the first parameter in the line using 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'. The parameters declared in the definition are called 'formal parameters' and the values passed in the lines 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. You can pass a mix of string and numeric parameters to the same 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 'Arguments at line 10' error message and the third would cause a 'Type mismatch at line 10' error to be reported.

a. Correct.

```
10 PROC_printit(1,"FRED",2)
20 END
30 :
40 DEF PROC_printit(num1,name$,num2)
50 PRINT num1,name$,num2
60 ENDPROC
```


b. Arguments Error.

```

10 PROC_printit(1,"FRED",2,4)
20 END
30 :
40 DEF PROC_printit(num1,name$,num2)
50 PRINT num1,name$,num2
60 ENDPROC

```

c. Type Mismatch.

```

10 PROC_printit(1,"FRED","JIM")
20 END
30 :
40 DEF PROC_printit(num1,name$,num2)
50 PRINT num1,name$,num2
60 ENDPROC

```

Local (Private) Variables

13. You can use the statement LOCAL to define variables which are private to individual procedures and functions. In addition, formal parameters are private to the procedure or function declaring them. These variables are only known locally to the defining procedure or function. They are not known to the rest of the program and they can only be changed from within the procedure or function where they are defined. Consequently, you can have 2 variables of the same name, say FLAG, in various parts of your program, and change the value of one without changing the other. This technique is used extensively in the example file handling programs in the second part of this manual.

14. Declaring variables as local, creates them locally and initialises them to zero/null.

15. Variables which are not formal variables or declared as LOCAL are known to the whole program, including all the procedures and functions. Such variables are called GLOBAL

16. Because the formal parameters which receive the passed parameters are private, all procedures and functions can be re-entrant. That is, they can call themselves. But for this feature, the short example program below would be very difficult to code. It is the often used example of a factorial number routine. (The factorial of a number n is $n * n-1 * n-2 * \dots * 1$. Factorial 6, for instance, is $6*5*4*3*2*1$)

```

10 REPEAT
20   INPUT "Give me an INTEGER number less than 35 "num
30   UNTIL INT(num)=num AND num<35
40   fact=FN_fact_num(num)
50   PRINT num,fact
60   END
70:
80   DEF FN_fact_num(n)
90   IF n=1 OR n=0 THEN =1   REM Return with 1 if n= 0 or 1
100  =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 private 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.

ASSEMBLERINTRODUCTION

1. BBCBASIC(Z80) has a Z80 assembler. It is accessed in the same way as the 6502 assembler on the BBC Micro. That is, '[' enters assembler mode and ']' exits assembler mode.

MNEMONICS

2. 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 number in 'IN' and 'OUT' are optional. Thus, both OUT (5),A and OUT 5,A are accepted. The pseudo-operations 'DEFB', 'DEFW' and 'DEFM' are included. DEFM is like EQU in the 6502 version. The instruction 'IN F,(C)' is not accepted but the equivalent object code is produced from 'IN (HL),(C)'.

ASSEMBLER STATEMENTS

3. An assembly language statement consists of 3 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.

4. Statements are terminated by a colon (:) or end of line (<RET>).

LABELS

5. Any BBCBASIC(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%).

6. In the example in paragraph 20, one external and one internal label are defined and used. Labels have the same rules as standard BBCBASIC variable names. Consequently, they should start with a letter and not start with a keyword.

COMMENTS

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

```
[;start assembler language program
etc
LD A,B ;An in-line comment:POP HL ;get start address
RET NZ ;Return if all done:JR loop ;else go back
etc
;end assembler language program:]
```

PROGRAM COUNTER

8. 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 uses the assembler. 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.

9. 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 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%
```

10. If you are going to use a machine code program in a number of your BBCBASIC(Z80) programs, the simplest way is to assemble it once, save it using *SAVE and load it from each of your programs using *LOAD. In order 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 BBCBASIC(Z80) program. However, altering LOMEM destroys ALL your dynamic variables (including file buffers) and is more risky.

11. Warning. The program counters, P%, and O% are initialised to zero. Using the assembler without first setting P% (and O%) is liable to corrupt the operating system area in page 0.

ASSEMBLER LISTING CONTROL

12. 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 3 bits of this number in the following manner:

- a. Bit 0 - LSB. Bit 0 controls the listing. If it is set, a listing is displayed.
- b. Bit 1. Bit 1 controls the error reporting. If it is set, errors are reported.
- c. 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%.

13. 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.

14. OPT Summary.

a. Code Assembled Starting at P%.

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.

c. Code Assembled Starting at 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.

BYTE, WORD AND STRING CONSTANTS

15. You can store constants within your assembler language program using the DEFB, DEFW and DEFM pseudo-operation commands.

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

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

will set 2 consecutive bytes of memory to 15 and 9 (decimal). The address of the first byte will be stored in the variable 'data'.

17. DEFW. DEFW can be used to set 2 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 example for DEFB.

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

```
.string    DEFM "This is a test message"
           DEFB &D
```

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
.string; leave assembler 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 pass*2
.continue; and carry on with assembler
```

THE ASSEMBLY PROCESS

19. The assembler works line by line through the machine code. When it finds a label declared it generates a BBCBASIC(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 BBCBASIC(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 2 passes through a (completely futile) demonstration program. Ten bytes of memory are reserved for the program. (If the program was run, it would 'doom-loop' from line 40 to 60 and back again.) The program disables error reporting by using OPT 1.

```
10 DIM code 20
20 P%=code
30 [OPT 1
40 .jim JP fred
50 DEFW &2345
60 .fred JP jim:]
```

This is the first pass through the assembly process.

```
>RUN
3B5E                                OPT 1
3B5E C3 5E 3B      .jim JP fred - Note that the JP
3B61 45 23          DEFW &2345      instruction jumps
3B63 C3 5E 3B      .fred JP jim    to itself.
```

This is the second pass through the assembly process.

```
3B5E                                OPT 1
3B5E C3 63 3B      .jim JP fred - Note that the JP
3B61 45 23          DEFW &2345      instruction jumps
3B63 C3 5E 3B      .fred JP jim    to the correct
                                   address.
```

20. Generally, if labels have been used, you must make 2 passes through the assembler 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 (OPT 7) 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 op-code. The correct address will be generated during the second pass.

LENGTH OF RESERVED MEMORY AREA

21. 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.

22. In the example below, a large amount of memory is initially reserved. The assembler is allowed to run once and the length of the assembly code determined (lines 100 to 120). After a CLEAR, the correct amount of memory is reserved (line 140) and a further 2 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.

```
- - -
10000 DEF PROC_ass(opt)
10010 P%=code
10020 [OPT opt
- - -
Assembler code program.
- - -
11000 ]
11010 ENDPROC
```

CONDITIONAL ASSEMBLY AND MACROS

23. Introduction. Most machine code assemblers allow conditional assemble and macro facilities. The assembler does not directly offer these features, but it is possible to implement them by using the facilities offered by BBCBASIC(Z80).

24. Optional 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 assembler 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 -:] ELSE[OPT pa
ss: - code for routine 2 -:]
:
  [OPT pass
  .more_code - - -
    - - - code - - -
    - - - :]
NEXT

```

25. 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 a 'normal CP/M' output routine, or one applicable to the Torch.

```

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

```

```

REM Include code depending on value of t_flag
:
DEF FN_select(t_flag)
IF t_flag PROC_torch ELSE PROC_normal
=pass
REM      Return original value of OPT. This is a bit
REM      artificial, but necessary to insert some
REM      BBCBASIC(Z80) code in the assembler code.
:
  DEF PROC_torch
  [OPT pass
  LD E,A
  RST &30
  DEFB 2
  RET:]
  ENDPROC
:
  DEF PROC_normal
  [OPT pass
  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.

SAVING AND LOADING MACHINE CODE PROGRAMS

26. As mentioned earlier, you can use machine code routines in a number of BBCBASIC(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 BBCBASIC(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 BBCBASIC(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 disk. You MUST specify the
*S. start address (aaaa) and either the length of the
 area of memory (l111) or its end address+1 (bbbb).
 The amount saved will always be rounded up to a
 multiple of 128. OSCLI can also be used to save a
 file.

```
*SAVE ufsp aaaa +l111
*SAVE ufsp aaaa bbbb
OSCLI "SAVE"+<str>+" "+STR$( <num> )+" "+STR$( <num> )

*SAVE "WOMBAT" 8F00 +80
*SAVE "WOMBAT" 8F00 8F80
OSCLI "SAVE"+ufn$+" "+STR$(add)+" "+STR$(len)
```

*LOAD Load the specified file into memory at hexadecimal
*L. address 'aaaa'. The load address MUST always be
 specified. The file length will always be a multiple
 of 128. 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 "+file_name$+" "+STR$(start_address)
```

LIMITATIONS

27. The assembler has been 'squeezed' so that it occupies as little room as possible. The main result of this is that, whilst it will correctly assemble all legal instructions, it will also assemble several illegal ones without issuing an error!

28. The instructions that fool it are logically correct within the structure of the Z80 mnemonic code, but not acceptable to the Z80 processor; they generally concern IX and IY. For example, the following 'non codes' will assemble without an error being reported.

```
ADD IX,HL
EX DE,IX
ADD IX,IY
PUSH SP
```

READING THE SCREEN

29. There is a machine code entry point (vector) at &lFD to enable you to read characters off the screen. It returns, in the Z80's A and L registers, the character at the current text cursor position. You can use this vector from within BBCBASIC(Z80) by using the USR function. USR returns an integer number comprising the Z80's H, L, H' and L' registers. Consequently, the result needs a little manipulation to move the L register byte to the least significant byte of the answer. For example,

```
character=(USR &lFD AND &FF0000) DIV &10000
```


STATEMENTS AND FUNCTIONS

The commands and statements are listed alphabetically. For ease of reference, they are not separated into 2 sections.

All statements can also be used as direct commands.

Where appropriate, the abbreviated form is shown under the statement.

The associated keywords are listed at the end of each explanation.

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

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 2 numbers without regard to the sign of the answer.

It is particularly useful when you want to know the difference between 2 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, a more elegant solution would be:

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

Associated Keywords:

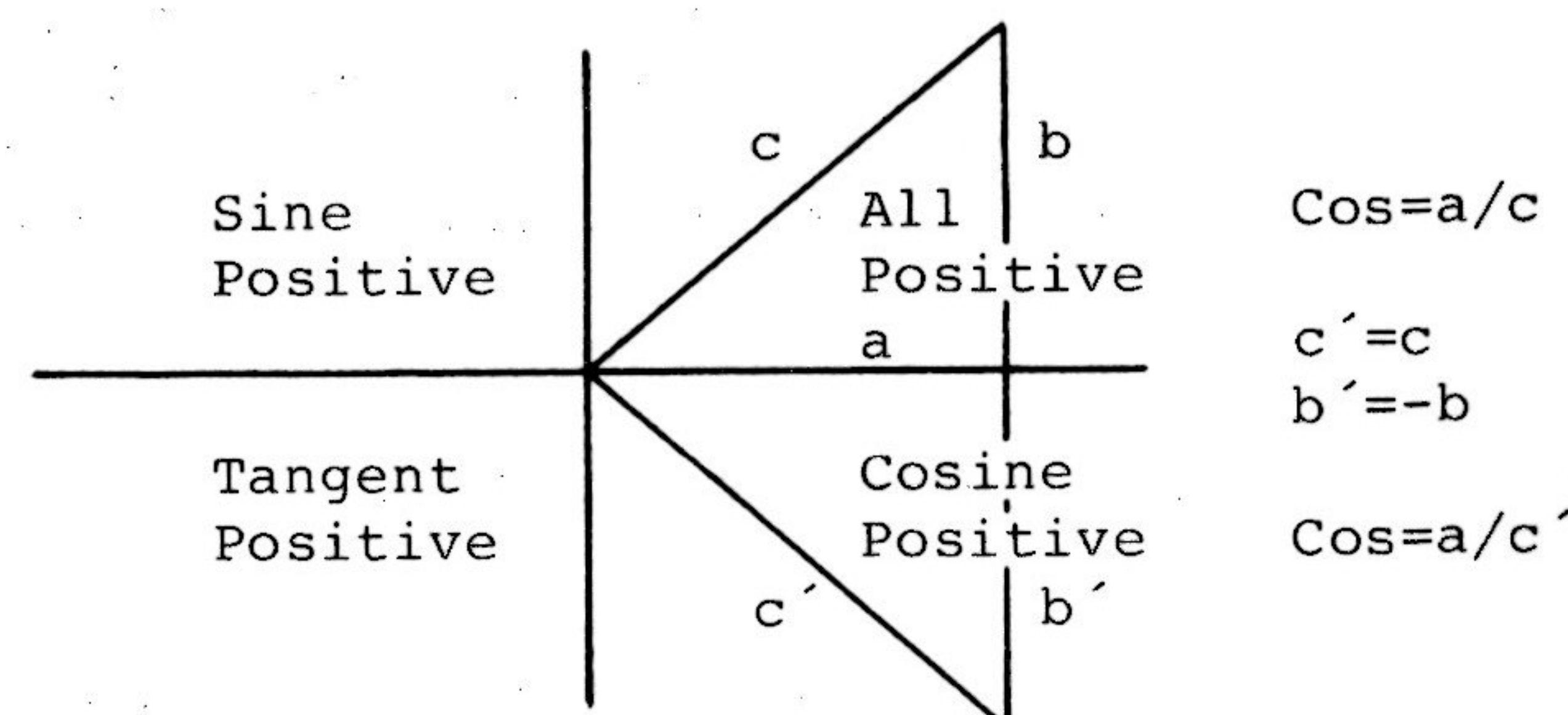
SGN

TOKEN = 94

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. As shown in the diagram below, 2 angles within the range $\pm\pi$ (± 180) can have the same cosine. This means that one cosine has 2 associated angles.



Within the 4 quadrants, there are 2 angles which have the same cosine, 2 with the same sine and 2 with the same tangent. When you are working back from the cosine, sine or tangent you don't know which of the 2 possible angles is correct. By convention, ACS gives a result in the top 2 quadrants (0 to π - 0 to 180 degrees) and ASN and ATN in the right-hand 2 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)
```

As shown below, you can convert the answer, to degrees by using the DEG function (or multiplying by $180/\pi$.)

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

Associated Keywords:

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

ADVAL

This function returns the value of the specified analogue to digital converter channel or the status of the fire buttons.

Positive Argument.

If its argument is between 1 and 4, ADVAL returns the last known value of the analogue to digital channel specified by its argument.

There are 4 analogue channels, numbered 1 to 4. Each channel has a resolution of 8 bits, but the value returned is scaled to 16 bits.

```
adcval=ADVAL(3)
level=ADVAL(chnumber)
```

The 4 analogue channels correspond to the following inputs:

ADVAL(1)	Channel 0	(M014 pin 1)
ADVAL(2)	Channel 1	(M014 pin 3)
ADVAL(3)	Channel 2	(M015 pin 1)
ADVAL(4)	Channel 3	(M015 pin 3)

Zero Argument.

If its argument is zero, ADVAL(0) returns a value which can be used to determine which, if any, of the 'fire' buttons is pressed on a games paddle.

```
fireno=ADVAL(0)
```

will return a value with the following meaning:

fireno=0	no button pressed
fireno=1	left fire button pressed (M014 pin 4)
fireno=2	right fire button pressed (M015 pin 4)
fireno=3	both fire buttons pressed

TOKEN=96.

AND
A.

The operation of integer bitwise logical AND between 2 items. The 2 operands are internally converted to 4 byte integers before the AND operation.

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

Unfortunately, BBCBASIC 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
IF answer THEN 110       time.
```

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
```

Associated Keywords:

EOR, OR, FALSE, TRUE, NOT

TOKEN = 80

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.

Different type of computer use different numbers for the characters. The codes used for CP/M computers are those defined by the American Standard for Information Interchange (ASCII). A list of the ASCII codes is at Annex A.

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\$.

Associated Keywords:

CHR\$, STR\$, VAL

TOKEN = 97

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 2 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))

Associated Keywords:

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

TOKEN = 98

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))
```

Associated Keywords:

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

TOKEN = 99

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.

Associated Keywords:

None

TOKEN = C6

BGET#
B.#

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

E=BGET#n

Before you use this statement you must have opened a file using OPENOUT, OPENIN or OPENUP. (See OPENOUT, OPENIN and the section on BBCBASIC(Z80) Disk Files for details.)

You can use BGET# to read single bytes from a disk file. This enables you to read back small integers which have been 'packed' into less 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. To make it more useful, the program could ask for the names of the input and output files at 'run time'.

```

10 REM Open input and output 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=&l3 THEN temp=&l9   :REM Convert ^S
110  BPUT#outfile,temp         :REM Write byte
120 UNTIL temp=&lA              :REM ^Z
130 CLOSE#0                    :REM Close all
140 END                        :REM files.
```

Associated Keywords:

OPENIN, OPENOUT, CLOSE#, EXT#, PTR#, PRINT#, INPUT#,
BPUT#, EOF#

BPUT#
BP.#

A statement which puts a byte to the data file whose channel number is the first argument. The second argument's 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 have opened a file for output using OPENOUT, OPENIN or OPENUP. (See these key-words and the section on BBCBASIC(Z80) Disk Files for details.)

You can use this statement to write single bytes to a disk 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 3 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 2 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.

```

100 BPUT#fnum,number%
110 BPUT#fnum,number% DIV 255
```

Associated Keywords:

OPENIN, OPENOUT, CLOSE#, EXT#, PTR#, PRINT#, INPUT#,
BGET#, EOF#

TOKEN = DS + 23

CALL
CA.

A statement to call a machine code subroutine.

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

CALL sets up a table in RAM containing details of the parameters. The IX register is set to the address of this parameter table. The IY register is set to the address of the machine code subroutine being called.

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

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

The parameter types are:

Code No	Parameter Type	
0:	byte (8 bits)	eg ?A%
4:	word (32 bits)	eg !A% or A%
5:	real (40 bits)	eg A
128:	fixed string	eg \$A% plus CHR\$(13)
129:	movable string	eg A\$

On entry to the subroutine the parameter table contains the following values:

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
Parameter address)	as necessary.

TOKEN D6

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.

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 2 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
↑		
Becomes &A0	because MSB always assumed set	

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).

See Annex E for more details of how parameters are stored.

Torch Only

In the BBC Micro, CALL and USR can be used to access routines in the machine operating system (MOS). In order to maximize compatibility with the BBC Micro, addresses between &FF00 and &FFFF are assumed by BBCBASIC(Z80) to refer to the 6502's MOS. Consequently, CALL and USR behave differently when addressing this area of memory.

See Annex F for details of the behaviour of CALL and USER when calling addresses above &FF00.

Access via a Z80 machine code routine to addresses in the range &FF00 to FFFF will NOT be passed to the 6502 via the Tube. Machine code access to the 6502 is possible by using the Torch Tube protocols. See Annex F and the Torch Programmers Guide for more details.

Associated Keywords:

USR

CHAIN CH.

A statement which will load and run the program whose name is specified in the argument. All but the static variables @% to Z% are CLEARED. The program file must be in tokenised format.

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

CHAIN sets ON ERROR OFF before chaining the specified program.

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 example program segment below 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 $(HIMEM+12)=out_file$
130 CHAIN "NEXTPROG"
```

RUN may be used as an alternative to CHAIN.

Associated Keywords:

LOAD, SAVE

TOKEN = 07

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. For instance:

```
CHR$(7)
```

will generate the ASCII character ^G. So,

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

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

CHR\$ is the complement of ASC.

Associated Keywords:

ASC, STR\$, VAL, VDU

TOKEN = 80

CLEAR
CL.

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

The CLEAR command tells BBCBASIC(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.

CLEAR will also make BBCBASIC(Z80) forget about any open files. Consequently, ALL DATA FILES should be closed before a CLEAR command is issued.

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.

Associated Keywords:

None

TOKEN = 88

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

CLOSE#file_num
 CLOSE#0

You use CLOSE# to tell BBCBASIC(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. (See the section on BBCBASIC(Z80) Disk Files for more information.)

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 or CHAIN another program module, you have no option but to close the data files first.

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

When a file is opened, the file control block (FCB) and the 128 byte file buffer are added to the heap. If the FCB and buffer are on the top of the heap when the file is closed, the heap space they occupied is recovered. (See Annex E for details of how memory is used and hints on memory management.)

Associated Keywords:

OPENIN, OPENOUT, EXT#, PTR#, PRINT#, INPUT#,
 BGET#, BPUT#, EOF#

TOKEN 09 + 23

CLG A statement which clears all the sprites from the screen. This statement **only** affects the sprites. Graphics created with DRAW and PLOT are not cleared.

Associated Keywords:

CLS, GCOL

TOKEN DA

CLS

A statement which clears the screen to the current text background colour (as set by the COLOUR statement) and moves the text cursor to the text origin (0,0 - top left).

This command has no affect on the graphics cursor.

If the text background colour is 'transparent', the screen will clear to the current backdrop colour(as set by the GCOL statement).

Associated Keywords:

CLG, COLOUR

TOKEN = DB

**COLOUR
(COLOR)**

A statement which selects the colour in which text and its background are printed.

The keyword COLOUR is followed by a number. If the number 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.

There are 16 colours, which are:

<u>Foreground</u>	<u>Background</u>	<u>Colour</u>
COLOUR 0	COLOUR 128	Transparent
COLOUR 1	COLOUR 129	Black
COLOUR 2	COLOUR 130	Medium green
COLOUR 3	COLOUR 131	Light green
COLOUR 4	COLOUR 132	Dark blue
COLOUR 5	COLOUR 133	Light blue
COLOUR 6	COLOUR 134	Dark red
COLOUR 7	COLOUR 135	Cyan
COLOUR 8	COLOUR 136	Medium red
COLOUR 9	COLOUR 137	Light red
COLOUR 10	COLOUR 138	Dark yellow
COLOUR 11	COLOUR 139	Light yellow
COLOUR 12	COLOUR 140	Dark green
COLOUR 13	COLOUR 141	Magenta
COLOUR 14	COLOUR 142	Grey
COLOUR 15	COLOUR 143	White

The COLOUR command sets the colours displayed in the individual character cells. It does not change the overall backdrop colour of the screen (see GCOL).

When the Einstein is powered up, the foreground colour is set to white (15) and the background colour is set to 'transparent' (128). Running other programs may change these colours. When BBCBASIC(Z80) is loaded, neither the background colour nor the foreground colour is changed.

Associated Keywords:

GCOL

TOKEN FB

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))

Associated Keywords:

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

TOKEN 9B

COUNT
COU.

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

```
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 either the *OPT command (or the *FX command - Torch version).

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 ....
```

Associated Keywords:

POS

TOKEN = 9C

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 example below 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.

```
90 ....
100 DATA FRED,17,BILL,21,ALISON,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"
170 ....
```

See READ for more details of this example.

Associated Keywords:

READ, RESTORE

TOKEN DC

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 them at the start of the program.

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

In order to make the text more readable (always a GOOD THING) 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 sub-section on Procedures and Functions in the General Information section.

Associated Keywords:

ENDPROC, FN, PROC

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 π 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.

Associated Keywords:

RAD, SIN, COS, TAN, ACS, ASN, ATN

TOKEN 90

TOKEN = DD

(FURTHER PART OF DEFN ETC)

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 <RETURN>.

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.

Associated Keywords:

```
EDIT, LIST, OLD, NEW
```

TOKEN C7

DIM

There are 2 quite different uses for the DIM statement: the first dimensions an array and the second reserves an area of memory for special applications.

Dimensioning Arrays

The DIM statement is used to declare arrays. Arrays must be pre-declared before use and they must not be re-dimensioned. Both numeric and string arrays may be multi dimensional.

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

After DIM, all elements in the array are 0/null.

The subscript base is 0, so DIM X(12) defines an array of 13 elements.

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 2 dimensional array. The 2 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 3 dimensional array. The third dimension would represent the 'move number'. Each move would use about 250 bytes of memory, so you could record 40 moves in about 10k bytes.

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

TOKEN DE

Reserving an Area of Memory

A DIM statement is used to reserve an area of memory which the interpreter will not then use. The variable in the DIM statement is set by BBCBASIC(Z80) to the start address of this memory area. This reserved area can be used by the indirection operators, machine code, etc.

The example below sets A% to reserve 68 bytes of memory with bytes A%?0 to A%?67 free for use by the program (68 bytes in all).

```
DIM A% 67
```

A 'DIM space' error will occur if a size of less than -1 is used (DIM P% -2). 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. Thus,

```
DIM P% -1
PRINT HIMEM-P%
```

is the equivalent of PRINT FREE(0) in other versions of BASIC.

See the section on Assembler for a more detailed description of the use of DIM for reserving memory for machine code programs.

Associated Keywords:

None

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, BBCBASIC(Z80) converts them to integers by truncation before DIViding them.

Associated Keywords:

MOD

70XW 81

DRAW

A statement which draws a line on the screen. The statement is followed by the X and Y co-ordinates of the end of the line.

The start of the line is the current position of the graphics cursor. This is generally either the end of the last line drawn or a point specified by a MOVE or PLOT statement.

On the Einstein, the screen is addressed as 0 to 1023 in the X axis and 0 to 767 in the Y axis. The default origin (X=0, Y=0) is the bottom left of the screen, but this may be changed using the PLOT -1 statement.

The line is drawn in the current graphics foreground colour. The background colour of the areas through which the line passes are changed to the current background colour. The graphics foreground, background and backdrop colours may be set by using the GCOL statement.

DRAW x,y

MOVE 200,300
DRAW 640,800

This statement is identical to PLOT 5.

See Section 17 of the Introduction to the Einstein Colour Micro Computer for more details about graphics.

Associated Keywords:

MODE, PLOT, MOVE, CLG, VDU, GCOL

TOWN OF

EDIT

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.

EDIT 230

EDIT 200,230

The following control functions are active in both the immediate and edit modes.

← Backspace the cursor one character position.

→ Advance the cursor one character position.

↑ Move the cursor to the line start.

↓ Move the cursor to the line end.

INS Insert a space at the cursor position.

CTRL F Delete the character at the cursor position.

DEL Backspace and delete.

CTRL U Delete from the cursor to the line end.

CTRL X Delete from left of the cursor to the line start.

To exit edit mode and replace the edited line, type carriage return.

To abort the edit and leave the line unchanged, type ESCAPE.

You can use this command to edit and join (concatenate) program lines. When you use it to join lines, remember to delete the unwanted original 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.

No TOWN

By only changing the line number, you can also use EDIT to duplicate a line.

Additional Functions

In addition to the editing functions, the following control characters have an effect:

CTRL A Dump the screen to the printer.

CTRL P Toggle the printer on and off (see the keyword PRINT).

Associated Keywords:

DELETE, LIST, OLD, NEW

ELSE
EL.

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

In an IF statement, if the test is FALSE, 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"WWWW" ELSE B=D:PRINT"QQQQ"
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 'Bad' if 'x' did not equal 3 OR if 'a' did not equal 'b'.

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

If you want to 'nest' the tests, you should use a procedure call. The following example, would print 'Bad' ONLY if x was equal to 3 AND 'a' was not equal to 'b'.

```
IF x=3 THEN PROC_ab_test
...

...
DEF PROC_ab_test
IF a=b THEN PRINT "OK" ELSE PRINT "Bad"
ENDPROC
```

ELSE also traps exceptions in ON.

```
ON number GOSUB 100,200,300 ELSE PRINT "Error"
```

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

Associated Keywords:

IF, THEN, ON

TOKEN 88

END

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

END tells BBCBASIC(Z80) 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 its a bit messy.

You can use END within, for instance, an IF...THEN...ELSE statement to stop your program if certain conditions are satisfied. You should also use END to stop BBCBASIC(Z80) 'running into' any procedure or function definitions at the end of your program.

Associated Keywords:

STOP

TOKEN = E0

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.

See the sub-section on Procedures and Functions for a more detailed explanation of procedures and local variables.

Associated Keywords:

DEF, FN, PROC, LOCAL

TOKEN = E1

ENVELOPE A statement which is used, in conjunction with the SOUND statement, to control the volume of the sound produced by the sound generator.

During its 'life' the volume and pitch of a sound may change. The variation of volume with time is called the amplitude envelope and the variation of pitch with time is called the pitch envelope. On the Einstein, you can control the duration of the sound and select one of 8 amplitude envelopes by using the ENVELOPE statement.

The ENVELOPE statement has no effect unless the second parameter of the SOUND statement (amplitude) is positive (=1).

The ENVELOPE statement has 2 parameters. The syntax of the statement and the meaning of the parameters are shown below.

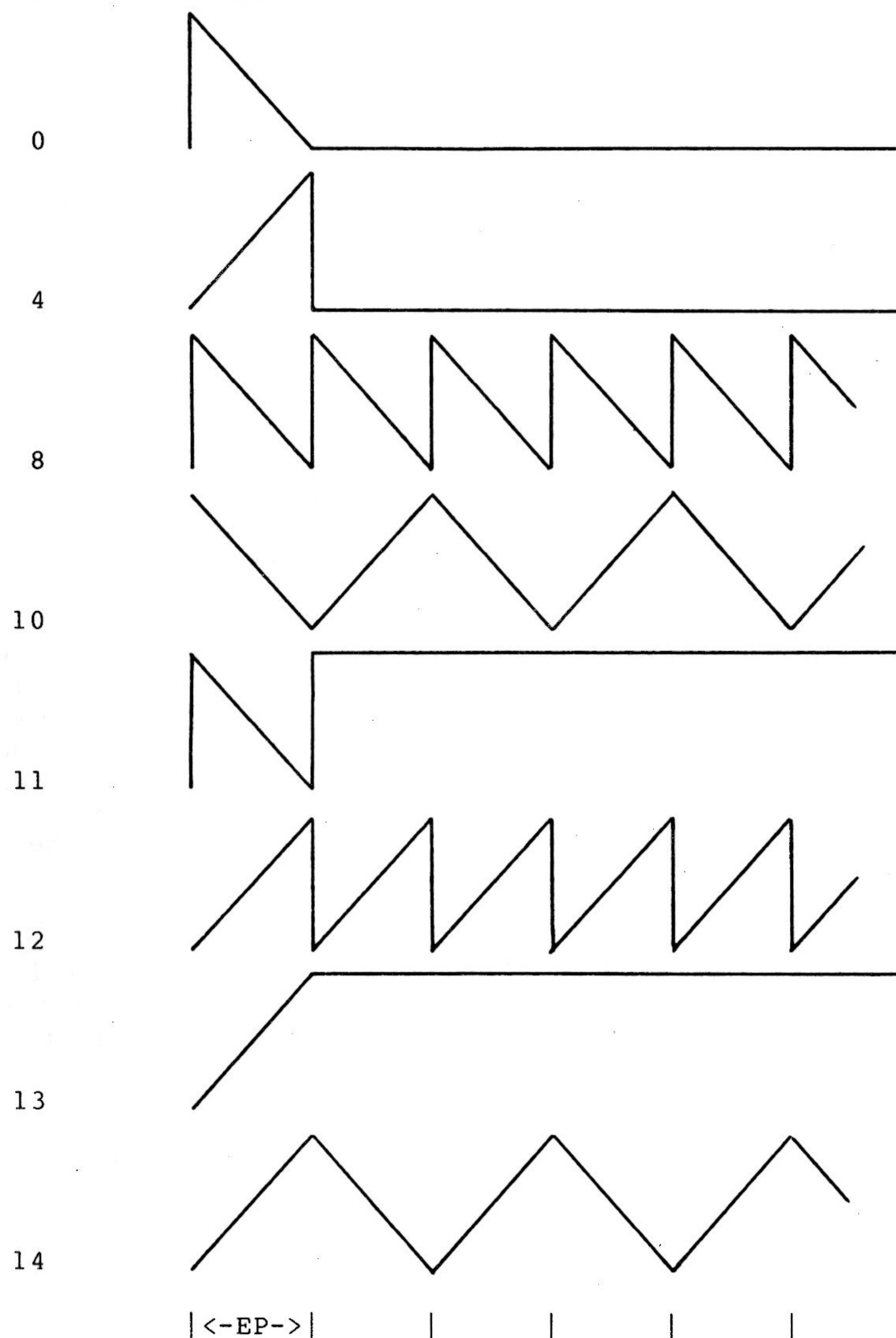
ENVELOPE durn,env

<u>Parameter</u>	<u>Range</u>	<u>Function</u>
durn	1 to 65535	The duration of one cycle of the sound envelope (envelope period - EP) in units of 128 microseconds. (durn=7812 gives a duration of approximately 1 second.)
env		Selects the volume envelope to be used. The envelope shapes and their associated numbers are shown below.

TOKEN E2

env

Shape



Associated Keywords:

ADVAL, SOUND

EOF#

A function which will return -1 (TRUE) if the data file whose channel number is the argument is at its end (the last byte of the last sector has been read).

In the case of a sparse random-access file EOF#file-num=TRUE indicates that the file pointer (PTR#file-num) is pointing to a block of the file which has not had data written to it.

IF EOF#filenum THEN

Because CP/M does not keep a precise record of the position of the end of a file, EOF# is an uncertain indicator that the end of the file has been reached. It is more of an error message telling you that there is definitely no more data available. However, it can be used successfully with files containing only character data.

EOF will also be true if an attempt has been made to read from an empty block of a sparse random access file. (In other words, a block has not yet been allocated to this record by CP/M.) The unused areas of blocks for which EOF is FALSE will contain nulls. By using EOF combined with a test for nulls, it is possible to make a certain check for the presence of character data.

Writing to a previously unused block will reset EOF# immediately, irrespective of whether the record is physically written to the disk at that time or not.

See the BBCBASIC(Z80) Disk Files section for an explanation of the limitations of EOF# and examples of how to use it.

Associated Keywords:

OPENIN, OPENOUT, EXT#, PTR#, PRINT#, INPUT#, BGET#, BPUT#, CLOSE#

TOKEN CS +23

EOR

The operation of bitwise integer logical exclusive-or between 2 items. The 2 operands are internally converted to 4 byte integers before the EOR operation. EOR will return a non-zero result if the 2 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, BBCBASIC 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 2 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
```

Associated Keywords:

NOT, AND, OR

TOKEN 82

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 BBCBASIC(Z80) reports an error.

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

Associated Keywords:

ON ERROR GOTO, ON ERROR OFF, REPORT, ERR

TOKEN 9E

ERR

A function returning the error code number of the last error which occurred (see Annex C).

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 sub-section on Error Handling for more details of error handling and correction.

Associated Keywords:

ON ERROR GOTO, ON ERROR OFF, ERL, REPORT

TOKEN 9F

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 BBCBASIC(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 (which is for the Torch) is a very simple program to accept an equation and plot the result.

```
10 MODE 0
20 VDU 29,640;512;
30 MOVE -640,0:DRAW 640,0
40 MOVE 0,-512:DRAW 0,512
50 INPUT "What is the function ",equation$
60 X=-100
70 MOVE -640,EVAL(equation$)
80 FOR X=-90 TO 100 STEP 10
90   DRAW 6.4*X,EVAL(equation$)
100 NEXT
110 END
```

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

Associated Keywords:

STR\$, VAL

TOKEN A0

EXP

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

Associated Keywords:

LN, LOG

TOKEN A1

EXT#

A function which returns the total length of the file whose channel number is its argument. The length will always be an exact multiple of 128 bytes.

In the case of a sparse random-access file, the value returned is the virtual file length. This may well be greater than the actual amount of data on the disk.

EXT performs a directory read and is therefore quite slow. If repeated use of the file length is necessary, use EXT# once and save the result in a variable.

```
length=EXT #F
```

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

Because of the limitations of CP/M, the file length is only given as a multiple of 128 bytes. Consequently, EXT# may not always give you the information you need. The limitations of CP/M and ways to overcome them are discussed at some length in the Disk Files Section.

Associated Keywords:

OPENIN, OPENOUT, CLOSE#, PTR#, PRINT#, INPUT#, BGET#, BPUT#,

TOKEN A2 + 23

FALSE
FA.

A function returning the value zero.

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

BBCBASIC(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 Variables sub-section for more details on Boolean variables and the keyword AND for logical tests and their results.

Associated Keywords:

TRUE, EOR, OR, AND, NOT

TOKEN A3

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). No spaces are allowed between the function name and the opening bracket of the parameter list (if any).

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.

```
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$
```

Two different functions with the same name, one with parameters and one without, are permitted (PLEASE don't do it).

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 example sets the print control variable to the parameter passed and returns a null string. (See PRINT for details.)

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

See the Procedures and Function sub-section for more details.

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 2 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). See the Program Flow Control sub-section for more details.

```
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 BBCBASIC(Z80) recognizes an integer value of a real variable.

TOKEN = E3

TOKEN A4

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 1 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. See the Program Flow Control sub-section for an explanation of this problem and an example of how to overcome it.

Associated Keywords:

TO, STEP, NEXT

GCOL GC.

A statement which sets the graphics backdrop and foreground/background colour to be used in all subsequent graphics operations.

GCOL bdrop,colour

The first argument selects the current backdrop colour. This is the colour of the border and the colour which 'shows through' any 'transparent' areas. If the first argument is 0, the backdrop is black.

The second argument selects the current foreground or background colour. If its value is greater than 127, then the background colour is set.

There are 16 colours, which are:

<u>Foreground</u>	<u>Background</u>	<u>Colour</u>
COLOUR 0	COLOUR 128	Transparent
COLOUR 1	COLOUR 129	Black
COLOUR 2	COLOUR 130	Medium green
COLOUR 3	COLOUR 131	Light green
COLOUR 4	COLOUR 132	Dark blue
COLOUR 5	COLOUR 133	Light blue
COLOUR 6	COLOUR 134	Dark red
COLOUR 7	COLOUR 135	Cyan
COLOUR 8	COLOUR 136	Medium red
COLOUR 9	COLOUR 137	Light red
COLOUR 10	COLOUR 138	Dark yellow
COLOUR 11	COLOUR 139	Light yellow
COLOUR 12	COLOUR 140	Dark green
COLOUR 13	COLOUR 141	Magenta
COLOUR 14	COLOUR 142	Grey
COLOUR 15	COLOUR 143	White

When the Einstein is powered up, the foreground colour is set to 15 (white), the background colour is set to 128 ('transparent') and the backdrop colour is set to 4 (dark blue). Running other programs may change these colours. When BBCBASIC(Z80) is loaded, none of these colours are changed.

Associated Keywords:

CLS, CLG, MODE, COLOUR, PLOT

TOKEN EG

GET
GET\$

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

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

GET and GET\$ wait for a key to be pressed and then return the ASCII number of the key (see Annex A) or a string containing the character of the pressed key.

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
```

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.

GET can also be used to input data from a Z80 port. The full 16-bit Z80 extended port addressing is available.

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

This extension of the language has not been approved by Acorn and cannot be guaranteed to remain unchanged in future releases.

Associated Keywords:

PUT, INKEY, INKEY\$

TOKEN GET = A5
TOKEN GET\$ = BE

GOSUB
GOS.

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. (See the Program Flow Control sub-section for details.)

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. Unless you wish to use ON GOSUB or calculate the line number of the subroutine, you will find that a PROCedures offer you all the facilities of subroutines and, by choosing their names carefully, you can make your programs much more readable.

Associated Keywords:

RETURN, ON, GOSUB

TOKEN EL

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 BBCBASIC(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.

Associated Keywords:

GOSUB, ON

TOKEN ES

HIMEM
H. A pseudo-variable which contains the address of the first byte that BBCBASIC(Z80) will not use.

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

HIMEM=HIMEM-40

BBCBASIC(Z80) uses the computer's memory to store your program and the variables that your program uses. When BBCBASIC(Z80) is first loaded and run it checks to find the start of the memory area used by the operating system. Since this area of memory is not available to BBCBASIC(Z80), HIMEM is set to this 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 BBCBASIC(Z80). See the Machine Code 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.

Associated Keywords:

LOMEM, PAGE, TOP

TOKEN D3

IF

A statement which sets up a test condition which can be used to control the subsequent flow of the program. It is part of the IF....THEN....ELSE structure. The word THEN is optional under most circumstances.

```
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=B, etc) is evaluated and the answer is either TRUE or FALSE. If the answer is TRUE, the rest of the line (up to the ELSE clause if there is one) is executed.

The '=' sign has 2 meanings. It can be used to assign a value to a variable or as part of a test. The example shows the 2 uses in one program line.

```
A=B=C
```

In english this reads "A becomes equal to the result of the test B=C". Thus if B does equal C, A will be set to TRUE (-1). However, if B does not equal C, A will be set to FALSE (0). The example below is similar, but A will be set to TRUE (-1) if 'age' is less than 21.

```
A=age<21
```

Since the IF statement evaluates the testable condition and acts on the result, you can use a previously set variable name in place of the test. The 2 examples below will print 'Under 21' if the value of 'age' is less than 21.

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

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

Associated Keywords:

THEN, ELSE

INKEY

A function which does a GET waiting for a maximum of 'num' clock ticks of 10ms each. If a key has been pressed, INKEY will return the ASCII value of the key pressed. If no key has been pressed within the time limit, INKEY will return -1. (A table of ASCII values is included at Annex A.)

```
key=INKEY(num)
N=INKEY(0)
temp=INKEY(200)
```

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.

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

The Einstein has a single key input buffer. If a key is pressed before the INKEY function is used, this key value will be returned. If you do not want this to happen you should empty the buffer before you look for a key to be pressed. You could do this by using INKEY and discarding the result. The example below clears the input buffer before it prompts for a response. You are then given one second to push a key. If you fail, you get a sarcastic message.

```
100 temp=INKEY(0)
110 PRINT "Push a key"
120 result=INKEY(100)
130 IF result = -1 THEN PRINT "Slow coach"
140 etc
```

Associated Keywords:

INKEY\$, GET, GET\$

TOKEN A6

INKEY\$

A function which does a GET\$ waiting for a maximum of 'num' clock ticks of 10ms each. If a key has been pressed, INKEY\$ will return the character pressed. If no key has been pressed within the time limit, INKEY\$ will return a null string.

```
key$=INKEY$(num)
N$=INKEY$(0)
temp$=INKEY$(200)
```

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.

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

The Einstein has a single key input buffer. If a key is pressed before the INKEY\$ function is used, this key value will be returned. If you do not want this to happen you should empty the buffer before you look for a key to be pressed. You could do this by using INKEY and discarding the result. The example below clears the input buffer before it prompts for a response. It then waits for the time period specified by 'delay' before going on.

```
100 temp=INKEY(0)
110 PRINT "Push a key"
120 result$=INKEY$(delay)
130 IF result$="" THEN PROC_too_slow
140 etc
```

Associated Keywords:

INKEY, GET, GET\$

TOKEN BF

INPUT
I.

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 return (enter) key is pressed to complete an entry, a NEW-LINE is generated. BBCBASIC 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 2 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 "ed 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.

Associated Keywords:

```
INPUT LINE, INPUT#, GET, INKEY
```

TOKEN E8

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\$

Associated Keywords:

INPUT

TOKEN E8+20+86

INPUT# A statement which reads data in internal format from a file and puts them in the specified variables.

INPUT #E,A,B,C,D\$,E\$,F\$

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 Disk Files section for more details and numerous examples of the use of INPUT#.

Associated Keywords:

INPUT

TOKEN E8+23

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"
```

Associated Keywords:

LEFT\$, MID\$, RIGHT\$, LEN

NO TOKEN

INT

A function converting a real number to the lower integer.

```
INT(99.8)=99, INT(-12)=-12, INT(-12.1)=-13.
X=INT(Y)
```

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.

Associated Keywords:

None

TOKEN A8

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.

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)
```

For example,

```
10 name$="BBCBASIC(Z80)"
20 FOR i=1 TO 15
30   PRINT LEFT$(name$,i)
40 NEXT
50 END
```

would print

```
B
BB
BBC
BBCB
BBCBA
BBCBAS
BBCBASI
BBCBASIC
BBCBASIC(
BBCBASIC(Z
BBCBASIC(Z8
BBCBASIC(Z80
BBCBASIC(Z80)
BBCBASIC(Z80)
BBCBASIC(Z80)
```

Associated Keywords:

RIGHT\$, MID\$, LEN, INSTR

NO TOKEN

LEN

A function which returns the length of the argument string.

```
X=LEN"fred"
X=LENAS
X=LEN(A$+B$)
```

This function 'counts' the number of characters in a string. For example,

```
length=LEN("BBCBASIC(Z80)  ")
```

would set 'length' to 16 since the string consists of the 13 characters of BBCBASIC(Z80) followed by 3 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)) ELSE flag=FALSE
80   NEXT
90   IF NOT flag THEN PRINT "Bad HEX"
100 UNTIL flag
```

Associated Keywords:

LEFT\$, MID\$, RIGHT\$, INSTR

TOKEN A9

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 emphasized 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'.

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.

Associated Keywords:

None

TOKEN E9

LIST A command which causes lines of the current program to be listed out with the automatic formatting options specified by LISTO.

L.

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.

With the standard CP/M version of BBCBASIC(Z80), the listing may be paused by pressing <CTRL> and 'S' (usually shown as ^S) and restarted by pushing any key. On the Torch, the listing may be paused by pressing <SHIFT> and <CTRL> together. The listing will resume when either or both keys are released. Also on the Torch, the listing may be 'paged' by pressing ^N before typing LIST. To display the next page, push <SHIFT>. You can return to the normal mode by pressing ^O.

Escape will abort the listing.

You can cause the listing to be printed by pressing ^P on the standard CP/M version or by pushing ^B on the Torch. Printing can be stopped by pushing ^P a second time (it's a 'toggle' action) or pushing ^C on the Torch. (Don't forget to select the printer and turn it on first.)

LIST may be included within a program, but it will exit to the command mode on completion of the listing.

Associated Keywords:

LISTO, OLD, NEW

TOKEN C9

LISTO

A command which controls the appearance of a LISTed program. The command controls the setting of the 3 least significant bits of the format control byte which can, therefore, be set to an integer 0 to 7 (0=all 3 bits 0, 7=all 3 bits 1).

a. 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.)

b. Bit 1. If Bit 1 is set, 2 extra spaces will be printed out on lines between FOR and NEXT. Two extra spaces will be printed for each depth of nesting.

c. Bit 2. If Bit 2 is set 2 extra spaces will be printed out on lines between REPEAT and UNTIL. 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 of the FOR..NEXT and REPEAT..UNTIL lines is done in the correct manner, in that the NEXT is aligned with the FOR and the REPEAT with the UNTIL.

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
```

TOKEN C9 + 4F

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
```

and

LISTO 2

will give

```
10A=20
20TEST$="FRED"
30FOR I=1 TO A
40  Z=2^Z
50  PRINT I,Z
60  REPEAT
70  PRINT TEST$
80  TEST$=LEFT$(TEST$,LEN(TEST$)-1)
90  UNTIL LEN(TEST$)=0
100NEXT
110END
```

Use the default setting, LISTO 7 - You know it makes sense.

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 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.

Associated Keywords:

LOG, EXP

TOKEN AA

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$
```

You use LOAD to bring a program in a disk file into memory. The key-word LOAD should be followed by the specifier (disk:name.extension) of the program file.

File names must conform to the standard CP/M format. However, if no extension is given, .BBC is assumed. See the Operating System Commands section for a more detailed description of valid file names.

Torch Only. The file specifier may be ambiguous. The file with the first entry in the directory which matches the specifier will be loaded.

```
LOAD "INT*"
```

Associated Keywords:

SAVE, CHAIN

TOKEN C8

LOCAL
LOC.

A statement to declare variables for local use inside a function (FN) or procedure (PROC). A null list of variables is not permitted.

LOCAL A\$,X,Y%

LOCAL saves the values of its arguments in such a way that they will be restored at '=' or ENDPROC.

If a function or a procedure is used recursively, the LOCAL variables will be preserved at each level.

The LOCAL variables are initialised to zero/null.

See the Procedures and Functions sub-section for full details.

Associated Keywords:

DEF, ENDPROC, FN, PROC

TOKEN EA

LOG

A function giving the base-10 logarithm of its argument.

$X = \text{LOG}(Y)$

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 as shown below.

If $x = \text{LOG}(y)$

then

$y = 10^x$

Associated Keywords:

LN, EXP

TOKEN AB

LOMEM
LOM.

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 and file buffers are stored in memory immediately after your program. You can change the address where BBCBASIC(Z80) starts to store these variables by changing LOMEM.

USE WITH CARE. Changing LOMEM in the middle of a program causes BBCBASIC(Z80) to lose track of all the variables you are using.

Do not change LOMEM while any data files are open.

Associated Keywords:

HIMEM, TOP, PAGE

TOKEN 02

MID\$

A string function which returns 'num' characters of the string starting from character 'start_posn'. If 'num' is not present or if there are insufficient characters in the string, then all the characters from 'start_posn' onwards are returned.

```
C$=MID$(A$,start_posn,num)
C$=MID$(A$,Z)
```

You can use this function to select any part of a string. For instance, if

```
name$="BBCBASIC(Z80)"
```

then

```
part$=MID$(name$,4,5)
```

would assign BASIC to 'part\$'.

If the last number is omitted or there are insufficient characters to the right of the specified position, MID\$ returns with the right hand part of the string starting at the specified position. Thus,

```
part$=MID$(name$,9)
```

would assign (Z80) to 'part\$'.

For example,

```
10 name$="BBCBASIC(Z80)"
20 FOR i=1 TO LEN(name$)-8
30   PRINT MID$(name$,i,11)
40 NEXT
```

would print

```
BBCBASIC(Z8
BCBASIC(Z80
CBASIC(Z80)
BASIC(Z80)
ASIC(Z80)
```

Associated Keywords:

LEFT\$, RIGHT\$, LEN, INSTR\$

NO TOKEN

MOD A binary operation giving the signed remainder of the integer division.

X=A MOD B

MOD is defined such that,

$A \text{ MOD } B = A - (A \text{ DIV } B) * B.$

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,

```
10 PRINT 23 DIV 3
20 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 BBCBASIC(Z80) calculates the result. Thus,

```
10 PRINT 23.1 DIV 3.9
20 PRINT 23.1 MOD 3.9
```

would give exactly the same results as the previous example.

Associated Keywords:

DIV

TOKEN 83

MODE A statement which selects the screen mode.

MODE 1
MODE mode_num

Mode	Graphics Resolution	Text Resolution
0	240x192	40x24
1	256x192	32x24

The screen is cleared to the current backdrop colour. The backdrop colour is unchanged. The text and graphics foreground and background colours are set to white and 'transparent' respectively.

All the sprites are cleared.

The text cursor is positioned at 0,0 (top left) and the graphics cursor is positioned at 0,0 (bottom left).

The graphics origin is set to 0,0 (bottom left).

Associated Keywords:

CLS, CLG

TOKEN EB

MOVE

A statement which moves the graphics cursor to an absolute position without drawing a line.

MOVE X,Y

MOVE 124,327

The MOVE statement is identical to PLOT 4.

The graphics screen is 1024 points (0-1023) wide and 768 points (0-767) high.

Associated Keywords:

DRAW, MODE, GCOL, PLOT

TOKEN EC

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.

This command effectively 'removes' a program from the computer's memory. In reality, the program is still there, but BBCBASIC(Z80) has been told to forget about it.

If you have done 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.

Associated Keywords:

OLD

TOKEN CA

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.) See the Program Flow Control sub-section for more details. 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.

Associated Keywords:

FOR, TO, STEP

TOKEN = ED

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

As explained in the Variables sub-section, BBCBASIC(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.

Associated Keywords:

None

TOKEN AC

OLD

A command which undoes the effect of NEW provided no lines have been typed in or deleted, and no variables have been created.

On standard CP/M computers, OLD works even if the BASIC has been re-loaded and re-started from CP/M after, for example, a 'fatal' BDOS error.

Unfortunately, such an error on the Torch clears down the operating system pointers and it may not be possible to recover the program after reloading BBCBASIC(Z80).

Associated Keywords:

NEW

TOKEN CB

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 3 other key-words: GOTO, GOSUB and ERROR. (ON ERROR is explained separately.)

ON action GOSUB 1000,2000,3000,4000

Exceptions may be trapped using the ELSE statement delimiter.

ON action GOTO 100,200,300,120 ELSE PRINT "Illegal"

ON B-46 GOSUB 100,200,(C/200) ELSE
PRINT "ERROR"

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.

You can use ON...GOTO and ON...GOSUB to execute the appropriate part of your program as the result of a menu selection. The following skeleton example offers a menu with 3 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 GOTO 1000,2000,3000 ELSE
110 .....etc
```

Associated Keywords:

ON ERROR, GOTO, GOSUB

TOKEN EE

ON ERROR A statement controlling error trapping. If an ON ERROR statement has been encountered, BBCBASIC(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 BBCBASIC(Z80).

```
ON ERROR PRINT "Suicide":END
ON ERROR GOTO 100
ON ERROR OFF
```

The ON ERROR statement can be used to trap out the escape key to prevent files being closed at the wrong time by its accidental use.

```
:
:
50 ON ERROR IF ERR=17 THEN 70
60 PRINT:REPORT:PRINT " at line ";ERR:END
70 :
:   etc.
```

See the Error Handling sub-section for more details.

Associated Keywords:

ON, GOTO, GOSUB

TOKEN EE + 20 + 85

OPENIN
OP.

A function which opens a disk data file for reading or update and returns the 'channel number' of the file. This number must be used in subsequent references to the file with BGET#, BPUT#, INPUT#, PRINT#, EXT#, PTR#, EOF# or CLOSE#.

A returned value of zero signifies that the specified file was not found on the disk.

```
X=OPENIN "jim"
X=OPENIN A$
X=OPENIN (A$)
X=OPENIN ("FILE1")
```

The example below reads data from disk into an array. If the data file does not exist, an error message is printed and the program ends.

```
10 DIM posn(10),name$(10)
20 fnum=OPENIN "TOPTEN"
30 IF fnum=0 THEN PRINT "No TOPTEN data file": END
40 FOR i=1 TO 10
50   INPUT#fnum,posn(i),name$(i)
60 NEXT
70 CLOSE#fnum
```

See Annex E for an explanation of how opening and closing files can use up memory.

Associated Keywords:

OPENOUT, OPENUP, CLOSE#, PTR#, PRINT#, INPUT#, BGET#
BPUT#, EOF#

TOKEN 8E

OPENOUT
OPENO. A function which opens a file for writing and returns the 'channel number' of the file. This number must be used in subsequent references to the file with BPUT#, PRINT#, EXT#, PTR# or CLOSE#.

```
X=OPENOUT(A$)
X=OPENOUT("DATAFILE")
```

A returned value of zero signifies that the file could not be opened because the directory was full.

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 disk. Under these circumstances, a 'Disk full' error would be reported when you tried to write data to the file for the first time.

Example of writing a top ten array:

```
A=OPENOUT "TOPTEN"
FOR Z=1 TO 10
  PRINT#A,N(Z),N$(Z)
NEXT
CLOSE#A
```

See Annex E for an explanation of how opening and closing files can use up memory.

Associated Keywords:

OPENIN, OPENUP, CLOSE#, PTR#, PRINT#, INPUT#, BGET#
 BPUT#, EOF#

TOKEN AE

OPENUP
OPENU. Because the CP/M operating system manages disk space in a more flexible manner than the BBC Micro's DFS, there is no need to distinguish between opening a file for input and opening it for update. Once a file is opened you can update it or extend it. The only limitation is the room available on the disk.

BBCBASIC(Z80) treats the OPENUP command in an identical manner to OPENIN.

See Annex E for an explanation of how opening and closing files can use up memory.

Associated Keywords:

OPENIN, OPENOUT, CLOSE#, PTR#, PRINT#, INPUT#, BGET#
 BPUT#, EOF#

TOKEN AD

OPT

An assembler pseudo operation controlling the method of assembly. OPT is followed by an expression with the following meanings:

Code Assembled Starting at P%

<u>Value</u>	<u>Action</u>
0	assembler errors suppressed; no listing.
1	assembler errors suppressed; listing.
2	assembler errors reported; no listing.
3(default)	assembler errors reported; listing.

Code Assembled Starting at O%

<u>Value</u>	<u>Action</u>
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:

Branch out of range.

Unknown label.

Byte errors do not exist.

See the section on Assembler for more details.

Associated Keywords:

None

No TOKEN

OR

The operation of bitwise integer logical OR between two items. The 2 operands are internally converted to 4 byte integers before the OR operation.

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

If you insist, you can leave out the space between OR and a preceding constant. Please don't do it - 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, BBCBASIC 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 last example, uses the OR in a similar fashion to the numeric operators (+, -, etc).

```
A=X OR 11
```

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

```
11111111 11111111 11111111 11101100
00000000 00000000 00000000 00001011
11111111 11111111 11111111 11101111 = -17
```

Associated Keywords:

AND, EOR, NOT

TOKEN 84

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 * commands. Using this statement, you can, for instance, erase and rename files whose names you only know at run-time.

Unfortunately, you cannot legally call a CP/M operating system function and return from whence you came. Consequently, you cannot use this statement to communicate with the operating system directly.

```
command$="ERA PHONE.DTA"
OSCLI command$
```

```
command$="REN ADDRESS.DTA=NAME.DTA"
OSCLI command$
```

Associated Keywords:

All operating system (*) commands.

TOKEN FF

**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 BBCBASIC(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 BBCBASIC(Z80).

If you make PAGE greater than HIMEM, a 'No room' error will occur if the program exits to command level.

The least significant byte of PAGE is always set to zero by BBCBASIC(Z80).

With care, several programs can be left around in RAM without the need for saving them.

USE WITH CARE.

Associated Keywords:

TOP, LOMEM, HIMEM

TOKEN DO

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, BBCBASIC(Z80) has 2 functions (RAD and DEG) which perform these conversions to a higher accuracy.

Associated Keywords:

None

TOKEN AF

PLOT
PL.

A multi-purpose point, line and ellipse drawing statement. Three numbers follow the PLOT statement: the first specifies the type of point, line or ellipse to be drawn; the second and third give the X and Y co-ordinates to be used.

PLOT Mode,X,Y

Lines are drawn from the current position of the graphics cursor to the specified X,Y co-ordinates. The origin is usually the bottom left-hand corner of the screen, but it can be changed by using the PLOT statement with the Mode set to -1. The graphics screen is 0-1024 points wide and 0-767 points high.

Text can be printed on the screen at a specified place using the TAB(X,Y) command.

The 2 most commonly used statements, PLOT 4 and PLOT 5, have the duplicate keywords MOVE and DRAW

The available PLOT modes are listed below.

Basic Modes

<u>Mode</u>	<u>Action</u>
0	Move relative to the last point.
1	Draw a line, in the graphics foreground colour, relative to the last point.
3	Draw a line, in the graphics background colour, relative to the last point.
4	Move to the absolute position.
5	Draw a line, in the graphics foreground colour, to the absolute co-ordinates specified by X and Y.
7	Draw a line, in the graphics background colour, to the absolute co-ordinates specified by X and Y.

TOKEN FO

Enhanced Modes

<u>Mode</u>	<u>Action</u>
8-15	As 0-7, but the lines are drawn dotted.....
16-23	As 0-7, but the lines are drawn dashed. - - - - -
24-31	As 0-7, but the lines are drawn dot-dashed.- - -
37	Draw an ellipse in the graphics foreground colour centred at the current position of the graphics cursor. The second and third parameters specify the X and Y radii of the ellipse.
39	Draw an ellipse in the graphics background colour centred at the current position of the graphics cursor. The second and third parameters specify the X and Y radii of the ellipse.
45/47	As 37 and 39, but the lines are drawn dotted.
53/55	As 37 and 39, but the lines are drawn dashed.
61/63	As 37 and 39, but the lines are drawn dot-dashed.
64-71	As 0 to 7, but plot a single point at the 'destination'.
96-103	As 0 to 7, but fill the area around the point x,y.
128-159	Position a sprite at the given co-ordinates. (Sprites number 0 to 31 respectively.)
-1	Move the graphics origin to the absolute position specified by the second and third parameters. All subsequent MOVE, DRAW, PLOT (absolute), and POINT operations work with respect to this value.

Drawing Circles

To draw a circle, draw an ellipse with the y-radius equal to 4/3 the x-radius. For example,

PLOT 37,150,200

Filling Shapes

PLOT modes 96 to 103 may be used to fill previously drawn shapes. The X and Y co-ordinates specify the point at which filling will start and they may be specified in either absolute or relative form depending on the mode used. The specified point should be within the shape to be filled. If there is a 'hole' in the shape, the fill will 'leak' out. If the point specified is outside a shape, all of the screen apart from enclosed shapes will be filled.

The fill colour may be either the current graphics foreground or background colour.

As with modes 0 and 5, modes 96 and 101 do not actually plot anything on the screen, they simply move the graphics cursor.

Draw Relative

The 'draw relative' commands act relative to the last point plotted. Suppose, for example, the current X and Y co-ordinates were 200, 300. The command

PLOT 1,0,150

would draw a line in the current graphics foreground colour from 200, 300 to 200,450.

Associated Keywords:

MODE, CLG, MOVE, DRAW, POINT, VDU, GCOL

POINT

A function which returns the state of the pixel at the co-ordinates specified.

There must not be a space between POINT and the opening bracket.

```
state=POINT(X,Y)
```

```
IF POINT(X,Y)=1 THEN 300
```

The value returned is:

1 if the point is in a foreground colour.

0 if the point is in a background colour.

-1 if the point is off the screen.

You can use POINT to find out the state of the screen at the specified point and take action accordingly. In an adventure game, for example, the swamps may be marked in a foreground colour. If the explorer ventured into a marked area, 'swamp type demons' would be activated.

Associated Keywords:

PLOT, DRAW, MOVE, GCOL

NO TOKEN

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. (For the Einstein this is 39 or 31 depending on the MODE selected.)

X=POS

COUNT will tell you the print head position of the printer.

See VPOS for an example of the use of POS and VPOS.

Associated Keywords:

COUNT, TAB, VPOS

TOKEN B1

PRINT
P.

A statement which prints characters on the VDU screen or printer depending on the output stream selected (see *OPT). The printer may be turned on and off manually by typing ^P. All the characters displayed on the VDU screen after the first ^P will be echoed to the printer. A second ^P will turn the printer off.

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

TOKEN = F1

Initially, numeric items are printed in decimal. If a divide sign (\div) 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. (The divide sign on the Einstein generates the code value &7E - the ASCII tilde (~).)

```

z10
012345678901234567890123456789
PRINT  $\div$ 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 2 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 BBCBASIC, 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.

Byte No	Range	Default	Purpose
3 (MSB)	0-1	0	STR\$ Format Control
2	0-2	0	Format Selection
1	?-?	9	Number of Digits Printed
0 (LSB)	0-FF	0A(10)	Zone and Print Field Width

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.

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.

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 1-0A 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

E 1-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 zeroes 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 1.056000000000000E1

F 0-0A 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

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
```

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	&01020A
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
```

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\$

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 Annex E). Strings are written as the bytes in the string (in the correct order) plus a carriage return.

Before you use this statement, you must have opened a file using OPENOUT, OPENIN 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 BBCBASIC(Z80) the format of the file is completely under your control. See the section on BBCBASIC(Z80) Disk Files for more details.

Associated Keywords:

PRINT

TOKEN F1 + 23

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). No spaces are allowed between the procedure name and the opening bracket of the parameter list (if any).

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.

```
INPUT "Number of discs " F
PROC_hanoi(F,1,2,3)
END
DEF PROC_hanoi(A,B,C,D)
IF A=0 THEN ENDPROC
PROC_hanoi(A-1,B,D,C)
PRINT "Move disk ";A" from pile ";B" to pile ";C
PROC_hanoi(A-1,D,C,B)
ENDPROC
```

See the Procedures and Functions sub-section for more details.

Associated Keywords:

DEF, ENDPROC, LOCAL

TOKEN = F2

PTR#

A pseudo-variable allowing the random-access pointer of the file whose channel number 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#**, **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 BBCBASIC(Z80) Disk 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 **OPENIN** or **OPENUP** may be extended by setting its pointer to its end (**PTR#fnum=EXT#fnum**) and then writing to it. (Unfortunately, life is not quite this simple with CP/M - see the BBCBASIC(Z80) Disk Files section for more details.) If you do this, you must remember to **CLOSE** the file 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 BBCBASIC(Z80) Disk Files section has a number of examples of the use of **PTR#**.

Associated Keywords:

INPUT#, **PRINT#**, **BGET#**, **BPUT#**, **OPENIN**, **OPENOUT**, **OPENUP**, **EXT#**, **EOF#**, **CLOSE#**

TOKEN CF + 23

PUT A statement to output data to a Z80 port. The full Z80 extended addressing is available.

PUT A,N :REM output N to port A.

This instruction gives direct access from BBCBASIC(Z80) to the computer's I/O hardware. Typically, you can use it to directly access I/O ports.

On some computers with more than 64k of RAM available, you may be able to use it to control memory paging.

The instruction is of little use on the Torch computers because all I/O is carried out by the BBC Micro via the Tube.

This extension to the language has not been approved by Acorn and cannot be guaranteed to remain unchanged in future releases of BBCBASIC(Z80).

Associated Keywords:

GET

TOKEN CE

RAD A function which converts degrees to radians.

X=RAD(Y)
X=SIN RAD(90)

Unlike humans, BBCBASIC(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.

Associated Keywords:

DEG

TOKEN BZ

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.

Associated Keywords:

DATA, RESTORE

TOKEN F3

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. BBCBASIC(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.
```

Associated Keywords:

None

TOKEN = F4

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 2 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 2 line numbers would be renumbered correctly, but the last 2 would be left unchanged.

```
ON action GOSUB 100,200,(type*10+300),400,500
```

RENUMBER may be used in a program, but it will exit to the command mode on completion.

Associated Keywords:

LIST

TOKEN CC

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 BBCBASIC(Z80) repeat a set number of instructions until some condition is satisfied.

```
REPEAT UNTIL GET=13 :REM wait for CR
```

```
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 for more details on the working of the program stack.

Associated Keywords:

UNTIL

TOKEN FS

**REPORT
REPO.**

A statement which prints out the error string associated with the last error which occurred.

You can use this statement within your own error handling routines to print out an error message for those errors you are not able to cope with.

The example below is an error handling routine designed to cope only with the <ESCAPE> key being pressed. All other errors are reported and the program terminated.

```

10 ON ERROR GOTO 1000
20 .....

970 .....
980 END
990 :
1000 PRINT:
1010 IF ERR=17 THEN PRINT "<ESCAPE>ignored":GOTO 20
1020 REPORT:PRINT " at line ";ERL

```

See the sub-section on Error Handling and the keywords ERR, ERL and ON ERROR for more details.

Associated Keywords:

ERR, ERL, ON ERROR

TOKEN F6

**RESTORE
RES.**

RESTORE can be used at any time in a program to set the line where DATA is read from.

RESTORE on its own resets the data pointer to the first data item in the program.

RESTORE followed by a parameter 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.

```

RESTORE
RESTORE 100
RESTORE (10*A+20)

```

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.

Associated Keywords:

READ, DATA

TOKEN F7

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 BBCBASIC(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 (RETURN) exit point.

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 BBCBASIC(Z80) as well. The sub-section on Program Flow Control explains why.

Associated Keywords:

GOSUB, ON GOSUB

TOKEN F8

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.

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)
```

For example,

```
10 name$="BBCBASIC(Z80)"
20 FOR i=1 TO 15
30 PRINT RIGHT$(name$,i)
40 NEXT
50 END
```

would print

```
)
0)
80)
Z80)
(Z80)
C(Z80)
IC(Z80)
SIC(Z80)
ASIC(Z80)
BASIC(Z80)
CBASIC(Z80)
BCBASIC(Z80)
BBCBASIC(Z80)
BBCBASIC(Z80)
BBCBASIC(Z80)
```

Associated Keywords:

LEFT\$, MID\$

No TOKEN

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 0 - &FFFFFFF.

RND(1) returns a real number in the range 0.0 to .99999999.

RND(n) returns an integer in the range 1 to n.

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.

Associated Keywords:

None

TOKEN = B3

RUN

Start execution of the program.

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 can be used as an alternative to CHAIN.

Associated Keywords:

NEW, OLD, LIST, CHAIN

TOKEN F9

SAVE
SA.

A statement which saves the current program area to a file, in internal (tokenised) format.

```
SAVE "FRED.DOC"
SAVE AS
```

You use SAVE to save your program for use at a later time. The program must be given a name and this name becomes the name of the file in which your program is saved.

The file must follow the normal CP/M naming conventions. It should consist of a name of no more than 8 characters, optionally followed by a full-stop and a 3 character extension.

Unless a different extension is specified in the filename, .BBC is automatically used. Thus,

```
SAVE "FRED"
```

would save the current program to a file called FRED.BBC.

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."
```

Associated Keywords:

LOAD, CHAIN

TOKEN CD

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
```

Associated Keywords:

ABS

TOKEN BY

SIN A function giving the sine of its radian argument.

$X = \text{SIN}(Y)$

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.

$Y = \text{SIN}(\text{RAD}(\text{degree_angle}))$

Associated Keywords:

COS, TAN, ACS, ASN, ATN, DEG, RAD

TOKEN BS

SOUND

A statement which generates sounds using the internal loudspeaker. Four sounds may be generated simultaneously. Each sound channel generates one sound.

The SOUND statement is followed by 4 numeric parameters which specify the sound channel to be used, the loudness of the note (the amplitude), the pitch of the note and whether the note is to start or stop.

SOUND Channel, Loudness, Pitch, Control

Channel.

Channel selects the sound generator channel number. The channel number can be 0, 1, 2 or 3.

Channels 1, 2 and 3 produce single notes, but channel 0 can produce various noises which are explained in the section headed 'Noise Channel'.

The 3 note channels operate independently except that if an envelope is used, all 3 channels are controlled by the one envelope.

Loudness.

You can set the loudness to any integer between -15 and 1.

Constant amplitude notes are produced by values between -15 (the loudest) and 0.

If the loudness is set to 1, the amplitude of the sound is controlled by the currently specified envelope.

TOKEN D4

Pitch.

For channels 1 to 3, the frequency of the note produced is determined by the Pitch, which can have any value between -44 and 252. The table below shows the value of Pitch needed to produce a given note.

A pitch value of 100 will sound Middle C.

To go up or down an octave, the Pitch must be changed by 48. To go up a perfect 5th, the Pitch must be increased by 28. Increasing the Pitch by 4 will increase the frequency of the note produced by 1 semitone.

Note	Octave Number						
	1	2	3	4	5	6	7
B	0	48	96	144	192	240	
A#	-4	44	92	140	188	236	
A	-8	40	88	136	184	232	
G#	-12	36	84	132	180	228	
G	-16	32	80	128	176	224	
F#	-20	28	76	124	172	220	
F	-24	24	72	120	168	216	
E	-28	20	68	116	164	212	
D#	-32	16	64	112	160	208	
D	-36	12	60	108	156	204	252
C#	-40	8	56	104	152	200	248
C	-44	4	52	100	148	196	244

The values have been chosen to maximize compatibility with the BBC Micro. Because of this, the least 2 significant bits of the pitch value are ignored. Thus, 1, 2, and 3 would give rise to the same pitch.

Control

The last parameter, Control, determines whether a sound is started or stopped. If the value of Control is TRUE (-1), then the sound is started; if the value is FALSE (0) then the sound is stopped. Unless an envelope is used, a sound, once started, will continue until another SOUND command is issued to stop it or the <ESCAPE> key is pressed.

Noise Channel

As mentioned earlier, channel number zero produces 'noises' rather than notes. Only Pitch and Control have any effect on channel zero. The value of Pitch, which can be from 0 to 31, controls the 'frequency' of the noise produced. A value of zero gives high frequency noise and a value of 31 gives low frequency noise.

The volume (or envelope) of channel zero is controlled by the most recent SOUND statement for one of the note channels. Consequently, channel zero can only be used in conjunction with one or more of the note channels. Fortunately, because the note channel SOUND statement does not need to enable that channel for channel zero to work, it is possible to generate a noise sound on its own.

If the value of Loudness for the controlling note channel is 1, the noise channel will be controlled by the currently specified envelope. The example below generates a low frequency noise for a period which is controlled by the FOR...NEXT loop.

```
10 SOUND 1,-15,0,0
20 SOUND 0,0,30,TRUE
30 FOR i=1 TO 1000:NEXT
40 SOUND 0,0,0,0
```

The example below sounds a high-frequency noise for about 1 second.

```
10 ENVELOPE 7812,0
20 SOUND 1,1,0,0
30 SOUND 0,0,5,TRUE
```

Associated Keywords:

ENVELOPE, ADVAL

Notes on ENVELOPE and SOUND. It takes a little experimentation to become familiar with ENVELOPE and SOUND and we suggest that you use this page for notes.

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$
```

Associated Keywords:

TAB, PRINT, INPUT

TOKEN 89

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.

Associated Keywords:

None

TOKEN B6

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
```

Associated Keywords:

FOR, TO, NEXT

TOKEN 88

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.

Associated Keywords:

END

TOKEN FA

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 a divide sign (/), then a hexadecimal conversion is carried out. (The divide sign on the Einstein generates the code value &7E - the ASCII tilde (~).)

A\$=STR\$(PI)

B\$=STR\$/100 :REM B\$ will be "64"

The opposite function to STR\$ is performed by the VAL function.

Associated Keywords:

VAL, PRINT

TOKEN C3

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
```

Associated Keywords:

None

NO TOKEN

TAB A statement available in PRINT or INPUT.

There are 2 versions of TAB, TAB(X) and TAB(X,Y) and they are effectively 2 different statements.

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 VDU orientated statement. It will move the cursor on the VDU 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$
```

Associated Keywords:

POS, VPOS, PRINT, INPUT

NO TOKEN

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))

Associated Keywords:

COS, SIN, ACS, ATN, DEG, RAD

TOKEN B7

THEN
TH.

An optional part of the IF... THEN ... ELSE statement. It introduces the action to be taken if the testable condition evaluates to TRUE.

IF A=B THEN 3000

IF A=B THEN PRINT "Equal" ELSE PRINT "Help"

You need to use THEN if it is followed by:

a. A line number.

IF a=b THEN 320

b. A 'star' (*) command.

IF a=b THEN *DIR

c. An assignment of a pseudo-variable.

IF a=b THEN TIME=0

or you wish to exit from a function as a result of the test. This is because BBCBASIC(Z80) can't work out what you mean in these circumstances if you leave the THEN out.

IF A=B PRINT "Equal" ELSE PRINT "Help"

DEF FN_test(num)

....

....

IF a=b THEN =num: REM Without the 'THEN' this line
.... REM would not make sense.

....

=num/256

Associated Keywords:

IF, ELSE

TOKEN = 8C

TIME
TI. A pseudo-variable which reads and sets the elapsed time clock.

X=TIME
TIME=100

You can use TIME to set and read BBCBASIC(Z80)'s internal clock.

This clock is a 32 bit integer which counts in one-hundredth of a second (centi-second) 'ticks'. Its accuracy depends on the computer.

TIMES A function which returns a 6 character string (hhmmss) giving the current value of the Einstein's real-time clock.

The real-time clock may be read, but not set, by BBCBASIC(Z80). Unless set by a suitable utility program, the real-time clock will contain the time since the Einstein was switched on or the reset button was last pushed.

```
run_time$=TIMES$
PRINT TIMES$
```

The string hhmmss is held in memory at &FB8C. The program SETTIME.BBC supplied on your distribution disk will enable you to set the clock.

Associated Keywords:

None

TOKEN DI + 24

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
60 END
```

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
40 END
```

will print

20

Associated Keywords:

FOR, NEXT, STEP

TOKEN = B8

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

Associated Keywords:

PAGE, HIMEM, LOMEM

token B8 + 50

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 <ESCAPE>.

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] >_
```

```
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] >_
```

Associated Keywords:

None

token FC

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

```

BBCBASIC(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.

Associated Keywords:

FALSE

TOKEN B9

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 <ESCAPE> is pressed.) The following example will continually ask for a number and print its square. The only way to stop it is to press <ESCAPE> or force 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.

Associated Keywords:

REPEAT

TOKEN EP

USR

A function allowing machine code to return a value directly for things which do not require the expense of CALL.

USR calls the machine code subroutine whose start address is its argument. Prior to calling the subroutine, the processor's A, B, C, D, E, F, H, and L registers are initialised to the least significant bytes of A%, B%, C%, D%, E%, F%, H% and L%.

USR returns a 32-bit integer result composed of H, L, H', L' (most significant to least significant).

X=USR(lift_down)

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 the Z80's registers and the machine code routine returns the result in the H, L, H', L' registers.

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.

Torch Only

In the BBC Micro, CALL and USR can be used to access routines in the machine operating system (MOS). In order to maximize compatibility with the BBC Micro, addresses between &FF00 and &FFFF are assumed by BBCBASIC(Z80) to refer to the 6502's MOS. Consequently, CALL and USR behave differently when addressing this area of memory.

See Annex F for details of the behaviour of CALL and USER when calling addresses above &FF00.

Associated Keywords:

CALL

TOKEN BA

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.

Associated Keywords:

STR\$, EVAL

TOKEN BB

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.

```
VDU 8,8           :REM cursor left two places.
VDU &0A0D;&0A0D; :REM CRLF twice
```

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. For instance, you might need to send ESCAPE followed by 'H' to a console to turn inverted video on. You could do this with the line

```
PRINT CHR$(27);"H";
```

However,

```
VDU 27,72
```

is simpler and you don't have to remember the semi-colons.

Torch Only

The VDU statement is of considerable use on the Torch. It is used, for example, to control the position of the graphics origin, the physical to logical colour assignment, the size and position of the graphics area, control the printer, etc. See your Torch Programmer's Guide for more details.

Associated Keywords:

CHR\$

VPOS
VP.

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
```

Associated Keywords:

POS

TOKEN BC

WIDTH
W.

A statement controlling output overall field width.

WIDTH 80

If the specified width is zero (the initial value) the interpreter will not attempt to control the overall field width.

WIDTH n will cause the interpreter to force a newline after n MOD 256 characters have been printed.

WIDTH also affects the output to the printer.

Associated Keywords:

COUNT

TOKEN FE

OPERATING SYSTEM COMMANDS

1. Introduction. As with the BBC micro computer, the star '*' commands provide access to the operating system. Since the BBC operating system is completely different to the Einstein's DOS there are a number of differences in the '*' commands. The operating system commands were originally designed to be compatible with those used by the CP/M operating system. Consequently, some DOS commands are available under both the CP/M and Einstein DOS names.

2. 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.

3. Case Conversion. Star commands and their associated qualifiers are converted from lower-case to upper-case if necessary. For example, *era wombat is converted to *ERA WOMBAT. This is in keeping with the general DOS philosophy and the BBC Micro's machine operating system (MOS).

4. Special Characters. Control characters, lower-case characters, DEL and quotation marks may be incorporated in filenames by using the 'escape' character '|'. However, there is no equivalent to the BBC Microcomputer's '|!' to set Bit 7.

```

| A gives ^A.
| a gives lower-case A.
| ? gives DEL.
| " gives the quote marks ".
| | gives the escape character |.

```


5. File Specifiers. File specifiers must comply with the standard DOS conventions.

drive:filename.extension

drive: The single number name (0 to 3) of the drive where the file will be found. If the drive name is omitted, the currently logged-on drive will be assumed. The colon is mandatory.

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.

Drives 0: to 3: are accepted in file specifications. Filenames in 'star' commands may optionally be enclosed in quotes; unmatched quotes will cause a 'Bad string' error. The standard DOS 'wild-cards' may be used when an ambiguous file specifier is acceptable.

? 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.

6. 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 or variable.
 <str> A string constant or variable.
 afsp An ambiguous file specifier.
 ufsp An unambiguous file specifier.
 d: A disk drive name.

*BYE
 *B.

Return to the operating system (DOS).

*CHAR

Redefine the character specified by the first argument (0 to 255 decimal) to a new shape. The subsequent 8 arguments are the decimal values of the 8 lines comprising the character, top to bottom. Although characters 0 to 31 are not normally displayed, they may be used as sprites.

*CHAR n,r0,r1,r2,r3,r4,r5,r6,r7

See the sub-section on 'Shapes' at Chapter 17 of the Introduction to the Einstein Colour Computer for full details of character shapes and sprites.

*DIR
 *.

List the disk directory. The syntax is similar to the DOS DIR command except that the extension defaults to .BBC if it is omitted.

*DIR [afsp]

*DIR List all .BBC files on the disk.
 DIR 1:. List all files on disk 1:.
 .. List all the files on the currently logged disk.

*DISP

Type the specified file on the VDU screen. This command is similar in action to the DOS DISP command (and the CP/M TYPE command) except that the extension .BBC is assumed if it is omitted.

*DISP ufsp

*DISP letter.txt

*DOS

Return to the operating system (DOS).

*DRIVE
*DR. Selects the drive (0 to 3) to be used as the default drive for subsequent disk operations. The colon is a mandatory part of the drive name. (The standard CP/M drive notation of A: to D: is also accepted.)

*DRIVE n:

*DRIVE 1:

*DRIVE B:

*ERA
*E. Erase the specified disk file or files. The syntax is similar to the DOS ERA command except that the extension defaults to .BBC if it is omitted. OSCLI may also be used to erase files.

*ERA afsp

OSCLI "ERA "+<str>

*ERA *.BAK

OSCLI "ERA "+"*.BAK"

*EXEC
*EX. Accept console input from the specified file instead of from the keyboard. If the extension is omitted, .BBC is assumed.

*EXEC ufsp

*EXEC commands

If the file does not exist, a 'Channel' error will occur.

*EXEC cannot be issued from within a procedure, function, subroutine or loop.

*KEY
*K. Redefine the function key specified by the first argument (0 to 15) to contain the string given as the second argument.

*KEY n string

*KEY 3 *DIR|M

Keys 8 to 15 are <SHIFT> and f0 to f7.

If there is insufficient room for the string, a 'Bad string' message is displayed and the key is loaded with as much of the string as would fit.

*LOAD
*L.

Load the specified file into memory at hexadecimal address 'aaaa'. The load address MUST always be specified. The file length will always be a multiple of 128. 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 "+file_name\$+" "+STR\$~(start_address)

*LOCK

Set the specified file to LOCKED (read only) status. If the extension is omitted, .BBC is assumed.

*LOCK ufsp

*LOCK names.dta

*MAG

Set the sprite magnification factor (0 to 3).

*MAG n

See the sub-section on Sprites at Chapter 17 of the Introduction to the Einstein Colour Computer for details of sprites.

*MOS Transfer control to the Machine Operating System.
(See the DOS/MOS Introduction for details.)

*OPT Select the "output stream". The default is OPT 0.
*O.

*OPT0 Console output.
*OPT 1 Auxiliary output.
*OPT 2 List output.

*OPT 2
PRINT "THIS WILL APPEAR ON THE PRINTER"
*OPT 0

*PSW Use the specified password for subsequent disk
*P. operations. The password must contain exactly 8
 characters. If the password is omitted, the normal
 state (no password) is restored.

*PSW password
*PSW

*PSW fredrica
*PSW

*REN A command to rename a disk file. If omitted, the
*R. extension defaults to .BBC. OSCLI can also be used
 to rename a file.

*REN ufspold TO ufspnew
OSCLI "REN "+<str>+" TO "+<str>

*REN OLDFILE TO NEWFILE
OSCLI "REN "+filename\$+".BBCTO "+filename\$+".BAK"

If an ambiguous filename is used, a 'Bad name' error
(ERR=204) will occur. If a file already exists with
the new name, a 'File exists' error (ERR=196) will
occur.

The standard CP/M syntax of 'REN ufspnew=ufspold' is
also accepted.

*RESET Reset the disk system. You should use *RESET after
*RES. you have changed a disk. This command does not close
 any files nor does it perform any other housekeeping
 function.

*SAVE Save an area of memory to disk. You MUST specify the
*S. start address (aaaa) and either the length of the
 area of memory (llll) or its end address+1 (bbbb).
 The amount saved will always be rounded up to a
 multiple of 128. There is no 'load address' or
 'execute address'. The default extension is .BBC.
 OSCLI can also be used to save a file.

*SAVE ufsp aaaa +llll
*SAVE ufsp aaaa bbbb
OSCLI "SAVE"+<str>+" "+STR\$(<num>)+" "+STR\$(<num>)

*SAVE "WOMBAT" 8F00 +80
*SAVE "WOMBAT" 8F00 8F80
OSCLI "SAVE"+ufn\$+" "+STR\$(add)+" "+STR\$(len)

If an ambiguous filename is used, a 'Bad name' error
(ERR=204) will occur.

*SPOOL Copy all subsequent console output to the specified
*SP. file. If the file name is omitted, any current spool
 file is closed and spooling is terminated. If the
 extension is omitted, .BBC is assumed.

*SPOOL filename
*SPOOL

If the directory is full, a 'Channel' error will
occur.

*SPOOL cannot be issued from within a procedure,
function, subroutine or loop. The *DIR (*) command
will not work properly while SPOOLing.

*SPRITE Create sprite number 'num' (0 to 31) with a shape corresponding to the character 'char_num' (0 to 255) and colour 'col' (0 to 15).

*SPRITE num,char_num,col

The shape of the character may be defined with the *CHAR command. If the sprite magnification (set by *MAG) is 2 or 3, then the sprite shape corresponds to that of 4 consecutive characters and four *CHAR commands will be needed.

Sprites are positioned using the PLOT statement.

See the sub-section on sprites at Chapter 17 of the Introduction to the Einstein Colour Micro Computer for more information about sprites.

*TYPE Type the specified file on the VDU screen. This command is similar in action to the DOS DISP command (and the CP/M TYPE command) except that the extension .BBC is assumed if it is omitted.

*TYPE ufsp

*TYPE FRED.DTA

*UNLOCK Set the specified file to 'unlocked (read-write) status. If the extension is omitted, .BBC is assumed.

*UNLOCK ufsp

*UNLOCK names.dta

BBCBASIC(Z80) DISK FILES

INTRODUCTION

1. These notes start with some basic information on files, and then go on to discuss program file manipulation, simple serial files, random and, finally, indexed files. The commands and functions used are explained, and followed by examples.

2. If you are new to BBCBASIC, or you are experiencing difficulty with disk files you should 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.

3. You will find the programs listed in these notes on your BBCBASIC(Z80) distribution disk (possibly with a few additions). They are **definitely NOT copyright** and, if you want to, you are free to incorporate any of the code in the programs you write. Use them, change them, or ignore them as you wish. There is only one proviso; the programs have been tested and used a number of times, but I cannot say with certainty that they are bug free. Remember, debugging is the art of taking bugs out - programming is the art of putting them in.

4. A number of the examples send control code strings to the VDU to clear to the end of the line/screen etc. These codes are computer dependent and they will probably have to be changed before the programs will run properly on your computer. In all but the simplest examples, an initialisation procedure is used to set variables to these control code strings and these variables are used in the rest of the program. Check and, if necessary, change these strings to suit your computer.

If you have any comments on these notes, suggestions for different file procedures, or better ways of explaining things, please send them to me at M-TEC Computer Services (UK).

Doug Mounter
Jan 83/Aug 84

THE STRUCTURE OF FILES

1. If you understand the way files work, skip paragraphs 2 to 4.
4. If you understand random and indexed files, skip paragraphs 5 and 6 as well.

BASICS

2. 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.

3. All computers are able to store and retrieve information from a non-volatile medium. (You don't lose the information when the power gets turned off.) Audio cassettes are used for small micro computers, 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 organized 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

4. Look upon the cassette or diskette 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.

5. 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.

6. Life is easier with a computer that uses diskettes (or disks). There is an index which tells the computer where to look for each of the files and the serial search for the file is not necessary. However, once you have found the file, you still need to read through it to find the record you want.

7. 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

8. 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.

9. 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.

10. 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 Zeds, 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.

11. The same happens with this sort of file on a diskette. A random file which has a lot of empty space in it is described as sparse. Most random files start this way and most never get more than about 3/4 full. Count the number of empty 'slots' in your address book and see what proportion this is of the total available.

Indexed Files

12. 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 organized except that we don't know the page number for each name.

13. 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.

14. 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.

15. 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).

16. 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 BBCBASIC 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 BBCBASIC(Z80)

Introduction

17. Conventional serial disk 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).

18. 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 on the disk 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.

19. Most versions of BASIC only offer serial and random files, but because of the way that disk files are handled by BBCBASIC (both on the BBC computer and CP/M computers using BBCBASIC(Z80)), 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

20. 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 disk operating system (DOS), but this need only concern you when you are organizing your program for maximum disk efficiency.

21. 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. BBCBASIC 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.

How Data are Stored

22. Numeric Data. In order to make the most efficient use of disk 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.

23. 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#.

The Limitations of CP/M

24. In the disk's directory there is a file control block (FCB) for every file on the disk. The FCB holds information about the length of a file and which physical areas of the disk it occupies. Unfortunately, the length is only held to the nearest sector and the physical location to the nearest block. The sector size is 128 bytes and the block size is generally 1, 2, 4 or 8 kbyte depending on the size of the disk. Thus, it is impossible to tell the exact length of a file by reference to the FCB and, in the case of a sparse random file, you cannot tell which sectors of the occupied blocks hold data and which don't. As a result, you cannot set the end-of-file flag (EOF#) with any precision by reference to the FCB in the directory. The best you can get is to the nearest sector for the end of a file and the nearest block for the middle of a sparse random file. In order to alleviate this problem for character data files, the unused area of blocks allocated to the file can be filled with zeros (nulls) by the CP/M DOS. This facility is used by BBCBASIC(Z80).

25. BBCBASIC(Z80) File Format. BBCBASIC provides the facility for completely free-format binary data files. Any CP/M file, from any source and in any format, can be processed using the BGET, BPUT and PTR functions. Because of this, it is not possible to have a discrete end-of-file marker, like control Z or null. (A numeric could have one of its bytes set to 1A or 00.) Unfortunately, this means that you cannot use a 'marker' to set EOF and, as a consequence, EOF cannot be used to indicate that you have just read the last data item in a file containing numeric or mixed character/numeric records.

Overcoming the Limitations of CP/M

26. When using simple serial BBCBASIC(Z80) data files one or more null items (zero value or null strings) may be read at the end of a file before EOF becomes true. If you can tolerate these nulls, then you can use EOF#; if not, you can include a deliberate end-of-file marker as the last record of a file. Alternatively, you can write the length of the file as the first record of the file.

27. The first method is quite satisfactory for character string files and the use of Control Z would go some way to making the file compatible with those produced by most other versions of BASIC.

28. For numeric or mixed character/numeric files, the most satisfactory method is to write the length of the file as the first record. This is no problem for BBCBASIC. When you open the file PTR is 0. Simply write a dummy numeric value as the first record and then write data to the file in the normal way. When you close the file, write the current value of PTR to the position occupied by the dummy value before issuing the CLOSE command.

29. Examples Serial-3, Serial-8, and Indexed illustrate these methods of finding the end of a file.

30. The examples in the manual were originally written for an 80 by 24 display. The example files included on your distribution disk have been altered to run on the Einstein's 40 by 24 display.

DISK FILE COMMANDS

1. Introduction. The commands and statements used in disk file manipulation are described below. They are not in alphabetical order, but in the order you are likely to want to use them.

2. Conventions. The CP/M operating system allows a composite file name in the following format:

FILENAME.EXTension

The file name can be up to 8 characters long, and the extension up to 3 characters. Whenever a file name without an extension is given, BBCBASIC(Z80) will append .BBC as the default extension.

3. Organization of Examples. Simple examples are given throughout this section with the explanation of the various commands. As printed in this manual, they are suitable for an 80 by 24 display, but the example files included on your distribution disk are tailored especially for the Einstein's 40 by 24 display. 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 filename Save the current program to a file, in internal (tokenised) format. The filename can be a variable or a string.

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.

Associated keywords:

LOAD, CHAIN

LOAD filename 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. File names must conform to the standard DOS format. However, if no extension is given, .BBC is assumed.

LOAD "FRED"

A\$="HEATING"
LOAD A\$

The file specifier may be ambiguous. The file with the first entry in the directory which matches the specifier will be loaded.

LOAD "INT*"

Associated keywords:

SAVE, CHAIN

CHAIN filename LOAD and RUN the program 'filename'. All the dynamic variables are cleared. The program must be in tokenised format.

CHAIN "GAME1"

A\$="PART2"
CHAIN A\$

Associated keywords:

LOAD, SAVE

MERGE

There are 2 ways of merging BBCBASIC(Z80) programs.

Using MERGE.BBC

MERGE.BBC is an UNLISTED BBCBASIC(Z80) program which combines 2 program files into a third program file. It asks you for the names of the 2 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 still 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 2 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<RETURN>

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<RETURN>
OLD<RETURN>

Associated keywords:

None

*ERA filename Erase the file 'filename'. Since variables are not allowed as arguments to * commands, the filename must be a constant.

*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 BBCBASIC 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.

command\$="ERA FRED"
OSCLI command\$

command\$="ERA PHONE.DTA"
OSCLI command\$

OSCLI "ERA FRED"

Do not erase a file which is open. BBCBASIC will report an error if the file for a particular channel number does not exist when you close it.

Associated keywords:

None

*REN f1 TO f2 Rename the file called 'f1' to be called 'f2'. If omitted, the extension defaults to .BBC.

*REN FRED1 TO FRED2
*REN PHONE TO PHONE.DTA

Once again, if you want to rename files whose names are only known at run-time, you must use the OSCLI command.

command\$="REN FRED1 TO FRED2"
OSCLI command\$

OSCLI "REN FRED1 to FRED2"

Do not rename an open file. As with erase, BBCBASIC will report an error if you subsequently try and do anything with the file. This is because its name no longer corresponds to the one recorded for the channel.

Associated keywords:

None

*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 DOS DIR command.

*DIR List *.BBC files on
the current drive.

DIR B:.DTA List *.DTA files on
drive B.

Associated keywords:

None

UNLIST

You may wish to protect your program from 'prying eyes'. You cannot do this directly from within BBCBASIC(Z80), but an UNLIST utility has been provided.

To use the UNLIST utility, save your program with the default .BBC extension to its name, exit from BBCBASIC(Z80) using *DOS (*BYE) and type

UNLIST filename<RETURN>

Do NOT add the extension .BBC to the file name.

To display a brief explanation of UNLIST, just type

UNLIST<RETURN>

For example, to unlist a program file called 'TEST.BBC', type

UNLIST TEST<RETURN>

If you want to use UNLIST, you must not use calculated line numbers in GOTO, GOSUB or RESTORE statements.

Associated keywords:

None

DISK DATA FILESIntroduction.

The statements and functions used for data files are:

OPENIN (and 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, BBCBASIC chooses the number for you.

When you open the file, the file (or channel) 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 the file number.)

You use the file number 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 number returned will be 0. This will occur if you try to open a non-existent file in the input mode (OPENIN) or if the directory is full when you try to open a file in the output mode (OPENOUT).

The 2 functions which open files are OPENIN and OPENOUT (and OPENUP). OPENOUT should be used to create new files, or overwrite old ones. OPENIN (or OPENUP) should be used for all other file input and output.

OPENOUT filename Open the file 'filename' for output and return the channel number 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.)

```
file_num=OPENOUT "PHONENUMS"
```

You always need to store the channel number 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$
```

You should be unable to open a file for output (channel number returned = 0) only if the directory is full.

Associated keywords:

```
OPENIN, OPENUP, CLOSE#, PTR#, PRINT#, INPUT#
BGET#, BPUT#, EOF#
```

OPENIN filename Open the file 'filename' for 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 OPENIN (or OPENUP) for reading serial and random files, adding to the end of serial files or writing to random files.

```
address=OPENIN "ADDRESS"
check_file=OPENIN check_file$
```

You will be unable to open for input (channel number returned = 0) if the file does not already exist.

OPENUP is identical to OPENIN in BBCBASIC(Z80).

Associated keywords:

```
OPENOUT, CLOSE#, PTR#, PRINT#, INPUT#, BGET#
BPUT#, EOF#
```


INPUT#fnum,var Read from the file opened as 'fnum' into the variable 'var'. Several variables can be read using the same INPUT# statement.

```
data=OPENIN "DATA"
:
:
INPUT#data,name$,age,height,sex$
:
:
```

READ# can be used as an alternative to INPUT#

Associated keywords:

OPENIN, OPENUP, OPENOUT, CLOSE#, PTR#, PRINT#
INPUT#, BGET#, BPUT#, EOF#

PRINT#fnum,var Write the variable 'var' to the file opened as 'fnum'. Several variables can be written using the same PRINT# statement.

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$,age,height,sex$
:
:
```

Associated keywords:

OPENIN, OPENUP, OPENOUT, CLOSE#, PTR#, INPUT#
INPUT#, BGET#, BPUT#, EOF#

CLOSE#fnum

Close the file opened as 'fnum'. CLOSE#0, END, untrapped errors or 'dropping off the end' of a program will close all files.

```
data=OPENOUT "DATA"
```

```
:
```

```
:
```

```
PRINT#data,name$,age,height,sex$
```

```
:
```

```
:
```

```
CLOSE#data
```

```
:
```

Associated keywords:

```
OPENIN, OPENUP, OPENOUT, PTR#, INPUT#, PRINT#
INPUT#, BGET#, BPUT#, EOF#
```

EOF#fnum

A function which will return TRUE (-1) if the last byte of the last sector of the file opened as 'fnum' has been read.

As mentioned earlier, EOF# is an uncertain indicator that the end of the file has been reached. It is more of an error message telling you that there is definitely no more data available. However, it can be used successfully with files containing only character data. An example is included in the section on serial files.

EOF will also be true if an attempt has been made to read from an empty block of a sparse random access file. (In other words, a block has not yet been allocated to this record by CP/M.) The unused areas of blocks for which EOF is FALSE will contain nulls. By using EOF combined with a test for nulls, it is possible to make a certain check for the presence of character data. Consequently, it is unnecessary to initialise a random file before writing to it.

Writing to a previously unused block will reset EOF immediately, irrespective of whether the record is physically written to the disk at that time or not.

Associated keywords:

```
OPENIN, OPENUP, OPENOUT, PTR#, INPUT#, PRINT#
INPUT#, BGET#, BPUT#, CLOSE#
```


PTR#fnum

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.

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#, PRINT#, 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 OPENIN may be extended by setting PTR# to its end, 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.

Associated keywords:

OPENIN, OPENUP, OPENOUT, EOF#, INPUT#, PRINT#
INPUT#, BGET#, BPUT#, CLOSE#

BGET#fnum

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.

```
byte=BGET#fnum
char$=CHR$(byte)
```

or, more expediently

```
char$=CHR$(BGET#fnum)
```

Associated keywords:

OPENIN, OPENUP, OPENOUT, EOF#, INPUT#, PRINT#
INPUT#, PTR#, BPUT#, CLOSE#

BPUT#fnum,var

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,&1B
BPUT#fnum,house_num
BPUT#fnum,ASC "E"
```

Associated keywords:

OPENIN, OPENUP, OPENOUT, EOF#, INPUT#, PRINT#
INPUT#, PTR#, BGET#, CLOSE#

EXT#fnum

Return the total length of the file opened as 'fnum'. Because of the limitations on the information available from the CP/M FCB, EXT# will only return the length of the file to the next largest multiple of 128.

In the case of a sparse random-access file the value returned is the virtual file length. This may well be greater than the actual amount of data on the disk.

EXT#,fnum performs a directory read and is therefore quite slow. If repeated use of the file length is necessary, use EXT once and save the result in a variable.

length=EXT#fnum

Associated keywords:

OPENIN, OPENUP, OPENOUT, EOF#, INPUT#, PRINT#
INPUT#, PTR#, BGET#, BPUT#, CLOSE#

SERIAL FILES

1. The section on serial files is split into 3 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 and introduces the concept of writing the file length as the first record. The final part describes conversion between BBCBASIC(Z80) format files and the file formats required/produced by other systems.

2. In a number of instances, the program lines are too long for the printed page. When this occurs, I have folded them round without adding a line number. This won't, of course, work in the actual program.

CHARACTER DATA FILES

3. The first 3 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.

4. When a file is closed after writing, the unused portion, if any, of the last block of the file is filled with nulls. This means that, unless the last data item exactly fills the last block, a null string will be read after the last data item before EOF becomes true. Thus, you can use EOF to terminate the reading of this type of data file if the possibility of an extra null string is acceptable.

5. You may notice that I have cheated a little in that I call a procedure to close the files, but I never return from it. This saves me using a dreaded 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.

6. Example 1 - Writing Serial Character Data

SAME AS DISC

```
10 REM F_WSER1
20 :
30 REM EXAMPLE OF WRITING TO A SERIAL CHARACTER DATA FILE
40 :
50 REM This program opens a data file and writes character
60 REM data to it. The use of OPENOUT ensures that, even if
70 REM the file existed before, it is cleared before being
80 REM written to.
85 :
90 :
100 phonenos=OPENOUT "PHONENOS"
110 PRINT "File Name PHONENOS Opened as Channel ";phonenos
120 PRINT
130 REPEAT
140 INPUT "Name ? " name$
150 IF name$="" THEN PROC_end
155 INPUT "Phone Number ? " phone$
160 PRINT
170 PRINT#phonenos,name$,phone$
180 UNTIL FALSE
190 :
200 DEF PROC_end
210 CLOSE#phonenos
220 END
```

7. Example 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 file and reads
55 REM it.
60 :
70 :
80 phonenos=OPENIN "PHONENOS"
90 PRINT "File Name PHONENOS Opened as Channel ";phonenos
100 PRINT
110 REPEAT
120 INPUT#phonenos,name$,phone$
130 IF name$<>"" PRINT name$,phone$
140 UNTIL EOF#phonenos
150 :
160 CLOSE#phonenos
170 END
```

130 IF EOF #phonenos THEN PROC_end

170 DEF PROC_end

8. Example 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 and then adds data to it. Because we can only get the length of the file to the nearest sector, we can't just set the PTR#file=EXT#file. (This would waste the unused part of the final sector.) However, we can set the pointer to 1 sector less than the extent (in this case 128 bytes) and then read through the file until we come to the end. The value of PTR before we tried to read the nulls at the end of the file is the place where our new data should be written.


```

10 REM F_WESER1
20 :
30 REM EXAMPLE OF WRITING TO THE END OF A CHARACTER DATA FILE
40 :
50 REM This program opens a file and sets PTR to the end
60 REM before writing more 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 Channel ";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 PTR is set to the last valid data byte+1 and the file
300 REM number is returned. If the file does not already
310 REM exist, it is still opened, but PTR is left at 0.
320 :
330 DEF FN_openend(name$)
340 LOCAL fnum
350 fnum=OPENIN(name$)
360 IF fnum=0 THEN fnum=OPENOUT(name$): =fnum
370 PTR#fnum=(EXT#fnum)-128
380 REPEAT: REM Skip valid data
390   last_PTR=PTR#fnum
400   INPUT#fnum,dummy$
410 UNTIL EOF#fnum AND dummy$=""
420 PTR#fnum=last_PTR
430 =fnum

```

MIXED NUMERIC/CHARACTER DATA FILES

9. The second 3 examples are also programs which write data to a file and read it back, but this time the data is mixed. With this type of file you cannot tolerate nulls, since they could be a valid data value. For the same reason, you cannot use a control Z as a file terminator. The most convenient way to overcome this problem is to store the length of the file as the first record. This is not difficult with BBCBASIC, since the file pointer may be set 0 and the length written there when the file is closed. All you have to remember to do is to write a dummy length immediately you open the file in order to reserve the space. (Alternatively, you could set the pointer to 5 before you started writing data to the file.) This way of storing and retrieving the file length makes opening 'at end' very much simpler.

10. Example 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 data file and writes mixed data to
60 REM it. The use of OPENOUT ensures that, even if the file
70 REM existed before, it is cleared before being written to.
80 :
90 REM When the file is opened a dummy length is written as
100 REM the first record and when the file is closed this is
110 REM changed to the final value of PTR. Functions are used
120 REM to open and close the files. The closing function
130 REM returns the final value of PTR (the file length).
140 :
150 REM Functions are also used to accept and validate the data
160 REM before writing it to the file.
170 :
180 :

```



```

190 stats=FN_open("STATS")
200 PRINT "File Name STATS Opened as Channel ";stats
210 PRINT
220 REPEAT
230   name$=FN_name
240   IF name$="" THEN PROC_end
250   age=FN_age
260   height=FN_height
270   sex$=FN_sex
280   PRINT
290   PRINT#stats,name$,age,height,sex$
300 UNTIL FALSE
310 :
320 DEF PROC_end
330 PRINT "The final byte was written at ";FN_close(stats)
340 END
350 :
360 :
370 REM Open the file and write a dummy length (0) as the first
380 REM record. Return the file number.
390 :
400 DEF FN_open(name$)
410 LOCAL fnum
420 fnum=OPENOUT(name$)
430 PRINT#fnum,0
440 =fnum
450 :
460 :
470 REM Write the length of the file to the first record and
480 REM then close the file. Return the length of the file.
490 :
500 DEF FN_close(fnum)
510 LOCAL EOFfnum
520 EOFfnum=PTR#fnum-1
530 PTR#fnum=0
540 PRINT#fnum,EOFfnum
550 CLOSE#fnum
560 =EOFfnum
570 :
580 :

```

```

590 REM Accept a name from the keyboard and make sure it
600 REM consists of spaces and upper or lower case characters.
610 REM Leading spaces are automatically ignored on input.
620 :
630 DEF FN_name
640 LOCAL name$,FLAG,n
650 REPEAT
660   FLAG=TRUE
670   INPUT "Name ? " name$
680   IF name$="" THEN 740
690   FOR I=1 TO LEN(name$)
700     n=ASC(MID$(name$,I,1))
710     IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123)
       THEN FLAG=FALSE
720   NEXT
730 IF NOT FLAG THEN PRINT "No funny characters please !!!"
740 UNTIL FLAG
750 =name$
760 :
770 :
780 REM Accept the age from the keyboard and round to one
790 REM place of decimals. Ages of 0 and less or 150 or more
800 REM are in error.
805 :
810 DEF FN_age
820 LOCAL age
830 REPEAT
840   INPUT "What age ? " age
850   IF age<=0 OR age >=150 THEN PRINT "No impossible ages
       please !!!"
860 UNTIL age>0 AND age<150
870 =INT(age*10+.5)/10
880 :
890 :
900 REM Accept the height in centimeters from the keyboard and
910 REM round to an integer. Heights of 50 or less and 230 or
920 REM more are in error.
930 :
940 DEF FN_height
950 LOCAL height
960 REPEAT
970   INPUT "Height in centimeters ? " height
980   IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
990 UNTIL height>50 AND height<230
1000 =INT(height+.5)
1010 :
1020 :
1030 REM Accept the sex from the keyboard. Only words beginning
1040 REM with upper or lower case M or F are acceptable. The
1050 REM returned string is truncated to 1 character.
1060 :

```



```

1070 DEF FN_sex
1080 LOCAL sex$,FLAG
1090 REPEAT
1100   FLAG=TRUE
1110   INPUT "Male or Female - M or F ? " sex$
1120   IF sex$<>" " THEN sex$=CHR$(ASC(sex$) AND 95)
1130   IF sex$<>"M" AND sex$<>"F" THEN FLAG=FALSE
1140   IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
1150 UNTIL FLAG
1160 =sex$

```

11. Example 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 data file and reads numeric and
60 REM character data from it.
70 :
80 REM When the file is opened its length is read from the
90 REM first record and used to determine when the end of the
100 REM data has been reached. A procedure is used to open the
110 REM file; it sets both the file number and the length of
120 REM the file. In effect, it is a function returning 2
130 REM values.
135 :
140 :
150 PROC_open("STATS"): REM Sets both file_num and length
160 stats=file_num
170 EOFstats=EOFfnum
180 PRINT "File Name STATS Opened as Channel ";stats
190 PRINT
200 REPEAT
210   INPUT#stats,name$,age,height,sex$
220   PRINT "Name ";name$
230   PRINT "Age ";age
240   PRINT "Height in centimeters ";height
250   IF sex$="M" THEN PRINT "Male" ELSE PRINT "Female"
260   PRINT
270 UNTIL PTR#stats>EOFstats
280 :
290 DEF PROC_end
300 CLOSE#stats
310 END

```

```

320 :
330 :
340 REM Open the file and read its length from the first record.
350 REM Set the file number (stats) and the length of the file
380 REM (length).
370 :
380 DEF PROC_open(name$)
390 file_num=OPENIN(name$)
400 INPUT#file_num,EOFfnum
410 ENDPROC

```

12. Example 6 - Writing 'AT END' of Mixed Files. This example has the same function as example 3, but for mixed data files.

```

10 REM F_WESER2
20 :
30 REM EXAMPLE OF WRITING AT THE END OF A MIXED NUMERIC/CHAR
40 REM DATA FILE
50 :
60 REM This program opens a data file, sets PTR to its end and
70 REM then writes numeric and character data to it.
80 :
90 REM If the file is being opened for the first time, a dummy
100 REM length is written as the first record. When the file
110 REM is closed the first record is changed to the value of
120 REM PTR. Functions are used to open and close the file.
130 REM The closing function returns the file length.
140 :
150 REM Functions are also used to accept and validate the data
160 REM writing it to the file.
170 :
180 :
190 stats=FN_open("STATS")
200 PRINT "File Name STATS Opened as Channel ";stats
210 PRINT
220 REPEAT
230   name$=FN_name
240   IF name$="" THEN PROC_end
250   age=FN_age
260   height=FN_height
270   sex$=FN_sex
280   PRINT
290   PRINT#stats,name$,age,height,sex$
300 UNTIL FALSE
310 :

```



```

320 DEF PROC_end
330 PRINT "The final byte was written at ";FN_close(stats)
340 END
350 :
360 :
370 REM Open the file and, if it exists, set PTR to the end and
380 REM return the file number. If it does not exist, open it,
390 REM write a dummy length as the first record and return the
395 REM file number.
400 :
410 DEF FN_open(name$)
420 LOCAL fnum
430 fnum=OPENIN(name$)
440 IF fnum>0 THEN INPUT#fnum,EOFFnum:PTR#fnum=EOFFnum+1: =fnum
450 fnum=OPENOUT(name$)
460 PRINT#fnum,0
470 =fnum
480 :
490 :
500 REM Write the length of the file to the first record and then
510 REM close the file. Return the length of the file.
520 :
530 DEF FN_close(fnum)
540 LOCAL EOFFnum
550 EOFFnum=PTR#fnum-1
560 PTR#fnum=0
570 PRINT#fnum,EOFFnum
580 CLOSE#fnum
590 =EOFFnum
600 :
610 :
620 REM Accept a name from the keyboard and make sure it
630 REM consists of spaces and upper or lower case characters.
640 REM Leading spaces are automatically ignored on input.
650 :
660 DEF FN_name
670 LOCAL name$,FLAG,n
680 REPEAT
690   FLAG=TRUE
700   INPUT "Name ? " name$
710   IF name$="" THEN 770
720   FOR I=1 TO LEN(name$)
730     n=ASC(MID$(name$,I,1))
740     IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123)
750       THEN FLAG=FALSE
750   NEXT
760   IF NOT FLAG THEN PRINT "No funny characters please !!!"
770 UNTIL FLAG
780 =name$
790 :
800 :

```

```

810 REM Accept the age from the keyboard and round to one place
820 REM of decimals. decimals. Ages of 0 and less or 150 or
830 more are in error.
835 :
840 DEF FN_age
850 LOCAL age
860 REPEAT
870   INPUT "What age ? " age
880   IF age<=0 OR age >=150 THEN PRINT "No impossible ages please !"
890 UNTIL age>0 AND age<150
900 =INT(age*10+.5)/10
910 :
920 :
930 REM Accept the height in centimeters from the keyboard and round
940 REM to an integer. Heights of 50 or less and 230 or more are
950 REM in error.
960 :
970 DEF FN_height
980 LOCAL height
990 REPEAT
1000   INPUT "Height in centimeters ? " height
1010   IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
1020 UNTIL height>50 AND height<230
1030 =INT(height+.5)
1040 :
1050 :
1060 REM Accept the sex from the keyboard. Only words beginning
1070 REM with upper or lower case M or F are acceptable. The
1080 REM returned string is truncated to 1 character.
1090 :
1100 DEF FN_sex
1110 LOCAL sex$,FLAG
1120 REPEAT
1130   FLAG=TRUE
1140   INPUT "Male or Female - M or F ? " sex$
1150   IF sex$<>" " THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
1160   IF sex$<>"M" AND sex$<>"F" THEN FLAG=FALSE
1170   IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
1180 UNTIL FLAG
1190 =sex$

```


COMPATIBLE DATA FILES

13. 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 2 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.

14. Example 7 - Writing a Compatible Data File

```

10 REM F_WSTD
20 :
30 REM EXAMPLE OF WRITING A COMPATIBLE DATA FILE
40 :
50 REM This program opens a file and writes numeric and
60 REM character data to it in a compatible format. Numerics
70 REM are changed to strings before they are written and the
80 REM data items are separated by commas. Each record is
90 REM terminated by CR LF and the file is terminated by a
100 REM Control Z.
105 :
110 :
120 REM Functions are used to accept and validate the data
130 REM before writing it to the file.
140 :
150 :
160 record$=STRING$(100," "): REM Reserve room for the longest
170 name$=STRING$(20," ") REM record necessary. It saves
180 : REM on string space.
190 compat=OPENOUT("COMPAT")
200 PRINT "File Name COMPAT Opened as Channel ";compat
210 PRINT

```

```

220 REPEAT
230   name$=FN_name
240   IF name$="" THEN PROC_end
250   age=FN_age
260   height=FN_height
270   sex$=FN_sex
280   PRINT
290   record$=name$+", "+STR$(age)+", "+STR$(height)+", "+sex$
300   PRINT#compat,record$
310   BPUT#compat,&0A
320 UNTIL FALSE
330 :
340 DEF PROC_end
350 BPUT#compat,&1A
360 CLOSE#compat
370 END
380 :
390 :
400 REM Accept a name from the keyboard and make sure it
410 REM consists only of spaces and upper or lower case
420 REM characters. Leading spaces are automatically ignored
430 REM on input.
435 :
440 DEF FN_name
450 LOCAL name$,FLAG,n
460 REPEAT
470   FLAG=TRUE
480   INPUT "Name ? " name$
490   IF name$="" THEN 550
500   FOR I=1 TO LEN(name$)
510     n=ASC(MID$(name$,I,1))
520     IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123)
530       THEN FLAG=FALSE
540   IF NOT FLAG THEN PRINT "No funny characters please !!!"
550 UNTIL FLAG
560 =name$
570 :
580 :
590 REM Accept the age from the keyboard and round to one place
600 REM of decimals. Ages of 0 and less or 150 or more are in
610 REM error.
615 :
620 DEF FN_age
630 LOCAL age
640 REPEAT
650   INPUT "What age ? " age
660   IF age<=0 OR age >=150 THEN PRINT "No impossible ages
670     UNTIL age>0 AND age<150
680   =INT(age*10+.5)/10

```



```

690 :
700 :
710 REM Accept the height in centimeters from the keyboard and
720 REM round to an integer. Heights of 50 or less and 230 or
730 REM more are in error.
740 :
750 DEF FN_height
760 LOCAL height
770 REPEAT
780   INPUT "Height in centimeters ? " height
790   IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
800 UNTIL height>50 AND height<230
810 =INT(height+.5)
820 :
830 :
840 REM Accept the sex from the keyboard. Only words beginning
850 REM with upper or lower case M or F are acceptable. The
860 REM returned string is truncated to 1 character.
870 :
880 DEF FN_sex
890 LOCAL sex$,FLAG
900 REPEAT
910   FLAG=TRUE
920   INPUT "Male or Female - M or F ? " sex$
930   IF sex$<>" " THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
940   IF sex$<>"M" AND sex$<>"F" THEN FLAG=FALSE
950   IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
960 UNTIL FLAG
970 =sex$

```

14. Example 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 BBCBASIC(Z80) can be used to manipulate any CP/M file by processing it on a character by character basis.

```

10 REM F_RSTD
20 :
30 REM EXAMPLE OF READING A COMPATIBLE DATA FILE
40 :
50 REM This program opens a data file and reads numeric and
60 REM character data from it. The data is read a byte at a
70 REM time and the appropriate action taken depending on
80 REM whether it is a character, a comma, or a control char.
90 :
100 :
110 compat=OPENIN("COMPAT")
120 PRINT "File Name COMPAT Opened as Channel ";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 centimeters ";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 commas and CRs
280 REM as data item terminators and Control Z as the file
290 REM terminator. Since we are not interested in reading a
300 REM record at a time, the record terminator CR LF is of no
302 REM special interest to us. We use the CR, along with
304 REM commas, as a data item separator and discard the LF.
310 :
320 DEF FN_read
330 LOCAL data$,byte
340 dat$=""
350 REPEAT
360   byte=BGET#compat
370   IF byte=&1A THEN CLOSE#compat: END
380   IF NOT(byte=&0A OR byte=&0D OR byte=&2C)
390     THEN data$=data$+CHR$(byte)
390 UNTIL byte=&0D OR byte=&2C
400 =data$

```


RANDOM (RELATIVE) FILES

1. There are 3 example random file programs. The first is so simple it is almost trivial, 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.

2. Example 1 - Simple Random Access File

```

10 REM F_RANDOM
20 :
30 REM VERY SIMPLE RANDOM ACCESS PROGRAM
40 :
50 REM This program maintains a random access file of names
60 REM and remarks. Each name can be up to a maximum of 30
70 REM characters long and each remark up to 50 characters.
80 REM The first byte of the record is set non zero (in fact
90 REM &FF) if there is a record present. This gives a
100 REM maximum record length of 1+31+51=83. (Don't forget
110 REM the CRs.)
120 :
130 :
140 @%=10
150 temp$=STRING$(50," ")
160 CLS
170 WIDTH 0
180 fnum=OPENIN "RANDZERO"
190 IF fnum=0 fnum=OPENOUT "RANDZERO"
200 IF fnum=0 PRINT "Directory Full":END
210 :

```



```

220 REPEAT
230   INPUT "Enter record number: "ans$
240   IF ans$="0" CLOSE#fnum:CLS:END
250   IF ans$="" record=record+1 ELSE record=VAL(ans$)
260   PTR#fnum=FN_ptr(record)
270   PROC_display
280   INPUT "Do you wish to change this record",ans$
290   PTR#fnum=FN_ptr(record)
300   IF FN_test(ans$) PROC_modify
310 UNTIL FALSE
320 END
330 :
340 :
350 DEF FN_test(A$) =LEFT$(A$,1)="Y" OR LEFT$(A$,1)="y"
360 :
370 DEF FN_ptr(record)=record*83
380 :
390 DEF PROC_display
400 PRINT "Record number ";record
410 flag=BGET#fnum
420 IF EOF#fnum OR flag=0 PRINT "Record not stored":ENDPROC
430 INPUT#fnum,name$,remark$
440 PRINT name$;" ";remark$
450 ENDPROC
460 :
470 DEF PROC_modify
480 PRINT "(Enter <RETURN> for no change)"
490 INPUT "Name ",temp$
500 temp$=LEFT$(temp$,30)
510 IF temp$<>"" name$=temp$
520 INPUT "Remark ",temp$
530 temp$=LEFT$(temp$,50)
540 IF temp$<>"" remark$=temp$
550 INPUT "Confirm update record",ans$
560 IF NOT FN_test(ans$) ENDPROC
570 BPUT#fnum,255
580 PRINT#fnum,name$,remark$
590 ENDPROC

```

3. Example 2 - Simple Random Access Database. The second program expands the previous program in to 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, name and size of the records you wish to use. If the file already exists this data is read from the records at the beginning of the file.

```

10 REM F_RAND1
15 :
20 REM SIMPLE DATABASE PROGRAM
25 :
30 REM Written by R T Russell. Jan 1983
40 :
50 REM This is a simple database program. You are asked for
60 REM the name of the file you wish to use. If the file
70 REM does not already exist, you are asked to specify the
80 REM number and format of the records. If the file does
90 REM already exist, the file specification is read from
100 REM the file. Considerable use is made of FOR NEXT loops
102 REM to input and print the data.
110 :
120 :
130 @%=10
140 CLS
150 WIDTH 0
160 INPUT "Enter the filename of the data file: "filename$
170 fnum=OPENIN(filename$)
180 IF fnum=0 PROC_setup ELSE PROC_readgen
190 base=PTR#fnum:PRINT
200 :
210 REPEAT
220   INPUT "Enter record number: "ans$
230   IF ans$="0" CLOSE#fnum:CLS:END
240   IF ans$="" record=record+1 ELSE record=VAL(ans$)
250   PTR#fnum=FN_ptr(record)
260   PROC_display
270   INPUT "Do you wish to change this record",ans$
280   PTR#fnum=FN_ptr(record)
290   IF FN_test(ans$) PROC_modify
300 UNTIL FALSE
310 END
320 :
330 DEF FN_test(A$) =LEFT$(A$,1)="Y" OR LEFT$(A$,1)="y"
340 :
350 DEF FN_ptr(record)=base+record*length
360 :

```



```

370 DEF PROC_setup
380 PRINT "New file."
390 fnum=OPENOUT(filename$)
400 IF fnum=0 PRINT "Sorry, disk directory full.":END
410 INPUT "Enter number of fields per record: "fields
420 DIM title$(fields),size(fields),A$(fields)
430 FOR field=1 TO fields
440   PRINT "Enter title of field number ";field;": ";
450   INPUT "title$(field)
460   PRINT
470   REPEAT
480     INPUT "Maximum size of field (characters)",size(field)
490     UNTIL size(field)>0 AND size(field)<256
500 NEXT field
510 length=1
520 PRINT#fnum,fields
530 FOR field=1 TO fields
540   PRINT#fnum,title$(field),size(field)
550   length=length+size(field)+1
560 NEXT field
570 ENDPROC
580 :
590 DEF PROC_readgen
600 length=1
610 INPUT#fnum,fields
620 DIM title$(fields),size(fields),A$(fields)
630 FOR field=1 TO fields
640   INPUT#fnum,title$(field),size(field)
650   length=length+size(field)+1
660 NEXT field
670 ENDPROC
680 :
690 DEF PROC_display
700 PRINT "Record number ";record
710 flag=BGET#fnum
720 IF EOF#fnum OR flag=0 PRINT "Record not stored":ENDPROC
730 FOR field=1 TO fields
740   INPUT#fnum,A$(field)
750   PRINT title$(field);" ";A$(field)
760 NEXT field
770 ENDPROC
780 :
790 DEF PROC_modify
800 PRINT "(Enter <RETURN> for no change)"
810 FOR field=1 TO fields
820   REPEAT
830     PRINT title$(field);" ";
840     INPUT LINE "A$
850     IF A$="" PRINT CHR$(11)title$(field);" ";A$(field)

```

Note: CHR\$(11) is cursor up for the computer on which this program was written. It would have been better to use a variable called, say, curUP\$ at this point. It should, of course, have been previously defined in the initialisation procedure, along with such other useful strings as bell\$, clearline\$ etc.

```

860 UNTIL LEN(A$)<=size(field)
870 IF A$<>"" A$(field)=A$
880 NEXT field
890 INPUT "Confirm update record",ans$
900 IF NOT FN_test(ans$) ENDPROC
910 BPUT#fnum,255
920 FOR field=1 TO fields
930   PRINT#fnum,A$(field)
940 NEXT field
950 ENDPROC

```


4. Random File Initialisation. The previous 2 examples did not initialise the file before starting to use it. This is perfectly acceptable with CP/M random files, but it can cause problems. If you have got, say, up to 2000 parts, you will eventually want to hold data on all of them. If you don't initialise the file at the start, you will not have reserved room on the disk for the full range of the file. In your enthusiasm, you may write other files to the disk. Then, all of a sudden, your inventory program will crash because there is insufficient room for the record you have just tried to write to disk. So, if you know how much room you want, initialise the file by writing a byte to each record (in this case the valid data/deleted data/no record indicator).

5. Example 3 - Random Access Inventory Program. The final example in this part is a full-blown inventory program. Rather than go through all its aspects here I have interspersed the listing with comments.

```

10 REM F_RAND2
20 :
30 REM Written by Doug Mounter - Jan 82
40 MODE 0: REM Selects a 80*32 display
50 REM EXAMPLE OF A RANDOM ACCESS FILE
60 :
70 REM This is a simple inventory program. It uses the
80 REM item's part number as the key and stores:
90 REM The item description - character max length 30
100 REM The quantity in stock - numeric
110 REM The re-order level - numeric
120 REM The unit price - numeric
130 REM In addition, the first byte of the record is used
140 REM as a valid data flag. Set to 0 if empty, D if the
150 REM record has been deleted or V if the record is
160 REM valid.
170 REM This gives a MAX record length of 47 bytes
180 REM (Don't forget the CR after the string)
190 :
200 :
210 PROC_initialise
220 inventory=FN_open("B:INVENTORY")

```

This is the command loop. You are offered a choice of functions until you eventually select function 0. Unfortunately, BBCBASIC has no case statement, and I have been forced to use ON GOSUB. Still it's not too bad. I have also used some forward jumps within procedures, etc to overcome the lack of a multi line IF statement. I could have used further procedures, but it 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);"Display the Inventory for One Part";
    TAB(55);"Type 2"
290 PRINT TAB(10);"Change the Stock Level of One Part";
    TAB(55);"Type 3"
300 PRINT TAB(10);"Display All Items Below Reorder Level";
    TAB(55);"Type 4"
310 PRINT TAB(10);"Recover a Previously 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$;
360 PRINT "Please enter the appropriate number
    (0 to 6 or 9) ";
370 function$=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#inventory
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 record ? ";
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 stock 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$;
1030  PRINT "Do you want to update any more stock levels ? ";
1040  UNTIL GET$="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 NUMBERS
1760 :
1770 RETURN
1780 :
1790 RETURN
1800 :
1810 :
```

1820 REM REINITIALISE THE INVENTORY DATA FILE

```

1830 :
1840 CLS
1850 PRINT TAB(0,3);bell$;"Are you sure you want to set up a
                                new inventory?"
1860 PRINT "You will DESTROY ALL THE DATA YOU HAVE ACCUMULATED
                                so far."
1870 PRINT "It would be safer to use a new disk in drive B and
                                start a new"
1880 PRINT "inventory file."
1890 PRINT "If you are SURE you want to do it, enter YES"
1900 PRINT "If you want to start a new inventory file, enter
                                NEW"
1910 INPUT "Otherwise, just hit return ",temp$
1920 IF temp$="YES" PROC_setup(inventory)
1930 IF temp$="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, when you want to use a different computer, you will be able to change them without searching through the whole \$\$\$\$\$\$ thing.

```

1980 :
1990 DEF PROC_initialise
2000 bell$=CHR$(7)
2010 return$=CHR$(13)
2020 rec_length=47
2030 partno=0
```

If you initially set strings to the maximum length you will ever use, you will save on memory usage.

```

2040 desc$=STRING$(30," ")
2050 temp$=STRING$(40," ")
2060 WIDTH 0
2070 ENDPROC
2080 :
2090 :
```



```

2100 REM OPEN THE FILE AND RETURN THE FILE NUMBER
2110 :
2120 REM If the file already exists, the largest permitted
2130 REM part number is read into maxpartno.
2140 REM If it is a new file, file initialisation is carried
2150 REM out and the largest permitted part number is
2160 REM written as the first record.
2170 :
2180 DEF FN_open(name$)
2190 fnum=OPENIN(name$)
2200 IF fnum>0 INPUT#fnum,maxpartno: =fnum
2210 fnum=OPENOUT(name$)
2220 IF fnum=0 PRINT "Disk Directory Full."!!:END
2230 CLS

```

It's a new file, so we won't go through the warning bit.

```

2240 PROC_setup(fnum)
2250 =fnum
2260 :
2270 REM SET UP THE FILE
2280 :
2290 REM Ask for the maximum part number required, write
2300 REM it as the first record and then write 0 in to
2310 REM the first byte of every record.
2320 :
2330 DEF PROC_setup(fnum)
2340 REPEAT
2350   PRINT TAB(0,12);bell$;;PROC_cteos
2360   INPUT "What is the highest part number required (Max
                                     5000) ",maxpartno
2370 UNTIL maxpartno>0 AND maxpartno<5000 AND INT(maxpartno)
                                     =maxpartno
2380 PTR#fnum=0
2390 PRINT#fnum,maxpartno
2400 FOR partno=1 TO maxpartno
2410   PTR#fnum=FN_ptr(partno)
2420   BPUT#fnum,0
2430 NEXT
2440 partno=0
2450 ENDPROC
2460 :
2470 :

```

2480 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.

```

2490 :
2500 DEF FN_getpartno
2510 REPEAT
2520   PRINT TAB(0,5);bell$;;PROC_cteos
2530   PRINT "Enter a Part Number Between 1 and ";maxpartno
2540   IF partno=maxpartno THEN 2610
2550   PRINT "The Next Part Number is ";partno+1;
2560   PRINT " Just hit RETURN to get this"
2570 :
2580   INPUT "What is the Part Number You Want ",partno$
2590   IF partno$<>" " partno=VAL(partno$):GOTO 2650
2600   IF partno=maxpartno partno=0 ELSE partno=partno+1
2610 :
2620   PRINT TAB(35,9);partno;;PROC_cteol
2630 UNTIL partno>0 AND partno<maxpartno+1 AND INT(partno)
                                     =partno
2640 =partno
2650 :
2660 :
2670 REM GET THE RECORD FOR THE PART NUMBER
2680 :
2690 REM Return TRUE if the record exists and FALSE if not
2700 REM If the record does not exist, load desc$ with "No
2710 REM Record". The remainder of the record is set to 0
2720 :
2730 DEF FN_getrec(partno,type)
2740 stock=0
2750 reord=0
2760 price=0
2770 PTR#inventory=FN_ptr(partno)
2780 test=BGET#inventory
2790 IF test=0 desc$="No Record": =FALSE
2800 IF test=type THEN 2870
2810 IF type=86 desc$="Record Deleted" ELSE desc$="Record
                                     Exists"
2820 =FALSE
2830 :
2840 INPUT#inventory,desc$
2850 INPUT#inventory,stock,reord,price
2860 =TRUE
2870 :
2880 :

```



```
2890 REM CALCULATE THE VALUE OF PTR FOR THIS RECORD
```

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.

```
2900 :
2910 DEF FN_ptr(partno)=partno*rec_length+5-rec_length
2920 :
2930 :
2940 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.

```
2950 :
2960 DEF FN_amend(partno)
2970 PRINT return$;:PROC_cteol:PRINT TAB(0,4);
2980 PRINT "Please Complete the Details for Part Number ";partno
2990 PRINT "Just hit Return to leave the entry as it is"
3000 flag=FALSE
3010 type=ASC"V"
3020 INPUT "Description - Max 30 Chars " temp$
3030 IF temp$="DELETE" type=ASC"D": =TRUE
3040 temp$=LEFT$(temp$,30)
3050 IF temp$<>" desc$=temp$:flag=TRUE
3060 IF desc$="No Record" OR desc$="Record Deleted" =FALSE
3070 INPUT "Current Stock Level " temp$
3080 IF temp$<>" stock=VAL(temp$):flag=TRUE
3090 INPUT "Reorder Level " temp$
3100 IF temp$<>" reord=VAL(temp$):flag=TRUE
3110 INPUT "Unit Price " temp$
3120 IF temp$<>" price=VAL(temp$):flag=TRUE
3130 =flag
3140 :
3150 :
```

```
3160 REM WRITE THE RECORD
```

Write the record to the file if necessary (flag=TRUE)

```
3170 :
3180 DEF PROC_write(partno,flag,type)
3190 IF NOT flag ENDPROC
3200 PTR#inventory=FN_ptr(partno)
3210 BPUT#inventory,type :REM write the indicator to byte 1
3220 PRINT#inventory,desc$,stock,reord,price
3230 ENDPROC
3240 :
3250 :
3260 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.

```
3270 :
3280 DEF PROC_display(flag)
3290 PRINT TAB(0,5);:PROC_cteos
3300 PRINT "Part Number ";partno
3310 PRINT "Description ";desc$
3320 IF NOT flag ENDPROC
3330 PRINT "Current Stock Level ";stock
3340 PRINT "Reorder Level ";reord
3350 PRINT "Unit Price ";price
3360 ENDPROC
3370 :
3380 :
```


The 2 following procedures are designed for a 80 by 32 display. If your computer uses a different size display you will need to amend lines 3480,3560 and 3570.

```

3390 REM Many computers do not have clear to end of line/screen
3400 REM vdu procedures. The following 2 procedures clear to
3410 REM the end of the line/screen. Care has been taken NOT
3420 REM to write to the last position on the screen (x=79, y=31)
3430 REM since this would cause the screen to scroll.
3440 :
3450 DEF PROC_cteol
3460 LOCAL x,y
3470 x=POS:y=VPOS
3480 IF y=31 PRINT STRING$(79-x," "); ELSE PRINT STRING$(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 STRING$(80," ");:NEXT
3570 PRINT STRING$(79-x," ");TAB(x,y);
3580 ENDPROC

```

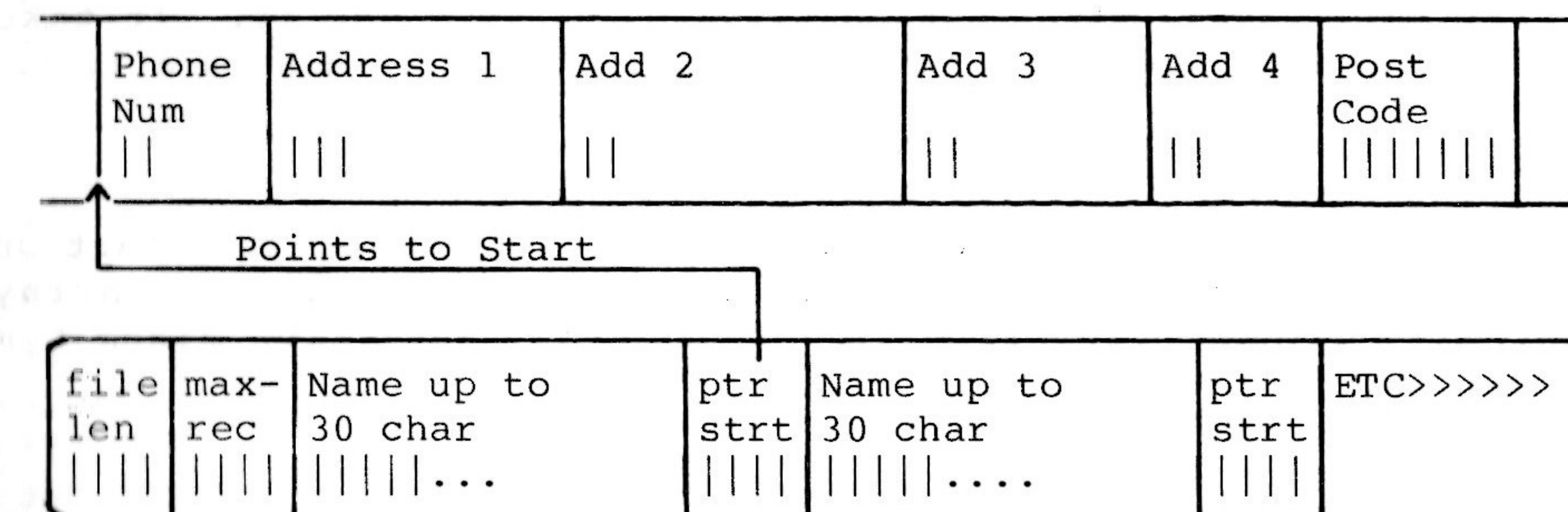
INDEXED DATA FILES

1. Deficiencies of Random 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 2 files, one as an index to the other. In order for this to work efficiently, you must have complete control over the file pointer. I don't know of any other version of BASIC which allows this control, but it is quite simple with BBCBASIC.

2. 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.

3. File Organization. The files are organized as shown below:

ADDRESS.DTA - All of the fields can be up to 30 characters long.



NAME.NDX

File len is the position of the pointer when the last byte was written to the file the last time it was used. It will be updated just before the file is closed this time. There is a similar record at the beginning of ADDRESS.DTA.

Max rec 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. If you want to write a disk access and sort program for the index - the best of luck. And please can I have a copy?

Ptr strt 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.

4. Program Organization. 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 a list.

5. The Index. The index is read into memory at the start and written back at the end. In memory, it consists of 2 arrays called index\$() and index(). Oh that we could have mixed type arrays!

6. Example - the LAST

```

10 REM F_INDEX
20 REM EXAMPLE OF AN INDEXED FILE
30 :
40 REM This is a simple address book filing system. It will
50 REM accept names, telephone numbers and addresses and store
60 REM them in a file called ADDRESS.DTA. The index is in
70 REM name order and is kept in a file called ADDRESS.NDX.
80 REM All the fields are character and the maximum length
90 REM of any field is 30.
100 MODE 3: REM Select a 80*25 display 100:
110 :
120 PROC_initialise
130 PROC_open_files
140 ON ERROR IF ERR<>17 PRINT:REPORT:PRINT" At line ";ERL:END
150 REPEAT
160 CLS
170 PRINT TAB(5,3);"If you want to:-" 0,3
180 PRINT TAB(10);"End This Session";TAB(55);"Type 0" 5 38
190 PRINT TAB(10);"Enter Data";TAB(55);"Type 1" 5 38
200 PRINT TAB(10);"Search For/Delete an Entry";TAB(55);"Type 2" 5 38
210 PRINT TAB(10);"List in Alphabetical Order";TAB(55);"Type 3" 5 38
220 PRINT TAB(10);"Reorganise the Data File and Index";TAB(55);"Type 4"; 5 38
230 REPEAT
240 PRINT TAB(5,11); 0
250 PRINT "Please enter the appropriate number (0 to 4) ";
260 function$=GET$
270 PRINT return$;:PROC_cteol PRINT cirlines$;
280 UNTIL function$>"/" AND function$<"5"
290 function=VAL(function$)
300 PRINT TAB(54,function+5);"<====<"; 35 "4"
310 ON function GOSUB 420,700,820,1030 ELSE 410,670,810,1020
320 UNTIL function=0
330 CLS
340 PROC_close_files
350 CLS
360 PRINT "Address Book Files Closed"
370 END
380 :
390 :

```



```

390 400 REM ENTER DATA
400 410 :
410 420 flag=TRUE
420 430 temp$=""
430 440 i=1
440 450 REPEAT
450 460 REPEAT
460 470 IF temp$="N" PROC message("Data NOT Accepted")
470 480 PROC_get_data
480 490 IF length=maxrec OR data$(1)="" flag=FALSE:GOTO 560 540
490 500 IF data$(1)="+" OR data$(1)="-"
500 510 PROC_message("Bad Data"):GOTO 560 540
510 520 i=FN_find_place(0,data$(1))
520 530 IF i>0 PROC_message("Duplicate Record")
530 540 PRINT "Is this data correct ? ";
540 550 temp$=FN_yesno GET$
550 560 UNTIL NOT flag OR temp$<>"N" = "y"
560 570 PROC_cteos
570 580 IF NOT flag THEN 640 610
580 590 PROC_put_index(i,data$(1),PTR#datanum)
590 600 FOR i=2 TO 7
600 610 PRINT#datanum,data$(i)
610 620 NEXT
620 630 :
630 640 UNTIL NOT flag
640 650 RETURN
650 660 :
660 670 :
670 680 REM SEARCH FOR AN ENTRY
680 690 :
690 700 i=0 1
700 710 REPEAT
710 720 PRINT TAB(0,11);:PROC_cteol cteol$;
720 730 INPUT "What name do you want to look for ",name$ PRINT
730 740 IF name$="" THEN 760 750
740 750 IF name$<>" "IF name$="DELETE" PROC_delete(i) ELSE
750 760 UNTIL name$=""
760 770 RETURN
770 780 :
780 790 :
790 800 PRINT TAB(0,13);cteol$;
800 810 INPUT NAME$

```

```

190 800 REM LIST IN ALPHABETICAL ORDER
800 810 :
810 820 entry=1
820 830 PRINT curoff$;
830 840 REPEAT
840 850 CLS
850 860 line_count=0
860 870 REPEAT
870 880 PRINT TAB(0,line_count);
880 890 PROC_read_data(entry)
890 900 PROC_print_data
900 910 entry=entry+1
910 920 line_count=line_count+8
920 930 temp$=INKEY$(0)
930 940 UNTIL entry>length OR line_count>16 OR temp$<>" "
940 950 PROC_message("Push any key to continue or E to end ")
950 960 UNTIL entry>length OR GET$="E"
960 970 PRINT curon$;
970 980 RETURN
980 990 :
990 1000 :
1000 1010 REM REORGANISE THE DATA FILE AND INDEX
1010 1020 :
1020 1030 entry=1
1030 1040 PRINT TAB(0,13);"Reorganising the Data File and Index"
1040 1050 newdata=OPENOUT"ADDRESS.BAK"
1050 1060 newindex=OPENOUT"NAME.BAK"
1060 1070 IF newdata=0 OR newindex=0 PRINT "Disk Directory Full":END
1070 1080 PRINT#newdata,0
1080 1090 REPEAT
1090 1100 PROC_read_data(entry)
1100 1110 index(entry)=PTR#newdata
1110 1120 FOR i=2 TO 7
1120 1130 PRINT#newdata,data$(i)
1130 1140 NEXT
1140 1150 entry=entry+1
1150 1160 UNTIL entry>length
1160 1170 EOFnewdata=PTR#newdata-1
1170 1180 PTR#newdata=0
1180 1190 PRINT#newdata,EOFnewdata
1190 1200 CLOSE#0

```


The time taken to rename a file is considerable.

```

12001210 PRINT "Re-naming the Data and Index Files"
12101220 *REN ADDRESS.***=ADDRESS.BAK
12201230 *REN NAME.***=NAME.BAK
12301240 *REN ADDRESS.BAK=ADDRESS.DTA
12401250 *REN ADDRESS.DTA=ADDRESS.***
12501260 *REN NAME.BAK=NAME.NDX
12601270 *REN NAME.NDX=NAME.***
12701280 indexnum=OPENUP "NAME.NDX"
12801290 datanum=OPENUP "ADDRESS.DTA"
12901300 INPUT#datanum,EOFdata
13001310 PTR#datanum=EOFdata+1
13101320 RETURN
13201330 :
13301340 :
13401350 REM INITIALISE VARIABLES AND ARRAYS
13501360 :
13601370 :
13701380 :
13801390 DEF PROC_initialise
1400 esc$=CHR$(27)
1410 bell$=CHR$(7)
1420 return$=CHR$(13)

```

1390 MODE 0
etcold\$ = chr\$(13)

The following cursor on/off strings are for the Torch computer.
(See page 77 of the BBC Micro User Guide.)

```

1430 curoff$=CHR$(23)+CHR$(1)+STRING$(8,CHR$(0))
1440 curon$=CHR$(23)+CHR$(1)+CHR$(1)+STRING$(7,CHR$(0))
1450 :
1460 REM The maximum record number, maxrec, is set/read in
1470 REM PROC_setup/PROC_read_index
1480 :
1490 DIM message$(7)
1500 FOR i=1 TO 7
1510 READ message$(i)
1520 NEXT
1530 DATA Name,Phone Number,Address,-- " --,-- " --,-- " --,
1540 :
1550 DIM data$(7)
1560 FOR i=1 TO 7
1570 data$(i)=STRING$(30," ")
1580 NEXT
1590 temp$=STRING$(255," ")
1600 temp$=""
1610 ENDPROC

```

1460 curoff\$ = chr\$(23)
1470 curon\$ = chr\$(23)

```

1620 :
1630 :
1640 REM OPEN THE FILES
1650 :
1660 DEF PROC_open_files
1670 indexnum=OPENUP"NAME.NDX"
1680 datanum=OPENUP"ADDRESS.DTA"
1690 IF indexnum=0 OR datanum=0 PROC_setup ELSE PROC_read_index
1700 ENDPROC
1710 :
1720 :
1730 REM SET UP NEW INDEX AND DATA FILES
1740 :
1750 DEF PROC_setup
1760 CLS
1770 PRINT TAB(0,13);"Setting Up Address Book"
1780 maxrec=200
1790 indexnum=OPENOUT"NAME.NDX"
1800 datanum=OPENOUT"ADDRESS.DTA"
1810 IF indexnum=0 OR datanum=0 PRINT "Disk Directory Full":END
1820 PRINT#indexnum,0,maxrec
1830 PRINT#datanum,0
1840 DIM index$(maxrec+1),index(maxrec+1)
1850 length=0
1860 index$(0)=""
1870 index(0)=0
1880 index$(1)=CHR$(&FF)
1890 index(1)=0
1900 ENDPROC
1910 :
1920 :
1930 REM READ INDEX AND LENGTH OF DATA FILE
1940 :
1950 DEF PROC_read_index
1960 CLS
1970 INPUT#indexnum,EOFindex,maxrec
1980 DIM index$(maxrec+1),index(maxrec+1)
1990 index$(0)=""
2000 index(0)=0
2010 length=0
2020 REPEAT
2030 length=length+1
2040 INPUT#indexnum,index$(length),index(length)
2050 UNTIL PTR#indexnum>EOFindex
2060 index$(length+1)=CHR$(&FF)
2070 index(length+1)=0
2080 INPUT#datanum,EOFdata
2090 PTR#datanum=EOFdata+1
2100 ENDPROC
2110 :
2120 :

```



```

2160 2130 REM WRITE INDEX, MAXREC AND NEW EOF POINTERS AND CLOSE FILES
2170 2140 :
2180 2150 DEF PROC_close_files
2190 2160 EOFdata=PTR#datanum-1
2200 2170 PTR#datanum=0
2210 2180 PRINT#datanum,EOFdata
2220 2190 PTR#indexnum=10
2230 2200 FOR i=1 TO length
2240 2210   PRINT#indexnum,index$(i),index(i)
2250 2220 NEXT
2260 2230 EOFindex=PTR#indexnum-1
2270 2240 PTR#indexnum=0
2280 2250 PRINT#indexnum,EOFindex,maxrec
2290 2260 CLOSE#0
2300 2270 ENDPROC
2310 2280 :
2320 2290 :
2330 2300 REM WRITE A MESSAGE AT LINE 23
2340 2310 :
2350 2320 DEF PROC_message(line$)
2360 2330 LOCAL x,y
2370 2340 x=POS
2380 2350 y=VPOS
2390 2360 PRINT TAB(0,23);:PROC_cteol:PRINT bell$;line$;
2400 2370 PRINT TAB(x,y);
2410 2380 ENDPROC
2420 2390 :
2430 2400 :
2440 2410 REM GET A Y/N ANSWER input DATA LIMIT TO 25 CHAR
2450 2420 :
2460 2430 DEF FN_yesno max_get_data
2470 2440 LOCAL temp$
2480 2450 temp$=GET$
2490 2460 IF temp$="y" OR temp$="Y" ="Y"
2500 2470 IF temp$="n" OR temp$="N" ="N"
2510 2480 =""
2520 2490 :
2530 2500 :
2540 2510 REM CLEAR 9 LINES FROM PRESENT POSITION
2550 2520 :
2560 2530 DEF PROC_clear9
2570 2540 LOCAL x,y,i
2580 2550 x=POS
2590 2560 y=VPOS
2600 2570 PRINT return$;
2610 2580 FOR i=1 TO 9
2620 2590   PRINT STRING$(80," ");
2630 2600 NEXT
2640 2610 PRINT TAB(x,y);
2650 2620 ENDPROC

```

Deleted

```

2630 :
2640 :
2650 2650 REM GET INPUT DATA - LIMIT TO 30 CHAR 25
2660 2660 :
2670 2670 DEF PROC_get_data
2680 2680 LOCAL i
2690 2690 PRINT TAB(0,13);cteos$
2700 2700 PROC_clear9
2710 2710 IF length=maxrec PROC_message("Address Book Full")
2720 2720 FOR i=1 TO 7
2730 2730   PRINT TAB(10);message$(i);TAB(25); 5 14
2740 2740   INPUT temp$
2750 2750   data$(i)=LEFT$(temp$,30) 25
2760 2760   IF data$(1)="" i=7
2770 2770 NEXT
2780 2780 ENDPROC
2790 2790 :
2800 2800 :
2810 2810 REM FIND AND DISPLAY THE REQUESTED DATA
2820 2820 :
2830 2830 DEF FN_display(i,name$)
2840 2840 PRINT TAB(0,12);:PROC_cteos cteos$
2850 2850 i=FN_find_place(i,name$)
2860 2860 IF i<0 PROC_message("Name Not Known - Next Highest Given")
2870 2870 PROC_read_data(i)
2880 2880 PRINT
2890 2890 PROC_print_data
2900 2900 =i
2910 2910 :
2920 2920 :
2930 2930 REM DELETE THE ENTRY FROM THE INDEX

```

Move everything below the entry you want deleted up one and subtract 1 from the length

```

2940 2940 :
2950 2950 DEF PROC_delete(i)
2960 2960 INPUT "Are you SURE ",temp$ input TAB(17,13) "Are you sure "temp$
2970 2970 PRINT TAB(0,VPOS-1);:PROC_cteos cteos$
2980 2980 IF temp$<>"YES" ENDPROC
2990 2990 IF i<0 i=-i
3000 3000 FOR i=i TO length
3010 3010   index$(i)=index$(i+1)
3020 3020   index(i)=index(i+1)
3030 3030 NEXT
3040 3040 length=length-1
3050 3050 ENDPROC
3060 3060 :
3070 3070 :

```


2800 3080 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.

```

2870 3090 :
2880 3100 DEF PROC_read_data(i)
2890 3110 PTRdata=PTR#datanum
2900 3120 IF i<0 i=-i
2910 3130 PTR#datanum=index(i)
2920 3140 data$(1)=index$(i)
2930 3150 FOR i=2 TO 7
2940 3160   INPUT#datanum,data$(i)
2950 3170 NEXT
2960 3180 PTR#datanum=PTRdata
2970 3190 ENDPROC
2980 3200 :
2990 3210 :
3000 3220 REM PRINT data$() ON VDU
3010 3230 :
3020 3240 DEF PROC_print_data
3030 3250 LOCAL i
3040 3260 FOR i=1 TO 7
3050 3270   IF data$(i)<>" PRINT TAB(10)5;message$(i);TAB(25)15;data$(i)
3060 3280   IF data$(1)=CHR$(&FF) i=7
3070 3290 NEXT
3080 3300 ENDPROC
3090 3310 :
3100 3320 :

```

3170 3330 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 3340 :
3190 3350 DEF PROC_put_index(i,entry$,ptr)
3200 3360 LOCAL j
3210 3370 IF i<0 i=-i
3220 3380 FOR j=length+1 TO i STEP -1
3230 3390   index$(j+1)=index$(j)
3240 3400   index(j+1)=index(j)
3250 3410 NEXT
3260 3420 index$(i)=entry$
3270 3430 index(i)=ptr
3280 3440 length=length+1
3290 3450 ENDPROC
3300 3460 :
3310 3470 :

```


3460 3480 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 2 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.

```
3270 3490 :
3280 3500 DEF FN_find_place(i,entry$)
3290 3510 LOCAL top,bottom
3300 3520 IF i<0 i=-i
3310 3530 IF entry$="+" AND i<length =i+1
3320 3540 IF entry$="+" AND i=length =1
3330 3550 IF entry$="-" AND i>1 =i-1
3340 3560 IF entry$="-" AND i<2 =length
```

Here, at last, THE BINARY CHOP

```
3350 3570 top=length+1
3360 3580 bottom=0
3370 3590 i=(top+1) DIV 2
3380 3600 IF entry$<>index$(i) i=FN_search(entry$)
3390 3610 REPEAT
3400 3620 IF entry$=index$(i-1) i=i-1
```

This bit moves the pointer up the index to the first of any duplicate entries.

```
3410 3630 UNTIL entry$<>index$(i-1)
3420 3640 IF entry$=index$(i) =i ELSE --i
3430 3650 :
3440 3660 :
3450 3670 REM DO THE SEARCHING FOR FN_find_place
3460 3680 :
3470 3690 DEF FN_search(entry$)
3480 3700 REPEAT
3490 3710 IF entry$>index$(i) bottom=i ELSE top=i
3500 3720 i=(top+bottom+1) DIV 2: REM round
3510 3730 UNTIL entry$=index$(i) OR top=bottom+1
3520 3740 =i
3530 3750
3540 3760
```


The 2 following procedures are designed for a 80 by 32 display. If your computer uses a different size display you will need to amend lines 3860,3940 and 3950.

```

3770 REM Some computers do not have clear to end of line/screen
3780 REM vdu procedures. The following 2 procedures clear to
3790 REM the end of the line/screen. Care has been taken NOT
3800 REM to write to the last position on the screen (x=79,y=24)
3810 REM since this would cause the screen to scroll.
3820 ;
3830 DEF PROC_cteol
3840 LOCAL x,y
3850 x=POS:y=VPOS
3860 IF y=24 PRINT STRING$(79-x," "); ELSE PRINT STRING$(80-x," ");
3870 PRINT TAB(x,y);
3880 ENDPROC
3890 ;
3900 ;
3910 DEF PROC_cteos
3920 LOCAL I,x,y
3930 x=POS:y=VPOS
3940 IF y<24 FOR I=y TO 23:PRINT STRING$(80," ");:NEXT
3950 PRINT STRING$(79-x," ");TAB(x,y);
3960 ENDPROC

```

*NOT
NEEDED*

Well, that's it. Apart from the following notes on the binary chop you have read it all.

THE BINARY CHOP

EXPLANATION

1. The quickest way to find an entry in an ORDERED list is not to search through the 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 bottom equal the pointer, if not make top equal to it. Then you repeat the process.

2. 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 (top+bottom)/2.
(4)	14	Is that entry 14? No it's not, it's
pointer>(5)	19	more, so set top to the current
(6)	23	value of pointer and repeat the
(7)	34	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 not, its
pointer>(3)	8	less, so set bottom to the current
(4)	14	value of 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
bottom> (3)	8	with the pointer set to the position
pointer> (4)	14	in the list of the number you are
top > (5)	19	looking for.
(6)	23	
(7)	34	
(8)	45	
(9)	61	

3. 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 2 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.

4. 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 2 'end markers'.

5. Have a look at the flow chart on the following page 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 8 goes to find the number. The AVERAGE number of comparisons required for a linear search would be 50.

BINARY CHOP FLOWCHART

ENTRY='THING' BEING
SEARCHED FOR

LENGTH=NO OF 'PROPER'
ENTRIES IN
THE ARRAY.
THE FIRST
'PROPER'
ENTRY IS IN
ARRAY(1).

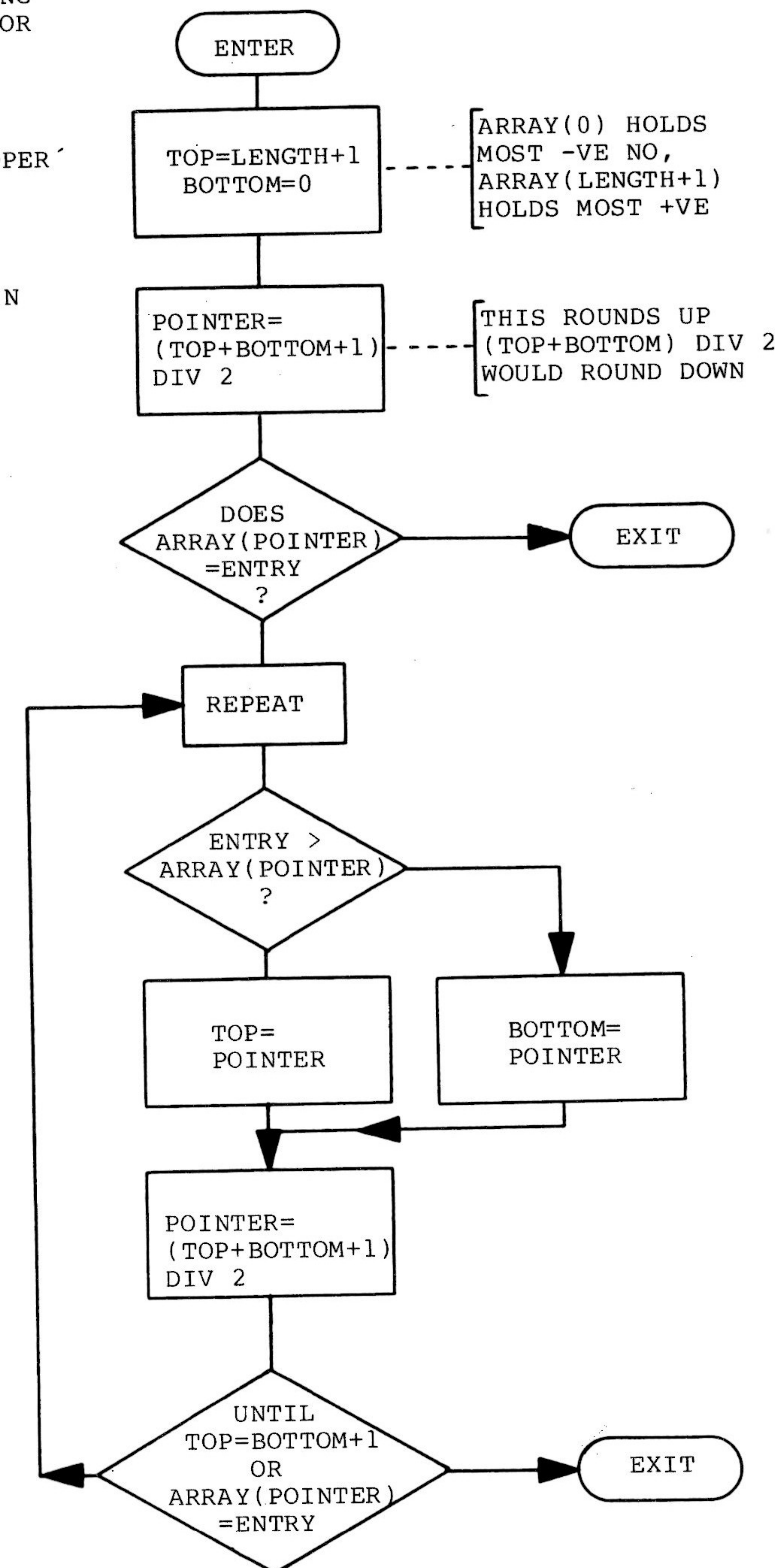


TABLE OF ASCII CODES

Binary	Hex	Dec	Character
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 Line Escape
00010001	11	17	^Q DC1 X On <i>cursor on</i>
00010010	12	18	^R DC2 Aux On
00010011	13	19	^S DC3 X Off
00010100	14	20	^T DC4 Aux Off <i>cursor off</i>
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 <i>WINSTEIN DELETE KEY</i>
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
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	/

ASCII Codes

Binary	Hex	Dec	Character
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	?
01000000	40	64	@
01000001	41	65	A
01000010	42	66	B
01000011	43	67	C
01000100	44	68	D
01000101	45	69	E
01000110	46	70	F
01000111	47	71	G
01001000	48	72	H
01001001	49	73	I
01001010	4A	74	J
01001011	4B	75	K
01001100	4C	76	L
01001101	4D	77	M
01001110	4E	78	N
01001111	4F	79	O
01010000	50	80	P
01010001	51	81	Q
01010010	52	82	R
01010011	53	83	S
01010100	54	84	T
01010101	55	85	U
01010110	56	86	V
01010111	57	87	W
01011000	58	88	X
01011001	59	89	Y
01011010	5A	90	Z

Binary	Hex	Dec	Character
01011011	5B	91	[
01011100	5C	92	\
01011101	5D	93]
01011110	5E	94	^
01011111	5F	95	_
01100000	60	96	`
01100001	61	97	a
01100010	62	98	b
01100011	63	99	c
01100100	64	100	d
01100101	65	101	e
01100110	66	102	f
01100111	67	103	g
01101000	68	104	h
01101001	69	105	i
01101010	6A	106	j
01101011	6B	107	k
01101100	6C	108	l
01101101	6D	109	m
01101110	6E	110	n
01101111	6F	111	o
01110000	70	112	p
01110001	71	113	q
01110010	72	114	r
01110011	73	115	s
01110100	74	116	t
01110101	75	117	u
01110110	76	118	v
01110111	77	119	w
01111000	78	120	x
01111001	79	121	y
01111010	7A	122	z
01111011	7B	123	{
01111100	7C	124	
01111101	7D	125	}
01111110	7E	126	~
01111111	7F	127	DEL Delete

ANNEX B TO BBCBASIC(Z80)

MATHEMATICAL FUNCTIONS

BBCBASIC has more intrinsic mathematical functions than many other versions of BASIC. Those that are not provided may be calculated as follows:

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	$ARCCOTH(X)=LN((X+1)/(X-1))/2$
$LOG_n(X)$	$LOG_n(X)=LN(X)*LN(n)$ $=LOG(X)*LOG(n)$

ERROR MESSAGES AND CODES

SUMMARY1. Untrappable - Error Code 0

Bad program	No room	Silly	Sorry
RENUMBER space		LINE space	

Strictly speaking 'Bad program' does not have an error code. It leaves ERR and ERL unchanged.

3. Trappable - Program.

No	Error	No	Error
1	Out of range	2	*
3	*	4	Mistake
5	Missing ,	6	Type mismatch
7	No FN	8	*
9	Missing "	10	Bad DIM
11	DIM space	12	Not LOCAL
13	No PROC	14	Array
15	Subscript	16	Syntax error
17	Escape	18	Division by zero
19	String too long	20	Too big
21	-ve root	22	Log range
23	Accuracy lost	24	Exp range
25	*	26	No such variable
27	Missing)	28	Bad HEX
29	No such FN/PROC	30	Bad call
31	Arguments	32	No FOR
33	Can't match FOR	34	FOR variable
35	*	36	No TO
37	*	38	No GOSUB
39	ON syntax	40	ON range
41	No such line	42	Out of DATA
43	No REPEAT	44	*
45	Missing #		

4. Trappable - Disk.

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
251	Bad key	253	Bad string
254	Bad command		

* Not applicable to BBCBASIC(Z80)

DETAILS

BBCBASIC(Z80)'s error messages and codes are briefly explained below in alphabetical order.

Accuracy lost 23

Before BBCBASIC(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, BBCBASIC(Z80) will issue an 'Accuracy lost' error message.

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.

Array 14

This error occurs when BBCBASIC(Z80) thinks it should be accessing an array, but does not know which one.

Bad call 30

This error indicates that a procedure or function has been incorrectly called.

Bad command 254

This error occurs when the command name is not recognized as a BBCBASIC(Z80) command. On the Torch, this error will also occur if a command preceded by '**' is not recognized by the BBC Micro. For example, trying to erase a non-existent file will give rise to this error.

Error Messages and Codes

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 HEX
28

Hexadecimal numbers can only include the numbers 0 to 9 and A to F. If you try to form a hex number with other characters this error will occur. For example:

&OF instead of &0F

Bad key
253

This error is generated if the key number given in a *KEY command is greater than 15.

Bad name
204

This error is generated if an ambiguous filename is encountered in a RENAME, SAVE or OPENOUT operation.

Bad program

From time to time BBCBASIC(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. This error will also occur if there was insufficient room for the string specified in a *KEY statement.

Error Messages and Codes

Can't match FOR
33

BBCBASIC(Z80) has been unable to find a FOR statement corresponding to the NEXT statement.

Channel
222

This error is generated by the disk 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 files specified cannot be closed because the disk has been changed whilst the files were open.

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

will generate this error.

Directory full
190

There is a limit to the number of directory entries. This limit depends on the implementation of CP/M, but is commonly 64 (8" single sided single density) 128 or 256. This error will be generated if a SAVE command attempts to exceed this limit.

Disk full
198

This error will occur if there is insufficient room on the disk for the data/program being written to it.

Error Messages and Codes

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 <Escape> key. You can trap this, and other errors, by using the ON ERROR GOTO statement. On the Torch computers, you can avoid this by reprogramming the <ESCAPE> key to return the ASCII value of escape (&1B) using *FX 229,1

Exp 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, BBCBASIC(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'.

Error Messages and Codes

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.

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
```

LINE space A program line is too long to be represented in BBCBASIC(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 BBCBASIC(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)
```


Error Messages and Codes

Missing "
9

This error message is generated if BBCBASIC(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 BBCBASIC(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 BBCBASIC(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
```

Mistake
4

This error will be generated if BBCBASIC(Z80) is unable to make any sense at all of the input line.

-ve root
21

This error message will occur if BBCBASIC(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 BBCBASIC(Z80) finds a RETURN statement without first encountering a GOSUB statement. (See the sub-section on Program Flow Control.)

Error Messages and Codes

No FN
7

If BBCBASIC(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.)

No FOR
32

This error message indicates that BBCBASIC(Z80) has found a NEXT statement without first encountering a FOR statement.

No PROC
13

If BBCBASIC(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.)

No REPEAT
43

This error message indicates that BBCBASIC(Z80) has found an UNTIL statement without first encountering a REPEAT statement.

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 BBCBASIC(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.

Error Messages and Codes

No such line
41 This error will occur if BBCBASIC(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$=""
```

No TO
36 This error message will be generated if BBCBASIC(Z80) encounters a FOR...NEXT loop with the 'TO' part missing.

Not LOCAL
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.)

Error Messages and Codes

ON range
40

This error will be generated if, in a simple ON GOTO/GOSUB/ERROR GOTO 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 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/-128 bytes or a RST to an illegal address.

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 MOD 255.

String too long 19 You will get this error message if your program tries to generate a string which is longer than 255 characters.

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
```

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', BBCBASIC(Z80) was able to recognize the start of the command.

Too big 20 This error will occur if a number is entered or calculated which is too big for BBCBASIC(Z80) to cope with.

Too many open files 192 This error will occur if you try to open more than 7 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 Function for details of parameter passing to functions and procedures.)

SYNTAX

1. Introduction. This Annex gives (in alphabetical order) abbreviated definitions for the commands and statements in BBCBASIC(Z80). 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 BBCBASIC(Z80) users. To those amongst you who would have preferred 'the real thing' - we apologize.

2. 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 BBCBASIC(Z80) statement.
- <var> Means a numeric or string variable.
- <exp> Means an expression like $\text{PI} \times \text{radius} \times \text{height} + 2$ or $\text{name}\$ + \text{"FRED"} + \text{CHR}\$(&0\text{D})$. It can also be a single variable or constant like 23 or "FRED".
- <l-num> Means a line number in a BBCBASIC(Z80) program.
- <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: $\text{PI} \times \text{radius} + 2.66$
- <s-const> Means a string constant like "FRED".

- <s-var> Means a string variable like 'address\$'.
- <string> 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 BBCBASIC 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.
- <v-name> Means a valid variable name.
- <afsp> Ambiguous file specifier.
- <ufsp> Unambiguous file specifier.
- <nchr> Character valid for use in a name. 0 to 9, A to Z, a to z and underline.

BBCBASIC(Z80) COMMANDS AND FUNCTIONS

ABS Absolute Value

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

Give the absolute value of the argument.

ACS Arc-cosine

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

Gives the arc-cosine of the argument (in radians).

ADVAL Analogue to Digital Converter Value

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

Return the last known value of the A to D channel given as the argument.

AND And

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

Perform and return an integer bitwise (and logical) AND between the 2 arguments.

ASC American Standard Code for Information Interchange (ASCII).

<n-var>=ASC(<string>)

Return the ASCII character value of the first character of the argument string, or -1 if the string is empty.

ASN Arc-sine
 <n-var>=ASN(<numeric>)
 Give the arc-sine of the argument (in radians).

ATN Arc-tangent
 <n-var>=ATN(<numeric>)
 Give the arc-tangent of the argument (in radians).

AUTO Automatic
 AUTO [<n-const> [,<n-const>]]
 Generate line numbers.
 AUTO [start[,increment]]

BGET# Get a Byte from File
 <n-var>=BGET#<n-var>
 Get a byte from the file whose channel number is given by the argument.

BPUT# Put a Byte to File
 BPUT#<n-var>,<numeric>
 Write LS byte of <numeric> to disk file specified by <n-var>.

CALL Transfer Control to a Machine Code Subroutine
 CALL<numeric>{,<n-var>|<s-var>}
 Call assembly language routine.
 CALL address{,arg}

CHAIN Chain Another Program
 CHAIN <string>
 Load and run a program.

CHR\$ Character String
 <s-var>=CHR\$(<numeric>)
 Get the character whose ASCII value is given by the argument.

CLEAR Clear Variables
 CLEAR
 Clear dynamic variables.

CLOSE# Close a File
 CLOSE#<numeric>
 Close the disk file specified by the argument. (IF argument=0, then close all files).

CLG Clear Graphics Screen
 CLG
 Clear graphics area of the screen.

CLS Clear the Text Screen
 CLS
 Clear the text area of the screen.

COLOUR Colour Select

COLOUR<numeric>

Select the text foreground and background logical colours.

COS Cosine

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

Give the cosine of the radian argument.

DATA Data Follows

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

Precedes data for a READ statement.

DEF Define

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

This is not strictly correct. A variable name cannot start with 1, but FN1 or PROC1 is valid. A more correct definition would be:

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

Precedes a declaration of a user function or procedure.

DEF FNname[(arg list)]
Define a function.

DEF PROCname[(arg list)]
Define a procedure.

DEG Degrees

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

Convert an angular argument in radians to degrees.

DELETE Delete Program Lines

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

Delete program lines.

DELETE start,end

DIM Dimension an Array/Reserve Memory

DIM <n-var>|<s-var>(<numeric>{,<numeric>})

Dimension arrays. Arrays must be dimensioned before use.

DIM var(sub1{,sub2...}){,...}

DIM <n-var> <numeric>

Reserve memory for special applications. (For assembler code for example.)

DIV Division of Whole Numbers

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

Gives the integer division of the first argument by the second.

DRAW Draw a Line

DRAW <numeric>,<numeric>

Draws a line on the screen from the previous graphics position to the X,Y co-ordinate defined by the arguments.

ELSE Alternative Action in IF...THEN...ELSE,
ON...GOSUB...ELSE and
ON...GOTO...ELSE
structures.

IF <t-cond> THEN <stmt> ELSE <stmt>
ON <n-var> GOTO<l-num>{,<l-num>} ELSE <stmt>
ON <n-var> GOSUB<l-num>{,<l-num>} ELSE <stmt>

Provides an alternative course of action in the
IF...THEN...ELSE and ON...GOTO/GOSUB structures.

EDIT Edit Line

EDIT <l-num>

Edit the specified program line.

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

Concatenate and edit the specified range of lines.

END End Program

END

The (optional) end-of-program statement. It may
occur as often as necessary. It terminates the
program and closes any open files.

ENDPROC End Procedure

ENDPROC

Return from a procedure. Part of the
PROC...ENDPROC structure.

ENVELOPE Envelope (Shape) of Sound

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

Controls the volume and pitch of a sound whilst it
is playing. Used with the SOUND statement.

EOF# End of File

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

Gives the end-of-file status of the specified
file.

EOR Exclusive OR

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

Gives the bitwise exclusive-or between the first
and second argument.

ERL# Error Line Number

<n-var>=ERL

Gives the line number where the last error
occurred.

ERR Error

<n-var>=ERR

Gives the error number of the last error which
occurred.

EVAL Evaluate

<n-var>=EVAL(<string>)
 <s-var>=EVAL(<string>)

Gives the numeric or string evaluation of the argument by applying the interpreter's expression evaluation program to the argument string.

EXP Exponent

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

Gives the value of e raised to the power of the argument.

EXT# Extent

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

Gives the number of bytes (to the next highest multiple of 128) allocated to the specified file.

FALSE Pseudo Boolean Function returning the value FALSE (0).

<n-var>=FALSE

A pseudo-boolean function returning the numeric value of 0.

FN

Function. Precedes a variable name indicating it is being used for a function.

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

This is not strictly correct. A variable name cannot start with 1, but FN1 is valid. A more correct definition would be:

<n-var>|<s-var>=FN<nchr>{<nchr>}[(<exp>{,<exp>})]

Uses a function to return a value.

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

or, more correctly

DEF FN<nchr>{<nchr>}[(<s-var>|<n-var>{,<s-var>|<n-var>})]

Defines a function.

FOR

Introduces FOR...NEXT Loop

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

Begin a FOR...NEXT loop.

GCOL

Graphics Colour

GCOL <numeric>,<numeric>

Sets the graphics colour to be used for all subsequent graphics operations.

GET

Get a Key Value or Get a Byte from a Port

<n-var>=GET

Waits for a key to be pressed and returns with the ASCII value of the character.

<n-var>=GET(<numeric>)

Reads from the specified Z80 I/O port.

GET\$

Get a Key

<s-var>=GET\$

Waits for a key to be pressed and returns with the character.

GOSUB

Go To a Subroutine

GOSUB <l-num>

GOSUB (<numeric>)

Calls a BASIC subroutine at the specified line number.

GOTO

Go To a Line Number

GOTO <l-num>

GOTO (<numeric>)

Branch to the specified line.

HIMEM

Highest Memory Location

HIMEM=<numeric>

<n-var>=HIMEM

A pseudo-variable which specifies the highest memory location used for the BBCBASIC(Z80) program plus one. (In other words, the first memory location above that known to BBCBASIC(Z80)).

IF

Conditional in IF...THEN...ELSE Structure

IF <t-cond> THEN <stmt>{:<stmt>} [ELSE <stmt>{:<stmt>}]
IF <exp> THEN <stmt>{:<stmt>} [ELSE <stmt>{:<stmt>}]

Do statement(s) if <t-cond> or <exp> = TRUE.

IF <t-cond> GOTO <l-num> [ELSE <l-num>]

IF <exp> GOTO <l-num> [ELSE <l-num>]

IF <t-cond> THEN <l-num> [ELSE <l-num>]

IF <exp> THEN <l-num> [ELSE <l-num>]

Branch if <t-cond> or <exp> = TRUE.

INKEY

Input the Number of the Key Depressed

<n-var>=INKEY(<numeric>)

Waits for a specified time for a key to be pressed. If a key is pressed within the time, the ASCII value of the key is returned; if not, -1 is returned.

INKEY\$ Input the Character Pressed

<s-var>=INKEY\$(<numeric>)

Waits for a specified time for a key to be pressed. If a key is pressed within the time, the key is returned; if not, an empty string is returned.

INPUT

To Input Information Into the Computer

INPUT [TAB(X[,Y])][SPC(n)]['] [<s-const>[,|;]]

<n-var>|<s-var>{,<n-var>|<s-var>}

Requests input from the user. A comma or a semi-colon after the prompt causes a question mark.

INPUT LINE [<s-const>[,|;]]

<n-var>|<s-var>{,<n-var>|<s-var>}

As INPUT, but accepts the whole line including commas, quotes etc.

INPUT# To Input Information Into the Computer From a Disk File

INPUT#<n-var>,<n-var>|<s-var>{,<n-var>|<s-var>}

Reads data from a file into the specified variables.

INSTR In String

<n-var>=INSTR(<string>,<string>[,<numeric>])

Gives the position of a sub-string (second argument) within a string (first argument). The starting position for the search may be specified (third argument).

INT Integer Part

<n-var>=INT<numeric>

Gives the whole number less than the value of the argument. For example, INT(3.45) will return 3 and INT(-3.45) will return -4.

LEFT\$ Left String

<s-var>=LEFT\$(<string>,<numeric>)

Gives the leftmost n characters (second argument) from a string (first argument).

LEN Length of a String

<n-var>=LEN(<string>)

Gives the length of the argument string.

LET Assignment

[LET] <var>=<exp>

An optional assignment command.

LIST List Program

LIST

LIST <n-const>

LIST <n-const>,<n-const>

LIST ,<n-const>

LIST <n-const>,<n-const>

List from the line specified by the first argument to the line specified by the second argument. If the first argument is omitted, list from the start of the program. If the last argument is omitted, list to the end.

LISTO List Option

LISTO <n-const>

Specifies the listing format. The default format is LISTO 7.

LN Natural Logarithm

<n-var>=LN<numeric>

Gives the natural logarithm of the argument.

LOAD Load a Program

LOAD <string>

Loads the specified program file from DISK.

LOCAL Specify Local Variables

LOCAL <s-var>|<n-var>{,<s-var>|<n-var>}

Declare variables local to function or procedure.

LOG Logarithm

<n-var>=LOG<numeric>

Gives the common logarithm of the argument (base 10).

LOMEM Lowest Memory Location

LOMEM=<numeric>
<n-var>=LOMEM

A pseudo-variable which specifies the start of the memory where the dynamic variables are stored.

MID\$ Middle String

<s-var>=MID\$(<string>,<numeric>[,<numeric>])

Gives a sub-string of the first argument. The second argument specifies the starting position for the sub-string within the first argument, and the third argument specifies the number of characters.

MOD Modulus

<n-var>=<numeric> MOD <numeric>

Gives the signed remainder of an integer division.

MODE Display Mode

MODE <numeric>

Sets the display mode.

MOVE Move Graphics Cursor

MOVE <numeric>,<numeric>

Moves the graphics cursor to the specified position without drawing a line.

NEW 'Remove' Old Program

NEW

Delete current program & variables.

NEXT Delimits FOR...NEXT Loop

NEXT [<n-var>{,<n-var>}]

End FOR...NEXT loop.

NOT Logical NOT Operation

<n-var>=NOT<numeric>

Performs a bitwise inversion and gives the ones complement of the argument.

OLD Recover Old Program

OLD

Recovers a program deleted by NEW or re-entering BBCBASIC(Z80).

ON Introduces a Program 'Switch'

ON <numeric> GOTO<l-num>{,<l-num>}
ON <numeric> GOSUB<l-num>{,<l-num>}

Provides multiple options in changing the order of execution of a program.

ON exp GOTO l-num,l-num.. [ELSE l-num]
Computed GOTO.

ON exp GOSUB l-num,l-num.. [ELSE l-num]
Computed GOSUB.

ON ERROR Assume/Relinquish Program Control of Error Handling.

ON ERROR <stmt>{:<stmt>}

Transfers control to <statement> when an error is reported. The <statement> can be GOTO a line number.

ON ERROR GOTO <l-num>

ON ERROR OFF

Restores default error handling.

OPENIN Open File for Input

<n-var>=OPENIN(<string>)

Attempts to open a file for input or random access and returns the allocated channel number. If the file does not exist, 0 is returned.

OPENOUT Open File for Output

<n-var>=OPENOUT(<string>)

Attempts to open a file for output and returns the allocated channel number. If there is insufficient room in the directory for another file, 0 is returned.

OPT Option

OPT <numeric>

Controls the output during assembly.

OPENUP Open File for Input to Computer

<n-var>=OPENUP(<string>)

This function is identical to OPENIN.

OR Logical OR

<n-var>=<numeric>OR<numeric>

Gives the bitwise integer logical OR between the 2 arguments.

OSCLI Operating System Command Line Interpreter

OSCLI <string>

Pass the argument to the 'operating system'.

PAGE Start Address for User's Program

PAGE=<numeric>

<n-var>=PAGE

A pseudo-variable giving the address of the start of the user's program. The least significant byte is always set to 0 by the computer.

PI The Constant 3.14159265

`<n-var>=PI`

A constant expressing the relationship between the circumference and the diameter of a circle.

PLOT Multi-purpose Point, Line and Triangle Drawing Statement.

`PLOT <numeric>,<numeric>,<numeric>`

Controls the drawing of points, lines and triangles on the screen.

POINT Find Colour of Point on Screen

`<n-var>=POINT(<numeric>,<numeric>)`

Gives the logical colour of the specified position on the screen. Returns -1 if the specified point is off the screen.

POS Position

`<n-var>=POS`

Gives the horizontal position of the cursor within the current text window.

PRINT Print

`PRINT {[TAB(x[,y])][SPC(n)]['[,;]]}[~]
[<string>|<numeric>]}`

Print data to the currently selected output stream.

PRINT# Print to File

`PRINT#<n-var>{,<numeric>|<string>}`

Write data to disk file.

PROC Procedure. Precedes a variable name indicating it is being used for a procedure.

`PROC<v-name>[(<exp>{,<exp>})]`

This is not strictly correct. A variable name cannot start with 1, but PROC1 is valid. A more correct definition would be:

`PROC<nchr>{<nchr>}[(<exp>{,<exp>})]`

Calls a procedure.

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

or, more correctly

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

Defines a procedure.

PTR# Pointer

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

Gives or sets the access pointer within a file.

PUT Output to a Hardware Port

`PUT <numeric>,<numeric>`

Output the least-significant byte of the second argument to the hardware port specified by the first argument.

RAD Radian

<n-var>=RAD <numeric>

Converts an angular argument in degrees to radians.

READ Read Data

READ<n-var>|<s-var>{,<n-var>|<s-var>}

Read data from DATA statement(s).

REM Remark

REM any text

Allows comments to be inserted.

RENUMBER Renumber Program

RENUMBER [<n-const>[,<n-const>]]

Renumber program lines. The first argument specifies the start number and the second the increment.

RENUMBER [start[,inc]]

REPEAT Introduces REPEAT...UNTIL Structure

REPEAT

Start of a REPEAT...UNTIL loop.

REPORT Report Error

REPORT

Print the last error, in words, to the currently selected output channel.

RESTORE Restore Data Pointer

RESTORE [<l-num>]

RESTORE [(**<numeric>**)]

Restore data pointer to the line number specified. Defaults to first data statement.

RETURN Delimit a Subroutine

RETURN

Causes a branch to the statement after the one which contained the GOSUB which caused the branch to the present subroutine.

RIGHT\$ Right String

<s-var>=RIGHT\$(<string>,<numeric>)

Gives the rightmost n characters (second argument) from a string (first argument).

RND Random<n-var>=RND[(**<numeric>**)]

Gets (generates) a random number.

RUN Run the Program

RUN

Causes the computer to execute the current program.

SAVE Save Program

SAVE <string>

Save the current program to disk with the specified file name.

SGN Sign

<n-var>=SGN(<numeric>)

Gives the sine of the argument. If the argument is positive, +1 is returned; if negative, -1 is returned; if zero, 0 is returned.

SIN Sine

<n-var>=SIN(<numeric>)

Gives the sign of the radian argument.

SOUND Generate a Sound

SOUND <numeric>,<numeric>,<numeric>,<numeric>

Generates the sound specified by the arguments.

SPC Space

PRINT SPC (<numeric>)
INPUT SPC (<numeric>)

Prints multiple spaces (number specified by the argument) to the currently selected output stream. It can only be used within PRINT and INPUT statements.

SQR Square Root

<n-var>=SQR(<numeric>)

Gives the square root of the argument.

STEP Optional Part of FOR...NEXT Structure

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

The amount by which the control variable is incremented for every pass through a FOR...NEXT loop.

STOP Stop Program

STOP

Interrupts a program which is running and prints the message:

STOP at line nnnn

STR\$ String

<s-var>=STR\$[~](<numeric>)

Gives the string form of the numeric argument as it would have been printed.

STRING\$ Generate a String`<s-var>=STRING$(<numeric>,<string>)`

Gives multiple concatenations of the string given as the second argument.

TAB Tabulation

```
PRINT TAB(<numeric>[,<numeric>])
INPUT TAB(<numeric>[,<numeric>])
```

Can only be used within PRINT or INPUT statements.

Single Argument

Prints spaces (and a new-line if necessary) to reach the column specified by the argument.

Double Argument

Moves the cursor directly to the specified co-ordinates.

TAN Tangent`<n-var>=TAN(<numeric>)`

Gives the tangent of the argument. The argument must be in radians.

THEN Selected Action in IF...THEN...ELSE Structure`IF <t-cond> THEN <stmt>{:<stmt>} [ELSE <stmt>{:<stmt>}]`

Do statement(s) if *<t-cond>* TRUE.

`IF exp GOTO <l-num> [ELSE <l-num>]``IF exp THEN <l-num> [ELSE <l-num>]`

Branch if *<t-cond>* TRUE.

THEN is optional unless followed by a line number, a pseudo-variable or a 'star' command.

`300 IF flag THEN TIME=0``400 IF flag THEN *LOAD WOMBAT 8000`**TIME** Time`TIME=<numeric>``<n-var>=TIME`

Sets or reads the internal timer.

TO Defines Final Value in FOR...NEXT Structure`FOR<n-var>=<numeric>TO<numeric>[STEP<numeric>]`

The limit value for the control variable before a FOR...NEXT loop is terminated.

TOP Top of Program`<n-var>=TOP`

Gives the first free memory location above the user's program.

TRACE Trace Action

TRACE ON|OFF|<l-num>
TRACE ON|OFF|(<numeric>)

Prints out the numbers of the program lines just before they are executed.

TRACE ON - Start trace mode.

TRACE OFF - End trace mode.

TRACE (<exp>) - Trace lines less than <exp>.

TRUE Pseudo Boolean Function returning the value TRUE (-1).

<n-var>=TRUE

A pseudo-boolean function returning a numeric value of -1.

UNTIL Delimit REPEAT...UNTIL Structure

UNTIL <t-cond>

Terminates a REPEAT...UNTIL loop if <t-cond> is non-zero (NOT FALSE).

USR User

<n-var>=USR(<numeric>)

Allows a machine code program to return a single value.

VAL Value

<n-var>=VAL(<string>)

Converts a character string representing a numeric value into numeric form.

VDU Visual Display Unit

VDU <numeric>{,|;<numeric>}[;]

Sends a stream of values to the currently selected output stream. If the arguments are separated by commas, single bytes are sent. If separated by semi-colons, the argument is sent as 2 bytes - the LSB is sent first.

VPOS Vertical Position of Cursor

<n-var>=VPOS

Gets the vertical position of the text cursor.

WIDTH Page Width

WIDTH <numeric>

Controls the overall output field width. It is initially set to 0 which disables auto new-lines.

OPERATING SYSTEM COMMANDS***BYE** ***BYE**

Return to the operating system.

***CPM** ***CPM**

Return to the operating system. (Not Torch.)

***DIR** ***DIR [<afsp>]**

List the files in the disk directory conforming with <afsp>. The default <afsp> is *.BBC.

***ERA** ***ERA <afsp>**

Erase the files conforming with <afsp>.

***LOAD** ***LOAD <ufsp> aaaa**

Load the specified file into memory at address aaaa. The address must be specified in hexadecimal.

***OPT** ***OPT <n-const>**

Specify the console output stream as follows:

*OPT 0 - Console output.

*OPT 1 - Auxiliary output.

*OPT 2 - List device output.

***REN** ***REN <ufsp1>=<ufsp2>**

Rename a file. The drives specified in the <ufsp>s must be the same.

***RESET** ***RESET**

Reset the disk system.

***SAVE** ***SAVE <ufsp> aaaa +1111**
***SAVE <ufsp> aaaa bbbb**

Save an area of memory to disk. The start address is aaaa and the end address is bbbb-1. Alternatively, the length may be specified as +1111.

***TYPE** ***TYPE <ufsp>**

Print the specified file to the screen.

The default extension is .BBC in all cases.

A "star" command cannot contain variable names and must be the last item on a program line.

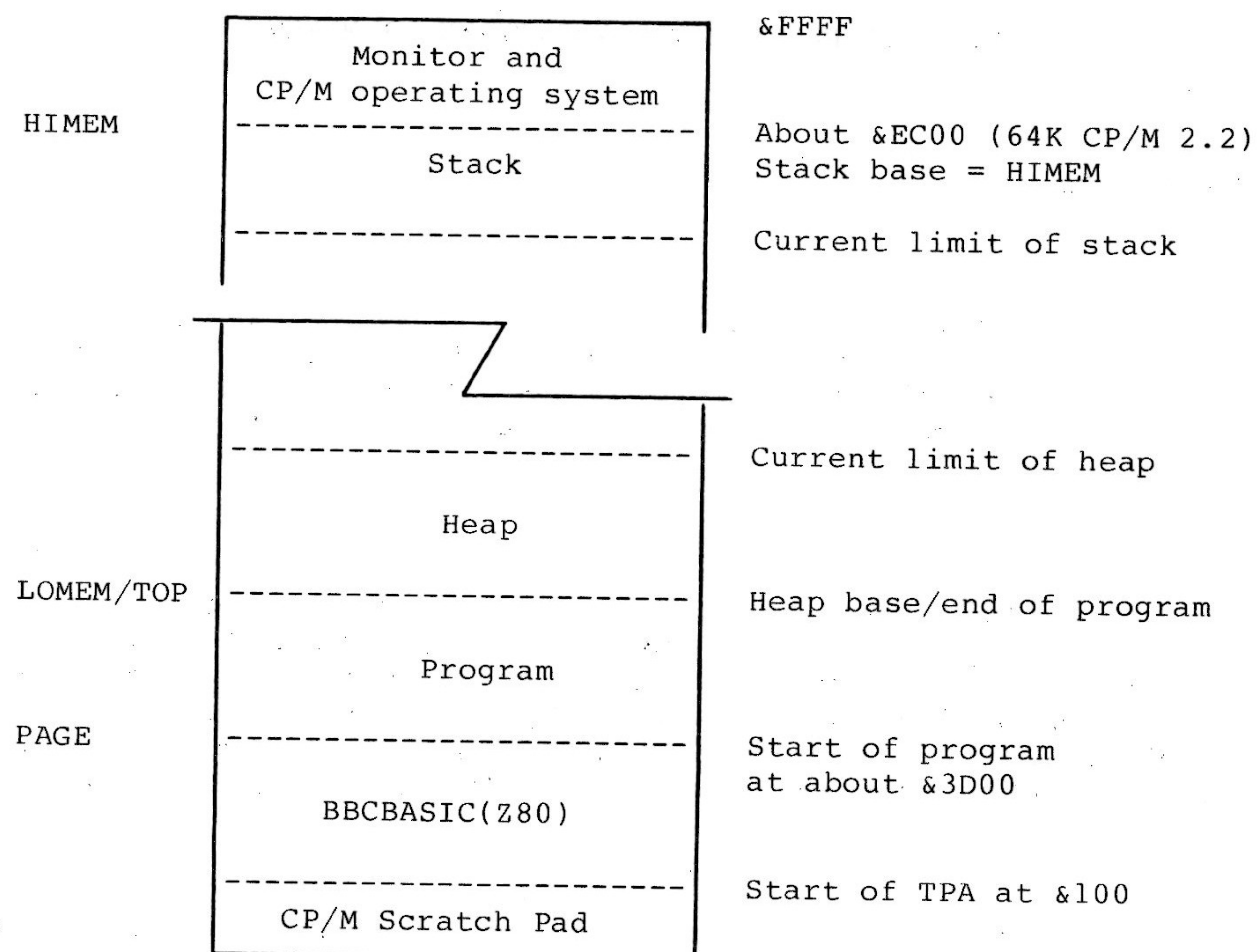
FORMAT OF PROGRAM AND VARIABLES IN MEMORY

MEMORY MAP

1. BBCBASIC(Z80) runs under the CP/M (MCP/CPN) operating system. It loads at the start of the transient program area (TPA) at 100H and occupies about 15k of memory. This leaves about 44k of memory for your program and its variables in a 64K system (CP/M 2.2).
2. By default, your program will start on the page boundary immediately following the BBCBASIC(Z80) interpreter and the 'dynamic data structures' (the variables and the file control blocks and buffers) 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.
3. As your program runs, the heap expands upwards towards the stack and the stack expands downwards towards the heap. If the 2 should meet, you get a 'No room' error. Fortunately, there is a limit to the amount by which the stack and the heap expand.
4. In general, the heap only expands whilst new variables are being declared. However, bad management of string variables and files can also cause the heap to expand.
5. In addition to running your program, the stack is also used 'internally' by the BBCBASIC(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 private variables in use.

Format of Program and Variables in Memory

6. The default memory map for a BBCBASIC(Z80) program is illustrated below.



7. The function of HIMEM, 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 and 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 BBCBASIC(Z80). The base of the program stack is set at HIMEM. (The first 'thing' stored on the stack goes at HIMEM-1.)

Format of Program and Variables in Memory

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 page boundary where your program starts. The least significant byte of PAGE is always zero.

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

8. 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, by good string variable management, and by closing files in the correct order.

Limiting the Number of Variables.

9. Since each new variable occupies room on the heap, limiting the number of variables 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.

10. 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 reassignments 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.

11. 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.

File Management

12. Memory Requirements for Files. When you open a file, a 38 byte file control block (FCB) and a 128 byte file buffer area are added to the heap. Every file opened needs an FCB and a buffer. Consequently, the heap grows by 166 bytes every time a file is opened. When you close a file, the space occupied by the FCB and the buffer are recovered if they are on the top of the heap.

13. Running the Program. As the program runs, variables are created, and files are opened and closed. If new variables have been created after a file has been opened, the heap space used for the file's FCB and buffer will not be recovered. Eventually however, things settle down, no new variables are created and the size of the heap fluctuates as files are opened and closed, but it does not keep expanding.

14. Bad Management. You can, however, force the heap to expand every time you open a file. Consider the following sequence:

```
X=OPENOUT "A"
Y=OPENOUT "B"
CLOSE#X      :REM Can't recover space - blocked by "B"
X=OPENOUT "C"
CLOSE#Y      :REM Can't recover space - blocked by "C"
Y=OPENOUT "D"
CLOSE#X      :REM Can't recover space - blocked by "D"
```

At this stage, there is only 1 file open, but we have consumed enough heap space for 4 files. If we continue with this sequence, we will eventually consume all the available heap space and get a 'No room' error. We could have compounded our error by creating new variables in between opening the files. That way, even if we had got the order right, recovery of the FCB and buffer area would have been blocked by the new variables.

15. Good Management. By re-arranging the sequence we can recover the space used by the 'old' files.

```
Y=OPENOUT "B"
X=OPENOUT "A"
CLOSE#X      :REM Space recovered
X=OPENOUT "C"
CLOSE#Y      :REM Can't recover space - blocked by "C"
CLOSE#X      :REM Space recovered for "C". Space for
Y=OPENOUT "D" :REM "B" cannot be recovered.
```

Better still,

```
X=OPENOUT "A"
Y=OPENOUT "B"
Z=OPENOUT "C"
CLOSE#Z
CLOSE#Y      :REM All the space is recovered
CLOSE#X
Y=OPENOUT "D"
```

Obviously, you can't always manage things quite so tidily, but a little planning saves a lot of memory.

Management Rules (OK?)

17. Most of the time you will have plenty of room for your program and all its variables. However, the following rules will stand you in good stead on the few occasions when you are short of space.

18. Variables.

a. Create your variables before you open any files (using dummy values if necessary). By doing this, the variables won't block recovery of the FCBs and buffer areas of closed files.

b. Keep in mind that every new variable adds to the heap.

19. Strings. The 2 previous rules apply. In addition:

a. The first time you create a string, give it a value which is as long as the longest value you will ever want to assign to it. Use a dummy value (STRING\$(nn," ")) if necessary.

b. If you forget the previous rule, remember not to assign a longer value to a string whilst any files are open.

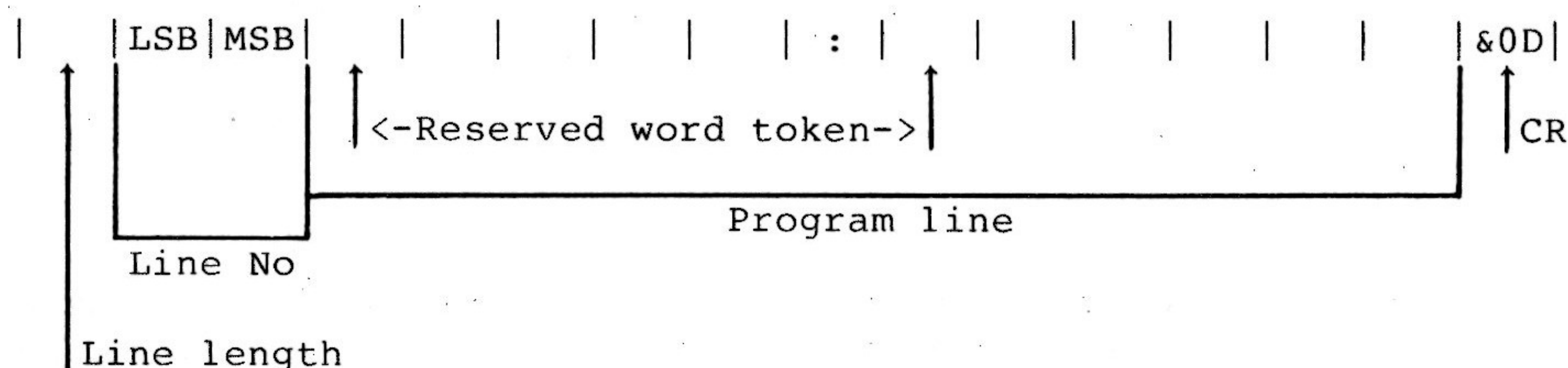
20. Files.

a. Plan the order in which you open and close files. In general close them in the reverse order to opening them. In other words, last opened - first closed.

b. Don't create any new variables whilst there are any open files.

PROGRAM STORAGE IN MEMORY

21. The program is stored in memory in the format shown below. The first program line commences at PAGE.



22. 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.

23. Line Number. The line number is stored in 2 bytes, LSB first. The end of the program is indicated by a line number of &FFFF and a line length of zero.

24. 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.

25. Line Terminator. Each program line (except the last) is terminated by a carriage-return (&0D).

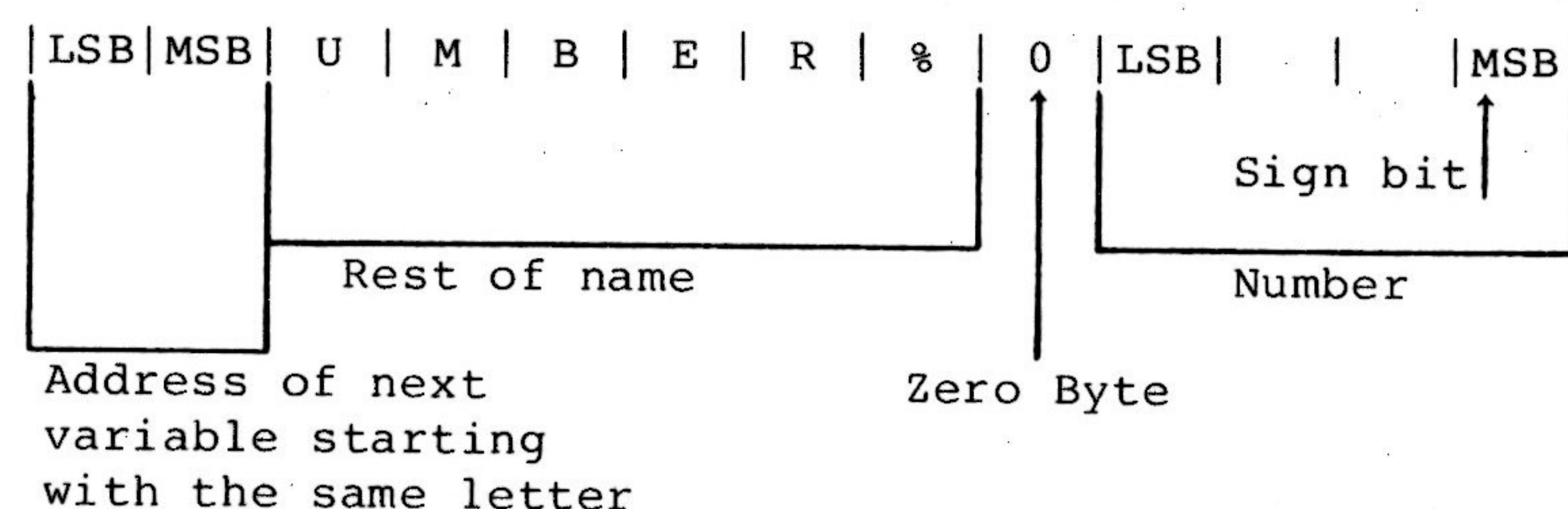
VARIABLE STORAGE IN MEMORY

26. Variables are held within memory as linked lists (chains). The first variable in each chain is accessed via an index which is maintained by BBCBASIC(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 (2 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.

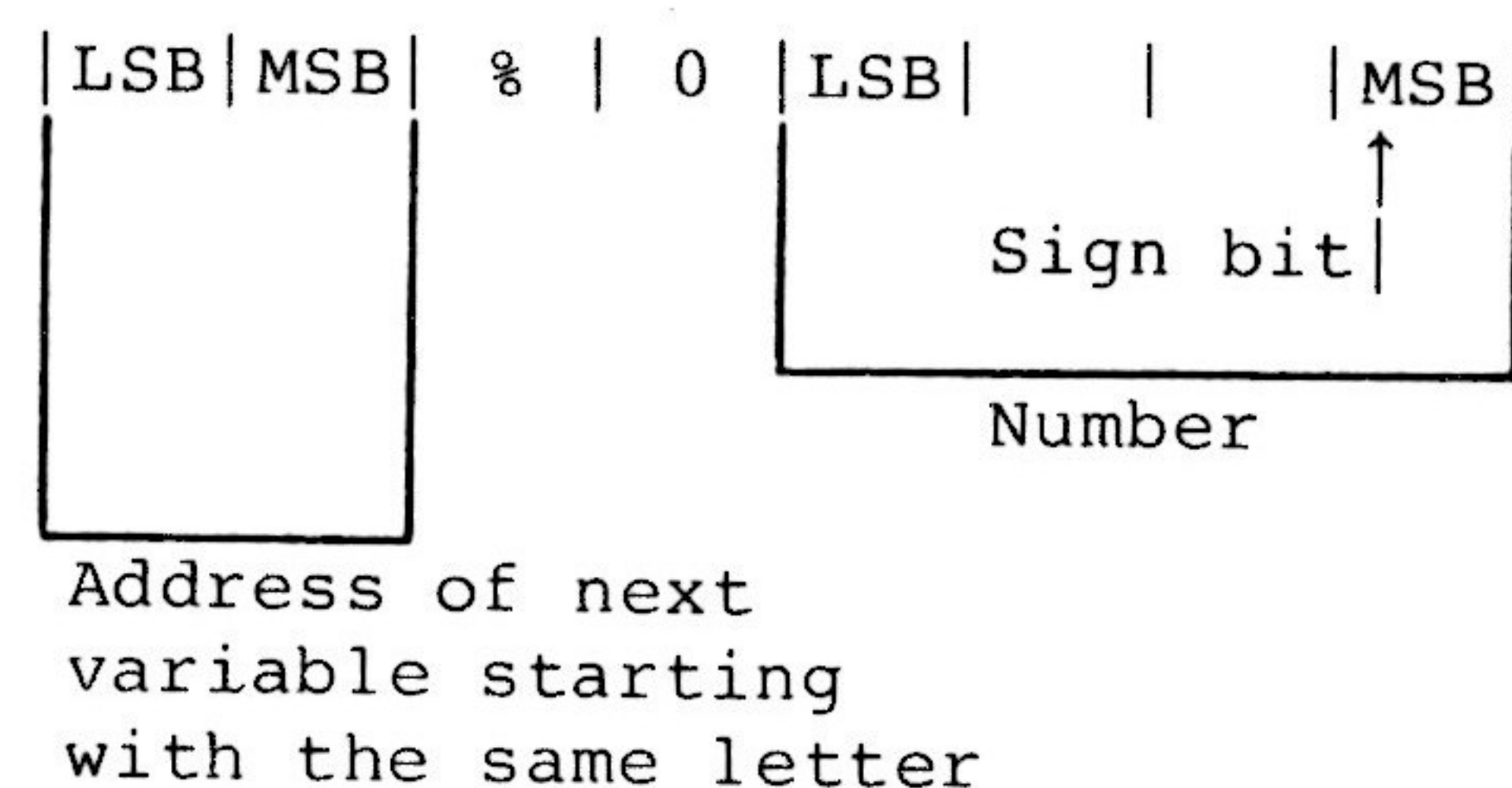
27. The first variable created for each starting character is accessed via the index and subsequently created variables are accessed via the index and the chain. Consequently, there is some speed advantage to be gained by arranging for all your variables to start with a different character. Unfortunately, this can lead to some pretty unreadable names and programs.

Integer Variables

28. Integers are held in 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.

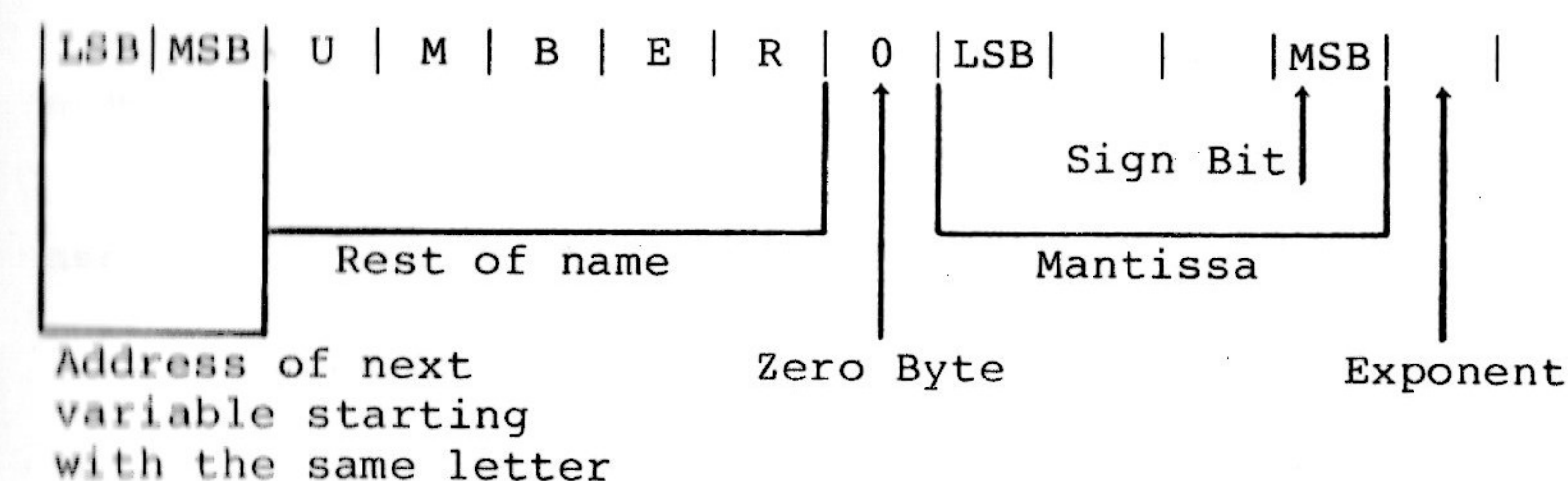


29. The smallest amount of space is taken up by a variable with a single letter name. The static integer variables, which are not included in the variable chains, use the names A% to Z%. Thus, the only single character names available for dynamic integer variables are a% to z% plus _% and `% (CHR\$(96)). As shown below, integer variables with these names will occupy 8 bytes.



Reals

30. 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.



31. 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.

32. In the following examples, the bytes are shown in the more human-readable manner with the MSB on the left.

33. The value 5.5 would be stored as shown below.

Mantissa	Exponent
.0011:0000 0000:0000 0000:0000 0000:0000	1000:0010
↑ Sign bit	
↑ Binary point	
& 30 00 00 00	& 82

Because the sign bit is assumed to be 1, this would become:

.1011:0000 0000:0000 0000:0000 0000:0000	1000:0010
& B0 00 00 00	& 82

The equivalent in decimal is:

$$\begin{aligned}
 & (0.5+0.125+0.0625) * 2^{(130-127)} \\
 & = 0.6875 * 2^3 \\
 & = 0.6875 * 8 \\
 & = 5.5
 \end{aligned}$$

34. BBCBASIC(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.

35. Depending on how it is put there, an integer value can be stored in a real variable in one of 2 ways. For example,

number=5

will store the integer &00 00 00 05 in the mantissa and set the exponent to zero. On the other hand,

number=5.0

will set the mantissa to &20 00 00 00 and the exponent to &82.

36. The 2 ways of storing an integer value are shown in the following examples.

number=5 & 00 00 00 05 00 Integer 5

number=5.0 & 20 00 00 00 82 Real 5.0

This is treated as

& A0 00 00 00 82 $= (0.5+0.125) * 2^{(130-127)}$
 $= 0.625 * 8$
 $= 5$

because the sign bit
is assumed to be 1.

number=-5 & FF FF FF FB 00

2's complement gives

& 00 00 00 05 00 Integer -5

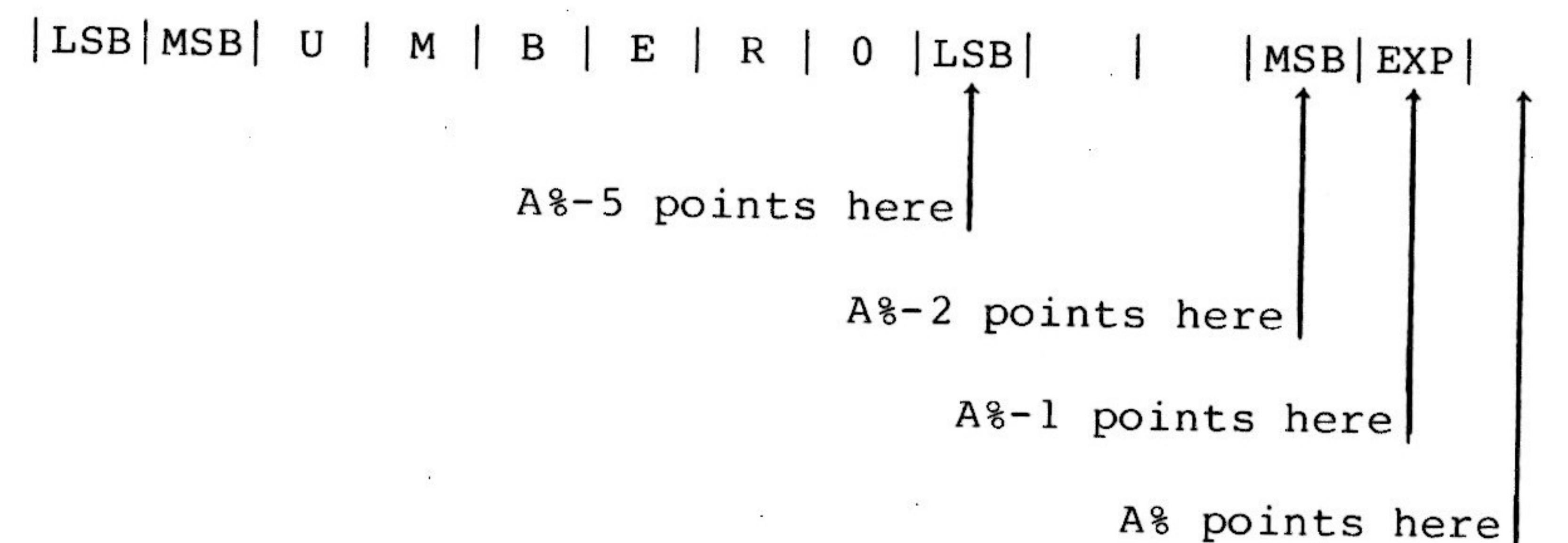
number=-5.0 & A0 00 00 00 82 Real -5.0

The sign bit is already 1. $= (0.5+0.125) * 2^{(130-127)}$
 $= 0.625 * 8$
 Magnitude =5

37. If all this seems a little complicated, try using the program below 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 sub-section for details.)

38. The layout of the variable 'NUMBER' in memory is shown below.



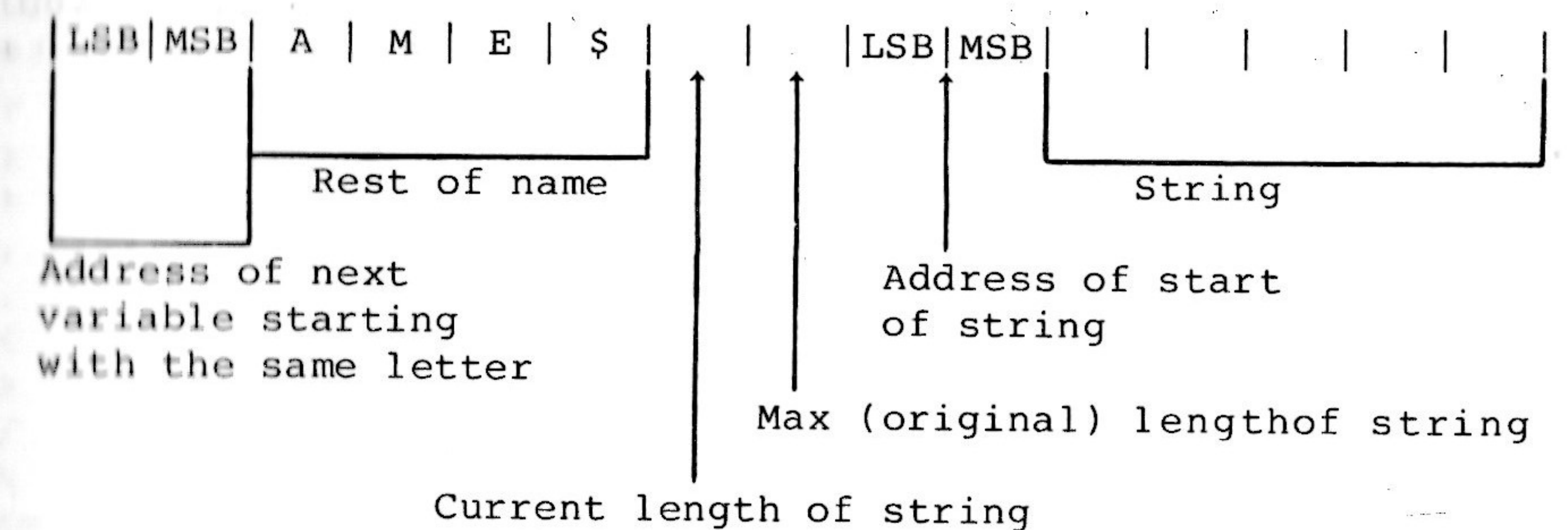
```

10 NUMBER=0
20 DIM A% -1
30 REPEAT
40   INPUT"NUMBER PLEASE "NUMBER
50   PRINT "& ";
60   :
70   REM Step through the 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

39. 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 is 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 BBCBASIC(Z80) needs information about the length of the string and the address in memory where it starts, the overheads for a string are more than for a numeric. The format of a string variable called 'NAME' is shown below.



40. When a string variable is first created in memory, the characters of the string follow immediately after the 2 bytes pointing to 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 2 address bytes. The original string space will remain, but it will be unusable. This unusable string space is called 'garbage'. See the Variables sub-section for ways to avoid creating 'garbage'.

41. 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

42. You can place a string starting at a given location in memory using the indirection operator '\$'. For example,

\$&5000="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. Thus in the above example, byte &5010 would be set to &0D.

BBCBASIC(Z80) INDEX

?	23
!	23
"	23
#	23
\$	23
%	23
&	23
'	23
(23
=	23
-	23
*	23
/	23
+	23
,	23
.	24
<	24
>	24
<	24
>	24
<	24
>	24
[24
]	24
^	24
~	24
*BYE	209, D-30
*CHAR	209
*DIR	209, 229, D-30
*DISP	209
*DOS	209
*DRIVE	210
*ERA	210, 228, D-30
*EXEC	210
*KEY	211
*LOAD	211, D-30
*LOCK	211
*MAG	212
*OPT	212, D-30
*PSW	212
*REN	212, 229, D-31
*RESET	213, D-31
*SAVE	52, 213, D-31
*BPOOL	213

*SPRITE	214
*TYPE	215, D-31
*UNLOCK	215
-ve Root	C-8
ABS	57, D-3
ACS	58, D-3
ADVAL	59, D-3
AND	61, D-3
ASC	62, D-3
ASCII Codes	Annex A
ASN	63, D-4
ATN	64, D-4
AUTO	65, D-4
Accuracy Lost	C-3
Arguments	C-3
Array	C-3
Arrays	10, 85
Assembler	41
- Byte Constants	45
- Comments	42
- Conditional Assembly and Macros	49
- DIM - Use of	43, 86
- Labels	42
- Length of Reserved Memory Area	48
- Limitations	53
- Mnemonics	41
- OPT Summary	45
- Program Counter	43
- Saving and Loading Programs	52
- String Constants	45
- Word Constants	45
- Listing Control	44
- Statements	41
Assembly Process	46
BBCBASIC(Z80) Disk Files	215
BBCBASIC(Z80) File Format	222
BGET#	66, 239, D-4
BPUT#	67, 239, D-4
Bad Call	C-3
Bad Command	C-3
Bad DIM	C-4
Bad HEX	C-4
Bad Key	C-4
Bad Name	C-4
Bad Program	C-4
Bad String	C-4
Basics - File Structure	217
Binary Chop	289

Binary Chop Flow Chart	291
Binary Operators - Use of	21
Boolean Variables	10
Byte Constants - Assembler	45
CALL	68, D-4
CHAIN	71, 226, D-5
CHAIN - Common Variables	71
CHR\$	72, D-5
CLEAR	73, D-5
CLG	75, D-5
CLOSE#	74, 236, D-5
CLE	76, D-5
COLOUR	77, D-6
COS	79, D-6
COUNT	80
CP/M - Limitations	222
CP/M - Limitations - Overcoming	223
Can't Match FOR	C-5
Case Conversion - Operating System Commands	207
Channel	C-5
Character Data Files	241
Clock - Real-time - Setting	Program SETTIME.BBC
Close Error	C-5
Commands - Disk Files	224
Comments - Assembler	42
Common Variables	71
Compatible Data Files	252
Compatible Data Files - Reading	254
Compatible Data Files - Writing	252
Conditional Assembly and Macros	49
Control Codes - Generation	5
Control Codes - Special	5
Control Codes and Functions	5
DATA	81, D-6
DEF	82, 35, D-6
DEG	83, D-7
DELETE	84, D-7
DIM	85, 43, D-7
DIM Space	C-5
DIV	87, D-7
DRAW	88, D-7
Data Files	231
- Character	241
- Compatible	252
- Indexed	275
- Mixed Numeric/Character Data	245
- Random - Deficiencies	275
Directory Full	C-5
Disk File Commands	224

Disk Files - BBCBASIC(Z80)	215
Disk Full	C-5
Division by Zero	C-6
Dollar	20
EDIT	89, D-8
ELSE	91, D-8
END	92, D-8
ENDPROC	93, D-8
ENVELOPE	94, D-9
EOF#	96, 237, D-9
EOR	97, D-9
ERL	98, D-9
ERR	99, D-9
EVAL	100, D-10
EXP	101, D-10
EXT#	102, 240, D-10
Error Codes and Messages	Annex C
- Details	C-3
- Trappable - Disk	C-2
- Trappable - Program	C-1
- Untrappable	C-1
Error Handling	27
Error Handling - Limitations	31
Escape	C-6
Ex 1.1 - Writing Serial Character Data	242
Ex 1.2 - Reading Serial Character Data	243
Ex 1.3 - Writing 'At End' - Character Files	243
Ex 1.4 - Writing a Mixed Data File	245
Ex 1.5 - Reading a Mixed Data File	248
Ex 1.6 - Writing 'At End' of Mixed Files	249
Ex 1.7 - Writing a Compatible Data File	252
Ex 1.8 - Reading a Compatible Data File	254
Ex 2.1 - Simple Random Access File	257
Ex 2.2 - Simple Random Access Database	258
Ex 2.3 - Random Access Inventory Program	262
Ex 3.1 - Indexed File Address Book	277
Exclamation	19
Exp Range	C-6
Expression Priority	6
FALSE	103, 6, D-10
FN	104, 34, D-11
FOR	105, D-11
FOR Variable	C-7
FOR...NEXT Loop - Changing Loop Variable	15
FOR...NEXT Loop - Leaving	15
FOR...NEXT Loop - Popping Inner Variable	15
Failed at nnnn	C-6
File Exists	C-7
File Not Found	C-7

File Specifiers	208
Files - BBCBASIC(Z80)	220, 222
- Character Data	241
- Command Conventions	224
- Commands	224
- Compatible Data	252
- Compatible Data - Reading	254
- Compatible Data - Writing	252
- Data	231
- Data - Indexed	275
- Good Memory Management	E-5
- How Numeric Data is Stored	221
- How Strings are Stored	222
- How Data is Read/Written	221
- Indexed	219
- Indexed - Address Book	275
- Memory Management	E-5
- Mixed Data - Reading	248
- Mixed Data - Writing	245
- Mixed Data - Writing 'At End'	249
- Mixed Numeric/Character Data	245
- Opening	231
- Organization of Examples	224
- Program - Manipulation	224
- Random (Relative)	257
- Random - Deficiencies	275
- Random Access	218
- Random Access - Initialisation	262
- Random Access - Simple	257
- Random Access Database - Simple	258
- Random Access Inventory	262
- Serial (Sequential)	217
- Serial Character Data - 'At End'	243
- Serial Character Data - Reading	243
- Serial Character Data - Writing	242
- Serial(Sequential)	241
- Structure - Basics	217
- Structure of	217
- The Limitations of CP/M	222
Fixed Strings - Storage	E-16
Format of Program and Variables in Memory	Annex E
Function Names	34
Functions and Procedures	33
Functions and Procedures - Defining	35
Functions and Procedures - Parameters	36
Functions and Statements - BBCBASIC(Z80)	55
GOAL	107, D-11
OPT	108, D-11
OPTS	108, D-12
GOBUB	109, D-12

GOTO	110, D-12
Garbage Generation	11, E-4
General Information About BBCBASIC(Z80)	5
HIMEM	111, D-12, E-2
Hints and Tips for Beginners	2
Hints and Tips for Torch Users	Annex F
IF	112, D-13
INKEY	113, D-13
INKEY\$	115, D-13
INPUT	116, D-13
INPUT LINE	118
INPUT#	119, 234, D-14
INSTR	120, D-14
INT	121, D-14
Indexed Files	219, 275
Indexed Files - Address Book Program	275
Indirection	17
Indirection Operators - Power of	22
Integer Variables	9
Integer Variables - Storage	E-10
Keywords	25
LEFT\$	122, D-14
LEN	123, D-14
LET	124, D-15
LINE Space	C-7
LIST	125, D-15
LISTO	126, D-15
LN	128, D-15
LOAD	129, 225, D-15
LOCAL	130, D-16
LOG	131, D-16
LOMEM	132, D-16, E-3
Labels - Assembler	42
Leaving Program Loops	14
Limiting Number of Variables	E-4
Line Numbers	6
Listing Controls - Assembler	44
Loading Machine Code Programs	52
Local (Private) Variables	16, 38
Log Range	C-7
Logical Operations	61, 97, 112, 139, 147
Loop Operation Errors	13
Loop Variable - Changing	15
MERGE	227
MID\$	133, D-16
MOD	134, D-16

MODE	135, D-17
MOVE	136, D-17
Macros - Assembler	49
Mathematical Functions	Annex B
Memory	Annex E
- Fixed Strings - Storage	E-16
- Integer Variables - Storage	E-10
- Program Storage	E-8
- Real Variables - Storage	E-11
- String Variables - Storage	E-15
- Allocation for String Variables	11, E-5
- Area Reserved for Assembler	48
- Management	E-4
- Management Rules	E-7
- Map	E-1
- Requirement - Files	E-5
- Usage - File Management	E-5
Missing "	C-8
Missing #	C-8
Missing)	C-8
Missing ,	C-7
Mistake	C-8
Mixed Data Files - Reading	248
Mixed Data Files - Writing	245
Mixed Data Files - Writing 'At End'	249
Mixed Numeric/Character Data Files	245
Mnemonics - Assembler	41
NEW	137, D-17
NEXT	138, D-17
NOT	139, D-17
Negative Root	C-8
No FN	C-9
No FOR	C-9
No GOSUB	C-8
No PROC	C-9
No REPEAT	C-9
No Room	C-9
No Such FN/PROC	C-9
No Such Line	C-10
No Such Variable	C-10
No TO	C-10
Not LOCAL	C-10
Numeric Arrays	12
Numeric Data - How Stored in Files	221
OLD	140, D-17
ON	141, D-18
ON ERROR	142, D-18
ON Range	C-11
ON Syntax	C-11

OPENIN	143, 233, D-18
OPENOUT	144, 232, D-18
OPENUP	145, 233, D-19
OPT	146, 45, D-19
OR	147, D-19
OSCLI	148, D-19
Opening Files	231
Operating System Commands	207
- Case Conversion	207
- General	209
- Special Chars	207
- Syntax	207
- Torch	213
Operators and Special Symbols	23
Optional Assembly	49
Organization of Examples - Disk Files	224
Out of DATA	C-11
Out of Range	C-11
Overcoming the Limitations of CP/M	223
PAGE	149, D-19, E-3
PI	150, D-20
PLOT	151, D-20
POINT	155, D-20
POS	156, D-20
PRINT	157, D-20
PRINT#	165, 235, D-21
PROC	166, 34, D-21
PTR#	167, 238, D-21
PUT	168, D-21
Private (Local) Variables	16, 38
Procedure Names	34
Procedures and Functions	33
Procedures and Functions - Defining	35
Procedures and Functions - Parameters	36
Program - Storage in Memory	E-8
Program Counter	43
Program File Manipulation	224
Program Flow Control	13
Program Loops - Leaving	14
Program Structure Limitations	13
Query	18
RAD	169, D-22
READ	170, D-22
REM	171, D-22
RENUMBER	172, D-22
REPEAT	173, D-22
REPEAT...UNTIL Forever	103, 201
REPEAT...UNTIL Loops - Leaving	14

REPORT	174, D-23
RESTORE	175, D-23
RETURN	176, D-23
RIGHT\$	177, D-23
RND	178, D-23
RUN	179, D-24
Random (Relative) Files	257
Random Access Database - Simple	258
Random Access File - Simple	257
Random Access File Initialisation	262
Random Access Files	218
Random Access Inventory	262
Random Files - Deficiencies	275
Reading a Screen Character	53
Real-time Clock - Setting	Program SETTIME.BBC
Real Variables	9
Real Variables - Storage	E-11
Relative (Random) Files	257
SAVE	180, 225, D-24
SGN	181, D-24
SIN	182, D-24
SOUND	183, D-24
SPC	187, D-25
SQR	188, D-25
STEP	189, D-25
STOP	190, D-25
STR\$	191, D-25
STRING\$	192, D-26
Saving Machine Code Programs	52
Saving and Loading Machine Code Programs	52
Screen - Reading a Character	53
Sequential (Serial) Files	217, 241
Serial (Sequential) Files	217, 241
Serial Character Data - Reading	243
Serial Character Data - Writing	242
Serial Character Data - Writing At End	243
Serial Files	241
Silly	C-11
Special (Static) Variables	9
Special Characters - OS Commands	207
Specification	9
Stack Pointer Control	16
Statement Separators	7
Statements - Assembler	41
Statements - BASIC	55
Statements and Functions - BBCBASIC(Z80)	55
String Constants - Assembler	45
String Management	E-4
String Too Long	C-11
String Variables - Storage	E-15

String Variables and Garbage	11
Strings	10
Strings - How Stored in Files	222
Structure of Files	217
Subscript	C-12
Syntax	Annex D
Syntax - Operating System Commands	207
Syntax Error	C-12
TAB	193, D-26
TAN	194, D-26
THEN	195, D-27
TIME	196, D-27
TIMES	196
TIMES - Setting	Program SETTIME.BBC
TO	197, D-27
TOP	198, D-27, E-3
TRACE	199, D-28
TRUE	200, 6, D-28
Too Big	C-12
Too Many Open Files	C-12
Trappable Errors	C-1
Type Mismatch	C-12
Types Allowed	9
UNLIST	230
UNTIL	201, D-28
USR	202, D-28
Untrappable Errors	C-1
VAL	203, D-28
VDU	204, D-29
VPOS	205, D-29
Variables	9
Variables - Storage in Memory	E-9
WIDTH	206, D-29
Word Constants - Assembler	45

DATA For CHARACTERS

127 = 255, 255, 255, 255, 255, 255, 255, 255

42 = 0, 16, 84, 56, 16, 56, 84, 16