Richard. T. Russell & Douglas J. Mounter

The BBC BASIC(Z80) Reference Manual for Cambridge Z88

Second edition

Credits

Our thanks to Richard Russell, Matthew Soar and Cambridge Computer Ltd for providing technical information and help in preparing this manual.

Preface

You are reading the second edition of this manual. It happened because Richard Russell decided to release the source code of the BBC BASIC(Z80) for CP/M in June 2019 and a few months later also the source code of the Z88 Patch under the same Zlib license.

It allowed us, the team of cambridgez88.jira.com, to finally release the source code of BBC BASIC(Z80) port for the Cambridge Z88 that also contained the integration of the Z88 Patch, released to the project in 2006 by Richard Russell as a “ROMable” binary. Previously, the patch was only available as a binary add-on program for OZ releases V2.2 UK - V4.0 UK. The integrated patch allowed OZ V4.3 and newer releases to enjoy the improved BBC BASIC application. Respecting copyright, we preserved the reverse-engineered and optimised version source code privately. This source code, however, lacked all the deeper understanding and comments of the original work.

All original source code labels and comments from the ZLib licensed files have now been applied to our reverse-engineered and optimized version of BBC BASIC(Z80) for Cambridge Z88.

You can find the source code here: https://bitbucket.org/cambridge/oz/src/develop/apps/bbcbasic/

The original Z88 User Guide contained introductory information about the BBC BASIC(Z80) V3.0 available for the Cambridge Z88. Many years have passed since the first edition of this manual was released by M-TEC, It is no longer available in print or purchase. The Z88 Patch contained an introduction readme text file, referencing the added star commands.

We decided to do the effort and preserve the legacy of BBC BASIC for Z88, by making this guide available with ALL the updated information from the patch and new improvements of OZ. This guide is based on mainly the online CP/M Z80 guide also using the extended random files examples, typed in from the original contents of the M-TEC first edition. It was then improved, spell-checked, clearer layout, improved Z88-specific descriptions (ie. screenshot for clarity) and added with the Z88 Patch command reference. All the original text from the M-TEC first edition has been checked against this second edition. It is targeted to be the complementary guide for BBC BASIC(Z80) on Cambridge Z88 ROM V4.7 and later releases.

Screenshots, code examples and Z88 Patch have been validated on OZvm - the Cambridge Z88 emulator.

On behalf of the cambridgez88.jira.com team, thank you Richard, for the generous decision to make the source code of BBC BASIC(Z80) and Z88 Patch available for everybody.

Gunther Strube, Vic Gerhardi and Jamie Bradbury
October 2019 - May 2020

Table of Contents

Introduction

Before You Start

This is a BBC BASIC reference manual, it is not intended to teach you BASIC nor tell you how the Z88 computer works. It gives a summary of the BBC BASIC(Z80)'s 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 BBC BASIC. A general knowledge of BASIC has been assumed.

File handling with BBC BASIC(Z80) is more flexible than on the BBC Micro and different in approach to most other versions of BASIC. Because of this, BBC BASIC(Z80) file handling has been covered in some detail.

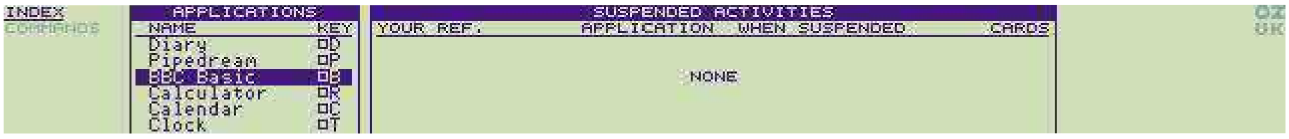
Please read your Z88's documentation before you try to use it seriously. We have included some (very) basic hints for beginners later in this section and these should be sufficient to get you going. However, if you are going to make the best use of your Z88, you need to understand what you are doing. Understanding does not come easy, but if you study the documentation and try things out for yourself you will be well rewarded.

Apart from sound, all the statements and functions specified for BBC BASIC are available. BBC BASIC(Z80) has been designed to be as compatible as possible with version 4 of the 6502 BBC BASIC resident in the BBC Micro Master series. The language syntax is not always completely identical to that of the 6502 version, but in most cases the Z80 version is more tolerant.

Running BBC BASIC(Z80)

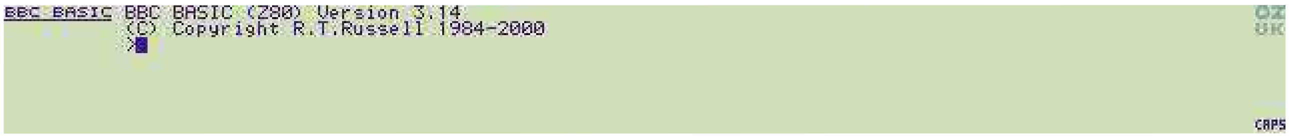
To run BBC BASIC(Z80) for the first time:

- Make sure the INDEX is displayed on your Z88. If necessary, press the `[INDEX]` key
- Use the cursor keys to select 'BBC BASIC' from the application list
- Press the `[ENTER]` key



```
INDEX     APPLICATIONS     SUSPENDED ACTIVITIES                                    OZ
COMMANDS  NAME        KEY  YOUR REF.  APPLICATION  WHEN SUSPENDED        CARDS     UK
          Diary       □D
          Pipedream   □P
          BBC Basic   □B                          NONE
          Calculator  □R
          Calendar    □C
          Clock       □T
```

After a moment, the BBC BASIC application will have started and the interpreter will reply:



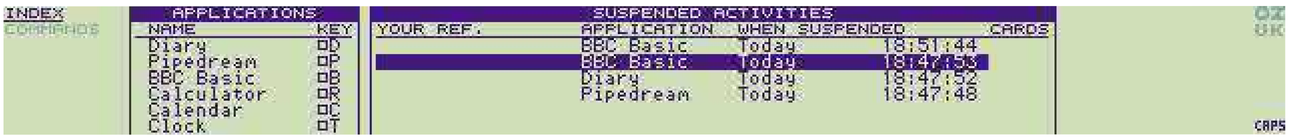
```
BBC BASIC BBC BASIC (Z80) Version 3.14                                      OZ
          (C) Copyright R.T.Russell 1984-2000                               UK
          >

                                                                            CAPS
```

You can now start to use BBC BASIC(Z80).

If you have previously run BBC BASIC(Z80), you can return to it by using the cursor keys to select BASIC from the list of suspended activities list in the INDEX. BBC BASIC(Z80) will appear exactly as you left it.

If you wish, you can have several instantiations (sessions or editions) of BBC BASIC(Z80). Each instantiation will appear in the suspended activities list and you may use the cursor control keys to select the appropriate instantiation.



```
INDEX     APPLICATIONS     SUSPENDED ACTIVITIES                                    OZ
COMMANDS  NAME        KEY  YOUR REF.  APPLICATION  WHEN SUSPENDED        CARDS     UK
          Diary       □D              BBC Basic    Today        18:51:44
          Pipedream   □P              BBC Basic    Today        18:47:53
          BBC Basic   □B              Diary        Today        18:47:52
          Calculator  □R              Pipedream    Today        18:47:48
          Calendar    □C
          Clock       □T                                                           CAPS
```

The easiest way to return to a suspended instantiation of BBC BASIC(Z80) by pressing □B. If you have more than one suspended instantiation of BBC BASIC(Z80), pressing □B will return you to the earliest suspended instantiation. Pressing □B a second time will take you to the next earliest, and so on.

Within the space available, we can't go into details of what an operating system is, how programs run, etc. Your Z88 user guide gives an explanation of how the Z88 works, what commands are available, how to run programs, etc. - please read it.

General information

Introduction

Control Codes and Functions

Generation

Control codes are generated by pressing the ◊ key followed by another key. You don't need to hold the ◊ key down. Thus 'control' A is generated by pressing the ◊ followed by the 'A' key. 'Control A' is shown as ◊A, etc.

What happens to Control Codes?

In the immediate mode on the BBC Micro, all control codes (with the exception of `[ESC]` and ◊U) are echoed to the micro's VDU software where they initiate the same functions as if they had been sent by the VDU statement. This does not happen on the Z88. In the immediate mode and in the response to an INPUT statement, some control codes have a special significance to the input line editor (see later); with the exception of `[ESC]`, the others are ignored. All control codes and special function key codes are available to GET and GET$.
Pressing the `[ESC]` key aborts a BBC BASIC(Z80) program or command.

Pausing the Display

Holding down the ◊ and `[SHIFT]` will pause the output to the screen.

Line Numbers

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

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

For example, the following will work.

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

Editing

Introduction

The single line editor is active in the immediate mode, and in response to an INPUT statement in a program and during the *EDIT command.
The following editing keys are available when the single line editor is active:

`[DEL]`	Backspace and Delete
⇦	Cursor Left
⇨	Cursor Right
◊`[DEL]`	Delete line
◊D	Delete to End of Line
◊G	Delete Character
◊M	[ENTER]
◊S	Swap Case
◊T	Delete Word
◊U	Insert Character
◊V	Insert/Overtype
◊ ⇦	Start of Line
◊ ⇨	End of Line
`[SHIFT]`⇦	Previous Word
`[SHIFT]`⇨	Next Word
`[SHIFT][DEL]`	Delete Character (same as ◊G)

The Program Editor

As described in the Z88 manual - section 7, you can use PipeDream to edit a BBC BASIC(Z80) program. This method is, however, a little long-winded if you want to edit a couple of lines.

No Program Editor available in Cambridge Z88 ROM releases OZ V2.2 - V4.0

The single editor is not available to BBC BASIC(Z80) when your Cambridge Z88 is shipped with ROM releases V2.2 - V4.0. Richard Russell supplied a Z88 Patch program for those ROM releases that would supply the ***EDIT** command. If you have not installed the patch program, it is possible to include a line-editing procedure in your BBC BASIC(Z80) programs.

The following program segment, which was developed by Cambridge Computer Ltd, provides the facilities of a line editor:

```
60000 END
60010 DEF PROCE(B)
60020 REM Cambridge Computer Ltd.
60030 IF B=0 THEN ENDPROC
60040 A=OPENOUT":RAM.0/EE.CLI"
60050 B$=":RAM.0/E.CLI"
60060 PRINT#A,".>"+B$
60070 PRINT#A,".J","LIST"+STR$(B),"PROCF"
60080 CLOSE#A
60090 *CLI .*:RAM.O/EE.CLI
60100 ENDPROC
60110 DEF PROCF
60120 A=INKEY(0)
60130 A=OPENIN B$
60140 INPUT#A,A$,A$
60150 CLOSE#A
60160 A=OPENOUT B$
60170 PRINT#A,".J",A$
60180 PTR#A=PTR#A-1
60190 BPUT#A,0
60200 CLOSE#A
60210 VDU 8
60220 OSCLI"*CLI .<"+B$
60230 ENDPROC
```

Once you have added the segment to your program, you can use it to edit program line ‘nnnn’ with the command:

```
    PROCE(nnnn)
```

The editing functions previously described are available when the line editing procedure is used. To save entering the program segment for every program you write, create a file containing the program segment using PipeDream and save it in plain text mode as EDBAS. In this case, add **.J** as the first line of the file. The start of the program segment would then look like this:

```
.J
60000 END
60010 DEF PROCE(B)
60020 REM Cambridge Computer Ltd.
60030 IF B=0 THEN ENDPROC
60040 A=OPENOUT":RAM.0/EE.CLI"
etc...
```

You can append the editor to your BBC BASIC programs with the following command:

```
*CLI .*EDBAS
```

Your program should not, of course, use line numbers 60000 and up. If it does, they will be overwritten when you load the line editor functionality.

To abort the single line editor when using PROCE(nnnn) and leave the line unchanged, press `[ESC]`.

The editor generates two working files in :RAM.0 called /EE.CLI and /E.CLI. The files can be erased after use. If you have a RAM card in slot 1 or slot 2 you could alter the program to save its working files in :RAM.1 or :RAM.2 by changing lines: 60040, 60050, and 60090.

Using PipeDream as Program Editor

As illustrated above, you can use PipeDream to write your BBC BASIC(Z80) programs. You should save the program files in plain text format (you will get some interesting error messages if you don't). Don't forget to include the CLI jammer command **.J** as the first line of the file. For complete (new) programs, you should include the BBC BASIC(Z80) command NEW as the second line of the file.
This will make sure that any program in memory will be deleted before the new one is typed in.
The first two lines of your PipeDream file will then look like this:

```
.J
NEW
```

If you saved (as plain text) the program to a file called BASPROG, you should load (type) it into BBC BASIC(Z80) with the following command:

```
*CLI .*BASPROG
```

Auto Numbering PipeDream files

If you use PipeDream to write your programs, you can arrange for the lines to be automatically numbered by BBC BASIC(Z80) as the file is 'loaded'. All you need to do is include the AUTO command as the third line of your program file. The first 3 lines of your PipeDream program file would now look like this:

```
.J
NEW
AUTO
```

Installing Z88 Patch with Program Editor and other commands

The version of BBC BASIC resident in the Cambridge Computer Z88 ROM releases V2.2 - V4.0 is deficient in a few respects, especially in the lack of any editing facilities or support for graphics operations. The purpose of the BASIC Patch program is to provide some of these capabilities.

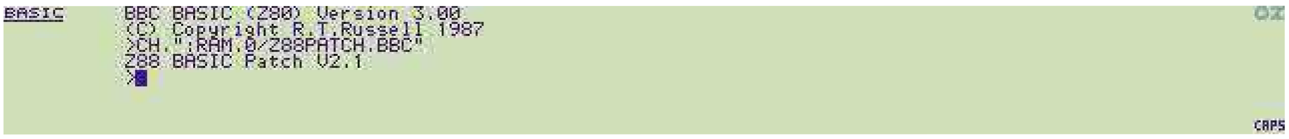
The patch provides *EDIT (Program Editor), MODE, CLG, DRAW, MOVE, PLOT commands and POINT() function; their syntax is compatible with other versions of BBC BASIC. Their syntax and functionality is described in the command reference section later in this manual.

The patch needs an expanded machine (at least 128 Kbytes of RAM installed in slot 1). If this is not the case, or if the available memory has been deliberately reduced by changing HIMEM, the message "No RAM" will be displayed when the program is CHAINed. The patch only works with Operating System ("OZ") versions V2.2-V3.0 and V4.0 (it will not work with foreign language versions V3.12 to V3.26 ROM's, because of a software bug in these machines).

The patch program is installed with a simple CHAIN command, and thereafter remains resident until the machine is reset or BASIC is KILLed. It occupies two kilobytes of memory, and results in the value of PAGE being raised to &2B00; when the graphics operations are used a further 2K is used for the display buffer.

Download the Z88 Patch ZIP file from R.T. Russell's BBC BASIC website here: http://www.bbcbasic.co.uk/bbcbasic/z88patch.html, then follow these steps:

- Extract Zip file contents and upload the Z88PATCH.BBC program to your Z88, for example to :RAM.0.
- Instantiate or activate a BBC BASIC(Z80) application with □B
- Enter the following command:
  ```
  CH.":RAM.0/Z88PATCH.BBC"
  ```
- The patch will be installed in the BBC BASIC(Z80) application



```
BASIC     BBC BASIC (Z80) Version 3.00
          (C) Copyright R.T.Russell 1987
          >CH.":RAM.0/Z88PATCH.BBC"
          Z88 BASIC Patch V2.1
          >
```

Installation of the BASIC Patch has a number of "side effects" of which you should be aware:

1. Changing HIMEM will have the effect of disabling the patch. If HIMEM has been changed it must be set back to &C000 before re-CHAINing the patch program (or alternatively KILL and restart BASIC from the INDEX). Changing HIMEM is not recommended in any case, since setting it to an unsuitable value will "crash" the machine (even without the patch).
2. The "Silly", "RENUMBER space" and "LINE space" errors will not appear; instead the "No room" message will be produced in each case.
3. You are advised to select MODE 0 before entering Pipedream, since it seems to get confused by the presence of the graphics window.

4. Since Cambridge Computer provided no "legal" method of installing a patch such as this, a rather "dirty" method has had to be adopted. This has some unfortunate, but unavoidable, consequences:
    a. The RUN and CHAIN commands may occasionally fail to work properly (on average fewer than 1 in 1000 times). If this happens no harm will be done; simply issue the command again.
    b. If you reply to the INPUT statement with a very long string (more than 252 characters) the machine will crash, so you must avoid doing so.
5. Using graphics statements in an ON ERROR routine may give anomalous results. For example:

```
10 ON ERROR MODE 0 : REPORT : END
20 MODE 1
30 REPEAT
40   DRAW RND(256)-1,RND(64)-1
50 UNTIL FALSE
```

   The above program can be aborted only by pressing `[ESC]`. The intention is that this will cause the display to clear and the message "Escape" to be displayed. In practice, the message actually displayed will be "Sorry, not implemented" since, although the Patch is active, the MODE statement still affects REPORT, ERR and ERL.

The Patch commands are integrated in OZ V4.3 and later releases, without the above mentioned “side effects”. PAGE address is not affected (remains at &2300). Later Z88 ROM versions, provided by the Open Source Z88 Development Project, can be accessed here (contains release notes and download links):

https://cambridgez88.jira.com/wiki/spaces/OZ/

Expression Priority

Order of Evaluation

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

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

Examples

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

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

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

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

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

Variables

Specification

Variable names may be of unlimited length and all characters are significant. Variable names must start with a letter. 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.

The following types of variable are allowed:

A	real numeric
A%	integer numeric
A$	string

Numeric Variables

Real Variables

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

Integer Variables

Integer variables are stored in 32 bits and have a range of +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.

Static Variables

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

Boolean Variables

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

```
PRINT NOT 1.5
    -2
```

The argument, 1.5, is truncated to 1 and the logical inversion of this gives -2

```
PRINT NOT -1.5
     0
```

The argument is truncated to -1 and the logical inversion of this gives 0

Two numeric functions, TRUE and FALSE, are provided. TRUE returns the value -1 and FALSE the value 0. These values allow the logical operators (NOT, AND, EOR and OR) to work properly.

However, anything which is non-zero is considered to be TRUE. This can give rise to confusion, since +1 is considered to be TRUE and NOT(+1) is -2, which is also considered to be TRUE.

Numeric Accuracy

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

String Variables and Garbage

Strings

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

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. BBC BASIC does not incorporate 'garbage collection' routines and it is possible to run out of room for variables even though there should be enough space.

Memory Allocation

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.

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 multidimensional array; you have to use one array for each type of variable you need.

Program Flow Control

Introduction

Whenever the BBC BASIC(Z80) comes across a FOR, REPEAT, GOSUB, FN or PROC statement, it needs to remember where it is in the program so that it can loop back or return there when it encounters a line with NEXT, UNTIL or RETURN statement or when it reaches the end of a function or procedure. These 'return addresses' tell BBC BASIC(Z80) where it is in the structure of your program.

Every time the BBC BASIC(Z80) encounters a FOR, REPEAT, GOSUB, FN or PROC statement it 'pushes' the return address on to a 'stack' and every time it encounters a NEXT, UNTIL, RETURN statement or the end of a function or procedure it 'pops' the latest return address of the stack and goes back there.

Unlike the BBC Micro, which has separate stacks for FOR...NEXT, REPEAT...UNTIL GOSUB...RETURN and FN/PROC operations, BBC BASIC(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.

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
```

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, BBC BASIC(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
```

BBC BASIC(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 on the BBC Micro when the control stack overflows.

Leaving Program Loops

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

REPEAT...UNTIL Loops

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
```

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_dem
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
```

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 BBC BASIC(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
```

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 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
```

Stack Pointer

The program 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.

Indirection

Introduction

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 BBC BASIC(Z80) enable you to read and write to memory in a far more flexible way. They provide a simple equivalent of PEEK and POKE, but they come into their own when used to pass data between CHAINed programs, build complicated data structures or for use with machine code programs.

The indirection operators

There are three indirection operators:

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

The examples that follow assume that a DIM statement has been used to reserve an area of memory and store the address of the first byte of the memory in a variable called ‘mem’. See the keyword DIM for more details. Or example:

```
DIM mem 20
```

If you know what you are doing, you can use the indirection operators to access the Z88’s system memory. However, because of the sophistication of the operating system, **THIS CAN BE DISASTROUS**.

Query

Byte Access

The query operator accesses individual bytes of memory. ?M means 'the contents of' memory location 'M'. The first example below write &23 to memory location `mem`, the second example sets ‘number’ to the contents of that memory location and the third example print the contents of that memory location.

```
?mem=&23
number=?mem
PRINT ?mem
```

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

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

Query as a Byte Variable

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

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

Exclamation

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
&12 into address M+3.

and

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

would give

12345678

Dollar

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

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

```
DIM memory 20
memory?5=&50
PRINT memory?5
```

Thus,

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

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

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

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

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

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

Power of Indirection Operators

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

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

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

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

Operators and Special Symbols

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

?	A unary and 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" 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 new lines in PRINT or INPUT.
( )	Objects in parentheses have the 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. Also used with STR$ to cause conversion to a hexadecimal string.

Keywords

Keywords are recognized before anything else. (For example, DEG and ASN in DEGASN are recognized, but neither is recognized in ADEGASN.) Consequently, you don't have to type a space between a keyword and a variable (but it does make it easier to read your program).

Although they are keywords, the names of pseudo variables such as PI, LOMEM, HIMEM, PAGE, TIME, etc, act as variables in that their names can form the first part of the name of another variable. For example, if A is a variable, then AB can also be a variable. Similarly, the name PI is not recognized in the name PILE; they are both unique variable names. 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.

Ninety-three out of the total of 123 keywords are not allowed in upper case at the start of a variable name (anything may be used in lower case). Those keywords that are allowed are shown in bold type.

Sound and colour commands are not available on the Z88. The commands are shown in small italics. If you use one of these commands, a “Sorry, not implemented” error will be reported.

Graphics commands are available via installation of external Z88 Patch for ROM releases V2.2 - V4.0. The Z88 Patch is integrated by default in ROM releases V4.3 and later.

If the Z88 Patch is not installed and you use one of the graphics commands, a “Sorry, not implemented” error will be reported.

Since the keywords must be in upper case, you may wish to use the `[CAPS LOCK]` key to lock the letter keys to uppercase (the default setting for BBC BASIC(Z80)). Alternatively, you can invert the action of the [SHIFT] key so that unshifted letters are in upper case and shifted letters are in lower case.

`[CAPS LOCK]`	Caps lock off/on
`□[CAPS LOCK]`	Invert shift action (`[SHIFT]` for lower case)
`◊[CAPS LOCK]`	Normal shift action (`[SHIFT]` for upper case)

Keywords Available

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

Error Handling

Introduction

Types of Errors

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 two kinds of errors which you could still encounter; errors of logic and run-time errors. Errors of logic are where BBC BASIC(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 BBC BASIC(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'. BBC BASIC(Z80) has no way of dealing with 'infinity' and it will report a 'Division by zero' error.

Trapping Errors

There is no way that BBC BASIC(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.

Reasons for Trapping Errors

Why would you want to take over responsibility for handling run-time errors? When BBC BASIC(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 BBC BASIC(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 BBC BASIC(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  Loop 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
```

Error Trapping Commands

The **ON ERROR** command directs BBC BASIC(Z80) to execute the statement(s) following ON ERROR when a trappable error occurs:

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

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

As explained in the Program Flow Control sub-section, every time BBC BASIC(Z80) encounters a FOR, REPEAT, GOSUB, FN or PROC statement it 'pushes' the return address on to a 'stack' and every time it encounters a NEXT, UNTIL, RETURN statement or the end of a function or procedure it 'pops' the latest return address of the stack and goes back there. The program stack is where BBC BASIC(Z80) records where it is within the structure of your program.

When an error is detected by BBC BASIC(Z80), the stack is cleared. Thus, you cannot just take any necessary action depending on the error and return to where you were because BBC BASIC(Z80) no longer knows where you were.

If an error occurs 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.

Error Reporting

There are two 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.

ERR

ERR returns the error number (see the Annex entitled Error Messages and Codes).

ERL

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

REPORT

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

Problems with Error Trapping

If there is an error in your ON ERROR statement, BBC BASIC(Z80) will go into an infinite loop. [ESC] (which generates error code 27) will not break you out of this loop. On most (single tasking) computers like the BBC Micro, you can escape from such a loop by pressing <BREAK> or turning the computer off.
Because the Z88 is a sophisticated computer which is capable of holding several programs in memory concurrently, this option has complications.
If there is an error in the error handling part of your program, you will end up in a ‘doom loop’. Unless you take special precautions, you will need to perform a soft or hard reset to escape from this situation. A hard reset will cause all the programs and data in RAM to be lost and a soft reset may possibly leave the computer in an unstable condition which will lead to an eventual software failure.

To overcome this problem, you should **ALWAYS** include a line similar to

```
dummy = INKEY(0)
```

as the **FIRST** line of any error handling routine. If you do so, you will be able to escape to the INDEX from a ‘doom loop’ by pressing the `[INDEX]` key. If you try to return to your BBC BASIC(Z80) program, you will find yourself in the same situation as when you left it. All you can do is to `◊KILL` the offending BBC BASIC(Z80) instantiation. You will lose your program, but everything else will be intact. For example:

```
10 ON ERROR GOTO 1000
20 FOR i= 1 TO 20
30 etc…

1000 dummy = INKEY(0)
1010 PRINT “Error!”
1020 etc..
```

Error Trapping Examples

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

```
  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 dummy = INKEY(0)
110 PRINT
120 PRINT "Error No ";ERR
130 REPORT:PRINT " at line ";ERL
140 END
```

Example run:

RUN
Type a number **1**
        1
Type a number **-2**
       -2
Error No 21
-ve root at line 30
RUN
Type a number **[Esc]**
Error No 17
Escape at line 20

The example below has been further expanded to include error trapping. The only 'predictable' error is that the user will try a negative number. Any other error is unacceptable, so it is reported and the program aborted. Consequently, when `[ESC]` is used to abort the program, it is reported as an error. However, a further test for ERR=17 could be included so that the program would halt on ESCAPE without an error being reported.

```
  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 dummy = INKEY(0)
110 PRINT
120 IF ERR=21 THEN PRINT "No negatives":GOTO 10
130 REPORT:PRINT " at line ";ERL
140 END
```

RUN
Type a number **5**
    5   2.23606798
Type a number **2**
    2   1.41421356
Type a number **-1**
   -1
No negatives
Type a number **4**
    4   2
Type a number **[Esc]**
Escape at line 20

The above example is very simple and was chosen for clarity. In practice, it would be better to test for a negative number before using SQR rather than trap the '-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 dummy = INKEY(0)
110 PRINT
120 IF ERR=28 THEN PRINT "Not hex":GOTO 20
130 REPORT:PRINT " at line ";ERL
140 END
```

Procedures and Functions

Introduction

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

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

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

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

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

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

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

Whilst BBC BASIC(Z80) has a large number of predefined functions (INT and LEN for example) it is very useful to be able to define your own to do something special. Suppose you had written a function called FN_discount to calculate the discount price from the normal retail price. You could write something similar to the following example anywhere in your program where you wished this calculation to be carried out.

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

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

Names

The names of procedures and functions MUST start with PROC or FN and, like variable names, they cannot contain spaces. (A space tells BBC BASIC(Z80) that it has reached the end of the word.) This restriction can give rise to some pretty unreadable names. However, the underline character can be used to advantage. Consider the procedure and function names below and decide which is 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.

Functions and Procedure Definitions

Starting a Definition

Functions and procedure definitions are 'signalled' to BBC BASIC(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.

The Function/Procedure Body

The 'body' of a procedure or function must not be executed directly - it must be performed (called) by another part of the program. Since BBC BASIC(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.

Ending a Definition

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

Single Line Functions/Procedures

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

```
500 DEF FN_average(n1,n2)=(n1+n2)/2
120 DEF PROC_clear:PRINT SPC(40-POS);:ENDPROC
```

Extending the Language

You can define a whole library of procedures and functions and include them in your programs. By doing this you can effectively extend the scope of the language. For instance, BBC BASIC(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 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 a 94 by 8 display. In order to prevent the display from scrolling, you must not write to the last column of the last row. The three 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<7 FOR i=y TO 6:PRINT SPC(94);:NEXT
160 REM Print spaces to end-1 of last line.
170 PRINT SPC(93-x);
180 PRINT TAB(x,y);
190 ENDPROC
```

Passing Parameters

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

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

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

Formal and Actual Parameters

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.

Correct

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

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
```

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 Variables

You can use the statement LOCAL to define variables which are only known locally to individual procedures and functions. In addition, formal parameters are local 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 two 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 this manual.

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

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.

Re-entrant Functions/Procedures

Because the formal parameters which receive the passed parameters are local, 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 *....* 1. Factorial 6, for instance, is 6*5*4*3*2*1).

```
 10 REPEAT
 20   INPUT "Enter an INTEGER 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
100 REM Return with 1 if n= 0 or 1
110 =n*FN_fact_num(n-1)
120 REM Else go round again
```

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

Assembler

Introduction

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

This section illustrates the way the BBC BASIC(Z80) assembler functions; it does not provide sufficient information to enable you to write assembler programs that interface with the Z88 operating system or hardware. In order to successfully write assembler language programs for the Z88 you will need a considerable amount of technical information about the machine. This information is available in the Z88 Developers' Notes, available on the **cambridgez88.jira.com wiki**.

The Z88 is a sophisticated computer which is capable of holding several programs in memory concurrently. This makes incorrectly written assembler language programs potentially dangerous. Whilst the Z88's operating system is quite robust, you could cause the Z88 to enter an undefined state if your program does not correctly interface with it. If this happens, you may corrupt all the data and programs in RAM and you will need to perform a soft or hard reset. A hard reset will cause all the programs and data in RAM to be lost and there is a possibility that a soft reset may leave the computer in an unstable condition which will lead to an eventual software failure.
If you wish to develop assembler language programs, we suggest you do so on a computer which does not does not hold programs or data that you cannot afford to lose. It would also be wise to download your program to another computer or an EPROM before testing it. If you do not take these precautions, you may eventually lose some irreplaceable programs or data.
We recommend using the Z88 emulator, OZvm, for assembler language application testing, which also includes debugging tools such as instruction single stepping and breakpoints, memory viewing and editing.

Instruction mnemonics

All standard Zilog mnemonics are accepted: 'ADD', 'ADC' and 'SBC', must be followed by 'A' or 'HL'. For example
`ADD A,C`
is accepted but
`ADD C`
is not. However, the brackets around the port address in 'IN' and 'OUT' are optional; thus both
`OUT (5),A` and **`OUT 5,A`**
are accepted. The instruction `'IN F,(C)'` is **not** accepted but the equivalent object code is produced from '`IN (HL),(C)`'.
The pseudo-operations 'DEFB', 'DEFW' and 'DEFM' are included. 'DEFM' is like 'EQUS' in the 6502 version.

Assembler Statements

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

Labels

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

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

Comments

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

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

Byte, Word and String Constants

You can store constants within your assembly language program using the define byte (DEFB), define word (DEFW) and define message (DEFM) pseudo-operation commands. These will create 1 byte, 2 byte and 'string' items respectively.

Define Byte - DEFB

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

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

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

Define Word - DEFW

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

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

will have the same result as the Byte Constant example.

String Constant - DEFM

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

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

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

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

Defined storage

Unfortunately, there is no ‘define storage’ directive. A second DIM statement may be used, or, for small amounts of storage, you can use DEFM with a dummy string. For example:

```
DEFM STRING$(100, “ “)
```

will reserve 100 bytes of storage.

Reserving Memory

The Program Counter

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

Using DIM to Reserve Memory

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

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

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

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

Moving HIMEM to Reserve Memory

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

Length of Reserved Memory

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

In the example below, a large amount of memory is initially reserved. To begin with, a single pass is made through the assembly code and the length needed for the code is calculated (lines 100 to

120). After a CLEAR, the correct amount of memory is reserved (line 140) and a further two passes of the assembly code are performed as usual. Your program should not, of course, subsequently try to use variables set before the clear statement. If you use a similar structure to the example and place the program lines which initiate the assembly function at the start of your program, you can place your assembly code anywhere you like and still avoid this problem.

```
100 DIM free -1, code HIMEM-free-1000
110 PROC_ass(0)
120 L%=P%-code
130 CLEAR
140 DIM code L%
150 PROC_ass(0)
160 PROC_ass(2)
- - -
Put the rest of your program here.
- - -
1000 DEF PROC_ass(opt)
10010 P%=code
10020 [OPT opt

- - -
Assembler code program.
- - -

11000 ]
11010 ENDPROC
```

Initial Setting of the Program Counter

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

The Assembly Process

OPT

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

Bit 0 - LSB

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

Bit 1

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

Bit 2

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

Assembly at a Different Address

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

OPT Summary

Code Assembled Starting at P%

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

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

Code Assembled Starting at O%

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

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

How the Assembler Works

The assembler works line by line through the machine code. When it finds a label declared it generates a BBC BASIC(Z80) variable with that name and loads it with the current value of the program counter (P%). This is fine all the while labels are declared before they are used. However, labels are often used for forward jumps and no variable with that name would exist when it was first encountered. When this happens, a 'No such variable' error occurs. If error reporting has not been disabled, this error is reported and BBC BASIC(Z80) returns to the direct mode in the normal way. If error reporting has been disabled (OPT 0, 1, 4 or 5), the current value of the program counter is used in place of the address which would have been found in the variable, and assembly continues. By the end of the assembly process the variable will exist (assuming the code is correct), but this is of little use since the assembler cannot 'back track' and correct the errors. However, if a second pass is made through the assembly code, all the labels will exist as variables and errors will not occur. The example below shows the result of two passes through a (completely futile) demonstration program. Twelve bytes of memory are reserved for the program. (If the program was run, it would 'doom-loop' from line 50 to 70 and back again.) The program disables error reporting by using OPT 1.

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

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

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

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

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

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

Saving and Loading Machine Code Programs

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

*SAVE

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

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

*LOAD

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

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

*LOAD ‟:RAM.0/WOMBAT” &8F00
OSCLI "LOAD "+f_name$+" "+STR$~(strt_address)
```

Saving and loading machine code files with PROC_save & PROC_load

*LOAD and *SAVE commands are only available with Z88 Patch, either installed manually with CHAIN command or when you are using OZ ROM V4.3 or later. See section "Installing Z88 Patch with Program Editor and other commands" on how to install the Z88 Patch.

Use the following two procedures in your machine code programs, as an alternative to *LOAD and *SAVE:

```
   10 REM Procedures to substitute for *LOAD and *SAVE
   20 REM R.T.Russell, June 1988
  100 DIM fname 255
  200 END
60000 DEF PROC_load($fname,E%):LOCAL H%,L%,D%,B%,C%:L%=fname
60010 H%=L% DIV 256:D%=E% DIV 256:B%=255:C%=255:CALL &FEA3:ENDPROC
60020 DEF PROC_save($fname,E%,C%):LOCAL H%,L%,D%,B%:L%=fname
60030 H%=L% DIV 256:D%=E% DIV 256:B%=C% DIV 256:CALL &FEA6:ENDPROC
```

Using PROC_save

Save an area of memory to a file. You MUST specify the start address (aaaa) and the length of the area of memory (1111). Both examples below uses hexadecimal notation.

```
PROC_save("ufsp", &aaaa, &1111)
PROC_save("WOMBAT",&8F00,&80)
```

Using PROC_load

Load the specified file into memory at hexadecimal address 'aaaa'. The load address MUST always be specified.

```
PROC_load("ufsp",&aaaa)
PROC_load("WOMBAT",&8F00)
```

Conditional Assembly and Macros

Introduction

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

Conditional Assembly

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

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

Macros

Within any machine code program it is often necessary to repeat a section of code a number of times and this can become quite tedious. You can avoid this repetition by defining a macro which you use every time you want to include the code. The example below uses a macro to pass a character to the screen or printer. Conditional assembly is used within the macro to select either the screen or the auxiliary output, depending on the value of op_flag.

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

This is illustrated in PROC_screen and PROC_aux in the example below.

```
DIM code 200
op_flag=TRUE
FOR pass=0 TO 3 STEP 3
  [OPT pass
  .start   - - -
           - - - code - - -
           - - -
:
  OPT FN_select(op_flag); Include code depending on op_flag
:
           - - -
           - - - code - - -
           - - -:]
NEXT
END
:
:
REM Include code depending on value of op_flag
:
DEF FN_select(op_flag)
IF op_flag PROC_screen ELSE PROC_print
=pass
REM Return original value of OPT.  This is a
REM bit artificial, but necessary to insert
REM some BBC BASIC code in the assembly code.
:
DEF PROC_screen
[OPT pass AND 6
RST &20: DEFB &27 ; OS_Out: A = character to send to screen
]
ENDPROC
:
DEF PROC_print
[OPT pass AND 6
; some code to send character to the printer
]
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.

Managing system error events on the Z88

Warning

If you write an application which contains bugs or tries to circumnavigate the operating system, then it is likely that all the other applications in the Z88 will be affected.
The effect may not be immediate, some indiscretions take weeks or even months to become apparent, but will usually be in the form of a system crash. You must always remember that the resources of the Z88 are not devoted exclusively to your application and therefore only use legal interfaces.

All internal applications follow all the rules and use no 'back door' techniques.
If you wish to write more than the simplest assembler code programs for the Z88, you will need a considerable amount of technical information about the machine. This information is available in the Developers' Notes on the **cambridgez88.jira.com** wiki.

Error handling

If your assembler language program makes any use at all of the Z88's facilities, it will be possible to suspend it. Under these circumstances, it is imperative that your program includes error handling code.
The following assembler language 'wrap around' provides the minimum acceptable error handling capability. It is also an example of how to use a macro for OZ system call API and how to open files to read date stamp properties.
The 'wrap around' recognises and acts on the following events:

- An Escape condition (the `[ESC]` key having been pressed whilst escape detection is enabled).
- The `[INDEX]` having been pressed or another application / popdown being activated
- The □ key having been pressed.
- The process having been killed (normally from INDEX)

You may wish to develop your own error handler, but you should always use one. If you don't, you could run into trouble.

```
.J
AUTO

DIM code 512                  : REM space for program
GN_Esp=&4C09                  : REM return pointer to system error message
GN_Soe=&3C09                  : REM Write string at extended address to standard
output
GN_Nln=&2E09                  : REM carriage return, linefeed to std. output
GN_Sop=&3A09                  : REM output string to std. output
GN_Opf=&6009                  : REM open file
OS_Erh=&75                    : REM install error handler
OS_Esc=&6F                    : REM examine special condition
GN_Err=&4A09                  : REM standard system error box
GN_Sdo=&0E09                  : REM date and time to standard output
OS_Dor=&87                    : REM DOR interface
dr_rd=&09                     : REM read DOR record
dr_fre=&05                    : REM free DOR handle
op_dor=&06                    : REM open file for DOR access
rc_quit=&67                   : REM KILL request error code
rc_esc=&01                    : REM escape detection error code

FOR pass=0 TO 2 STEP 2
P%=code
[
OPT  pass
LD   HL,0
ADD  HL,SP                    \ get current BASIC stack pointer
LD   (bstk),HL                \ preserve it
LD   SP,(&1FFE)               \ install system (safe) stack pointer
XOR  A
LD   B,A
LD   HL,errhan                \ address of error handler
OPT  FNsys(OS_Erh)            \ install new error handler
LD   (obou),A                 \ save old error handler call level
LD   (oerr),HL                \ save old error handler address

\ Here is the call to your assembler language routine which should be
\ included at the end of this code.
CALL main                     \ call main routine
```

```
.exit
LD   HL,(oerr)                \ address of old error handler
LD   A,(obou)                 \ old call level
LD   B,0
OPT  FNsys(OS_Erh)            \ restore previous error handler
LD   SP,(bstk)                \ restore BASIC stack pointer
RET                           \ return to BBC BASIC interpreter

.errhan
RET  Z
CP   rc_esc                   \ ESC pressed?
JR   NZ,err1
OPT  FNsys(OS_Esc)            \ acknowledge ESC
LD   A,rc_esc
OR   A                        \ return rc_esc back to main program
RET                           \ Fc = 0, Fz = 0

.err1
CP   rc_quit                  \ KILL request?
JR   NZ,err2

LD   HL,(oerr)                \ re-install old error handler
LD   A,(obou)                 \ old call level
OPT  FNsys(OS_Erh)

LD   SP,(bstk)                \ install BASIC stack pointer
LD   HL,(oerr)

LD   A, rc_quit               \ reload A with RC_QUIT
OR   A                        \ Fz = 0
SCF                           \ Fc = 1
JP   (HL)                     \ jump to BASIC's error handler

.err2                         \ write error message if possible
OPT  FNsys(GN_Esp)            \ Get ext. pointer to system error message
OPT  FNsys(GN_Soe)            \ Write error message to std. output
OPT  FNsys(GN_Nln)            \ New line to std. output
OR   A                        \ Fc = 0
RET

.bstk DEFW 0                  \ storage for BASIC stack pointer
.obou DEFB 0                  \ storage for old call level
.oerr DEFW 0                  \ storage for old error handler address

\ ---------------------------------------------------------------------
\ main routine starts here
.main
LD   HL,scratch_1             \ holds address of file to open
LD   DE,scratch_2             \ explicit name buffer
LD   C,40                     \ size of explicit name buffer
LD   B,0                      \ HL string pointer is local
LD   a, op_dor                \ get DOR handle
OPT  FNsys(GN_Opf)            \ open...
JR   NC,opened_OK
OPT  FNsys(GN_Err)            \ report error in standard window
RET
.opened_OK
LD   A,dr_rd                  \ read DOR record
LD   B,ASC"U"                 \ read update information
LD   C,6                      \ 3 byte internal date, 3 byte int. time
LD   DE,scratch_1             \ store returned information at (DE)
OPT  FNsys(OS_Dor)            \ fetch update date
LD   A,dr_fre
OPT  FNsys(OS_Dor)            \ free DOR handle
LD   HL,scratch_2             \ display explicit filename
OPT  FNsys(GN_Sop)            \ to standard output
LD   HL,tab_str
OPT  FNsys(GN_Sop)            \ tab to column 40
```

```
LD   HL,scratch_1
OPT  FNsys(GN_Sdo)              \ output returned update date
OPT  FNsys(GN_Nln)              \ display newline
RET                             \ back to BASIC

.scratch_1 DEFM STRING$(40,"X")
.scratch_2 DEFM STRING$(40,"X")
.tab_str DEFM CHR$1+"2X"+CHR$(32+40)+CHR$0
\ main routine ends here
\ ---------------------------------------------------------------------
]
NEXT pass

CLS
PRINT "Read File Update Date and Time"
INPUT "Filename:"A$
IF LEN(A$)>40 THEN PRINT "String too long": END

A$=A$+CHR$0  : REM null-terminate filename string
$scratch_1=A$
CALL code
END
:
DEF FNsys(arg)
IF arg>255 THEN PROC_Rst20Defw(arg) ELSE PROC_Rst20Defb(arg)
=pass
:
DEF PROC_Rst20Defw(arg)
[OPT pass
RST &20: DEFW arg
]
ENDPROC
DEF PROC_Rst20Defb(arg)
[OPT pass
RST &20: DEFB arg
]
ENDPROC
```

You can test the error handling code by replacing the above main routine with the following program as the other ‘main’ routine (also remove the filename input code after the assembler section which is no longer necessary).

```
.main
LD   B,10                       \ Loop 10 times
.loop
LD   A,B                        \ loop value
ADD  A,ASC”0”-1                 \ convert to Ascii digit
OPT  FNsys(OS_Out)              \ display digit to std. output
\
.inp
OPT  FNsys(OS_In)               \ Read a character from keyboard
JR   C,inp                      \ if Fc = 1, read error code
DJNZ loop
RET
```

Try running the complete program and observe the effect of:

- Suspending (eg. activate INDEX) and then either killing or reentering
- Pressing `[ESC]`
- Switching the Z88 off and on (both `[SHIFT]` keys)

Statements and functions

Introduction

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

All statements can also be used as direct commands.

Where appropriate, the abbreviated form is shown to the right of the statement.

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

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

Syntax

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

Symbols

The following symbols have special meaning in the syntax definitions.

{ }	The enclosed item may be repeated zero or more times.
[ ]	The items enclosed are optional, they may occur zero or one time.
|	Indicates alternatives; one of which must be used.
<stmt>	Means a BBC BASIC(Z80) statement.
<var>	Means a numeric or string variable.
<exp>	Means an expression like PI*radius*height+2 or name$+"FRED"+CHR$(&0D). It can also be a single variable or constant like 23 or "FRED".
<l-num>	Means a line number in a BBC BASIC(Z80) program.
<k-num>	Means the number of one of the programmable keys.
<n-const>	Means a numeric constant like '26.4' or '256'.
<n-var>	Means a numeric variable like 'size' or 'weight'.
<numeric>	Means a <n-const> or a <n-var> or an expression combining them. For example: PI*radius+2.66

<s-const>	Means a string constant like "FRED".
<string>	Means an unquoted string of characters.
<s-var>	Means a string variable like 'address$'.
<str>	Means a <s-const> or a <s-var> or an expression combining them. For example: name$+add$+"Phone".
<t-cond>	Means a 'testable condition'. In other words, something which is either TRUE or FALSE. Since BBC BASIC does not have true Boolean variables, TRUE and FALSE are numeric (with a value of -1 and 0). Consequently, a <numeric> can be used anywhere a <t-cond> is specified.
<name>	Means a valid variable name.
<d:>	Means a device drive name. eg. :RAM.0 (See the ‘Operating System Interface’ section for details of valid devices).
<afsp>	Means an ambiguous file specifier.
<ufsp>	Means an unambiguous file specifier.
<nchr>	Means a character valid for use in a name. 0 to 9, A to Z, a to z and underline.
PI	The mathematical constant **π** (3.14159 etc.).

ABS

A function giving the absolute value of its argument.

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

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

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

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

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

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

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

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

Syntax

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

Associated Keywords

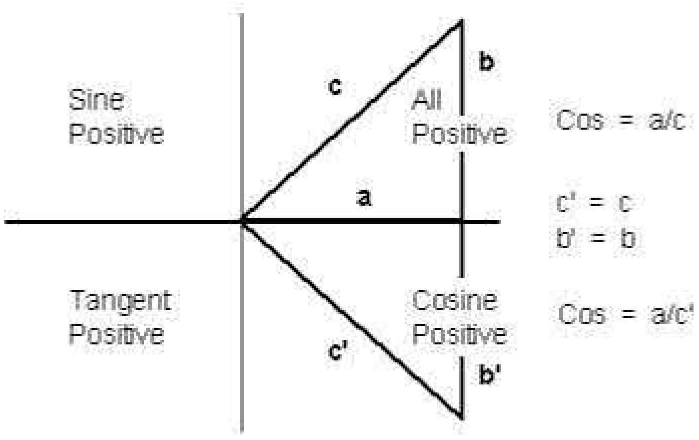
SGN

ACS

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

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

The following diagram illustrates the problem:



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

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

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

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

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

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

Syntax

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

Associated Keywords

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

ADVAL

Not available on the Z88

AND (A.)

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

```
answer=num1 AND num2
char=byte AND &7F
IF (num AND &F0)
test=(count=3 AND total=5)
```

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

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

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

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

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

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

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

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

or

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

(without brackets this 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
```

Syntax

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

Associated Keywords

EOR, OR, FALSE, TRUE, NOT

ASC

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

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

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

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

ASC is the complement of CHR$.

Syntax

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

Associated Keywords

CHR$, STR$, VAL

ASN

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

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

If you know the sine of the angle, this function will tell you the angle (in radians). Unfortunately, you cannot do this with complete certainty because one sine has two associated angles. (See ACS for details.)

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

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

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

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

Syntax

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

Associated Keywords

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

ATN

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

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

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

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

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

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

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

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

Syntax

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

Associated Keywords

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

AUTO (AU.)

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

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

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

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

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

```
AUTO start_number,step_size
```

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

A hyphen is an acceptable alternative to a comma.

Syntax

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

Associated Keywords

None

BGET# (B.#)

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

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

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

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

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

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

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

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

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

Syntax

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

Associated Keywords

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

BPUT# (BP.#)

A statement which puts a byte to the data file whose file handle 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
BPUT#4,prn
```

Before you use this statement you must normally have opened a file for output using OPENOUT or OPENUP. See these keywords and the BBC BASIC(Z80) Files section 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 three bytes are 'masked off'. Each byte written is numeric, but you can use ASC(character$) to convert (the first character of) 'character$' to a number.

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

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

Syntax

```
BPUT#<numeric>,<numeric>
```

Associated Keywords

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

CALL (CA.)

A statement to call a machine code subroutine.

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

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

Parameter Table

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

Variables included in the parameter list need not have been declared before the CALL statement. The parameter types are:

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

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

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

Parameter Formats

Integer variables are stored in two's complement format with their least significant byte first.

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

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

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

```
    & 00 00 00 05 00   Integer 5
```

or

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

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

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

See the Annex entitled 'Format of Program and Variables in Memory' for details of how parameters are stored.

Syntax

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

Associated Keywords

USR

CHAIN (CH.)

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

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

The program file must be in tokenised format.

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

CHAIN sets ON ERROR OFF before chaining the specified program.

RUN may be used as an alternative to CHAIN.

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

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

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

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

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

Syntax

```
CHAIN <str>
```

Associated Keywords

LOAD, RUN, SAVE

CHR$

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

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

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

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

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

```
CHR$(7)
```

will generate the ASCII character ^G. So,

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

will print the message 'ERROR' and sound the PC's 'bell'. CHR$ is the complement of ASC.

Syntax

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

Associated Keywords

ASC, STR$, VAL, VDU

CLEAR (CL.)

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

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

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

Syntax

CLEAR

Associated Keywords

None

CLOSE# (CLO.#)

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

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

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

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

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

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

Syntax

```
CLOSE#<numeric>
```

Associated Keywords

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

CLG

This clears the graphics window (only); it does not affect the position of the graphics cursor. Note that CLS can be used to clear the text window and leave the graphics window unchanged.

This statement is available when installed with Z88 Patch via CHAIN command. See “Installing Z88 Patch” section previously in this guide, on how to obtain the Z88 Patch. Available by default when BBC BASIC(Z80) is used with Z88 ROM release V4.3 and later.

Syntax CLG
Associated Keywords MODE, CLS

CLS

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

Syntax

CLS

Associated Keywords

CLG

COLOUR

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

Not available on the Z88

COS

A function giving the cosine of its radian argument.

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

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

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

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

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

Syntax

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

Associated Keywords

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

COUNT (COU.)

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

```
char_count=COUNT
```

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

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

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

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

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

Syntax

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

Associated Keywords

POS

DATA (D.)

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

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

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

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

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

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

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

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

Syntax

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

Associated Keywords

READ, RESTORE

DEF

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

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

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

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

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

In order to make the text more readable (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 Procedures and Functions sub-section in the General Information section.

Syntax

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

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 PI radians is 180 degrees.

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

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

Syntax

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

Associated Keywords

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

DELETE (DEL.)

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

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

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

```
DELETE 10,15
```

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

```
DELETE 0,120
```

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

```
DELETE 2310,65535
```

A hyphen is an acceptable alternative to a comma.

Syntax

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

Associated Keywords

EDIT, LIST, OLD, NEW

DIM

There are two 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 two dimensional array. The two dimensions would represent the row (numbered 0 to 7) and the column (also numbered 0 to 7). The contents of each 'cell' of the array would indicate the presence (if any) of a piece and its value.

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

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

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

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 BBC BASIC(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 reserves 68 bytes of memory and sets A% equal to the address of the first byte. Thus A%?0 to A%?67 are free for use by the program (68 bytes in all):

```
DIM A% 67
```

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 some other versions of BASIC. See also EXT#-1. See the Assembler section for a more detailed description of the use of DIM for reserving memory for machine code programs.

Syntax

```
DIM <n-var>|<s-var>(<numeric>{,<numeric>})
DIM <n-var> <numeric>
```

Associated Keywords

CLEAR

DIV

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

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

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

```
21 DIV 4
```

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

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

Syntax

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

Associated Keywords

MOD

DRAW

Draws a straight line (in black) between the current position of the graphics cursor and the specified coordinates, then moves the graphics cursor to the specified position. This statement is identical to PLOT 5,x,y.

This statement is available when installed with Z88 Patch via CHAIN command.
See “Installing Z88 Patch” section previously in this guide, on how to obtain the Z88 Patch.

Available by default when BBC BASIC(Z80) is used with Z88 ROM release V4.3 and later.

Syntax

```
DRAW <numeric>,<numeric>
```

Associated Keywords

PLOT, MODE

*EDIT (*E.)

This command allows you to edit a specified program line. It results in the line being displayed with the cursor positioned at the end, and you can then edit the line using any of the usual line-editing features on the Z88.

To enter the edited line into the program press ENTER; to abandon the edit and leave the line unchanged press ESC.

You can also use *EDIT to concatenate two or more program lines, by specifying the first line and last line separated by commas (e.g. *EDIT 10,30). This process will stop when the concatenated line length exceeds 255. You will have to edit out the line numbers of the second and subsequent lines (and delete the old lines afterwards).

*EDIT may be abbreviated to *E. (the dot is required).

This statement is available when installed with Z88 Patch via CHAIN command.

See “Installing Z88 Patch” section previously in this guide, on how to obtain the Z88 Patch. Available by default when BBC BASIC(Z80) is used with Z88 ROM release V4.3 and later.

Syntax

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

Associated Keywords

DELETE, LIST, OLD, NEW

ELSE (EL.)

A statement delimiter which provides an alternative course of action in IF...THEN, ON...GOSUB, ON...GOTO and ON...PROC statements. In an IF statement, if the 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"WWW" ELSE B=D:PRINT"QQQ"
IF A=B THEN B=C ELSE IF A=C THEN...............
```

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

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

If you want to 'nest' the tests, you should use a 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 a$ ELSE PRINT er$
ENDPROC
```

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

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

Syntax

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

Associated Keywords

IF, THEN, ON

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 BBC BASIC(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 it's 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 BBC BASIC(Z80) 'running into' any procedure or function definitions at the end of your program.

Syntax

END

Associated Keywords

STOP

ENDPROC

A statement denoting the end of a procedure.

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

Syntax

ENDPROC

Associated Keywords

DEF, FN, PROC, LOCAL

ENVELOPE

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

Not available on the Z88

EOF#

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

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

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

If you attempt to read beyond the current end of file, you will get an ‘End Of file’ error.

In an unexpanded Z88, BBC BASIC(Z80) has a workspace of 8Kbytes. In an expanded Z88 (containing at least 128Kbytes in slot 0 or 1) the workspace is increased to 40Kbytes.

EOF#-1 returns TRUE for an expanded machine and FALSE for an unexpanded machine.

Syntax

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

Associated Keywords

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

EOR

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

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

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

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

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

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

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

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

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

```
A=X EOR 11
```

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

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

Syntax

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

Associated Keywords

NOT, AND, OR

ERL

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

```
        X=ERL
```

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

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

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

Syntax

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

Associated Keywords

ON ERROR GOTO, ON ERROR OFF, REPORT, ERR

ERR

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

```
        X=ERR
```

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

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

Syntax

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

Associated Keywords

ON ERROR GOTO, ON ERROR OFF, ERL, REPORT

EVAL (EV.)

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

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

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

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

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

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

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

In the following example, EVAL would print 'Hello world!' if you entered 'FN_FRED' in response to the prompt. The program will continue until an error occurs (pressing `[ESC]` for example).

```
10 REPEAT
20   INPUT "Enter an expression: ",a$
30   PRINT EVAL a$
40 UNTIL FALSE
50:
60 DEF FN _FRED
70 PRINT "Hello World!"
80 =-1
```

Example run (user entry in **bold**):

```
>RUN[ENTER]
Enter an expression: 2*5[ENTER]
      10
Enter an expression: FN_FRED[ENTER]
      Hello World!
Enter an expression: JIM[ENTER]
No such variable at line 20
>
```

Syntax

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

Associated Keywords

STR$, VAL

EXP

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

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

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

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

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

Syntax

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

Associated Keywords

LN, LOG

EXT#

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

```
      length=EXT#f_num
```

In the case of a sparse random-access file, the value returned is the complete file length from byte zero to the last byte written. This may well be greater than the actual amount of data in the file, but it is the amount of disk space allocated to the file by the Z88's operating system.

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

EXT#-1 returns an estimate of the amount of free memory in the Z88. See the Annex entitled 'Format of Program and Variables in Memory' for more details.

Free memory is available to the device **`:RAM.-`** and to applications (such as PipeDream), but not necessarily to the RAM filing system. Consequently, you cannot reliably use EXT#-1 to discover how much space is left in the current filing system device. (**`:RAM.0`** for example).

Syntax

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

Associated Keywords

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

FALSE (FA.)

A function returning the value zero.

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

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

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

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

```
10 terminator=10
20 REPEAT: PRINT "An endless loop": 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: PRINT "An endless loop": UNTIL FALSE
```

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

```
10 REPEAT: PRINT "An endless loop": UNTIL 0
```

See the keyword AND for logical tests and their results.

Syntax

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

Associated Keywords

TRUE, EOR, OR, AND, NOT

FN

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

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

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

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

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

```
DEF FN_mean(Q1,Q2,Q3,Q4)=(Q1+Q2+Q3+Q4)/4
DEF FN_fact(N) IF N<2 =1 ELSE =N*FN_fact(N-1)

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

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

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

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

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

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

Thus the PRINT statement

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

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

Syntax

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

Associated Keywords

ENDPROC, DEF, LOCAL

FOR (F.)

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

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

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

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

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

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

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

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

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

```
DIM X 0
FOR ?X=0 TO 16: PRINT ~?X: NEXT
DIM X 3
FOR !X=0 TO 16 STEP 4: PRINT ~!X: NEXT
```

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

Syntax

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

Associated Keywords

TO, STEP, NEXT

GCOL (GC.)

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

Not available on the Z88

GET/GET$

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

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

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

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

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

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

The values returned by the cursor and other ‘special’ keys are listed below. Many of the keys return 2 bytes, a zero byte followed by another value.

Keys	**Normal Codes**	**with [SHIFT]**	**with ◊**	**with □**
`[ENTER]`	13	0,209	0,193	0,177
`[TAB]`	9	0,210	0,194	0,178
`[DEL]`	127	0,211	0,195	0,179
⇦	0,252	0,248	0,244	0,240
⇨	0,253	0,249	0,245	0,241
⇩	0,254	0,250	0,246	0,242
⇧	0,255	0,251	0,247	0,243
£	163	126 (-)	30	158
=	61	43	0,0	Nothing
`[SPACE]`	32	32	160	32
‘	39	35 (“)	96	Nothing

GET can also be used to input data from an I/O port. Full 16-bit port addressing is available :

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

This is an addition to the original language specification.

Syntax

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

Associated Keywords

PUT, INKEY, INKEY$

GOSUB

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

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

The only limit placed on the depth of nesting is the room available for the stack.

You may calculate the line number. However, if you do, the program should not be RENUMBERed. A calculated value must be placed in brackets.

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

Subroutines are similar to PROCedures, but they are called by line number not by name. This can make the program difficult to read because you have no idea what the subroutine does until you have followed it through. You will probably find that PROCedures offer you all the facilities of subroutines and, by choosing their names carefully, you can make your programs much more readable.

Syntax

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

Associated Keywords

RETURN, ON, PROC

GOTO (G.)

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

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

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

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

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

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

Syntax

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

Associated Keywords

GOSUB, ON

HIMEM

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

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

```
        HIMEM=HIMEM-40
```

BBC BASIC(Z80) uses the computer's memory to store your program and the variables that your program uses. The default value of HIMEM is the highest memory address available for use by BBC BASIC(Z80). On an expanded Z88 (one containing at least 128 Kbytes of RAM in slot 0 or 1) HIMEM is initially set to &C000. On an unexpanded Z88, it is set to &4000. In an unexpanded Z88, BBC BASIC(Z80) has a workspace of 8 Kbytes. In an expanded Z88, the workspace is increased to 40 Kbytes.

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

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

USE WITH CARE.

Syntax

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

Associated Keywords

LOMEM, PAGE, TOP

IF

A statement which 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.

```
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 word THEN is optional under most circumstances.

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 two meanings. It can be used to assign a value to a variable or as part of a test. The example shows the two 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 two 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"
```

Syntax

```
IF <t-cond> THEN <stmt>{:<stmt>} [ELSE <stmt>{:<stmt>}]
IF <exp> THEN <stmt>{:<stmt>} [ELSE <stmt>{:<stmt>}]
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>]
```

Associated Keywords

THEN, ELSE

INKEY/INKEY$

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

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

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

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

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

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

INKEY with a negative argument is not available on the Z88.

Same key codes are returned as for GET. Please refer to GET reference for more details.

Syntax

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

Associated Keywords

GET, GET$

INPUT

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

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

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

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

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

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

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

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

The statement

```
INPUT A
```

is exactly equivalent to

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

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

```
INPUT A$
A=EVAL(A$)
```

Strings in quoted form are taken as they are, with a possible error occurring for a missing closing quote.

A semicolon following a prompt string is an acceptable alternative to a comma.

Syntax

```
INPUT [TAB(X[,Y])][SPC(<numeric>)]['][<s-const>[,|;]]
                    <n-var>|<s-var>{,<n-var>|<s-var>}
```

Associated Keywords

INPUT LINE, INPUT#, GET, INKEY

INPUT LINE

A statement of identical syntax to INPUT which uses a new line for each item to be input. The item input is taken as is, including commas, quotes and leading spaces.

```
      INPUT LINE A$
```

Syntax

```
INPUT LINE[TAB(X[,Y])][SPC(<numeric>)]['][<s-const>[,|;]]
                             <s-var>{,<s-var>}
```

Associated Keywords

INPUT

INPUT#

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

```
INPUT #E,A,B,C,D$,E$,F$
INPUT #3,aux$
```

It is possible to read past the end-of-file without an error being reported. You should always include some form of check for the end of the file.

READ# can be used as an alternative to INPUT#.

Whilst BBC BASIC(Z80) stores numbers in internal format, text is stored in the usual way as a string of characters followed by a carriage-return. Consequently, you can use the INPUT# command to read text from, say, a PipeDream file.

See the ‘BBC BASIC(Z80) Files’ section for more details and numerous examples of the use of INPUT#.

Syntax

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

Associated Keywords

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

INSTR

A function which returns the position of a sub-string within a string, optionally starting the search at a specified place in the string. The leftmost character position is 1. If the sub-string is not found, 0 is returned.

The first string is searched for any occurrence of the second string.

There must not be any spaces between INSTR and the opening bracket.

```
X=INSTR(A$,B$)
position=INSTR(word$,guess$)
Y=INSTR(A$,B$,Z%) :REM START AT POSITION Z%
```

You can use this function for validation purposes. If you wished to test A$ to see if was one of the set 'FRED BERT JIM JOHN', you could use the following:

```
set$="FRED BERT JIM JOHN"
IF INSTR(set$,A$) PROC_valid ELSE PROC_invalid
```

The character used to separate the items in the set must be excluded from the characters possible in A$. One way to do this is to make the separator an unusual character, say CHR$(127).

```
z$=CHR$(127)
set$="FRED"+z$+"BERT"+z$+"JIM"+z$+"JOHN"
```

Syntax

```
<n-var>=INSTR(<str>,<str>[,<numeric>])
```

Associated Keywords

LEFT$, MID$, RIGHT$, LEN

INT

A function truncating a real number to the lower integer.

```
X=INT(Y)

INT(99.8)  =99
INT(-12)   =-12
INT(-12.1) =-13
```

This function converts a real number (one with a decimal part) to the nearest integer (whole number) less than the number supplied. Thus,

```
INT(14.56)
```

gives 14, whereas

```
INT(-14.5)
```

gives -15.

Syntax

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

Associated Keywords

None

LEFT$

A string function which returns the left 'num' characters of the string. If there are insufficient characters in the source string, all the characters are returned.

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$="BBC BASIC(Z80)"
20 FOR i=3 TO LEN(name$)
30   PRINT LEFT$(name$,i)
40 NEXT
50 END
```

would print

```
BBC
BBCB
BBCBA
BBCBAS
BBCBASI
BBC BASIC
BBC BASIC(
BBC BASIC(Z
BBC BASIC(Z8
BBC BASIC(Z80
BBC BASIC(Z80)
```

Syntax

```
<s-var>=LEFT$(<str>,<numeric>)
```

Associated Keywords

RIGHT$, MID$, LEN, INSTR

LEN

A function which returns the length of the argument string.

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

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

```
length=LEN("BBC BASIC(Z80)   ")
```

would set 'length' to 17 since the string consists of the 14 characters of ‘BBC BASIC(Z80)’ followed by three spaces.

LEN is often used with a FOR....NEXT loop to 'work down' a string doing something with each letter in the string. For example, the following program looks at each character in a string and checks that it is a valid hexadecimal numeric character.

```
 10 valid$="0123456789ABCDEF"
 20 REPEAT
 30   INPUT "Type in a HEX number" hex$
 40   flag=TRUE
 50   FOR i=1 TO LEN(hex$)
 60     IF INSTR(valid$,MID$(hex$,i,1))=0 flag=FALSE
 80   NEXT
 90   IF NOT flag THEN PRINT "Bad HEX"
100 UNTIL flag
```

Syntax

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

Associated Keywords

LEFT$, MID$, RIGHT$, INSTR

LET

LET is an optional assignment statement.

LET is not permitted in the assignment of the pseudo-variables LOMEM, HIMEM, PAGE, PTR# and TIME.

LET was mandatory in early versions of BASIC. Its use emphasised that when we write

```
X=X+4
```

we don't mean to state that X equals X+4 - it can't be, but rather 'let X become equal to what it was plus 4':

```
LET X=X+4
```

Most modern versions of BASIC allow you to drop the 'LET' statement. However, if you are writing a program for a novice, the use of LET makes it more understandable.

Syntax

```
[LET] <var>=<exp>
```

Associated Keywords

None

LIST (L.)

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

`LIST`	lists the entire program
`LIST ,111`	lists up to line 111
`LIST 111,`	lists from line 111 to the end
`LIST 111,222`	lists lines 111 to 222 inclusive
`LIST 100`	lists line 100 only

A hyphen is an acceptable alternative to a comma.

The listing may be paused by pressing ◊ and `[SHIFT]` keys together. With the standard CLI active, the screen output will halt at the end of each page until the `[SPACE]` key is pressed.

Escape will abort the listing.

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

Syntax

```
LIST
LIST <n-const>
LIST <n-const>,
LIST ,<n-const>
LIST <n-const>,<n-const>
```

Associated Keywords

LISTO, OLD, NEW

LISTO

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

Bit Settings

Bit 0 (LSB)

If Bit 0 is set, a space will be printed between the line number and the remainder of the line. (All leading spaces are stripped when the line is originally entered.)

Bit 1

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

Bit 2

If Bit 2 is set two 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
```

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
```

Syntax

`LISTO <n-const>`

Associated Keywords

LIST

LN

A function giving the natural logarithm of its argument.

```
X=LN(Y)
```

This function gives the logarithm to the base 'e' of its argument. The 'natural' number, 'e', is approximately 2.71828183.

Logarithms are 'traditionally' used for multiplication (by adding the logarithms) and division (by subtracting the logarithms). For example,

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

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

Syntax

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

Associated Keywords

LOG, EXP

LOAD (LO.)

A command which loads a new program from a file and CLEARs the variables of the old program. The program file must be in 'internal' (tokenised) format.

```
LOAD "PROG1"
LOAD A$
```

File names must conform to the standard Z88 format. If no device and/or path are given, the current device and/or path are assumed. See the ‘Operating System Interface’ section for a more detailed description of valid file names.

You use LOAD to bring a program in a file into memory. The keyword LOAD should be followed by the name of the program file. If the program file is in the current directory, only the file name needs to be given. If the program is not in the current directory, its full device, path and file name must be specified. For example:

```
LOAD ":RAM.0/bbcprogs/demo"
```

would load the program 'demo' from the directory 'bbcprogs' on device **`:RAM.0`** .

Syntax

`LOAD <str>`

Associated Keywords

SAVE, CHAIN

LOCAL

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 keyword ON ERROR LOCAL for details of local error trapping.

Syntax

```
LOCAL <n-var>|<s-var>{,<n-var>|<s-var>}
```

Associated Keywords

DEF, ENDPROC, FN, PROC

LOG

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

```
      X = LOG(Y)
```

This function calculates the common (base 10) logarithm of its argument. Inverse logarithms (anti-logs) can be calculated by raising 10 to the power of the logarithm. For example, if x=LOG(y) then y=10^x.

Logarithms are 'traditionally' used for multiplication (by adding the logarithms) and division (by subtracting the logarithms). For example,

```
10 log1=LOG(2.5)
20 log2=LOG(2)
30 log3=log1+log2
40 answer=10^log3
50 PRINT answer
```

Syntax

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

Associated Keywords

LN, EXP

LOMEM

A pseudo-variable which controls where in memory the dynamic data structures are to be placed. The default is TOP, the first free address after the end of the program.

```
LOMEM=LOMEM+100
PRINT ~LOMEM :REM The ~ makes it print in HEX.
```

Normally, dynamic variables are stored in memory immediately after your program. (See the Annex entitled ‘Format of Program and Variables in Memory’.) You can change the address where BBC BASIC(Z80) starts to store these variables by changing LOMEM.

USE WITH CARE.

Changing LOMEM in the middle of a program causes BBC BASIC(Z80) to lose track of all the variables you are using.

Syntax

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

Associated Keywords

HIMEM, TOP, PAGE

MID$

A 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$="BBC BASIC(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$="BBC BASIC(Z80)"
20 FOR i=1 TO LEN(name$)
30   PRINT MID$(name$,i,10)
40 NEXT
```

would print

```
BBC BASIC(
BC BASIC(Z
C BASIC(Z8
 BASIC(Z80
BASIC(Z80)
ASIC(Z80)
SIC(Z80)
IC(Z80)
C(Z80)
(Z80)
Z80)
80)
0)
)
```

Syntax

`<s-var>=MID$(<str>,<numeric>[,<numeric>])`

Associated Keywords

LEFT$, RIGHT$, LEN, INSTR

MOD

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

```
X=A MOD B
```

MOD is defined such that,

```
A MOD B = A - ( A DIV B ) * B.
```

If you are doing integer division (DIV) of whole numbers it is often desirable to know the remainder. (A 'teach children to divide' program for instance.) For example, 23 divided by 3 is 7, remainder 2. Thus,

```
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 BBC BASIC(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.

Syntax

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

Associated Keywords

DIV

MODE (MO.)

The MODE statement allows selection of the normal text-only mode (MODE 0) or a text-and-graphics mode (MODE 1). In MODE 1 the display is split into two parts: a text-window on the left and a graphics-window on the right. The text window consists of 8 rows of 50 characters, and the graphics window is 64 pixels high by 256 pixels wide; you cannot (normally) mix text and graphics in the same window.

Points in the graphics window are addressed as x,y coordinates from 0,0 (the bottom-left corner) to 255,63 (the top-right corner), although the origin can be moved using the PLOT -1 statement (q.v.).

Although MODE 1 sets up the window positions and sizes as described, it is possible to change these using the VDU statement. However the method of doing this is outside the scope of this reference guide. It is not advisable to cause the text and graphics windows to overlap, although this may occasionally be useful.

MODE clears the display (both text and graphics windows), moves the text cursor to 0,0 (the top left of the text window), resets the graphics origin and moves the graphics cursor to 0,0 (the bottom left of the graphics window).

In MODE 0 (the normal 94-column text mode) the other graphics statements have no effect.

MODE cannot be used within a PROCedure, function, FOR..NEXT or REPEAT.. UNTIL loop. Doing so will result in the 'Bad MODE' error (code 153).

MODE 1 changes the value of HIMEM address 2 Kbytes lower and MODE 0 changes the value of HIMEM address 2 Kbytes higher (allocating and releasing a graphics buffer).

This statement is available when installed with Z88 Patch via CHAIN command. See “Installing Z88 Patch” section previously in this guide, on how to obtain the Z88 Patch.

Available by default when BBC BASIC(Z80) is used with Z88 ROM release V4.3 and later.

Syntax

```
MODE (<numeric>)
```

Associated Keywords

CLS,CLG

MOVE

Moves the graphics cursor to the specified coordinates, but does not affect what is displayed.

This statement is identical to PLOT 4,x,y.

This statement is available when installed with Z88 Patch via CHAIN command. See “Installing Z88 Patch” section previously in this guide, on how to obtain the Z88 Patch.

Available by default when BBC BASIC(Z80) is used with Z88 ROM release V4.3 and later.

Syntax

```
MOVE <numeric>,<numeric>
```

Associated Keywords

DRAW, MODE, GCOL, PLOT

NEW

A command which initialises the interpreter for a new program to be typed in. The old program may be recovered with the OLD command provided no new program lines have been typed in or deleted and no variables have been created.

```
       NEW
```

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

If you have made a mistake, you can recover your old program by typing OLD. However, this won't work if you have begun to enter a new program.

Syntax NEW
Associated Keywords OLD

NEXT (N.)

The statement delimiting FOR...NEXT loops. NEXT takes an optional control variable.

```
    NEXT
    NEXT J
```

If the control variable is present then FOR....NEXT loops may be 'popped' automatically in an attempt to match the correct FOR statement (this should not be necessary). If a matching FOR statement cannot be found, a 'Can't match FOR' error will be reported.

Leaving out the control variable will make the program run quicker, but this is not to be encouraged.

See the keyword FOR for more details about the structure of FOR....NEXT loops.

Syntax

```
NEXT [<n-var>{,<n-var>}]
```

Associated Keywords

FOR, TO, STEP

NOT

This is a high priority unary operator (the same priority as unary -). It causes a bit-by-bit binary inversion of the numeric to its right. The numeric may be a constant, a variable, or a mathematical or boolean expression. Expressions must be enclosed in brackets.

```
A=NOT 3
flag=NOT flag
flag=NOT(A=B)
```

NOT is most commonly used in an IF....THEN....ELSE statement to reverse the effect of the test.

```
IF NOT(rate>5 AND TIME<100) THEN .....
IF NOT flag THEN .....
```

BBC BASIC(Z80) does not have true boolean variables; it makes do with numeric variables. This can lead to confusion because the testable condition in an IF....THEN....ELSE statement is evaluated mathematically and can result in something other than -1 (TRUE) or 0 (FALSE).

When the test in an IF....THEN....ELSE is evaluated, FALSE=0 and anything else is considered to be TRUE. If you wish to use NOT to reverse the action of an IF statement it is important to ensure that the testable condition does actually evaluate to -1 for TRUE.

If the testable condition evaluates to 1, for example, the result of the test would be considered to be TRUE and the THEN part of the IF....THEN....ELSE statement would be carried out. However, using NOT in front of the testable condition would not reverse the action. NOT 1 evaluates to -2, which would also be considered to be TRUE.

Syntax

```
<n-var>=NOT<numeric>
```

Associated Keywords

NONE

OLD

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

```
    OLD
```

OLD works even if BBC BASIC(Z80) has been re-loaded and re-started from CP/M-80. However, it will only work if no other programs have been run and BBC BASIC(Z80) loads at the same address as before.

Syntax OLD
Associated Keywords NEW

ON

A statement controlling a multi-way switch. The line numbers in the list may be constants or calculated and the 'unwanted' ones are skipped without calculation. The ON statement is used in conjunction with four other key-words: GOTO, GOSUB, PROC and ERROR. (ON ERROR is explained separately.)

```
ON option GOTO 1000,2000,3000,4000
ON action GOSUB 100,3000,200,5000,30
ON choice PROC_add,PROC_find,PROC_delete
```

The ON statement alters the path through your program by transferring control to one of a selection of line numbers depending on the value of a variable. For example,

```
200 ON number GOTO 1000,2000,500,100
```

would send your program to line 1000 if 'number' was 1, to line 2000 if 'number' was 2, to line 500 if 'number' was 3 and to line 100 if 'number' was 4.

Exceptions may be trapped using the ELSE statement delimiter.

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

If there is no statement after the ELSE, the program will 'drop through' to the following line if an exception occurs. In the following two examples, the program would drop through to the error handling part of the program if 'choice' or 'B-46' was less than one or more than 3.

```
ON choice PROC_add,PROC_find(a$),PROC_delete ELSE
     PRINT "Illegal Choice - Try again"
ON B-46 GOSUB 100,200,(C/200) ELSE PRINT "ERROR"
```

You can use ON...GOTO, ON...GOSUB, and ON...PROC to execute the appropriate part of your program as the result of a menu selection. The following skeleton example offers a menu with three choices.

```
 20 CLS
 30 PRINT "SELECT THE ACTION YOU WISH TO TAKE"
 40 PRINT "1 OPEN A NEW DATA FILE"
 50 PRINT "2 ADD DATA TO THE FILE"
 60 PRINT "3 CLOSE THE FILE AND END"''
 70 REPEAT
 80   INPUT TAB(10,20)"WHAT DO YOU WANT ? "choice
 90 UNTIL choice>0 AND choice<4
100 ON choice PROC_open,PROC_add,PROC_close ELSE
110 .....etc
```

Limitations

If a statement terminator (: or the token for ELSE) appears within the line, the interpreter assumes that the ON... statement is terminated. For example, you cannot pass a colon as a literal string parameter in an ON...PROC command. The program line

```
      ON entry PROC_start,PROC_add(":"),PROC_end
```

would be interpreted as

```
      ON entry PROC_start,PROC_add(":"),PROC_end
```

and give rise to an interesting crop of error messages.

Syntax

```
ON <numeric> GOTO <l-num>{,<l-num>}
             [ELSE <stmt>{:<stmt>}]
ON <numeric> GOSUB <l-num>{,<l-num>}
             [ELSE <stmt>{:<stmt>}]
ON <numeric> PROC<name>[(<exp>{,<exp>})]
             {,PROC<name>[(<exp>{,<exp>})]}
             [ELSE <stmt>{:<stmt>}]
```

Associated Keywords

ON ERROR, ON ERROR LOCAL, GOTO, GOSUB, PROC

ON ERROR

A statement controlling error trapping. If an ON ERROR statement has been encountered, BBC BASIC(Z80) will transfer control to it (without taking any reporting action) when an error is detected. This allows error reporting/recovery to be controlled by the program. However, the program control stack is still cleared when the error is detected and it is not possible to RETURN to the point where the error occurred.

ON ERROR OFF returns the control of error handling to BBC BASIC(Z80).

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

If there is an error in your error handling routine, your program will enter a loop. Unless you take the precautions described below, the only way to escape from this is with a soft or hard reset. You can avoid having to perform a reset by including a line such as:

```
dummy = INKEY(0)
```

at the beginning of your error handling routine. You can then exit from BBC BASIC(Z80) by pressing the `[INDEX]` key and ◊KILL the offending instantiation of BBC BASIC(Z80).

Error handling is explained more fully in the General Information section.

Syntax

```
ON ERROR <stmt>{:<stmt>}
ON ERROR OFF
```

Associated Keywords

ON, GOTO, GOSUB, PROC

OPENIN (OP.)

A function which opens a file for reading and returns the file handle of the file. This number must be used in subsequent references to the file with BGET#, INPUT#, 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$(10)
20 fnum=OPENIN "TOPTEN"
30 IF fnum=0 THEN PRINT "No TOPTEN data": END
40 FOR i=1 TO 10
50   INPUT#fnum,posn(i),name$(i)
60 NEXT
70 CLOSE#fnum
```

You can also use OPENIN to access the devices **`:COM.0`**, etc. See the ‘Operating System Interface’ and ‘BBC BASIC(Z80) Files’ sections for more details.

Syntax

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

Associated Keywords

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

OPENOUT

A function which opens a file for writing and returns the file handle of the file. This number must be used in subsequent references to the file with BPUT#, PRINT#, EXT#, PTR# or CLOSE#. If the specified file does not exist it is created. If the specified file already exists it is truncated to zero length.

A returned value of zero indicates that the specified file could not be created.

```
X=OPENOUT(A$)
X=OPENOUT("DATAFILE")
X=OPENOUT("LPT1")
```

You can also read from a file which has been opened using OPENOUT. This is of little use until you have written some data to it. However, once you have done so, you can move around the file using PTR# and read back previously written data.

Data is not written to the file at the time it is opened. Consequently, it is possible to successfully open a file on a full device. Under these circumstances, a 'Device full' error would be reported when you tried to write data to the file for the first time.

The example below writes the contents of two arrays (tables) to a file called 'TOPTEN'.

```
10 A=OPENOUT "TOPTEN"
20 FOR Z=1 TO 10
30   PRINT#A,N(Z),N$(Z)
40 NEXT
50 CLOSE#A
60 END
```

You can also use OPENOUT to access the devices **`:COM.0`**, **`:PRT.0`**, etc. See the ‘Operating System Interface’ and ‘BBC BASIC(Z80) Files’ sections for more details.

Syntax

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

Associated Keywords

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

OPENUP

A function which opens a file (or device) for update (reading and writing) and returns the file handle of the file. This number must be used in subsequent references to the file with BGET#, BPUT#, INPUT#, PRINT#, EXT#, PTR#, EOF# or CLOSE#.

A returned value of zero signifies that the specified file (or device) was not found.

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

See the random file examples **F-RAND0** in the ‘BBC BASIC(Z80) Files’ section for examples of the use of OPENUP.

You can also use OPENUP to access the devices **`:COM.0`**, **`:PRT.0`**, etc. See the ‘Operating System Interface’ and ‘BBC BASIC(Z80) Files’ sections for more details.

Syntax

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

Associated Keywords

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

OPT

An assembler pseudo operation controlling the method of assembly. (See the Assembler section for more details.) OPT is followed by an expression with the following meanings:

Code Assembled Starting at P%

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

Code Assembled Starting at O%

Value	Action	
4	assembler errors suppressed;	no listing.
5	assembler errors suppressed;	listing.
6	assembler errors reported;	no listing.
7	assembler errors reported;	listing.

The possible assembler errors are:

```
Out of range - error code 40.
No such variable - error code 26.
```

Syntax

```
OPT <numeric>
```

Associated Keywords

None

OR

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

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

You can leave out the space between OR and a preceding constant, but it makes your programs difficult to read.

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

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

In the example below, the operands are boolean (logical). In other words, the result of the tests (IF) A=2 and (IF) B=3 is either TRUE or FALSE. The result of this example will be TRUE if A=2 or B=3.

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

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

The last example, uses the OR in a similar fashion to the numeric operators (+, -, etc). Suppose X was -20 in the following example,

```
A=X OR 11
```

the OR operation would be:

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

Syntax `<n-var>=<numeric> OR <numeric>`
Associated Keywords AND, EOR, NOT

OSCLI

This command allows a string expression to be passed to the operating system. It overcomes the problems caused by the exclusion of variables in the star (*) commands. Using this statement, you can, for instance, erase and rename files whose names you only know at run-time.

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

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

See the Operating System Interface section for more details. If the command is not recognised by the Z88's operating system, a 'Bad command' error will be reported.

Syntax

```
OSCLI <str>
```

Associated Keywords

All operating system (*) commands.

PAGE (PA.)

A pseudo-variable controlling the starting address of the current user program area. It addresses the area where a program is (or will be) stored.

```
PAGE=&3100
PRINT ~PAGE
PAGE=TOP+&100: REM Move to start of next page.
```

PAGE is automatically initialised by BBC BASIC(Z80) to the address of the lowest available page in RAM, but you may change it.

If you make PAGE less than its original value or greater than the original value of HIMEM, you will get a 'Bad program' error when you try to enter a program line and you may well crash BBC BASIC(Z80).

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

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

USE WITH CARE.

Syntax

```
<n-var>=PAGE
```

Associated Keywords

TOP, LOMEM, HIMEM

PI

A function returning 3.14159265.

```
X=PI
```

You can use PI to calculate the circumference and area of a circle. The example below calculates the circumference and area of a circle of a given radius.

```
10 CLS
20 INPUT "What is the radius of the circle ",rad
30 PRINT "The circumference is: ";2*PI*rad
40 PRINT "The area is: ";PI*rad*rad
50 END
```

PI can also be used to convert degrees to radians and radians to degrees.

```
radians=PI/180*degrees
degrees=180/PI*radians
```

However, BBC BASIC(Z80) has two functions (RAD and DEG) which perform these conversions to a higher accuracy.

Syntax

```
<n-var>=PI
```

Associated Keywords

RAD, DEG

PLOT (PL.)

PLOT is a multi-purpose drawing statement. Three numbers follow the PLOT statement: the first specifies the type of point, line, or triangle to be drawn; the second and third give the X and Y coordinates to be used.

n	action
-1	Move the graphics origin to x,y.
0	Move the graphics cursor relative to the last point.
1	Draw a line, in 'black', relative to the last point.
2	Draw a line, in 'inverse', relative to the last point.
3	Draw a line, in 'white', relative to the last point.
4	Move the graphics cursor to the absolute position x,y.
5	Draw a line, in 'black', to the absolute position x,y.
6	Draw a line, in 'inverse', to the absolute position x,y.
7	Draw a line, in 'white', to the absolute position x,y.
8-15	As 0-7, but plot the last point on the line twice (i.e. in the 'inverting' modes omit the last point).
16-31	As 0-15, but draw the line dotted.
32-63	As 0-31, but plot the first point on the line twice (i.e. in the 'inverting' modes omit the first point).
64-71	As 0-7, but plot a single point at x,y.
72-79	Draw a horizontal line left and right from the point x,y until the first 'lit' pixel is encountered, or the edge of the window. This can be used to fill shapes.
80-87	Plot and fill a triangle defined by the two previously visited points and the point x,y.
88-95	Draw a horizontal line to the right of the point x,y until the first 'unlit' pixel is encountered, or the edge of the window. This can be used to 'undraw' things.
96-103	Plot and fill a rectangle whose opposite corners are defined by the last visited point and the point x,y.

This statement is available when installed with Z88 Patch via CHAIN command. See "Installing Z88 Patch" section previously in this guide, on how to obtain the Z88 Patch.

Available by default when BBC BASIC(Z80) is used with Z88 ROM release V4.3 and later.

Syntax

```
PLOT <numeric>,<numeric>,<numeric>
```

Associated Keywords

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

POINT

This function returns the state of the pixel at the specified location, as 0 (unlit) or 1 (lit). If the specified point is outside the graphics window (taking into account the position of the graphics origin), or if MODE 0 is selected, the value -1 is returned.

This statement is available when installed with Z88 Patch via CHAIN command. See "Installing Z88 Patch" section previously in this guide, on how to obtain the Z88 Patch.

Available by default when BBC BASIC(Z80) is used with Z88 ROM release V4.3 and later.

Syntax

```
<n-var>=POINT(<numeric>,<numeric>)
```

Associated Keywords

PLOT, DRAW, MOVE, GCOL

POS

A function returning the horizontal position of the cursor on the screen. The left hand column is 0 and the right hand column is one less than the width of the display.

```
        X=POS
```

COUNT will tell you the print head position of the printer. It is an uncertain indicator of the horizontal position of the cursor on the screen. (See the keyword COUNT for details.)

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

Syntax

```
<n-var>=POS
```

Associated Keywords

COUNT, TAB, VPOS

PRINT

A statement which prints characters on the screen. You can also echo the characters to the printer. Printer echo on the Z88 is controlled with the following key sequences:

□ + P	Printer echo On. All characters sent subsequently to the screen are echoed to the printer
□ - P	Printer echo 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
```

Initially numeric items are printed in decimal. If a tilde (~) is encountered in the print list, the numeric items which follow it are printed in hexadecimal. If a comma or a semi-colon is encountered further down the print list, the format reverts to decimal.

```
                    z10
                    012345678901234567890123456789
PRINT ~10 58,58     ,,,,,,,,,A,,,,,,,,3A,,,,,,,,58
```

A comma (,) causes the cursor to TAB to the beginning of the next print zone unless the cursor is already at the start of a print zone. A semi-colon causes the next and following items to be printed on the same line immediately after the previous item. This 'no-gap' printing continues until a comma (or the end of the print list) is encountered. An apostrophe (') will force a new line. TAB(X) and TAB(Y,Z) can also be used at any position in the print line to position the cursor.

```
                    z10
                    012345678901234567890123456789
PRINT "HELLO",24.2  HELLO     ,,,,,,24.2
PRINT "HELLO";24.2  HELLO24.2
PRINT ;2 5 4.3,2    254.3     ,,,,,,,,,2
PRINT "HELLO"'2.45  HELLO
                    ,,,,,,2.45
```

Unlike most other versions of BASIC, a comma at the end of the print list will not suppress the new line and advance the cursor to the next zone. If you wish to split a line over two or more PRINT statements, end the previous print list with a semicolon and start the following list with a comma or end the line with a comma followed by a semicolon.

```
                    z10
                    012345678901234567890123456789
PRINT "HELLO" 12;   HELLO,,,,,,,,12,,,,,,,,,,23.67
PRINT ,23.67
```

or

```
     PRINT "HELLO" 12,;
     PRINT 23.67
```

Printing a string followed by a numeric effectively moves the start of the print zones towards the right by the length of the string. This displacement continues until a comma is encountered.

```
                    z10
                    012345678901234567890123456789
PRINT "HELLO"12 34  HELLO,,,,,,,,12,,,,,,,,34
PRINT "HELLO"12,34  HELLO,,,,,,,,12     ,,,,,,,,34
```

Print Format Control

Although PRINT USING is not implemented in BBC BASIC, similar control over the print format can be obtained. The overall width of the print zones and print field, the number of figures or decimal places and the print format may be controlled by setting the print variable, @%, to the appropriate value. The print variable (@%) comprises 4 bytes and each byte controls one aspect of the print format. @% can be set equal to a decimal integer, but it is easier to use hexadecimal, since each byte can then be considered separately.

```
     @%=&SSNNPPWW
```

Byte	Range	Default	Purpose
SS	00-01	00	STR$ Format Control
NN	00-02	00	Format Selection
PP	??-??	09	Number of Digits Printed
WW	00-0F	0A(10)	Zone and Print Field Width

STR$ Format Control - SS

SS effects the format of the string generated by the STR$ function. If SS is 01 the string will be generated according to the format set by @%, otherwise the G9 format will be used.

Format Selection - NN

Byte 2 selects the general format as follows:

00	General Format (G).
01	Exponential Format (E).
02	Fixed Format (F).

G Format Numbers that are integers are printed as such.
Numbers in the range 0.1 to 1 will be printed as such.
Numbers less than 0.1 will be printed in E format.
Numbers greater than the range set by Byte 1 will be printed in E format. In which case, the number of digits printed will still be controlled by Byte 1, but according to the E format rules.

The earlier examples were all printed in G9 format.

E Format Numbers are printed in the scientific (engineering) notation.

F Format Numbers are printed with a fixed number of decimal places.

Number of Digits - PP

PP controls the number of digits printed in the selected format. The number is rounded (NOT truncated) to this size before it is printed. If PP is set outside the range allowed for by the selected format, it is taken as 9. The effect of PP differs slightly with the various formats.

Format	Range	Control Function
G	01-0A	The maximum number of digits which can be printed, excluding the decimal point, before changing to the E format. `                    01234567890123456789` `&030A - G3z10` `(00'00'03'0A)` `PRINT 1000.31       ,,,,,,,1E3` `PRINT 1016.31       ,,,,1.02E3` `PRINT 10.56         ,,,,,,10.6`
E	01-FF	The total number of digits to be printed excluding the decimal point and the digits after the E. Three characters or spaces are always printed after the E. If the number of significant figures called for is greater than 10, then trailing zeros will be printed. `01030A - E3z10` `(00'01'03'0A)` `                    01234567890123456789` `PRINT 10.56         ,,1.06E1` `&010F0A - E15z10` `(00'01'0F'0A)` `                    01234567890123456789` `PRINT 10.56         1.05600000000000E1`
F	00-0A	The number of digits to be printed after the decimal point. `&02020A - F2z10` `(00'02'02'0A)` `                    01234567890123456789` `PRINT 10.56         ,,,,,10.56` `PRINT 100.5864      ,,,,100.59` `PRINT .64862        ,,,,,,0.65`

Zone Width - WW

Byte 0 sets the width of the print zones and field.

```
&020208 - F2z8
(00'00'02'08)
```

followed by

```
&020206 - F2z6
(00'02'02'06)
                    01234567890123456789
PRINT 10.2,3.8      ,,,10.20,,,,3.80
PRINT 10.2,3.8      ,10.20,,3.80
```

Changing the Print Control Variable

It is possible to change the print control variable (@%) within a print list by using the function:

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

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

```
     PRINT FN_pformat(&90A) x FN_pformat(&2020A) y
```

will print x in G9z10 format and y in F2z10 format.

Examples

```
G9z10                    G2z10
&00090A                  &00020A
012345678901234          012345678901234
1111.11111               ,,,,,1.1E3
13.7174211               ,,,,,,,,14
,1.5241579               ,,,,,,,1.5
1.88167642E-2            ,,,,1.9E-2
2.09975158E-3            ,,,,2.1E-3

F2z10                    E2z10
&02020A                  &0102A
012345678901234          012345678901234
,,,1111.11               ,,,1.1E3
,,,,,13.72               ,,,1.4E1
,,,,,,1.52               ,,,1.5E0
,,,,,,0.02               ,,,1.9E-2
,,,,,,0.00               ,,,2.1E-3
```

The results obtained by running the following example program show the effect of changing the zone width. The results for zone widths of 5 and 10 (&0A) illustrate what happens when the zone width is too small for the number to be printed properly. The example also illustrates what happens when the number is too large for the chosen precision.

```
 10 test=7.8123
 20 FOR i=5 TO 25 STEP 5
 30   PRINT
 40   @%=&020200+i
 50   PRINT "@%=&000";~@%
 60   PRINT STRING$(3,"0123456789")
 70   FOR j=1 TO 10
 80     PRINT test^j
 90   NEXT
100   PRINT '
110 NEXT
120 @%=&90A
```

```
&00020205
012345678901234567890123456789
 7.81
61.03
476.80
3724.91
29100.11
227338.75
1776038.54
13874945.89
1.083952398E8
8.46816132E8

&0002020A
012345678901234567890123456789
      7.81
     61.03
    476.80
   3724.91
  29100.11
 227338.75
1776038.54
13874945.89
1.083952398E8
8.46816132E8
```

```
&0002020F
012345678901234567890123456789
           7.81
          61.03
         476.80
        3724.91
       29100.11
      227338.75
     1776038.54
    13874945.89
   1.083952398E8
    8.46816132E8

&00020214
012345678901234567890123456789
                7.81
               61.03
              476.80
             3724.91
            29100.11
           227338.75
          1776038.54
         13874945.89
       1.083952398E8
        8.46816132E8

&00020219
012345678901234567890123456789
                     7.81
                    61.03
                   476.80
                  3724.91
                 29100.11
                227338.75
               1776038.54
              13874945.89
            1.083952398E8
             8.46816132E8
```

Syntax

```
PRINT {[TAB(<numeric>[,<numeric>])][SPC(<numeric>]
       ['][,][;][~][<str>|<numeric>]}
```

Associated Keywords

PRINT#, TAB, POS, STR$, WIDTH, INPUT, VDU

PRINT# (P.#)

A statement which writes the internal form of a value out to a data file (or device).

```
PRINT#E,A,B,C,D$,E$,F$
PRINT#4,prn$
```

The format of the variables as written to the file differs from the format used on the BBC Micro. All numeric values are written as five bytes of binary real data (see the Annex entitled ‘Format of Program and Variables in Memory’). Strings are written as the bytes in the string (in the correct order) plus a carriage return.

Before you use this statement, you must normally have opened a file using OPENOUT or OPENUP.

You can use PRINT# to write data (numbers and strings) to a data file in the 'standard' manner. If you wish to 'pack' your data in a different way, you should use BPUT#. You can use PRINT# and BPUT# together to mix or modify the data format. For example, if you wish to write a 'compatible' text file, you could PRINT# the string and BPUT# a line-feed. This would write the string followed by a carriage-return and a line-feed to the file.

Remember, with BBC BASIC(Z80) the format of the file is completely under your control.

Syntax

```
PRINT#<numeric>{,<numeric>|<str>}
```

Associated Keywords

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

PROC

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

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

A procedure may be defined with any number of parameters of any type.

A procedure definition is terminated by ENDPROC.

A procedure does not have to be declared before it is called.

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

```
 10 INPUT"Number of discs "F
 20 PROC_hanoi(F,1,2,3)
 30 END
 40 :
 50 DEF PROC_hanoi(A,B,C,D)
 60 IF A=0 THEN ENDPROC
 70 PROC_hanoi(A-1,B,D,C)
 80 PRINT"Move disk ";A" from ";B" to ";C
 90 PROC_hanoi(A-1,D,C,B)
100 ENDPROC
```

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

Syntax

`PROC<name>[(<exp>{,<exp>})]`

Associated Keywords

DEF, ENDPROC, LOCAL

PTR#

A pseudo-variable allowing the random-access pointer of the file whose file handle is its argument to be read and changed.

```
      PTR#F=PTR#F+5 :REM Move pointer to next number
      PTR#F=recordnumber*recordlength
```

Reading or writing (using BGET#, BPUT#, INPUT# or PRINT#) takes place at the current position of the pointer. The pointer is automatically updated following a read or write operation.

You can use PTR# to select which item in a file is to be read or written to next. In a random file (see the section on BBC BASIC(Z80) Files) you use PTR# to select the record you wish to read or write.

If you wish to move about in a file using PTR# you will need to know the precise format of the data in the file.

A file opened with OPENUP may be extended by setting its pointer to its end (PTR#fnum = EXT#fnum) and then writing to it. If you do this, you must remember to CLOSE the file when you have finished with it in order to update the directory entry.

By using PTR# you have complete control over where you read and write data in a file. This is a simple concept, but it may initially be difficult to grasp its many ramifications. The BBC BASIC(Z80) Files section has a number of examples of the use of PTR#.

PTR#-1 Returns the number of file handles still available for the entire Z88 (not just BBC BASIC(Z80)) and the ROM release number.

If you are going to display this information, you will need to do so in hexadecimal because the one (4 byte) number contains two items of information. For example:

```
      PRINT ~PTR#-1
            5A0004
```

The last 3 digits (least significant 2 bytes) are the ROM release number. The first 2 digits (most significant 2 bytes) are the number of files handles still available for use by the filing system (&5A=90).

Syntax

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

Associated Keywords

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

PUT

A statement to output data to an output port. Full 16bit addressing is available.

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

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

It is strongly recommended that you do not try to control the Z88's hardware with this command - a mistake can be disastrous. However, if you insist on directly accessing the Z88's hardware, you will need to study the Z88 Developers' Notes 'Manipulating the Blink Registers' section. This guide is available online via cambridgez88.jira.com/wiki/spaces/DN.

This command is an addition to the original language specification and it cannot be guaranteed to remain unchanged in future releases.

Syntax

```
PUT <numeric>,<numeric>
```

Associated Keywords

GET

RAD

A function which converts degrees to radians.

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

Unlike humans, BBC BASIC(Z80) wants angles expressed in radians. You can use this function to convert an angle expressed in degrees to radians before using one of the angle functions (SIN, COS, etc).

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

See COS, SIN and TAN for further examples of the use of RAD.

Syntax

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

Associated Keywords

DEG

READ

A statement which will assign to variables values read from the DATA statements in the program. Strings must be enclosed in double quotes if they have leading spaces or contain commas.

```
      READ C,D,A$
```

In many of your programs, you will want to use data values which do not change frequently. Because these values are subject to some degree of change, you won't want to use constants. On the other hand, you won't want to input them every time you run the program either. You can get the best of both worlds by declaring these values in DATA statements at the beginning or end of your program and READing them into variables in your program.

A typical use for DATA and READ is a name and address list. The addresses won't change very often, but when they do you can easily amend the appropriate DATA statement.

See DATA for more details and an example of the use of DATA and READ.

Syntax

```
READ <n-var>|<s-var>{<n-var>|<s-var>}
```

Associated Keywords

DATA, RESTORE

REM

A statement that causes the rest of the line to be ignored thereby allowing comments to be included in a program.

You can use the REM statement to put remarks and comments in to your program to help you remember what the various bits of your program do. BBC BASIC(Z80) completely ignores anything on the line following a REM statement.

You will be able to get away without including any REMarks in simple programs. However, if you go back to a lengthy program after a couple of months you will find it very difficult to understand if you have not included any REMs.

If you include nothing else, include the name of the program, the date you last revised it and a revision number at the start of your program.

```
10 REM WSCONVERT REV 2.30
20 REM 5 AUG 84
30 REM Converts 'hard' carriage-returns to 'soft'
40 REM ones in preparation for use with WS.
```

Syntax

```
REM any text
```

Associated Keywords

None

RENUMBER (REN.)

A command which will renumber the lines and correct the cross references inside a program.

The options are as for AUTO, but increments of greater than 255 are allowed.

You can specify both the new first number (n1) and/or the step size (s). The default for both the first number and the step size is 10. The two parameters should be separated by a comma or a hyphen.

RENUMBER
RENUMBER n1
RENUMBER n1,s
RENUMBER ,s

For example:

`RENUMBER`	First number 10, step 10
`RENUMBER 1000`	First number 1000, step 10
`RENUMBER 1000-5`	First number 1000, step 5
`RENUMBER ,5`	First number 10, step 5
`RENUMBER -5`	First number 10, step 5

RENUMBER produces error messages when a cross reference fails. The line number containing the failed cross-reference is renumbered and the line number in the error report is the new line number.

If you renumber a line containing an ON GOTO/GOSUB statement which has a calculated line number, RENUMBER will correctly cope with line numbers before the calculated line number. However, line numbers after the calculated line number will not be changed.

In the following example, the first two line numbers would be renumbered correctly, but the last two would be left unchanged.

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

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

Syntax

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

Associated Keywords

LIST

REPEAT (REP.)

A statement which is the starting point of a REPEAT...UNTIL loop. A single REPEAT may have more than one UNTIL, but this is bad practice.

The purpose of a REPEAT...UNTIL loop is to make BBC BASIC(Z80) repeat a set number of instructions until some condition is satisfied.

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

X=0
REPEAT
  X=X+10
  PRINT "What do you think of it so far?"
UNTIL X>45
```

You must not exit a REPEAT...UNTIL loop with a GOTO. If you jump out of a loop with a GOTO (How could you!!!) you should jump back in. If you must jump out of the loop, you should use UNTIL TRUE to 'pop' the stack. For (a ghastly) example:

```
 10 i=1
 20 REPEAT: REM Print 1 to 100 unless
 30   I=I+1: REM interrupted by the
 40   PRINT i: REM space bar being pressed
 50   x=INKEY(0): REM Get a key
 60   IF x=32 THEN 110:REM exit if <SPACE>
 70 UNTIL i=100
 80 PRINT "****"
 90 END
100 :
110 UNTIL TRUE: REM Pop the stack
120 PRINT "Forced exit":REM Carry on with program
130 FOR j=1000 TO 1005
140   PRINT j
150 NEXT
```

See the keyword UNTIL for ways of using REPEAT...UNTIL loops to replace unconditional GOTOs for program looping.

See the sub-section on ‘Program Flow Control’ in the ‘General Information’ section for more details on the working of the program stack.

Syntax

REPEAT

Associated Keywords

UNTIL

REPORT (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.

Syntax

REPORT

Associated Keywords

ERR, ERL, ON ERROR

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.

Syntax

```
RESTORE [<l-num>]
RESTORE [(<numeric>)]
```

Associated Keywords

READ, DATA

RETURN (R.)

A statement causing a RETURN to the statement after the most recent GOSUB statement.

You use RETURN at the end of a subroutine to make BBC BASIC(Z80) return to the place in your program which originally 'called' the subroutine.

You may have more than one return statement in a subroutine, but it is preferable to have only one entry point and one exit point (RETURN).

Try to structure your program so you don't leave a subroutine with a GOTO. If you do, you should always return to the subroutine and exit via the RETURN statement. If you insist on using GOTO all over the place, you will end up confusing yourself and maybe confusing BBC BASIC(Z80) as well. The sub-section on ‘Program Flow Control’ explains why.

Syntax

RETURN

Associated Keywords

GOSUB, ON GOSUB

RIGHT$

A string function which returns the right 'num' characters of the string. If there are insufficient characters in the string then all are returned.

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$="BBC BASIC(Z80)"
20 FOR i=3 TO LEN(name$)
30   PRINT RIGHT$(name$,i)
40 NEXT
```

would print

```
80)
Z80)
(Z80)
C(Z80)
IC(Z80)
SIC(Z80)
ASIC(Z80)
BASIC(Z80)
 BASIC(Z80)
C BASIC(Z80)
BC BASIC(Z80)
BBC BASIC(Z80)
```

Syntax

`<s-var>=RIGHT$(<str>,<numeric>)`

Associated Keywords

LEFT$, MID$

RND

A function which returns a random number. The type and range of the number returned depends upon the optional parameter.

`RND`	returns a random integer 1 - &FFFFFFFF.
`RND(n)`	returns an integer in the range 1 to n (n>1).
`RND(1)`	returns a real number in the range 0.0 to .99999999.

If n is negative the pseudo random sequence generator is set to a number based on n and n is returned.

If n is 0 the last random number is returned in RND(1) format.

```
X=RND(1)
X%=RND
N=RND(6)
```

The random number generator is initialised by RUN (or CHAIN). Consequently, RND will return zero until the RUN (or CHAIN) command is first issX=RND(1)

Syntax

`<n-var>=RND[(<numeric>)]`

Associated Keywords

None

RUN

Start execution of the program.

```
RUN
```

RUN is a statement and so programs can re-execute themselves.

All variables except @% to Z% are CLEARed by RUN.

If you want to start a program without clearing all the variables, you can use the statement

```
GOTO nnnn
```

where nnnn is the number of the line at which you wish execution of the program to start.

RUN "filename" can be used as an alternative to CHAIN "filename".

Syntax

```
RUN [<str>]
```

Associated Keywords

NEW, OLD, LIST, CHAIN

SAVE (SA.)

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

```
    SAVE "FRED.BBC"
    SAVE A$
```

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

The name (file-specifier) must follow the normal naming conventions of Z88 filing system for folders and filenames. See the Operating System Interface section for a description of a file-specifier (name).

Syntax

```
SAVE <str>
```

Associated Keywords

LOAD, CHAIN

SGN

A function returning -1 for negative argument, 0 for zero argument and +1 for positive argument.

```
X=SGN(Y)
result=SGN(answer)
```

You can use this function to determine whether a number is positive, negative or zero.

SGN returns:

+1	for positive numbers
0	for zero
-1	for negative numbers

Syntax

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

Associated Keywords

ABS

SIN

A function giving the sine of its radian argument.

```
    X=SIN(Y)
```

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

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

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

```
    Y=SIN(RAD(degree_angle))
```

Syntax

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

Associated Keywords

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

SOUND

A statement which generates sounds using the internal loudspeaker.

Not available on the Z88

SPC

A statement which prints a number of spaces to the screen (or currently selected console output stream). The argument specifies the number of spaces (up to 255) to be printed.

SPC can only be used within a PRINT or INPUT statement.

```
PRINT DATE;SPC(6);SALARY
INPUT SPC(10) "What is your name ",name$
```

Syntax

```
PRINT SPC(<numeric>)
INPUT SPC(<numeric>)
```

Associated Keywords

TAB, PRINT, INPUT

SQR

A function returning the square root of its argument.

```
        X=SQR(Y)
```

If you attempt to calculate the square root of a negative number, a '-ve root' error will occur. You could use error trapping to recover from this error, but it is better to check that the argument is not negative before using the SQR function.

Syntax

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

Associated Keywords

None

STEP (S.)

Part of the FOR statement, this optional section specifies step sizes other than 1.

```
      FOR i=1 TO 20 STEP 5
```

The step may be positive or negative. STEP is optional; if it is omitted, a step size of +1 is assumed.

You can use this optional part of the FOR...TO...STEP...NEXT structure to specify the amount by which the FOR...NEXT loop control variable is changed each time round the loop. In the example below, the loop control variable, 'cost' starts as 20, ends at 5 and is changed by -5 each time round the loop.

```
10 FOR cost=20 TO 5 STEP -5
20   PRINT cost,cost*1.15
30 NEXT
```

Syntax

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

Associated Keywords

FOR, TO, NEXT

STOP

Syntactically identical to END, STOP also prints a message to the effect that the program has stopped.

You can use STOP at various places in your program to aid debugging. If your program is going wrong, you can place STOP commands at various points to see the path taken by your program. (TRACE is generally a more useful aid to tracing a program's flow unless you are using formatted screen displays.)

Once your program has STOP’ed you can investigate the values of the variables to find out why things happened the way they did.

STOP DOES NOT CLOSE DATA FILES. If you use STOP to exit a program for debugging, CLOSE all the data files before RUNning the program again. If you don't you will get some most peculiar error messages.

Syntax

STOP

Associated Keywords

END

STR$

A string function which returns the string form of the numeric argument as it would have been printed.

If the most significant byte of @% is not zero, STR$ uses the current @% description when generating the string. If it is zero (the initial value) then the G9 format (see PRINT) is used.

If STR$ is followed by ~ (tilde) then a hexadecimal conversion is carried out.

```
    A$=STR$(PI)
    B$=STR$~(100) :REM B$ will be "64"
```

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

Syntax

```
<s-var>=STR$[~](<numeric>)
```

Associated Keywords

VAL, PRINT

STRING$

A function returning N concatenations of a string.

```
A$=STRING$(N,"hello")
B$=STRING$(10,"-*-")
C$=STRING$(Z%,S$)
```

You can use this function to print repeated copies of a string. It is useful for printing headings or underlinings. The last example for PRINT uses the STRING$ function to print the column numbers across the page. For example,

```
PRINT STRING$(4,"-=*=-")
```

would print

```
-=*=--=*=--=*=--=*=-
```

and

```
PRINT STRING$(3,"0123456789")
```

would print

```
012345678901234567890123456789
```

Syntax

```
<s-var>=STRING$(<numeric>,<str>)
```

Associated Keywords

None

TAB

A keyword available in PRINT or INPUT.

There are two versions of TAB: TAB(X) and TAB(X,Y) and they are effectively two different keywords.

TAB(X) is a printer orientated statement. The number of printable characters since the last new-line (COUNT) is compared with X. If X is equal or greater than COUNT, sufficient spaces to make them equal are printed. These spaces will overwrite any characters which may already be on the screen. If X is less than COUNT, a new-line will be printed first.

TAB(X,Y) is a screen orientated statement. It will move the cursor on the screen to character cell X,Y (column X, row Y) if possible. No characters are overwritten and COUNT is NOT updated. Consequently, a TAB(X,Y) followed by a TAB(X) will give unpredictable (at first glance) results.

The leftmost column is column 0 and the top of the screen is row 0.

```
PRINT TAB(10);A$
PRINT TAB(X,Y);B$
```

Syntax

```
PRINT TAB(<numeric>[,<numeric>])
INPUT TAB(<numeric>[,<numeric>])
```

Associated Keywords

POS, VPOS, PRINT, INPUT

TAN (T.)

A function giving the tangent of its radian argument.

```
      X = TAN(Y)
```

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

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

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

```
      Y=TAN(RAD(degree_angle))
```

Syntax

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

Associated Keywords

COS, SIN, ACS, ATN, ASN, DEG, RAD

THEN (TH.)

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

```
IF a=b THEN 320
```

A 'star' (*) command.

```
IF a=b THEN *DIR
```

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 BBC BASIC(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 THEN required on this line
=num/256
```

Syntax

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

Associated Keywords

IF, ELSE

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 BBC BASIC(Z80)'s internal clock. The value of the clock is returned in centi-seconds (one-hundredths of a second) and it is quite accurate. A delay loop such as

```
REPEAT UNTIL TIME = T
```

is likely to fail. The compound condition test

```
REPEAT UNTIL TIME >= T
```

should always be used.

On starting BBC BASIC(Z80), TIME may be found to be 'stuck' at a large value. To cure this, it should be initialised (TIME=0), for example.

The following example is a simple program to provide a 24 hour clock. Lines 20 to 40 get the correct time, lines 50 and 60 calculate the number of centi-seconds and set TIME, and lines 110 to 130 convert the value in TIME to hours, minutes and seconds. Line 90 stops the time being printed unless it has changed by at least one second.

```
 10 CLS
 20 INPUT "HOURS ",H
 30 INPUT "MINUTES ",M
 40 INPUT "SECONDS ",S
 50 PRINT "PUSH ANY KEY TO SET THE TIME ";:X=GET
 60 TIME=((H*60+M)*60+S)*100
 70 T=0
 80 REPEAT
 90   IF TIME DIV 100=T DIV 100 THEN 150
100   T=TIME
110   S=(T DIV 100) MOD 60
120   M=(T DIV 6000) MOD 60
130   H=(T DIV 360000) MOD 24
140   PRINT TAB(0,23) H;":";M;":";S;
150 UNTIL FALSE
```

Syntax

TIME=<numeric>
<n-var>=TIME

Associated Keywords

TIME$

TIME$

A 24 character long string pseudo-variable which reads and sets the system clock. The format of the character string is:

```
Day date Month year, hh:mm:ss
```

Where:

`Day`	is the day of the week in full (Monday, Tuesday, etc)
`date`	is the day of the month (1st, 2nd, etc)
`Month`	is the month name (January, February, etc)
`year`	is the year (1986, 1987, etc)
`hh`	is hours (00 to 23)
`mm`	is minutes (00 to 59)
`ss`	is seconds (00 to 59)

```
clock$=TIME$
TIME$=”Sunday 2nd February 1986, 18:33:30”
```

The date or both date and time may be set as shown below:

```
TIME$=”Day date Month year”
TIME$=”Day date Month year, hh:mm:ss”
```

When setting the clock, the day of the week may be omitted.

The Z88 is remarkable in that it will accept almost any date format. The following examples demonstrate this to some extent. Neither of them conforms to the default format, but they work. The first example below sets only the date and the second sets the date and the time.

```
TIME$=”2 Feb 86”
clock$=”Mon 03 Feb 1986,22:31:15”
TIME$=clock$
```

In general, you will find it easier to use []T (Clock popdown) to set the date and time.

If the Z88 cannot make sense of the string, or the day/date is impossible, a ‘Bad syntax’ error will be reported.

Syntax

```
TIME$=<str>
<s-var>=TIME$
```

Associated Keywords

TIME

TO

The part of the FOR ... TO ... STEP statement which introduces the terminating value for the loop. When the loop control variable exceeds the value following 'TO' the loop is terminated. For example,

```
10 FOR i=1 TO 5 STEP 1.5
20   PRINT i
30   NEXT
40 PRINT "**********"
50 PRINT i
```

will print

```
         1
       2.5
         4
**********
       5.5
```

Irrespective of the initial value of the loop control variable and the specified terminating value, the loop will execute at least once. For example,

```
10 FOR i= 20 TO 10
20   PRINT i
30 NEXT
```

will print

```
    20
```

Syntax

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

Associated Keywords

FOR, NEXT, STEP

TOP

A function which returns the value of the first free location after the end of the current program.

The length of your program is given by TOP-PAGE.

```
PRINT TOP-PAGE
```

Syntax

<n-var>=TOP

Associated Keywords

PAGE, HIMEM, LOMEM

TRACE (TR.)

TRACE ON causes the interpreter to print executed line numbers when it encounters them.

TRACE X sets a limit on the size of line numbers which will be printed out. Only those line numbers less than X will appear. If you are careful and place all your subroutines at the end of the main program, you can display the main structure of the program without cluttering up the trace with the subroutines.

TRACE OFF turns trace off. TRACE is also turned off if an error is reported or you press <Esc>.

Line numbers are printed as the line is entered. For example,

```
10 FOR Z=0 TO 2:Q=Q*Z:NEXT
20 END
```

would trace as

```
[10] [20] >_
```

whereas

```
10 FOR Z=0 TO 2
20   Q=Q*Z:NEXT
30 END
```

would trace as

```
[10] [20] [20] [20] [30] >_
```

and

```
10 FOR Z=0 TO 3
20 Q=Q*Z
30 NEXT
40 END
```

would trace as

```
[10] [20] [30] [20] [30] [20] [30] [40] >_
```

Syntax

```
TRACE ON|OFF|<l-num>
TRACE ON|OFF|(<numeric>)
```

Associated Keywords

None

TRUE

A function returning the value -1.

```
10 flag=FALSE
....
100 IF answer$=correct$ flag=TRUE
....
150 IF flag PROC_got_it_right ELSE PROC_wrong
```

BBC BASIC(Z80) does not have true Boolean variables. Instead, numeric variables are used and their value is interpreted in a 'logical' manner. A value of 0 is interpreted as false and NOT FALSE (in other words, NOT 0 (= -1)) is interpreted as TRUE.

In practice, any value other than zero is considered TRUE. This can lead to confusion; see the keyword NOT for details.

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

Syntax

```
<n-var>=TRUE
```

Associated Keywords

FALSE

UNTIL (U.)

The part of the REPEAT ... UNTIL structure which signifies its end.

You can use a REPEAT...UNTIL loop to repeat a set of program instructions until some condition is met.

If the condition associated with the UNTIL statement is never met, the loop will execute for ever. (At least, until `[Esc]` is pressed or some other error occurs.)

The following example will continually ask for a number and print its square. The only way to stop it is by pressing `[Esc]` or forcing a 'Too big' error.

```
10 z=1
20 REPEAT
30   INPUT "Enter a number " num
40   PRINT "The square of ";num;" is ";num*num
50 UNTIL z=0
```

Since the result of the test z=0 is ALWAYS FALSE, we can replace z=0 with FALSE. The program now becomes:

```
20 REPEAT
30   INPUT "Enter a number " num
40   PRINT "The square of ";num;" is ";num*num
50 UNTIL FALSE
```

This is a much neater way of unconditionally looping than using a GOTO statement. The program executes at least as fast and the section of program within the loop is highlighted by the indentation.

See the keyword REPEAT for more details on REPEAT...UNTIL loops. See the Variables sub-section for more details on Boolean variables and the keyword AND for logical tests and their results.

Syntax

`UNTIL <t-cond>`

Associated Keywords

REPEAT

USR

A function which allows a machine code routine to return a value directly.

USR calls the machine code subroutine whose start address is its argument. The processor's A, B, C, D, E, F, H and L registers are initialised to the least significant words of A%, B%, C%, D%, E%, F%, H% and L% respectively (see also CALL).

USR provides you with a way of calling a machine code routine which is designed to return one integer value. Parameters are passed via the processor's registers and the machine code routine returns a 32-bit integer result composed of the processor's HLH'L' registers. The HL register forms the most significant word of the result.

```
      X=USR(lift_down)
```

Unlike CALL, USR returns a result. Consequently, you must assign the result to a variable. It may help your understanding if you look upon CALL as the machine code equivalent to a PROCedure and USR as the equivalent to Function.

Syntax

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

Associated Keywords

CALL

VAL

A function which converts a character string representing a number into numeric form.

```
      X=VAL(A$)
```

VAL makes the best sense it can of its argument. If the argument starts with numeric characters (with or without a preceding sign), VAL will work from left to right until it meets a non numeric character. It will then 'give up' and return what it has got so far. If it can't make any sense of its argument, it returns zero.

For example,

```
      PRINT VAL("-123.45.67ABC")
```

would print

```
      -123.45
```

and

```
      PRINT VAL("A+123.45")
```

would print

```
      0
```

VAL will NOT work with hexadecimal numbers. You must use EVAL to convert hexadecimal number strings.

Syntax

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

Associated Keywords

STR$, EVAL

VDU (V.)

A statement which takes a list of numeric arguments and sends their least-significant bytes as characters to the display.

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 screen.

It also differs from PRINT CHR$(n) in that you cannot use it to end codes or characters to the printer.

Syntax

```
VDU <numeric>{,|;<numeric>}[;]
```

Associated Keywords

CHR$

VPOS

A function returning the vertical cursor position. The top of the screen is line 0.

```
        Y=VPOS
```

You can use VPOS in conjunction with POS to return to the present position on the screen after printing a message somewhere else. The example below is a procedure for printing a 'status' message at line 23. The cursor is returned to its previous position after the message has been printed.

```
1000 DEF PROC_message(message$)
1010 LOCAL x,y
1020 x=POS
1030 y=VPOS
1040 PRINT TAB(0,23) CHR$(7);message$;
1050 PRINT TAB(x,y);
1060 ENDPROC
```

Syntax

```
<n-var>=VPOS
```

Associated Keywords

POS

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 new line after n MOD 256 characters have been printed.

WIDTH also affects the output to the printer.

Syntax

```
WIDTH <numeric>
```

Associated Keywords

COUNT

The Screen Driver

Introduction

As with the BBC Micro, the VDU command may be used to send characters to the Z88 Screen Driver. Since the Z88 is very different to the BBC Micro, there are considerable differences in the action of the VDU commands.

In many ways the VDU command is similar to a PRINT CHR$(num) command, but it involves less typing and a number of characters may be easily sent. It is most often used for sending characters that control the action of the Screen Driver.

The VDU statement takes a list of numeric arguments (constants or variables) and sends their least significant bytes as characters to the screen.

A 16 bit value (word) can be sent if the value is followed by a semi-colon. It is sent as a pair of characters, the least significant byte first.

You cannot use the VDU command to send control sequences to a device (`:PRT.0`, for example).

This section lists and briefly describes the control codes and their function.

Char	Number	Meaning
NUL	0	Null - it does nothing
SOH	1	Escape character for screen. Preceed special functions. See the 'Description of Escape Sequences' sub-section
BEL	7	Bell - make a short 'beep'
BS	8	Move the text cursor backwards one character
HT	9	Move the text cursor forwards one character
LF	10	Move the text cursor down one line
VT	11	Move the text cursor up one line
FF	12	Clear the text area - identical to CLS
CR	13	Move the text cursor to the start of the current line
DEL	127	Black square

Escape Sequences

The character ASCII code 1 (SOH) is used to introduce special character combinations called 'Escape Sequences' which print special characters on the screen, toggle the current printing mode, position the cursor on the screen, define windows and perform various display control functions.

Description of VDU Codes

VDU 0

Does nothing. In other words, it is ignored.

VDU 1

This is the ‘Escape’ code which introduces numerous screen printing and control sequences which are described in the ‘Description of Escape Sequences’ sub-section.

VDU 7

VDU 7 causes a short ‘beep’ from the speaker.

VDU 8

VDU 8 moves the cursor one character to the left. If the cursor was at the start of a line, it moves to the end of the previous line (right edge of the text window). If it was also at the top line of the text window, and scroll is enabled, the window contents scrolls down. If a window has been defined, the cursor is constrained to remain within the window.

VDU 9

VDU 9 moves the cursor down one line. If the cursor was on the bottom line of the window and scroll is enabled, the window contents scrolls up. If a window has been defined, the cursor is constrained to remain within the window.

VDU 10

VDU 10 moves the cursor down one line. If the cursor was on the bottom line of the window and scroll is enabled, the window contents scrolls up. If a window has been defined, the cursor is constrained to remain within the window.

VDU 12

VDU 12 clears the text window and moves the cursor to the top left corner of the window; it is identical to CLS command.

VDU 13

VDU 13 moves the cursor to the left edge of the window, but does not move it vertically. If a window has been defined, the cursor is constrained to remain within the window.

VDU 127

Prints a black square.

Description of Escape Sequences

Introduction

The character ASCII code 1 is used as an 'Escape' character to introduce special code sequences that:

- Generate special characters
- Position the cursor
- Set text attributes
- Define text justification
- Define windows
- Provide miscellaneous screen control operations

The escape character is followed by

> A single character

or

> A sequence of codes, the first of which specifies the number of parameters (bytes) to follow.

The parameter count may take one of two forms.

> It may be the ASCII code for a single decimal digit, "1" (49) to "9" (57).

or

> It may be a binary number with bit 7 set (128 added to the number).

Most escape sequences have less than 10 following parameters, and the first method of specifying the parameter count is generally used. Numbers, not characters, are used as parameters in the VDU command, so the ASCII character '2', for example, would appear as 50 (its ASCII value), or as ASC"2".

Where appropriate, the escape sequences are shown in 2 ways (both of which send identical code sequences to the screen driver). The first is the easiest to remember since it uses characters that are related to the name of the function the sequence is performing (a mnemonic). However, this method uses the ASC function to convert the characters into their ASCII codes. The second method uses the actual codes sent to the screen driver. Whilst this is more difficult to remember, it involves less typing.

The following table lists the special display characters such as 'INDEX', 'MENU', 'HELP', etc. The 'Code' column shows both ASC code (where applicable) and the Screen driver mnemonic (see Developers' Notes).

Special characters

Code	*Sequence*	*Value (H)*	*Description*	*Width*	*Boldable*
' '	VDU 1,32	&20	Exact symbol	1	
'!'	VDU 1,33	&21	Bell symbol	3	
'''	VDU 1,39	&27	Grave accent	1	yes
'*'	VDU 1,42	&2A	Square	1	yes
'+'	VDU 1,43	&2B	Diamond	1	yes
'-'	VDU 1,45	&2D	SHIFT symbol	3	
'|'	VDU 1,124	&7D	Vertical unbroken bar	1	yes
SD_SPC	VDU 1,224	&E0	SPACE symbol	3	
SD_ENT	VDU 1,225	&E1	ENTER symbol	3	
SD_TAB	VDU 1,226	&E2	TAB symbol	3	
SD_DEL	VDU 1,227	&E3	DEL symbol	3	
SD_ESC	VDU 1,228	&E4	ESC symbol	3	
SD_MNU	VDU 1,229	&E5	MENU symbol	3	
SD_INX	VDU 1,230	&E6	INDEX symbol	3	
SD_HLP	VDU 1,231	&E7	HELP symbol	3	
SD_OLFT	VDU 1,240	&F0	Outline arrow Left	2	
SD_ORGT	VDU 1,241	&F1	Outline arrow Right	2	
SD_ODWN	VDU 1,242	&F2	Outline arrow Down	2	
SD_OUP	VDU 1,243	&F3	Outline arrow Up	2	
SD_BLFT	VDU 1,244	&F4	Bullet arrow Left	1	
SD_BRGT	VDU 1,245	&F5	Bullet arrow Right	1	
SD_BDWN	VDU 1,246	&F6	Bullet arrow Down	1	
SD_BUP	VDU 1,247	&F7	Bullet arrow Up	1	
SD_PLFT	VDU 1,248	&F8	Pointer arrow Left	1	yes
SD_PRGT	VDU 1,249	&F9	Pointer arrow Right	1	yes
SD_PDWN	VDU 1,250	&FA	Pointer arrow Down	1	yes
SD_PUP	VDU 1,251	&FB	Pointer arrow Up	1	yes

Box Characters

VDU 1, '2', '*', <char>

(where <char> is from 'A' to 'O') draws various characters such as arrows or box construction shapes (which are all boldable), with the following logic. Each of the bottom 4 bits of 'char' represents a direction:

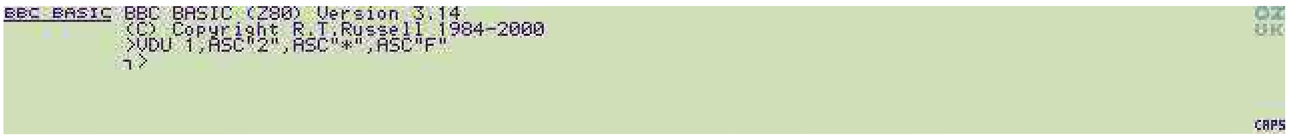
bit 0	decimal 1 Left
bit 1	decimal 2 Down
bit 2	decimal 4 Right
bit 3	decimal 8 Up

If one bit is set, a pointer arrow in the relevant direction is drawn. If two bits are set, two sides of a square or a line will be drawn. If three bits are set, a 'T' shape will be drawn. If all four bits are set, a cross will be drawn.

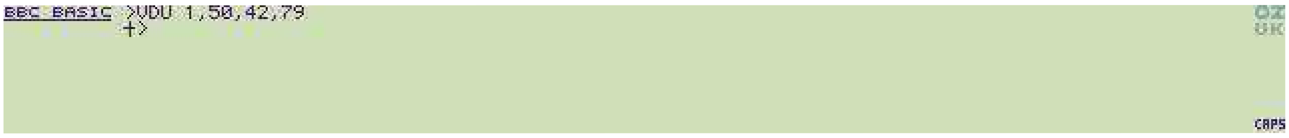
For example, the corner generated:

VDU 1, ASC"2", ASC"*", ASC"F"

makes a reasonable logical NOT sign.


```
BBC BASIC  BBC BASIC (Z80) Version 3.14
           (C) Copyright R.T.Russell 1984-2000
           >VDU 1,ASC"2",ASC"*",ASC"F"
           ¬>
```

This example, VDU 1,50,42,79 - without using ASC, will draw a cross:

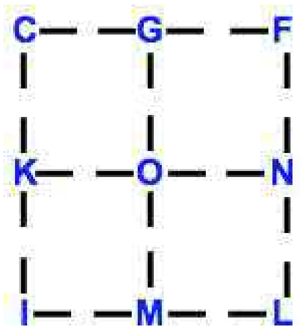

```
BBC BASIC  >VDU 1,50,42,79
           +>
```

Here's the complete VDU's:

Code	Up	Left	Down	Right	Symbol
`VDU 1,ASC”2”,ASC”*”,ASC”A”`	0	0	0	1	Pointer arrow Right
`VDU 1,ASC”2”,ASC”*”,ASC”B”`	0	0	1	0	Pointer arrow Down
`VDU 1,ASC”2”,ASC”*”,ASC”C”`	0	0	1	1	Corner Down Right
`VDU 1,ASC”2”,ASC”*”,ASC”D”`	0	1	0	0	Pointer arrow Left
`VDU 1,ASC”2”,ASC”*”,ASC”E”`	0	1	0	1	Horizontal bar

`VDU 1,ASC"2",ASC"*",ASC"F"`	0	1	1	0	Corner Left Down
`VDU 1,ASC"2",ASC"*",ASC"G"`	0	1	1	1	T Down
`VDU 1,ASC"2",ASC"*",ASC"H"`	1	0	0	0	Pointer arrow Up
`VDU 1,ASC"2",ASC"*",ASC"I"`	1	0	0	1	Corner Up Right
`VDU 1,ASC"2",ASC"*",ASC"J"`	1	0	1	0	Vertical unbroken bar
`VDU 1,ASC"2",ASC"*",ASC"K"`	1	0	1	1	T Right
`VDU 1,ASC"2",ASC"*",ASC"L"`	1	1	0	0	Corner Up Left
`VDU 1,ASC"2",ASC"*",ASC"M"`	1	1	0	1	T Up
`VDU 1,ASC"2",ASC"*",ASC"N"`	1	1	1	0	T Left
`VDU 1,ASC"2",ASC"*",ASC"O"`	1	1	1	1	Cross

To draw a horizontally and vertically divided window:



Cursor Positioning

The cursor position, ie. the next print position, may be moved by the following sequences - x and y are the column and row respectively, with (0,0) being the top left of the current window:

`VDU 1, ASC"3", ASC"@", 32+x, 32+y`	move to x,y
`VDU 1, ASC"2", ASC"X", 32+x`	horizontal tab
`VDU 1, ASC"2", ASC"Y", 32+y`	vertical tab

The origin of the window (see later) is at the top left-hand corner; its coordinates are 0,0.

Display Attributes

These combinations toggle various display modes of the current window (applying to subsequently written characters):

VDU 1, ASC"B"	Bold
VDU 1, ASC"C"	Cursor visible
VDU 1, ASC"F"	Flash
VDU 1, ASC"G"	Grey
VDU 1, ASC"L"	Caps Lock
VDU 1, ASC"R"	Reverse video
VDU 1, ASC"S"	Vertical scrolling
VDU 1, ASC"T"	Tiny font
VDU 1, ASC"U"	Underline
VDU 1, ASC"W"	Horizontal scrolling

With vertical scrolling on, the window scrolls when the cursor tries to go outside of the window else the cursor wraps from top row to bottom row or vice versa.

With horizontal scrolling on, the row scrolls between the left and right margins when the cursor tries to go outside of the window margins else the cursor wraps from left column to right column or vice versa.

Horizontal scrolling disable vertical scrolling.

Rather than toggling these modes, they may be set or reset explicitly by prefixing them with '+' (on) or '-' (off), and a count byte of two to indicate that there are two parameters. For instance:

    VDU 1, ASC"2",ASC"+",ASC"B"

sets bold on. When written in this form, modes may be combined in a list, for example:

    VDU 1, ASC"5",ASC"-",ASC"B", ASC"F", ASC"T", ASC"U"

resets the bold, flash, tiny, underline toggles. Finally SD_DTS (&7F) deletes all toggle settings, ie. sets all toggles to off:

    VDU 1, SD_DTS

Changing Display Attributes

The display modes are usually set at the time of writing text to the screen, however it is possible to apply various effects to text already present. This approach is used in the menu system to highlight commands and by the Filer to highlight files. The technique can only be used with the hardware attributes ie. flash, grey, inverse and underline. The two commands, apply and eor, work over the next <n> characters where <n> is the second parameter and offset from 32:

VDU 1, ASC”2”, ASC”A”, 32+<n>	Apply toggles
VDU 1, ASC”2”, ASC”E”, 32+<n>	EOR toggles

The following sequence inverts a 20 character bar at cursor position (0,0):

```
VDU 1, ASC”2”, ASC”+”, ASC”R”, 1, ASC”3”, ASC”@”, 32, 32, 1, ASC”2”, ASC”E”, 52
```

Cancelling Display Attributes

All the attributes can be set back to the system default (off) with the command:

```
VDU 1 , 127
```

Changing Display Attributes

It is possible to change the 'hardware' attributes (flash, grey, reverse and underline) of text already printed on screen.

Sequence	Description

`VDU 1, ASC"2",ASC"A",32+n`
`VDU 1, 50,65,32+n`
	Apply the current toggles over the next 'n' characters

`VDU 1,ASC"2",ASC"E",32+n`
`VDU 1,50,69,32+n`
	Invert (EOR) the current toggles over the t next 'n' characters

The 'current toggles' are the current toggle settings, not the attributes of the character under the cursor.

For example, the following code sequence would invert the first 20 characters on line 0 starting at column 0.

```
PRINT TAB (O, 0);

VDU 1, 50, 43, 82:REM set invert toggle

VDU 1, 50, 65, 52: REM Apply toggle for 20 (52=32+20) characters
```

The last 2 lines could be combined as:

```
VDU 1, 50, 43, 82, 1, 50, 65, 52
```

Text Justification and Margins

You can control how the text is positioned within a window with the following justification Escape sequences.

Sequence	Description

`VDU 1,ASC”2”,ASC"J",”N”`
`VDU 1,50,74,78`
	Set normal justification (the default).

`VDU 1,ASC”2”,ASC"J",”C”`
`VDU 1,50,74,67`
	Centre text between the margins.

`VDU 1,ASC”2”,ASC"J",”L”`
`VDU 1,50,74,76`
	Left align the text.

`VDU 1,ASC”2”,ASC"J",”R”`
`VDU 1,50,74,82`
	Right align the text.

`VDU 1,ASC”2”,ASC"L",32+n`
`VDU 1,76,32+n`
	Set the left margin to 'n'. (Notice no parameter count.)

`VDU 1,ASC”2”,ASC"R",32+n`
`VDU 1,82,32,+n`
	Set the right margin to 'n'. (Notice no parameter count.)

Windows

Up to 6 display windows may be defined by the user on the Z88; they are referred to by a single ASCII character '1' to '6'. Windows '7' and '8' are used by the Z88 operating system. Window '7' is the 'Topic' area and window '8' is used by a number of system calls for error processing.

Windows 'remember' their attribute settings (see earlier), but if a window area is overwritten then the window's contents are lost.

The OZ window contains special characters which control the scanning of the screen and if these are disturbed, the display will not work properly. Consequently it is very important that windows do not go beyond the width of the screen and overwrite the ‘OZ.' window. To avoid this, windows should not be more than 94 6-pixel characters from the edge of the application window (104 6-pixel characters from the leftmost edge of the screen).

Defining a Window

The following Escape sequence command defines a window.

```
VDU 1 ASC"7",ASC"#",n,32+x,32+y,32+w,32+h,t
VDU 1, 55,35,n,32+x,32+y,32+w,32+h,t
```

Where 'n' is the ASCII code for the window number (49 to 54).
'x' is the start column (left-hand) of the window.
'y' is the start row (top) of the window.
'w' is the width of the window.
'h' is the height of the window.

't' is the 'type' of window, made up as follows:

bit-0 = 1 sets left and right vertical bars on.
bit-1 = 1 sets the shelf brackets on.
bits-2 to 6 are ignored.
bit-7 must be set to 1.

The 'x ' and 'y' parameters are usually relative to the top left-hand corner of the applications area. They may be made absolute (relative to the top left-hand corner of the screen) by using 128+x and 128+y in the Escape sequence. The 'type' parameter is optional. If the parameter count is set to 54 (ASC"6") it may be omitted and a window without bars or brackets will be defined.

Selecting a Window

Once defined, output may be directed to window ‘n’ (‘1’ to ‘6’ - ASCII 49 to 54) with the following Escape sequences:

Sequence	**Description**

`VDU 1,ASC"2",ASC"H",ASC"n"`
`VDU 1,50,72, n`
	Direct output to window ‘n’ maintaining (Holding) the display attributes previously set for this window

`VDU 1,ASC"2",ASC"I",ASC"n"`
`VDU 1,50,73, n`
	Direct output to window ‘n’ resetting (Initializing) the display attributes for this window to their default value (cursor off, scrolling desabled, etc).

`VDU 1,ASC"2",ASC"C",ASC"n"`
`VDU 1,50,67, n`
	Direct output to window ‘n’ resetting the display attributes for this window to their default value and clear the window.

The Z88 system uses windows extensively. The filer, for example, uses a window with ‘shelf brackets’ and tiny/inverted/underlined text to produce a banner heading for the window. The following BBCBASIC(Z80) program demonstrates a ‘bannered’ window.

```
REM Define window 1 to be 40 wide and 8 high
REM with left and right bars and shelf brackets
REM starting at x=1, y=0 with reference to the
REM applications area.

VDU 1,55,35,49,33,32,72,40,131

REM Select window 1 and reset its attributes

VDU 1,50,73,49

REM Set window attributes to Tiny Font,
REM Underline, Inverse Video

VDU 1,52,43,84,85,82

REM Centre the text in the window

VDU 1,50,74,67

REM Print window title in centre of first line

PRINT TAB(0,0);"Central Window Title";TAB(0,0);

REM Apply current toggles over next 40 characters

VDU 1,50,65,72

REM Redefine window to exclude top line (window
REM title area). This time, only left and right
REM side-bars are specified (129)

VDU 1,55,35,49,33,33,72,39,129

REM Select window 1 and reset its attributes

VDU 1,50,73,49

REM Display the cursor and enable vertical
REM scrolling

VDU 1,51,43,67,83
```

User Defined Characters

You can define characters based on a 6x8 matrix with the following Escape sequence:

```
VDU 1,138,ASC"=",char_code,r0,r1,r2,r3,r4,r5,r6,r7

VDU 1,138,61,char_code,r0,r1,r2,r3,r4,r5,r6,r7
```

The character code, char code, must be between 64 (`@') and 127 ([DEL]).

The parameters ‘r0' to 'r7' are the numeric values of the top to bottom rows of the character with bit-7 set. Their values must, therefore, lie between 128 (bit-7 set to 1) and 191 (bit-7 and bits 1 to 5 set to 1). Bit-5 is the left-hand bit and bit-0 the right. The standard characters have their left-hand column (bit-5) blank. If user-defined characters are to be mixed with standard characters, they should follow this convention.

Notice that the parameter count is 10 and it is consequently specified in the 'alternative' way.

If a lesser number of parameters are specified, the unspecified rows will be set to 0. This is useful, since many characters have a blank bottom row.

User defined characters co-exist with standard characters, they do not overwrite them as on the BBC Micro.

Printing User Defined Characters

User defined characters may be printed using the following Escape sequence:

```
VDU 1,ASC"2",ASC"?",char_code
VDU 1,50,63,char_code
```

Limitations

On an unexpanded Z88 (one without 128k or more in slot 1), only 16 characters may be defined without encroaching on the PipeDream map. Characters above the limit of 16 may be overwritten by map information when PipeDream is run. If the PipeDream map width is less than 65 pixels, then all the User Defined Characters may be used.

Miscellaneous Functions

Window Scrolling

The following Escape sequences control window scrolling.

Sequence	Description
VDU 1,254	Scroll current window downwards
VDU 1,255	Scroll current window upwards.

Grey Window

The following Escape sequences control window greying.

Sequence	Description
VDU 1, ASC"2" , ASC"G", ASC"+" VDU 1,50,71,43	Grey the current window
VDU 1, ASC"2", ASC"G" , ASC"-" VDU 1,50,71,45	Ungrey the current window

Multiple Output

The following Escape sequences send the same code a number of times.

Sequence	Description
VDU 1, 51, 32+n,m	Output 'n' copies of the code 'm'.

Multiple Bell

The following Escape sequences rings the 'bell' a number of times.

Sequence	Description
VDU 1,52,33,32+r, 32+s, 32+m	Generate a series of r 'beeps' m*10 milliseconds long with a wait of s*10 milliseconds between each 'beep'.
VDU 1,52,33,35,82,232	For example, this VDU command will sound 3 'beeps' of 2 seconds each with 1/2 a second between each 'beep'.

Operating System Interface

Introduction

As with the BBC micro computer, the star (*) commands provide access to the operating system. Since the BBC operating system is very different to the Z88's, there are considerable differences in the star commands.

When a star command is issued, BBCBASIC(Z80) passes it to the Z88's operating system for action. If it is not recognised, a 'Bad command' error is reported.

File Specifiers

File and device specifiers must comply with the standard Z88 filer conventions. These conventions are described at Appendix D to the Z88 manual and summarised briefly below.

```
[:device][/][path/]filename.extension
```

`:device` The memory bank where the file is located or the name of a physical device. This can be:

:RAM.0	RAM in slot 0 (internal memory).
:RAM.1	RAM in slot 1 (external memory).
:RAM.2	RAM in slot 2 (external memory).
:RAM.3	RAM in slot 3 (external memory).
:RAM.-	Any RAM memory (RAM.0 to RAM.3). Used by the CLI for temporary files. Lost on reset.
:SCR.0	Screen
:ROM.0	ROM.
:PTR.0	Printer.(Serial port via the printer driver so that any special codes and escape sequences are interpreted.)
:COM.0	Communications (serial) port, but NOT via the printer driver.
:INP.0	Standard input
:OUT.0	Standard output
:NUL.0	Null. (Absorbs output and acts like an empty file on input.)

The final .0 of a device name may be omitted if the device is unique. For example, :COM.0 may be abbreviated to :COM

There is a bug in versions of the filer up to and including version 3 that will cause the Z88 to crash if you use the device :RAM.- and then perform a soft reset whilst any files are still present in the device. You can avoid this problem by deleting any files in :RAM.- as soon as you have finished with them.

`path`	The list of directories which must be followed in order to find the specified file. The names of each directory in the path must be separated by the slash (divide) character. If the path is omitted, the current directory on the specified drive is assumed.

```
BBCBASIC/PROGS
```

`filename`	The name of the file. The length of the name must not exceed 12 characters.
`extension`	The optional extension of the file. If an extension is used; it must be separated from the filename by a full-stop.

The standard Z88 '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 `^{4}' to the end of the name or extension.
//	Matches any number of directories (or none).

Symbols

The following symbols and abbreviations are used as part of the explanation of the operating system commands.

ufsp	Unambiguous file specifier (:device/path/name.extension).
str	A string constant. For example: "FRED".
string	A string constant or variable or an expression combining them. For example: name$+"Phone".

Accessing Star Commands

The star commands may be accessed directly or via the OSCLI statement. The 2 examples below both delete the file 'WOMBAT'.

```
*DELETE WOMBAT

OSCLI("DELETE WOMBAT")
```

Syntax

A star command must be the last (or only) command on a program line and its argument may not be a variable. If you need to use one of these commands with a variable as the argument, use the OSCLI statement. Examples of the use of the OSCLI statement are given later.

Case Conversion

Star commands and their associated qualifiers are converted from lower-case to upper-case if necessary. For example, *delete wombat is converted to *DELETE WOMBAT. This is in keeping with the general Z88 philosophy and the BBC Micro's machine operating system (MOS).

Star Commands

*CLI

Pass the following string to the Command Line Interpreter (CLI) for action. The use of the CLI is explained later in this section..

```
*CLI string

*CLI .D 1000        (Delay for 10 seconds)
```

See the 'Command Line Interpreter' sub-section for more details.

*DELETE

Delete the specified file.

```
*DELETE ufsp

*DELETE GAME1
```

This command will delete only one file at a time; wild-cards are not permitted in the file specifier. If you wish to delete more than one file, you should use the 'Erase' function of the 'Filer'.

*ERASE

This command is synonymous with `*DELETE`.

*NAME

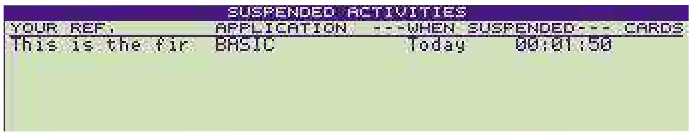
Assign a 'name' to the current instantiation of BBCBASIC(Z80).

```
*NAME string
```

The list of suspended activities shown when the Z88's index is selected has space for 'Your Reference' for each of the suspended activities. This reference may be up to 15 characters long. The *NAME command allows you to specify the reference for the current instantiation of BBCBASIC(Z80). If the reference exceeds 15 characters, it is truncated. For example:

```
*NAME This is the first instantiation
```

would appear in the list of suspended activities as:



```
                      SUSPENDED ACTIVITIES
YOUR REF.          APPLICATION    ---WHEN SUSPENDED--- CARDS
This is the fir    BASIC                Today    00:01:50
```

*SPOOL and *EXEC

*SPOOL and *EXEC are not available on the Z88. However, you can perform similar functions using the CLI as shown below. (The '+'represents the '=/+' key - you don't have to use the [SHIFT] key.)

>LIST ☐ +S [ENTER] .... **>☐ -S**	The equivalent of *SPOOL. It produces a text file of the program called S.SGN in :RAM.- . Remember to delete it when you have finished with it.

Use PipeDream to add '.J' as the first line. Remember to load and save the file in plain text mode. You can now use the command

```
>*CLI .*:RAM.-/S.SGN[ENTER]
```

to '*EXEC' the file back into BBCBASIC(Z80)

You can use this technique to produce a text file for extensive editing in PipeDream.

*RENAME

Rename a file. OSCLI can also be used to rename a file. Using OSCLI has the advantage that the file names may be string variables.

```
*RENAME ufspold ufspnew
OSCLI "RENAME "+string+" "+string

*RENAME OLDFILE NEWFILE

*OSCLI("RENAME "+f_name1$+" "+f_name1$+".BAK")
*OSCLI(“RENAME "+fname1$+" "+fname2$)
```

The Command Line Interpreter (CLI)

The Z88 Command Line Interpreter (CLI) is very powerful. It 'sits' between the keyboard and the current application and interprets the keys that you press. In most cases, the characters for the keys that you press are passed directly to the application that is currently active. However, some keys, [INDEX] ❑ and ◇ for example, have a particular significance to the CLI and they cause subsequent key presses to be interpreted as CLI commands and actioned accordingly. For example, pressing the ❑ key followed by the **F** key will take you into the filer popdown.

You can use the CLI to make changes to the way the Z88 behaves. Any such changes are confined to the application that was active at the time.

The CLI takes its input from the keyboard or from a file. Thus, the CLI acts in many ways like a programming language for controlling the way the computer functions.

CLI Command Files

The CLI may receive commands from a file as well as from the keyboard (or the current input stream). For example, from within the FILER popdown, the following key sequence

```
◇ EXprnt.cli
```

will invoke (run) the CLI command file `pmt.cli'.

A CLI command file is terminated when the end of the file is reached or it is suspended (see later). Control is then returned to the CLI.

You can use the command `.*filename`

within one CLI command file to invoke another CLI command file. When the second CLI command file terminates, control is returned to the first CLI command file.

Cancelling a CLI File

The only way to cancel a CLI command file whilst it is running or suspended is by pressing the [SHIFT] and [ESC] keys together. Since this sequence cannot be represented in any other way (see below), you cannot cancel a suspended CLI command file from within a program.

You can cancel all CLI command files by pressing ◇[ESC].

Special Character Sequences

Some of the keys on the keyboard, '❑' and '◇' for example, have no ASCII code. However, some way of representing these keys is required when input is taken from a CLI command file (or redirected). The ASCII character sequences listed below represent many of the `non-printable' keys. Unfortunately not all key sequences may be represented in this manner. For instance, there is no way of representing [CAPS LOCK] key or ◇[ESC].

Sequence	**Represents**
`#`	Holding down the ❑ key and pressing the next key in the sequence.
`|`	Holding down the ◇ key and pressing the next key in the sequence.
`~A`	Pressing ❑ and releasing it before pressing another key.
`~C`	Pressing ◇ and releasing it before pressing another key.
`|[`	[ESCAPE] key.
`~E`	[ENTER] key.
`~S`	[SHIFT]. This is only generated if <SHIFT> would have any effect.
`~I`	[INDEX] key.
`~M`	[MENU] key.
`~H`	[HELP] key.
`~X`	[DELETE] key.
`~U`	↑ (cursor up) key.
`~D`	↓ (cursor down) key.
`~L`	← (cursor left) key.
`~R`	→ (cursor right) key.
`##`	A single '#'.
`||`	A single '|'.
`~~`	A single '~'.
`~.`	Used to generate a single '.' at the beginning of a line without it being taken as the start of a CLI command.
`.;`	Treat the rest of the line as a remark.

File Control Commands

A full stop at the beginning of a line in a CLI command file introduces the **I/O** redirection and additional file control commands as described below.

I/O Redirection

The standard input and output for the Z88 initially come from and go to the keyboard and the screen. In computer jargon, they are 'bound' to the keyboard and the screen.

You can change the input and output binding (redirect the I/O) with the following CLI commands:

.<filename	Take input from the file (or device) 'filename'
.>filename	Send output to the file (or device) 'filename'

Don't forget that as far as the Z88 is concerned there is no difference between a 'real' file and a device. For example, the device :COM.0 is treated in the same way as a file by the Z88.

In addition to completely redirecting the input or output, you can also copy the input or output stream to a file with the following CLI commands:

.T<filename	Send a copy of the input to the file (or device) `filename'
.T>filename	Send a copy of the output to the file (or device) `filename'

The previous output redirection commands apply to the standard output. If you want to redirect the Printer output, you may so do with the following CLI commands:

.=filename	Redirect the pre-filter output to the file (or device) `filename'
.T=filename	Send a copy of the pre-filter output to the file (or device) 'filename'

Additional CLI File Commands

The following additional commands are only effective from within a CLI file or from redirected input.

Sequence	**Effect**
`.S`	Suspend the current CLI, but maintain all re-bindings
`.D`	Delay for 'n' centiseconds. If [ESC] is pressed during a delay, all subsequent delays for that CLI will fail.
`.J`	Jammer. Ignore all special sequences for the rest of the CLI. For example, after this command, #P will generate the characters `#P' and not invoke PipeDream.
`.*file`	Invoke (run) the CLI command file 'file'.

Re-binding Within a CLI Command File

Any re-bindings are in effect only for the duration of the CLI command file. You may suspend a CLI command file and maintain the bindings with the ``.S`' command. However, once you have done so, the only way of cancelling the CLI is by pressing [SHIFT]/[ESCAPE] on the keyboard.

Accessing The CLI From BBCBASIC(Z80)

You can access the CLI from within BBCBASIC(Z80) by using the *CLI or OSCLI commands. The following examples perform identically and cause a 10 second delay.

```
    *CLI .D 1000
```

Or

```
    OSCLI("CLI .D 1000")
```

You can invoke a CLI file from within BBCBASIC(Z80) using either the *CLI or OSCLI commands. For example

```
    *CLI .*filename
```

Or

```
    OSCLI("CLI .*filename")
```

will invoke the CLI command file 'filename'.

The advantage of using the OSCLI command is that you can use a variable for the file name and the command does not need to be the last (or only) one on a line. For example, the following command will invoke the CLI command file whose name is held in the variable `fname$'.

OSCLI("CLI .*"+fname$)

A command passed to the CLI is not executed until the CLI gains control, and this only happens when input is expected from the keyboard (or redirected input file). This is no problem if you are in BBCBASIC(Z80)'s immediate mode where the computer is waiting for you to type something. However, if you are in a program, the commands queued to the CLI will not be actioned until the program pauses for input, or ends and exits to immediate mode. You can force the CLI to look for input by appending an INKEY(0) command to the end of your OSCLI command line. For example, the following command will invoke the CLI command file `prntr.cli' and execute it, instigating the redirection which it sets up.

```
    OSCLI("CLI .*prntr.cli"):dummy=INKEY(0)
```

The file 'prntr.cli may be created using PipeDream and saved as a plain text file with the name `prntr.cli. The file contains the following commands, for example.

```
    #+P
    .S
```

The use of this file is explained more fully in the 'Printing' section of the manual.

Don't forget that the only way to remove a suspended CLI is by pressing [SHIFT]/[ESC] on the keyboard. You cannot issue a command from within a program that will remove it. For this reason, you should avoid the use of suspended CLI files wherever possible.

CLI Command Examples

Example 1

From within BBCBASIC(Z80), the following command line will enter the Panel popdown, turn sound off and return to BASIC. The line may be included in a BBC BASIC program or entered in the direct mode.

```
*CLI #S~R~R~DN~E
```

Example 2

As previously mentioned, CLI commands are not executed until control is passed to the CLI. You can demonstrate this with a short program incorporating the previous example. When you RUN the following program, you will notice that 'LINE 10' and 'LINE 30' are printed before the Panel popdown is entered.

```
10 PRINT "LINE 10"
20 *CLI #S~R~R~DN~E
30 PRINT "LINE 30"
```

Changing line 20 to

```
OSCLI("CLI #S~R~R~DN~E"):x=INKEY(0)
```

forces the execution of the CLI command before 'LINE 30' is printed.

Example 3

The following BBCBASIC(Z80) program is the program line editor (by Cambridge Computer Ltd) introduced in the ‘Program editor’ in the **General Information** section. It makes extensive use of CLI commands. The listing below is interspersed with comments that do not form part of the program.

You do not need to use this program when using ROM V4.3 or later. The *EDIT command is available, replacing the need for this manual approach using the CLI.

The editor consists of 2 procedures. The first creates a CLI command file that lists the line to a file and then calls the second procedure which cleans it up and uses it as redirected input before exiting. The line to be edited is then left in the input buffer. The Z88's line editor may then be used to edit the line as if it had just been typed in.

```
60000 END
```

Make sure that the program does not end up here accidentally and then define the first procedure.

```
      60010 DEF PROCE(B)
      60020 REM Cambridge Computer Ltd.
```

The procedure is called with the line number to edit. Make sure this is not zero.

```
      60030 IF B=0 THEN ENDPROC
```

Build a CLI command file in :RAM.0 for later execution. The file is called EE.CLI. It will contain the following lines:

`.>:RAM.0/E.CLI`	(This will send subsequent output to the file E.CLI)
`.J`	(`Jam' the CLI so that nothing is mistaken as a CLI command)

```
 LIST  'line number B as an ASCII string'
 PROCF
```

```
      60040 A=OPENOUT ":RAM.0/EE.CLI"
      60050 B$=":RAM.0/E.CLI"
      60060 PRINT#A,".>"+B$
      60070 PRINT#A,".J","LIST"+STR$(B),"PROCF"
      60080 CLOSE#A
```

Execute the CLI command file `EE.CLI' and then return. The CLI command file is not actually executed by the CLI until control is passed to it. As previously explained, this is when keyboard input is expected. Consequently, the commands in the CLI command file will not be executed until BBCBASIC(Z80) returns to the immediate mode (after the ENDPROC).

```
      60090 *CLI .*:RAM.0/EE.CLI
      60100 ENDPROC
```

Having built the CLI command file EE.CLI and queued it for action by the CLI, PROCE terminates and BBCBASIC(Z80) returns to the immediate mode. When this happens, the CLI command file EE.CLI is executed and it sends:

```
      LIST nnn
      The program line 'nnn'
```

to the file E.CLI. It then causes 'PROCF' to be executed. This is the procedure where the line to be edited is cleaned up and turned into an input file.

```
       60110 DEF PROCF
```

Force a keyboard input to signal to the CLI that the command file EE.CLI has terminated.

```
       60120 A=INKEY(0)
```

The line to be edited is the second line in the file E.CLI. Open the file, throw away the first line

(LIST `nnn'), read the line to be edited and close the file.

```
60130 A=OPENIN B$
60140 INPUT#A,A$,A$
60150 CLOSE#A
```

We have the line that we want to edit in A$. Now we use it to write a file (E.CLI again) that we can use as redirected input.

```
60160 A=OPENOUT B$
60170 PRINT#A,".J",A$
```

Change the CR at the end of the line to a ‘null’. We are going to use the file as input and if we left the CR at the end, the line would be entered before we had a chance to edit it.

```
60180 PTR#A=PTR#A-1
60190 BPUT#A,0
60200 CLOSE#A
```

Get rid of the 'gash' file EE.CLI and move the cursor left ready for editing.

```
60210 *ERASE :RAM.0/EE.CLI
60220 VDU 8
```

Redirect input to be taken from the file E.CLI and exit.

```
60230 OSCLI "*CLI .<"+B$
60240 ENDPROC
```

The line to be edited is now in the input buffer and may be edited as if it had just been typed in.

Printing

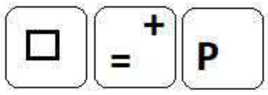
Introduction

There are several ways of sending output from BBCBASIC(Z80) to the printer, either by echoing output to the screen or by directing the output exclusively to the printer.

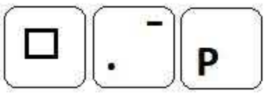
This section describes the various ways of sending output to the printer along with the advantages and drawbacks of each method. It also describes the 'Printer Filter' and the actions of the various control codes and 'Escape' sequences.

Keyboard Control

You can echo characters which are sent to the screen to the printer. Printer echo is controlled with the following key sequences.

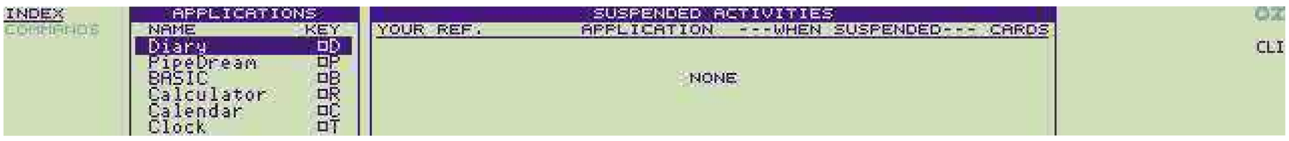


Printer echo on. All subsequent characters sent to the screen are echoed to the printer.



Printer echo off.

This is not quite as straightforward as it at first appears. In fact, ❑+P starts a CLI command that sends a copy of characters sent to the screen to the printer as well. The key sequence ❑-P removes that CLI command.



```
INDEX     APPLICATIONS        SUSPENDED ACTIVITIES                                  OZ
COMMANDS  NAME        KEY     YOUR REF.   APPLICATION  ---WHEN SUSPENDED---  CARDS
          Diary       □D                                                            CLI
          PipeDream   □P
          BASIC       □B                    NONE
          Calculator  □R
          Calendar    □C
          Clock       □T
```

Whilst the CLI command is active, the 'CLI' indicator appears in the OZ window and page-waits do not occur during a program listing.

In addition to ❑-P, you can cancel the ❑+P CLI command in the normal way by pressing the [SHIFT]/[ESCAPE] or ◇[ESCAPE].

Using printer echo is an easy way of listing a program on the printer. In the following example, the ‘+’ represents the `=/+' key; you don't have to use the [SHIFT] key.

```
LIST❑+P[ENTER]
```

……….

Program listed to screen and printer.

……….

❑-P

From Within a Program

There are 2 ways of sending data to the printer from within a program. The first involves the use of the Command Line Interpreter and the second used the BBCBASIC(Z80) command, PRINT#. The former echoes characters sent to the screen to the printer. It provides all the normal BBCBASIC(Z80) print control features, but it is more complicated to set up. The latter sends characters to the printer only. It is simple to set up, but it becomes more difficult to use as the complexity of the printed line increases.

Using the CLI

The use of the Command Line Interpreter is explained in Section Seven of your Z88 manual https://cambridgez88.jira.com/wiki/x/RAAkAg and in the `Operating System Interface' section of this manual.

Using CLI commands to send data to the printer has the advantage that the screen format is duplicated on the printer and all BBCBASIC(Z80)'s print format controls work. You do not, however, have access to the special print controls provided by the Z88's Printer Filter and you are left with a suspended CLI command file (explained in the 'Operating System Interface' section). Consequently, this is the least favoured method unless you especially want to echo screen output to the printer.

It is possible to emulate pressing the key sequences ❑+P and ❑-P from within BBCBASIC(Z80) by sending these key sequences to the CLI. At first glance, the following program line should work. Remember that the `#' symbol is interpreted as ❑ by the CLI.

```
*CLI #+P
```

This does not achieve the desired effect within a program because the CLI command does not become active until the next attempt to read a character from the keyboard. In order to activate the CLI command, the line needs to have an INKEY command at the end. Since a star command must be the last (or only) command in a program line, you need to use the OSCLI command to pass the CLI command to the operating system. Thus, the line becomes;

```
OSCLI("CLI #+P"):dummy=INKEY(0)
```

Try this line out on the Z88. You will see the CLI indicator in the status panel flash on and off again indicating that a CLI command sequence is momentarily active. Unfortunately, the print echo is only active whilst the CLI indicator is present, so this is not a lot of use to us!

In order to keep print echo active, we need to suspend the CLI command sequence after we have issued the print echo command. Unfortunately, this needs 2 command lines and must be done using a CLI command file.

You will need to use PipeDream to create the following CLI command file. Save it as ‘plain text’ to a file named pon.cli.

```
#+P
.S
```

Remember to save it as ‘plain text’ - you will get all sorts of interesting error messages if you don't.

Once you have created this file, you can turn the printer on and off from within your program with the following program lines:

Printer On

```
OSCLI("CLI .*:RAM.0/pon.cli"):dummy=INKEY(0)
```

All subsequent PRINT statements will print to the screen and the printer.

The first line of the CLI file turns printer echo on by sending the command ❑+P to the CLI. In order to stop the printer echo being turned off at the end of the CLI command file, the CLI is suspended with the ‘.S' command on the next line. Without the INKEY(0) command at the end of the program line, print echo would not be turned on until the program attempted to read from the keyboard or the program terminated and BBCBASIC(Z80) returned to the immediate mode.

Remember, you cannot cancel a suspended CLI command file from within a program. The only way is to press the [SHIFT]/[ESC] or ◇[ESC] keys on the keyboard. Consequently, you should avoid using this method of sending characters to the printer unless you really need to.

The use of CLI command files is explained more fully in the ‘Operating System Interface' section of the manual.

Printer Off

```
OSCLI ("CLI #-P"):dummy=INKEY(0)
```

Subsequent PRINT statements will now only print to the screen.

This line sends ❑-P to the CLI and turns printer echo off. Unfortunately, this leaves the CLI suspended and there is no way to reactivate it from within your program. You can, however, reactivate the CLI manually by pressing [SHIFT]/[ESCAPE] when your program has terminated.

You may not initially notice the effect of having a suspended CLI command file. However, when you list a program you will find that the listing no longer pauses automatically after every page.

Using PRINT#

You can also send output to the printer using the devices (hardware names) :PRT.0 and :COM.0. The :PRT.0 device filters output through a 'filter' (device driver), the :COM.0 device sends output direct to the printer (serial port).

This method of printing does allow you to make use of the 'Printer Filter', but as the complexity of the line to be printed increases, this method becomes more difficult.

To send data to the printer in this way, you first open a file to the device and then use the PRINT# statement to send the characters to be printed to the printer. Characters printed in this way do not appear on the screen.

When you send characters to the printer device :PRT.0 in this way, they are processed by the Printer Filter and all the codes described in the 'Print Filter' sub-section are recognised and acted upon.

If you are using the :PRT.0 device, you must send the printer-on Escape sequence ENQ [(ASCII codes 5 & 91) to turn printing on before you send anything you want printed. Remember to turn the printer off with the Escape sequence ENQ] (ASCII codes 5 & 93) when you have finished.

The following example opens a file and prints to it via the Printer Filter. It uses some printer Escape sequences which are discussed in the 'Printer Escape Sequences' sub-section and it assumes that the Printer Editor has been used to set 'Off at CR' to `No' for bold and underline.

```
 10 REM Open the printer file
 20 prntr=OPENOUT ":PRT"
 30 :
 40 ENQ$=CHR$(5):LF=10
 50 REM Send the Print Filter On Escape sequence
 60 PRINT#prntr,ENQ$+"["
 70 :
 80 REM Set bold print on
 90 PRINT#prntr,ENQ$+"B"
100 :
110 REM Set underline on
120 PRINT#prntr,ENQ$+"U"
130 :
140 REM Print 'Hello World' and send a line-feed
150 PRINT#prntr,"Hello World":BPUT#prntr,LF
160 :
170 REM Send the Print Filter Off Escape sequence
180 PRINT#prntr,ENQ$+"]"
190 :
200 REM Close the printer file
210 CLOSE#prntr
220 END
```

PRINT# is intended for sending data to a file. Consequently, it works in a different way to PRINT. Items separated by commas or spaces are treated as different data fields and a carriage-return is sent between each field. Consequently, you need to build and send a complete line at a time. In addition, you must explicitly send the line-feed character at the end of the line.

If you are using the :COM.0 device, everything is sent to the printer and you don't need to turn it on and off. However, you must send the printer specific control codes for underline, etc since the Printer Filter is not being used.

The Printer Filter

All output explicitly sent to the printer device :PRT.0 goes via the ‘Printer Filter'. This filter examines every character to see if it has some special meaning or if it is to be translated into another character or sequence of characters. The code translation and command codes may be edited using the 'Printer Editor' (❑E) as described in your Z88 Users' Guide.

Most printers have a set of control codes that switch the printer into similar modes. However, whilst most printers provide a number of similar modes, the codes used are often different. The 'Printer Filter' provides a universal set of print format controls which may be used irrespective of the printer in use. Provided you use the 'Printer Editor' (❑E) to configure the Print Filter for your printer, your programs will work in exactly the same way irrespective of the printer in use.

The 'Printer Filter On' and 'Printer Filter Off' Escape sequences also turn the printing on and off. So remember to use them before and after printing.

The tables in the following sub-sections include program segments that will send the appropriate codes to the printer file. These assume that the printer file handle is in the variable `pf' and that ENQ$ is CHR$(5).

Printer Control Codes

Escape Sequence	**Meaning**

`PRINT#pf,ENQ$+"["`
`BPUT#pf,5:BPUT#pf,91`
	Printer ON.

`PRINT#pf,ENQ$+"]"`
`BPUT#pf,5:BPUT#pf,93`
	Printer OFF.

`PRINT#pf,ENQ$+"2P"+CHR$(32+n)`
`BPUT#pf,5:BPUT#pf,50:BPUT#pf,80:BPUT#pf,32+n`
	Set page length to 'n' lines.

`PRINT#pf,ENQ$+"2H"+CHR$(32+n)`
`BPUT#pf,5:BPUT#pf,50:BPUT#pf,72:BPUT#pf,32+n`
	Microspace 'n' units of 1/120 inch.

`PRINT#pf,ENQ$+"5S" BPUTfpf,5:BPUT#pf,83`	Reset attributes that would be reset by a new-line.

`PRINT#pf,ENQW53$"+CHRS(x)+CHRS(y)`
`BPUT#pf,5:BPUT#pf,51:BPUT#pf,36:BPUT#pf,x:BPUT#pf,y`
	Send the hex values 'x' and 'y' to the printer.

Attributes

The following codes toggle printer attributes such as bold print, underline, etc. These toggles can be automatically reset at the next carriage-return by setting the 'Off at CR' option in the Printer Editor to 'Yes'. By default, all the attribute toggles with the exception of ‘Alternate Font' (`A') and 'User Defined' (`E') are automatically reset when a carriage-return is sent.

Escape Sequence	Meaning

`PRINT#pf,ENQ$+"U"`
`BPUT#pf,5:BPUT#pf,85`
	Underline

`PRINT#pf,ENQ$+"B"`
`BPUT#pf,5:BPUT#pf,66`
	Bold

`PRINT#pf,ENQ$+"X"`
`BPUT#pf,5:BPUT#pf88`
	Extended sequence

`PRINT#pf,ENQ$+"I"`
`BPUT#pf,5:BPUT#pf,73`
	Italics

`PRINT#pf,ENQ$+"L"`
`BPUT#pf,5:BPUT#pf,76`
	Subscript

`PRINT#pf,ENQ$+"R"`
`BPUT#pf5:13PUT#pf,82`
	Superscript

Escape Sequence	Meaning

`PRINT#pf,ENQ$+"A"`
`BPUT#pf,5:BPUT#pf,65`
	Alternate font

`PRINT#pf,ENQ$+"E"`
`BPUT#pf,5:BPUT#pf,69`
	User defined

Unless you have used the Printer Editor to set 'Off at CR' to 'No', you should use the `BPUT#' version of the commands. If you use the 'PRINT#' version of the commands, the terminating CR will undo its effect (excepting Printer On/Off and Alternate and User Defined fonts).

Unlike the screen control codes, the printer control codes cannot be strung together or explicitly set by preceding them with a '+' or ‘-’. Neither is there a 'reset all toggles' command.

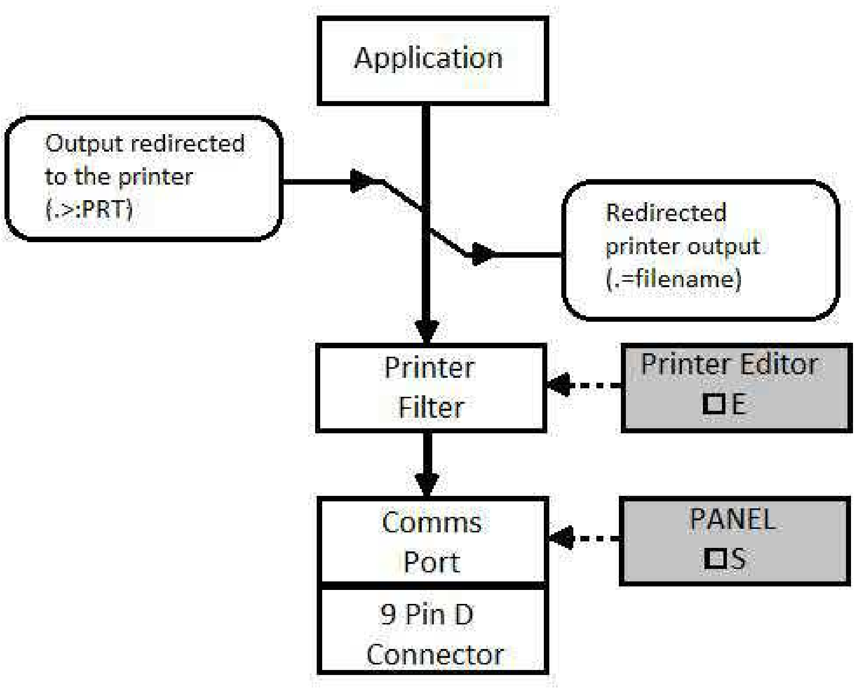
Untrapped Characters

The following control characters are not treated as commands by the Print Filter. Like the normal printable characters (Space to Delete &20 to &7F) they may be translated, but they are otherwise sent directly to the printer.

Name	**Hex Value**
NUL	&00
BEL	&07
BS	&08
HT	&09
LF	&10
VT	&OB
FF	&OC
CR	&OD

Printer Redirection

The 'Operating System Interface' section of the manual describes how input and output may be redirected. The diagram below may help you to visualise how and where redirection takes place. It also illustrates how the Printer Editor and the PANEL affect the output.


Application
Output redirected to the printer (.>:PRT)
Redirected printer output (.=filename)
Printer Filter
Printer Editor ❑E
Comms Port
PANEL ❑S
9 Pin D Connector

Since you cannot issue CLI 'dot' commands from the keyboard, you will need to write a CLI command file using PipeDream in order to redirect the printer input or output. The CLI command file shown below would redirect printer output to a file called `pfile'.

```
.=pfile
.S
```

If this file had been saved (in ‘plain text’ format) as ‘predif, you could initiate printer output redirection by issuing the command

```
◊EXpredir
```

from the filer (❑F), or

```
*CLI .*predir
```

in the immediate mode from within BBCBASIC(Z80).

If you wished to issue the command from within a BBCBASIC(Z80) program, you would need the following program line.

```
OSCLI("CLI .*predir"):x=INKEY(0)
```

See the 'Operating System Interface' section of the manual for an explanation of CLI command files.

The Serial Port

Hardware Connections

Z88 9 Way D Type (Male)				Printer 25 Way D Type (Male)	
1	Unswitched +5v at 10 uA		output		
2	TX	Transmit data	output	3	RX
3	RX	Receive data	input	2	TX
4	RTS	Ready to send	output	5	CTS
5	CTS	Clear to send	input	20	DTR
6		Reserved for future use			
7	GND	Signal ground		7	GND
8	DCD	Data carrier detect	input	20	DTR
9	DTR	Data terminal ready	output	6,8	DSR, DCD

DTR is high when the Z88 is awake. The Z88 is always awake when the screen is active. Even if the Z88 is asleep, it will wake every minute or so to carry out various housekeeping tasks (checking the alarms, for example). At these times, DTR will go high.

Pin 1 can be used to indicate that power is available to the Z88.

You can use the PANEL (❑S) to set up the parameters of the serial port (speed, parity, etc).

Flow Control

Output from and input to the serial port can be controlled either by software or hardware. The hardware handshaking is always active, so if you only want to use software handshaking, you will need to wire a cable to set the handshaking lines high at all times. You can do this by connecting pins 5, 8 and 9 together on the Z88 9-pin connector.

Output

An external device (printer, modem, etc) can ask the Z88 to stop sending by either bringing the CTS line (pin 5) low or sending an XOFF (CHR$(19)) character to the Z88.

There is potentially a slight delay when using software handshaking. Transmission will stop only after the XOFF character has been recognised and, since the Z88 has an input buffer, this can only happen after previously received characters have been processed. With hardware flow control, transmission stops at the end of the next character.

Transmission is resumed when an XON (CHR$(17)) character is received or the CTS line is brought high.

The output buffer is around 95 bytes long.

Input

If software flow control is used, the Z88 will send XOFF to an external device once the receive buffer is more than half full. Characters will continue to be received until there are only 15 spaces left in the buffer. At this point, an XOFF character will be sent for every character subsequently sent by the external device. If the receive buffer overflows, then the received data is lost. The Z88 will send the XON character when the receive buffer has been cleared to less than half full.

If hardware flow control is used, the Z88 will bring RTS low when the receive buffer becomes more than half full and bring RTS high when the receive buffer is less than a quarter full.

The input buffer is around 127 bytes long.

BBC BASIC(Z80) Files

Introduction

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

If you are new to BBC BASIC(Z80), or you are experiencing difficulty with disk files you might find these notes useful. Some of the concepts and procedures described are quite complicated and require an understanding of file structures. If you have trouble understanding these parts, don't worry. Try the examples and write some programs for yourself and then go back and read the notes again. As you become more comfortable with the fundamentals, the complicated bits become easier.

The programs given in this manual are for demonstration and learning purposes; they are not intended to be taken and used as working programs without modification to suit your needs. 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 we 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.

The Structure of Files

If you understand the way files work, skip the next two paragraphs. If you understand random and indexed files, skip the following two paragraphs as well.

Basics

Many people are confused by the jargon that is often used to describe the process of storing and retrieving information. This is unfortunate, because the individual elements are very simple and the most complicated procedures are only a collection of simple bits and pieces.

All computers are able to store and retrieve information from a non-volatile medium. In other words, you don't lose the information when the power gets turned off. Remember, pressing both `[SHIFT]` keys does not actually turn your Z88 off; removing the batteries would. Audio cassettes are used for small micro computers, diskettes for medium sized systems and magnetic tape and large disks for big machines. Some computers, like the Z88, are designed so that the contents of RAM are not lost when the machine is ‘shut down’ (pressing both `[SHIFT]` keys). These computers can use RAM instead of tape or disk as a non-volatile medium. RAM has several advantages of disks. File access is very much quicker and there aren’t any disks to lose; spill coffee over, etc.

In order to be able to find the information you want, the information has to be organised in some way. All the information on one general subject is gathered together into a FILE. Within the file, information on individual items are grouped together into RECORDS.

Serial (Sequential) Files

Look upon the cassette or diskette or the Z88’s RAM as a drawer in a filing cabinet. The drawer is full of folders called FILES and each file holds a number of enclosures called RECORDS. Sometimes the files are in order in the drawer, sometimes not. If you want a particular file, you start at the beginning of the drawer and search your way through until you find the file you want. Then you search your way through the records in the file until you find the record you want.

This is very similar to the way a cassette is searched for a particular file. You put the cassette in the recorder, type in the name of the file you want and push play. You then go and make a cup of tea whilst the computer reads through all the files until it comes to the one you want. Because the cassette is read serially from start to end, it's very difficult to do it any other way.

Life is easier with a computer that uses diskettes or RAM. 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.

There are a number of ways to overcome this problem. We will consider the two simplest; random access (or relative) files and indexed files.

Random Access Files

The easiest way to find the record you want is to identify each record with a number, like an account number. You can then ask for, say, the 23rd record. This is similar to turning to page 23 in the account book. This works very well at first. Every time you get a new customer you start a new page. Most of the pages have a lot of empty space, but you must have the same amount of space available for each account, otherwise your numbering system won't work. So, even at the start, there are a lot of gaps.

What happens when you close an account? You can't tear out the page because that would upset the numbering system. All you can do is draw a line through it - in effect, turn it into a blank page. Before long, quite a number of pages will be 'blank' and a growing proportion of your book is wasted.

With other forms of 'numbering', say by the letters of the alphabet, you could still not guarantee to fill all the pages. You would have to provide room for the Zs, but you may never get one. When you started entering data, most of the pages would be blank and the book would only gradually fill up.

The same happens with this sort of file stored in the Z88's RAM. 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 ¾ full. Count the number of empty 'slots' in your address book and see what proportion this is of the total available.

Indexed Files

Suppose we want to hold our address book on the computer. We need a number of records each holding the name, address, telephone number, etc of one person. In our address book, we have one or two pages per letter of the alphabet and a number of 'slots' on each page. With this arrangement, the names are in alphabetical order of their first letter. This is very similar to the way the accounts book was organised except that we don't know the page number for each name.

If we had an index at the front of the book we could scan the index for the name and then turn to the appropriate page. We would still be wasting a lot of space because some names, addresses etc are longer than others and our 'slots' must be large enough to hold the longest.

Suppose we numbered all the character positions in the book and we could easily move to any character position. We could write all the names, addresses, etc, one after the other and our index would tell us the character position for the start of each name and address. There would be no wasted space and we would still be able to turn directly to the required name.

What would happen when we wanted to cancel an entry? We would just delete the name from the index. The entry would stay in its original place in the book, but we would never refer to it. Similarly, if someone changed their address, we would just write the name and the new address immediately after the last entry in the book and change the start position in the index. Every couple of years we would rewrite the address book, leaving out those items not referenced in the index and up-date the index (or write another one).

This is not a practical way to run a paper and pencil address book because it's not possible to turn directly to the 3423rd character in a book, and the saving in space would not be worth the tedium involved. However, with BBC BASIC you can turn to a particular character in a file and the tedium only takes a couple of seconds, so it's well worth doing.

Files in BBC BASIC(Z80)

Introduction

Conventional serial 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).

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.

Most versions of BASIC only offer serial and random files, but because of the way that disk files are handled by BBC BASIC (both on the BBC computer and the Z88 using BBC BASIC(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

As far as the programmer is concerned, data can be written to and read from a file a data item or a character (byte) at a time. In fact, there is a buffer between the program and the operating system, but this should only concern you when you are organising your program for maximum file access efficiency.

Because of the character by character action of the write/read process, it is possible (in fact, necessary) to keep track of your position within the file. BBC BASIC does this for you automatically and provides a pointer PTR (a pseudo-variable) which holds the position of the NEXT character (byte) to be written/read. Every time a character is written/read PTR is incremented by 1, but it is possible to set PTR to any number you like. This ability to 'jump around' the file enables you to construct both random (relative) and indexed files.

BBC BASIC provides the facility for completely free-format binary data files. Any file which can be read by the computer, from any source and in any data format, can be processed using the BGET, BPUT and PTR functions.

How Data is Stored

Data files written by the PRINT# statement and read by the INPUT# statement have different format to files produced by similar statements on the BBC Micro. This only becomes significant when calculating record sizes for random access files. Record sizing is discussed at the ‘Random (Relative) Files’ sub-section.

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.

How Strings are Stored

Strings are stored in a data file as the ASCII bytes of the string followed by a carriage-return. A string, therefore, occupies one more byte than the length of the string. If no carriage-return (CR) is found within 256 characters when reading a string value, a null string is returned.

If you need a line feed as well, it's no problem to add it using the Byte-Put function BPUT#. Similarly, extraneous characters included in files produced by other programs can be read and, if necessary, discarded using BGET#.

How Files are Referred To

We refer to a file (or device) by its name. Unfortunately, this is too complicated for the Z88's Operating System. Consequently, the only time OZ refers to a file by its name is when it opens the file. From then on, it refers to the file by the number it allocated to it when it was opened. This number is called the 'file handle'.

The Z88 FIling System

The Z88 has a uniform device independent I/O system. Although procedures do exist that will explicitly send data directly to, for example, the serial port, it is more logical to think of these devices as 'files' and use the standard file I/O interface.

For example, to send data to the communications port (serial port without the printer filter), you can open a file to :COM.0 and PRINT to it just as if you were PRINTing to a file.

```
10 com=OPENOUT(":COM.0")
20 PRINT#com,"This is going to the comms device"
30 BPUT#com,10: REM Send the linefeed
40 CLOSE#com
```

Remember, you need to explicitly send the line-feed. See the 'Operating System Interface' and 'Printing' sections for more details.

Logically, you can only write to and read from the I/O devices in a sequential manner. It makes no sense, for example, to try to set the file pointer with PTR#.

File Access Commands

Introduction

The commands and statements used in file manipulation are described below. They are not in alphabetical order, but in the order you are likely to want to use them. Whilst these notes repeat much of the material covered in the Statements and Functions section, additional information has been added and they are presented in a more readable order.

Filenames

Please refer to your 'Z88 Users' Guide' for a full explanation of device, directory and filenames. The explanation below is only intended as a brief reference guide.

The Z88's operating system allows a composite file name in the following format:

```
:DEVICENAME/PATHNAME/FILENAME.EXTENSION
```

The **devicename** is the name of a RAM memory or a physical device. This can be:

`:RAM.0`	Files stored in RAM in slot 0 (internal memory)
`:RAM.1`	Files stored in RAM in slot 1 (external memory)
`:RAM.2`	Files stored in RAM in slot 2 (external memory)
`:RAM.3`	Files stored in RAM in slot 3 (external memory)
`:RAM.-`	Files stored in any RAM (:RAM.0 - :RAM.3). Used by the CLI for temporary files. Lost on Reset
`:SCR.0`	Screen
`:ROM.0-3`	Application names available in internal ROM or external application cards, such as EPROM or Flash cards.
`:EPR.0-3`	Files stored on external EPROM or Flash Cards, formatted via Filer. This device is only available in OZ V5.0 or later.
`:PRT.0`	Printer. (Serial port via the printer driver so that any special codes and escape sequences are interpreted)
`:COM.0`	Communications (serial) port, but NOT via the printer driver.
`:INP.0`	Standard input
`:OUT.0`	Standard output.
`:NUL.0`	Null. (Absorbs output and acts like an empty file on input.

The final .0 of a device name may be omitted if the device is unique. For example, :COM.0 may be abbreviated to :COM.

There is a bug in all ROM versions before V4.0, in the Filer popdown, that will cause the Z88 to crash if you use `:RAM.-` device to store files - and then perform a soft reset whilst any files are still present in the device. You can avoid this problem by deleting any files in :RAM.- before issuing a soft reset.

The **pathname** is the name of the directory or the path to the directory in which the file will be found or created.

The filename can be up to 12 characters long, and the extension up to 3 characters.

Organisation of Examples

Simple examples are given throughout this section with the explanation of the various commands. The following sections contain examples of complete programs for serial files, random files and, finally, indexed files. If you have problems understanding the action of any of the commands you may find the examples helpful. The best way to learn is to do - so have a go.

Program File Manipulation

SAVE

Save the current program to a file, in internal (tokenised) format. The filename can be a variable or a string.

```
SAVE filename
SAVE "FRED"

A$="COMPOUND"
SAVE A$
```

The first example will save the program to a file named FRED. The second will save COMPOUND.

You can specify a device name and a path as well as the file name. The following example will save the current program to a file called “TEST” in a directory called PROGS which is in a directory called BBCBASIC on device :RAM.0.

```
SAVE “:RAM.0/BBCBASIC/PROGS/TEST”
```

LOAD

Load the program 'filename' into the program area. The old program is deleted (as if a NEW command had been given prior to the LOAD) and all the dynamic variables are cleared. The program must be in tokenised format. File names must conform to the Z88 file format.

```
LOAD filename
LOAD "FRED"

A$="HEATING"
LOAD A$
```

As with SAVE, you can specify a device name and path. The example below loads the program saved previously as an example of the SAVE command.

```
LOAD ":RAM.0/BBCBASIC/PROGS/TEST"
```

CHAIN

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

```
CHAIN filename
CHAIN "GAME1"

A$="PART2"
CHAIN A$
```

As with SAVE and LOAD, you can specify a device name and/or a path.

***DELETE**

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

```
*DELETE filename

*DELETE FRED
*DELETE PHONE.DTA
```

To delete a file whose name is known only at run-time, use the OSCLI command. It's a bit clumsy, but a lot better than the original specification for BBC BASIC allowed. This time all of the command, including the ERA, must be supplied as the argument for the OSCLI command. You can use OSCLI for erasing a file whose name is a constant, but you must include all of the command line - in quotes this time.

```
fname$="FRED"
OSCLI "DELETE "+fname$

fname$="PHONE.DTA"
command$="DELETE "
OSCLI command$+fname$

OSCLI "DELETE FRED"
```

You can include a device name and/or path in both the *DELETE and *OSCLI command formats.

***RENAME**

Rename 'file1' to be called 'file2'.

```
      *RENAME file2=file1
      *RENAME FRED2=FRED1
      *RENAME PHONE.DTA=PHONE
```

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

```
fname1$="FRED1"
fname2$="FRED2"
OSCLI "RENAME "+fname2$+"="+fname1$
```

If you attempt to rename an open file, an ‘in use’ error will be reported.

Files and Devices

Introduction

The statements and functions used for data files are:

```
OPENIN
OPENUP
OPENOUT
EXT#
PTR#
INPUT#      BGET#
PRINT#      BPUT#
CLOSE#      END
EOF#
```

Opening Files

You cannot use a file until you have told the system it exists. In order to do this you must OPEN the file for use. Other versions of BASIC allow you to choose the file number. In order to improve efficiency, BBC BASIC(Z80) chooses the number for you.

When you open the file, a file handle (an integer number) is returned by the interpreter and you will need to store it for future use. (The open commands are, in fact, functions which open the appropriate file and return its file handle.)

You use the file handle for all subsequent access to the file. (With the exception of the STAR commands outlined previously.)

If the system has been unable to open the file, the handle returned will be 0. This will occur if you try to open a non-existent file in the input mode (OPENIN or OPENUP).

The BBC BASIC(Z80) for the Z88 imposes a limit on the number of files you can have open at any one time. The limit is 10 for BBC BASIC(Z80) and more than 90 for the whole filing system (other application instantiations opening files), depending on which ROM version you are using on your Z88. If you attempt to have more files open at one time than allowed for BBC BASIC(Z80), the file will not be created and you will get a ‘Too many files open’ error. If you try to exceed the limit for the filing system, you will get a ‘No Room’ error.

File Opening Functions

The three functions which open files are OPENIN, OPENUP and OPENOUT. OPENOUT should be used to create new files, or overwrite old ones. OPENIN should be used for input only and OPENUP should be used for input/output.

OPENOUT

Open the file 'filename' for output and return the file handle allocated. The use of OPENOUT destroys the contents of the file if it previously existed. (The directory is updated with the length of the new file you have just written when you close the file.)

```
x=OPENOUT filename
x=OPENOUT devicename

file_num=OPENOUT "PHONENUMS"
file_num=OPENOUT ":COM"
```

You always need to store the file handle because it must be used for all the other file commands and functions. If you choose a variable with the same name as the file, you will make programs which use a number of files easier to understand.

```
phonenums=OPENOUT "PHONENUMS"
opfile=OPENOUT opfile$
```

OPENIN

Open the file 'filename' for input only. Unlike the Z80 version of BBC BASIC, you cannot write to a file opened with OPENIN.

```
x=OPENIN filename
x=OPENIN devicename

address=OPENIN "ADDRESS"
check_file=OPENIN check_file$
comms=OPENIN ":COM"
```

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

If you try to write to a file opened for input you will get a ‘Write protected’ error (252).

OPENUP

Open the file 'filename' for update (input or output) without destroying the contents of the file. The file may be read from or written to. When the file is closed, the directory is updated to show the maximum used length of the file. None of the previously written data is lost unless it has been overwritten. Consequently, you would use OPENUP for reading serial and random files, adding to the end of serial files or writing to random files.

```
x=OPENUP filename
x=OPENUP devicename

address=OPENUP "ADDRESS"
check_file=OPENUP check_file$
comms=OPENUP ":COM"
```

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

CLOSE#

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

```
CLOSE#fnum
```

INPUT#

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

```
data=OPENIN "DATA"
:
INPUT#data,name$,age,height,sex$
:
READ# can be used as an alternative to INPUT#
```

PRINT#

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

```
PRINT#fnum,var
```

String variables are written as the character bytes in the string plus a carriage-return. Numeric variables are written as 5 bytes of binary data.

```
data=OPENOUT "DATA"
:
PRINT#data,name$,age,height,sex$
```

EXT#

Return the total length of the file opened as 'fnum'.

```
EXT#fnum
```

In the case of a sparse random-access file the value returned is the length of the file to the last byte actually written to the file. Although much of the file may well be unused, writing this 'last byte' reserved physical space in RAM for a file of this length. Thus it is possible to write a single byte to a file and get a 'No Room' error.

PTR#
A pseudo-variable which points to the position within the file from where the next byte to be read will be taken or where the next byte to be written will be put.

```
        PTR#fnum
```

When the file is OPENED, PTR# is set to zero. However, you can set PTR# to any value you like. (Even beyond the end of the file - so take care).

Reading or writing, using INPUT# and PRINT#, (and BGET# and BPUT# - explained later), takes place at the current position of the pointer. The pointer is automatically updated following a read or write operation.

A file opened with OPENUP may be extended by setting PTR# to its end (PTR# = EXT#), and then writing the new data to it. You must remember to CLOSE such a file in order to update its directory entry with its new length. A couple of examples of this are included in the sections on serial and indexed files.

PTR#-1 Returns the number of file handles still available for the entire Z88 (not just BBC BASIC(Z80)) and the ROM release number.

If you are going to display this information, you will need to do so in hexadecimal because the one (4 byte) number contains two items of information. For example:

```
        PRINT ~PTR#-1
            5A0004
```

The last 3 digits (least significant 2 bytes) are the ROM release number. The first 2 digits (most significant 2 bytes) are the number of files handles still available for use by the filing system (&5A=90).

EOF#
A function which will return -1 (TRUE) if the data file whose file handle is the argument is at (or beyond) its end. In other words, when PTR# points beyond the current end of the file.

```
        eof=EOF#fnum
```

Attempting to read beyond the current end of file will not give rise to an error. Either zero or a null string will be returned depending on the type of variable read.

EOF# is only really of use when dealing with serial (sequential) files. It indicates that PTR# is greater than the recorded length of the file (found by using EXT#). When reading a serial file, EOF# would go true when the last byte of the file had been read.

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

Writing to a byte beyond the current end of file updates the file length immediately, whether the record is physically written to the disk at that time or not.

EOF#-1 returns TRUE for an expanded Z88 and FALSE for an unexpanded Z88 (32K in slot 0).

BGET#

A function which reads a byte of data from the file opened as 'fnum', from the position pointed to by PTR#fnum. PTR#fnum is incremented by 1 following the read. A positive integer between 0 and 255 is returned (as you might expect). This can be converted into a string variable using the CHR$ function.

```
BGET#fnum

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

or, more expediently

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

BPUT#

Write the least significant byte of the variable 'var' to the file opened as 'fnum', at the position pointed to by PTR#fnum. PTR#fnum is incremented by 1 following the write.

```
BPUT#fnum,var

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

Serial Files

Introduction

The section on serial files is split into three parts. The first deals with character data files. These are the simplest type of files to use and the examples are correspondingly short. The second part looks at mixed numeric/character data files. The final part describes conversion between BBC BASIC(Z80) format files and the file formats required/produced by other systems.

Character Data Files

The first three examples are programs to write data in character format to a serial file and to read the data back. All the data is in character format and, since the files will not be read by other versions of BASIC, no extra control characters have been added.

You may notice that we have cheated a little in that a procedure is called to close the files and end the program without returning. This saves using a GOTO, but leaves the return address on the stack. However, ending a program clears the stack and no harm is done. You should not use this sort of trick anywhere else in a program. If you do you will quickly use up memory.

Ex 1 - Writing Serial Character Data

```
 10 REM F-WSER1
 20 :
 30 REM WRITING TO A SERIAL CHARACTER DATA FILE
 40 :
 50 REM This program opens a data file and writes
 60 REM serial character data to it.  The use of
 70 REM OPENOUT ensures that, even if the file
 80 REM existed before, it is cleared before
 90 REM being written to.
100 :
110 phonenos=OPENOUT "PHONENOS"
120 PRINT "File Name PHONENOS Opened as Handle ";phonenos
130 PRINT
140 REPEAT
150   INPUT "Name ? " name$
160   IF name$="" THEN PROC_end
170   INPUT "Phone Number ? " phone$
180   PRINT
190   PRINT#phonenos,name$,phone$
200 UNTIL FALSE
210 :
220 DEF PROC_end
230 CLOSE#phonenos
240 END
```

Ex 2 - Reading Serial Character Data

```
 10 REM F-RSER1
 20 :
 30 REM EXAMPLE OF READING A SERIAL CHARACTER FILE
 40 :
 50 REM This program opens a previously written
 60 REM serial file and reads it.
 70 :
 80 :
 90 phonenos=OPENIN "PHONENOS"
100 PRINT "File Name PHONENOS Opened as Handle ";phonenos
110 PRINT
120 REPEAT
130   INPUT#phonenos,name$,phone$
140   PRINT name$,phone$
150 UNTIL EOF#phonenos
160 :
170 CLOSE#phonenos
180 END
```

Ex 3 - Writing 'AT END' of Character Files

The next example extends the write program from Example 1. This new program opens the file, sets PTR# to the end (line 380) and then adds data to it. A procedure is used to open the file. This has the advantage of making the program more understandable by putting the detailed 'open at end' coding out of the main flow of the program.

```
 10 REM F-WESER1
 20 :
 30 REM EXAMPLE OF WRITING TO THE END OF A SERIAL DATA FILE
 40 :
 50 REM The program opens a file and sets PTR
 60 REM to the end before writing data to it.
 70 :
 80 REM A function is used to open the file.
 90 :
100 :
110 phonenos=FN_openend("PHONENOS")
120 PRINT "File Name PHONENOS Opened as Handle ";phonenos
130 PRINT
140 REPEAT
150   INPUT "Name ? " name$
160   IF name$="" THEN PROC_end
170   INPUT "Phone Number ? " phone$
180   PRINT
190   PRINT#phonenos,name$,phone$
200 UNTIL FALSE
210 :
220 DEF PROC_end
230 CLOSE#phonenos
240 END
250 :
260 :
270 REM Open the file 'AT END'.
280 :
290 REM If the file does not already exist, it
300 REM is created with OPENOUT.  PTR# is left
310 REM at zero and the handle is returned.  If
320 REM the file exists, PTR# is set to the end
330 REM and the file handle returned.
340 DEF FN_openend(name$)
350 LOCAL fnum
360 fnum=OPENUP(name$)
370 IF fnum=0 THEN fnum=OPENOUT(name$): =fnum
380 PTR#fnum=EXT#fnum
390 =fnum
```

Mixed Numeric/Character Data Files

The second three examples are also programs which write data to a file and read it back, but this time the data is mixed. They are simply extensions of the previous examples which illustrate the handling of mixed data.

Ex 4 - Writing a Mixed Data File

```
 10 REM F-WSER2
 20 :
 30 REM EXAMPLE OF WRITING TO A MIXED NUMERIC/CHAR DATA FILE
 40 :
 50 REM This program opens a file and writes
 60 REM numeric and char data to it.  The use
 70 REM of OPENOUT ensures that, even if the
 80 REM file exists, it is cleared before
 90 REM being written to.  Functions
100 REM are used to accept and validate
110 REM the data before writing it to the file.
120 :
130 :
140 stats=OPENOUT("STATS")
150 PRINT "File Name STATS Opened as Handle ";stats
160 PRINT
170 REPEAT
180   name$=FN_name
190   IF name$="" THEN PROC_end
200   age=FN_age
210   height=FN_height
220   sex$=FN_sex
230   PRINT
240   PRINT#stats,name$,age,height,sex$
250 UNTIL FALSE
260 :
270 DEF PROC_end
280 PRINT "The file is ";EXT#stats;" bytes long"
290 CLOSE#stats
300 END
310 :
320 :
330 REM Accept a name from the keyboard and make
340 REM sure it consists only of spaces and
350 REM upper or lower case characters. Leading
360 REM spaces are ignored on input.
370 :
380 DEF FN_name
390 LOCAL name$,FLAG,n
400 REPEAT
```

```
410   FLAG=TRUE
420   INPUT "Name ? " name$
430   IF name$="" THEN 490
440   FOR I=1 TO LEN(name$)
450     n=ASC(MID$(name$,I,1))
460     IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123) THEN FLAG=FALSE
470   NEXT
480   IF NOT FLAG THEN PRINT "No funny characters please !!!"
490 UNTIL FLAG
500 =name$
510 :
520 :
530 REM Accept the age from the keyboard and
540 REM round to one place of decimals.  Ages
550 REM of 0 or less, or 150 or more are
560 REM considered to be in error.
570 DEF FN_age
580 LOCAL age
590 REPEAT
600   INPUT "What age ? " age
610   IF age<=0 OR age >=150 THEN PRINT "No impossible ages please !!!"
620 UNTIL age>0 AND age<150
630 =INT(age*10+.5)/10
640 :
650 :
660 REM Accept the height in centimetres from
670 REM the keyboard and round to an integer.
680 REM Heights of 50 or less and 230 or more
690 REM are considered to be in error.
700 DEF FN_height
710 LOCAL height
720 REPEAT
730   INPUT "Height in centimetres ? " height
740   IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
750 UNTIL height>50 AND height<230
760 =INT(height+.5)
770 :
780 :
790 REM Accept the sex from the keyboard.  Only
800 REM words beginning with upper or lower case
810 REM M or F are OK.  The returned string is
820 REM truncated to 1 character.
830 DEF FN_sex
840 LOCAL sex$,FLAG
850 REPEAT
860   FLAG=TRUE
870   INPUT "Male or Female - M or F ? " sex$
880   IF sex$<>"" THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
```

```
890   IF sex$<>"M" AND sex$<>"F" THEN FLAG=FALSE
900   IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
910 UNTIL FLAG
920 =sex$
```

Ex 5 - Reading a Mixed Data File

```
 10 REM F-RSER2
 20 :
 30 REM EXAMPLE OF READING FROM A MIXED NUMERIC/CHAR DATA FILE
 40 :
 50 REM This program opens a file and reads
 60 REM numeric and character data from it.
 70 :
 80 :
 90 stats=OPENIN("STATS")
100 PRINT "File Name STATS Opened as Handle ";stats
110 PRINT
120 REPEAT
130   INPUT#stats,name$,age,height,sex$
140   PRINT "Name ";name$
150   PRINT "Age ";age
160   PRINT "Height in centimetres ";height
170   IF sex$="M" THEN PRINT "Male" ELSE PRINT "Female"
180   PRINT
190 UNTIL EOF#stats
200 :
210 CLOSE#stats
220 END
```

Ex 6 - Writing 'AT END' of Mixed Files

This example is similar to Example 3, but for a mixed data file.

```
 10 REM F-WESER2
 20 :
 30 REM EXAMPLE OF WRITING AT THE END OF A
 40 REM MIXED NUMERIC/CHAR DATA FILE
 50 :
 60 REM This program opens a file, sets PTR
 70 REM to its end and then writes numeric and
 80 REM character data to it.
 90 :
100 REM Functions are used to accept and
110 REM validate the data before writing it to
120 REM the file.
130 :
140 stats=FN_open("STATS")
150 PRINT "File Name STATS Opened as Handle ";stats
160 PRINT
170 REPEAT
```

```
 180   name$=FN_name
 190   IF name$="" THEN PROC_end
 200   age=FN_age
 210   height=FN_height
 220   sex$=FN_sex
 230   PRINT
 240   PRINT#stats,name$,age,height,sex$
 250 UNTIL FALSE
 260 :
 270 DEF PROC_end
 280 PRINT "The file is ";EXT#stats;" bytes long"
 290 CLOSE#stats
 300 END
 310 :
 320 :
 330 REM Open the file.  If it exists, set PTR#
 340 REM to the end and return the handle.  If
 350 REM it does not exist, open it, leave PTR#
 360 REM as it is and return the file handle.
 370 DEF FN_open(name$)
 380 LOCAL fnum
 390 fnum=OPENUP(name$)
 400 IF fnum=0 THEN fnum=OPENOUT(name$): =fnum
 410 PTR#fnum=EXT#fnum
 420 =fnum
 430 :
 440 :
 450 REM Accept a name from the keyboard and make
 460 REM sure it consists of spaces and upper or
 470 REM lower case characters.  Leading spaces
 480 REM are automatically ignored on input.
 490 DEF FN_name
 500 LOCAL name$,FLAG,n
 510 REPEAT
 520   FLAG=TRUE
 530   INPUT "Name ? " name$
 540   IF name$="" THEN 600
 550   FOR I=1 TO LEN(name$)
 560     n=ASC(MID$(name$,I,1))
 570     IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123) THEN
FLAG=FALSE
 580  NEXT
 590   IF NOT FLAG THEN PRINT "No funny characters please !!!"
 600 UNTIL FLAG
 610 =name$
 620 :
 630 :
 640 REM Accept the age from the keyboard and
 650 REM round to one place of decimals. Ages of
 660 REM 0 or less or 150 or more are in error.
 670 :
 680 DEF FN_age
 690 LOCAL age
 700 REPEAT
 710   INPUT "What age ? " age
 720   IF age<=0 OR age >=150 THEN PRINT "No impossible ages please
!!!"
 730 UNTIL age>0 AND age<150
 740 =INT(age*10+.5)/10
 750 :
 760 :
 770 REM Accept the height in centimetres from
 780 REM the keyboard and round to an integer.
 790 REM Heights of 50 or less or 230 or more
 800 REM are in error.
 810 DEF FN_height
 820 LOCAL height
 830 REPEAT
 840   INPUT "Height in centimetres ? " height
 850   IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
 860 UNTIL height>50 AND height<230
 870 =INT(height+.5)
 880 :
 890 :
 900 REM Accept the sex from the keyboard.  Only
 910 REM words beginning with upper or lower
 920 REM case M or F are valid.  The returned
 930 REM string is truncated to 1 character.
 940 DEF FN_sex
 950 LOCAL sex$,FLAG
 960 REPEAT
 970   FLAG=TRUE
 980   INPUT "Male or Female - M or F ? " sex$
 990   IF sex$<>"" THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
1000   IF sex$<>"M" AND sex$<>"F" THEN FLAG=FALSE
1010   IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
1020 UNTIL FLAG
1030 =sex$
```

Compatible Data Files

The next example tackles the problem of writing files which will be compatible with other versions of BASIC. The most common format for serial files is as follows:

- Data is written to the file as ASCII characters.

- Data items are separated by commas.
- Records are terminated by the two characters CR and LF.
- The file is terminated by a Control Z (&1A).

The example program accepts data from the keyboard and writes it to a file in the above format.

Ex 7 - Writing a Compatible Data File

```
 10 REM F-WSTD
 20 :
 30 REM EXAMPLE OF WRITING A COMPATIBLE FILE
 40 :
 50 REM This program opens a file and writes
 60 REM numeric and character data to it in a
 70 REM compatible format.  Numerics are changed
 80 REM to strings before they are written and
 90 REM the data items are separated by commas.
100 REM Each record is terminated by CR LF and
110 REM the file is terminated by a Control Z.
120 :
130 REM Functions are used to accept and
140 REM validate the data before writing it to
150 REM the file.
160 :
170 record$=STRING$(100," "): REM Reserve room for the longest
180 name$=STRING$(20," "): REM record necessary.
190 : REM It saves on string space.
200 compat=OPENOUT("COMPAT")
210 PRINT "File Name COMPAT Opened as Handle ";compat
220 PRINT
230 REPEAT
240   name$=FN_name
250   IF name$="" THEN PROC_end
260   age=FN_age
270   height=FN_height
280   sex$=FN_sex
290   PRINT
300   record$=name$+","+STR$(age)+","+STR$(height)+","+sex$
310   PRINT#compat,record$
320   BPUT#compat,&0A
330 UNTIL FALSE
340 :
350 DEF PROC_end
360 BPUT#compat,&1A
370 CLOSE#compat
380 END
390 :
400 :
410 REM Accept a name from the keyboard and make
```

```
 420 REM sure it consists only of spaces and
 430 REM upper or lower case characters. Leading
 440 REM spaces are ignored on input.
 450 :
 460 DEF FN_name
 470 LOCAL name$,FLAG,n
 480 REPEAT
 490   FLAG=TRUE
 500   INPUT "Name ? " name$
 510   IF name$="" THEN 570
 520   FOR I=1 TO LEN(name$)
 530     n=ASC(MID$(name$,I,1))
 540     IF NOT(n=32 OR n>64 AND n<91 OR n>96 AND n<123) THEN FLAG=TRUE
 550   NEXT
 560   IF NOT FLAG THEN PRINT "No funny characters please !!!"
 570 UNTIL FLAG
 580 =name$
 590 :
 600 :
 610 REM Accept the age from the keyboard and
 620 REM round to one place of decimals.  Ages
 630 REM of 0 or less or 150 or more are
 640 REM considered to be in error.
 650 DEF FN_age
 660 LOCAL age
 670 REPEAT
 680   INPUT "What age ? " age
 690   IF age<=0 OR age >=150 THEN PRINT "No impossible ages please
!!!"
 700 UNTIL age>0 AND age<150
 710 =INT(age*10+.5)/10
 720 :
 730 :
 740 REM Accept the height in centimetres from
 750 REM the keyboard and round to an integer.
 760 REM Heights of 50 or less and 230 or more
 770 REM are considered to be in error.
 780 DEF FN_height
 790 LOCAL height
 800 REPEAT
 810   INPUT "Height in centimetres ? " height
 820   IF height<=50 OR height>=230 THEN PRINT "Very funny !!!"
 830 UNTIL height>50 AND height<230
 840 =INT(height+.5)
 850 :
 860 :
 870 REM Accept the sex from the keyboard. Only
 880 REM words beginning with upper or lower
 890 REM case M or F are valid.  The returned
 900 REM string is truncated to 1 character.
 910 DEF FN_sex
 920 LOCAL sex$,FLAG
 930 REPEAT
 940   FLAG=TRUE
 950   INPUT "Male or Female - M or F ? " sex$
 960   IF sex$<>"" THEN sex$=CHR$(ASC(MID$(sex$,1,1)) AND 95)
 970   IF sex$<>"M" AND sex$<>"F" THEN FLAG=FALSE
 980   IF NOT FLAG THEN PRINT "No more sex(es) please !!!"
 990 UNTIL FLAG
1000 =sex$
```

Ex 8 - Reading a Compatible Data File

The last example in this section reads a file written in the above format and strips off the extraneous characters. The file is read character by character and the appropriate action taken. This is a simple example of how BBC BASIC(Z80) can be used to manipulate any file by processing it on a character by character basis.

```
 10 REM F-RSTD
 20 :
 30 REM EXAMPLE OF READING A COMPATIBLE FILE
 40 :
 50 REM This program opens a data file and reads
 60 REM numeric and character data from it.  The
 70 REM data is read a byte at a time and the
 80 REM appropriate action taken depending on
 90 REM whether it is a character, a comma, or
100 REM a control char.
110 compat=OPENUP("COMPAT")
120 PRINT "File Name COMPAT Opened as Handle ";compat
130 PRINT
140 REPEAT
150   name$=FN_read
160   PRINT "Name ";name$
170   age=VAL(FN_read)
180   PRINT "Age ";age
190   height=VAL(FN_read)
200   PRINT "Height in centimetres ";height
210   sex$=FN_read
220   IF sex$="M" THEN PRINT "Male" ELSE PRINT "Female"
230   PRINT
240 UNTIL FALSE
250 :
260 :
270 REM Read a data item from the file.  Treat
280 REM commas and CRs as data item terminators
290 REM and Control Z as the file terminator.
300 REM Since we are not interested in reading a
310 REM record at a time, the record terminator
320 REM CR LF is of no special interest to us.
330 REM We use the CR, along with commas, as a
332 REM data item separator and discard the LF.
334 :
340 DEF FN_read
350 LOCAL data$,byte$,byte
360 data$=""
370 REPEAT
380   byte=BGET#compat
390   IF byte=&1A OR EOF#compat THEN CLOSE#compat: END
400   IF NOT(byte=&0A OR byte=&0D OR byte=&2C) THEN
```

```
data$=data$+CHR$(byte)
410 UNTIL byte=&0D OR byte=&2C
420 =data$
```

Random (Relative) Files

Introduction

There are three example random file programs. The first is very simple, but it demonstrates the principle of random access files. The second expands the first into quite a useful database program. The final example is an inventory program. Although it does not provide application dependent features, it would serve as it stands and it is sufficiently well structured to be expanded without too many problems.

Designing the File

Unlike other versions of BASIC, there is no formalised record structure in BBC BASIC. A file is considered to be a continuous stream of bytes (characters) and you can directly access any byte of the file. This approach has many advantages, but most files are logically considered as a sequence of records (some of which may be empty). How then do we create this structure and access our logical records?

Record Structure

Creating the structure is quite simple. You need to decide what information you want to hold and the order in which you want to store it. In the first example, for instance, we have two items of information (fields) per logical record; the name and the remarks. The name can be a maximum of 30 characters long and the remarks a maximum of 50 characters. So our logical record has two fields, one 30 characters long and the other 50 characters long. When the name string is written to disk it will be terminated by a CR - and so will the remarks string. So each record will be a maximum of 82 characters long.

We haven't finished yet, however. We need to be able to tell whether any one record is 'live' or empty (or deleted). To do this we need an extra byte at the start of each record which we set to one value for 'empty' and another for 'live'. In all the examples we use 0 to indicate 'empty' and NOT 0 to indicate 'live'. We are writing character data to the file so we could use the first byte of the name string as the indicator because the lowest ASCII code we will be storing is 32 (space). You can't do this for mixed data files because this byte could hold a data value of zero. Because of this, we have chosen to use an additional byte for the indicator in all the examples.

Our logical record thus consists of:

1	indicator byte
31	bytes for the name
51	bytes for the remarks

Thus the maximum amount of data in each record is 83 bytes. Because we cannot tell in advance how big each record needs to be (and we may want to change it later), we must assume that ALL the records will be this length. Since most of the records will be smaller than this, we are going to waste quite a lot of space in our random access file, but this is the penalty we pay for convenience and comparative simplicity.

When we write the data to the file, we could insist that each field was treated as a fixed length field by packing each string out with spaces to make it the 'correct' length. This would force each field to start at its 'proper' byte within the record. We don't need to do this, however, because we aren't going to randomly access the fields within the record; we know the order of the fields within the record and we are going to read them sequentially into appropriately named variables. We can write the fields to the file with each field following on immediately behind the previous one. All the 'spare' room is now left at the end of the record and not split up at the end of each field.

Accessing The Records

In order to access any particular record, you need to set PTR# to the first byte of that record. Remember, you can't tell BBC BASIC(Z80) that you want 'record 5', because it knows nothing of your file and record structure. You need to calculate the position of the first byte of 'record 5' and set PTR# to this value.

To start with, let's call the first record on the file 'record zero', the second record 'record 1', the third record 'record 2', etc. The first byte of 'record zero' is at byte zero on the file. The first byte of 'record 1' is at byte 83 on the file. The first byte of 'record 2' is at byte 166 (2*83) on the file. And so on. So, the start point of any record can be calculated by:

```
first_byte= 83*record_number
```

Now, we need to set PTR# to the position of this byte in order to access the record. If the record number was held in 'recno' and the file handle in 'fnum', we could do this directly by:

```
PTR#fnum=83*recno
```

However, we may want to do this in several places in the program so it would be better to define and use a function to set PTR# as illustrated below.

```
190 ...
200 PTR#fnum=FN_ptr(recno)
210 ...
etc

900 DEF FN_ptr(record)=83*record
```

Whilst the computer is quite happy with the first record being 'record zero', us mere humans find it a little confusing. What we need is to be able to call the first record 'record 1', etc. We could do this without altering the function which calculates the start position of each record, but we would waste the space allocated to 'record 0' since we would never use it. We want to call it 'record 1' and the program wants to call it 'record 0'. We can change the function to cater for this. If we subtract 1 from the record number before we multiply it by the record length, we will get the result we want. Record 1 will start at byte zero, record 2 will start at byte 83, etc. Our function now looks like this:

```
DEF FN_ptr(record)=83*(record-1)
```

In our example so far we have used a record length of 83. If we replace this with a variable 'rec_len' we have a general function which we can use to calculate the start position of any record in the file in any program. (You will need to set rec_len to the appropriate value at the start of the program.) The function now becomes:

```
DEF FN_ptr(record)=rec_len*(record-1)
```

We use this function (or something very similar to it) in the following three example programs using random access files.

Ex 9 - Simple Random Access File

```
 10 REM F-RAND1
 20 :
 30 REM VERY SIMPLE RANDOM ACCESS PROGRAM
 40 :
 50 REM This program maintains a random access
 60 REM file of names and remarks.  There is
 70 REM room for a maximum of 20 entries. Each
 80 REM name can be up to a max of 30 chars
 90 REM long and each remark up to 50 chars.
100 REM The first byte of the record is set non
110 REM zero (in fact &FF) if there is a record
120 REM present.  This gives a maximum record
130 REM length of 1+31+51=83. (Including CRs)
140 :
150 bell$=CHR$(7)
160 temp$=STRING$(50," ")
170 maxrec=20
180 rec_len=83
190 ans$=""
200 CLS
210 WIDTH 0
220 fnum=OPENUP "RANDONE"
230 IF fnum=0 fnum=FN_setup("RANDONE")
240 REPEAT
250   REPEAT
260     INPUT '"Enter record number: "ans$
270     IF ans$="0" CLOSE#fnum:CLS:END
280     IF ans$="" record=record+1 ELSE record=VAL(ans$)
290     IF record<1 OR record>maxrec PRINT bell$;
300   UNTIL record>0 AND record<=maxrec
310   PTR#fnum=FN_ptr(record)
320   PROC_display
330   INPUT '"Do you wish to change this record" ,ans$
340   PTR#fnum=FN_ptr(record)
350   IF FN_test(ans$) PROC_modify
360 UNTIL FALSE
```

```
370 END
380 :
390 :
400 DEF FN_test(A$) =LEFT$(A$,1)="Y" OR LEFT$(A$,1)="y"
410 :
420 :
430 DEF FN_ptr(record)=rec_len*(record-1)
440 REM This makes record 1 start at PTR# = 0
450 :
460 :
470 DEF PROC_display
480 PRINT '"Record number ";record'
490 flag=BGET#fnum
500 IF flag=0 PROC_clear:ENDPROC
510 INPUT#fnum,name$,remark$
520 PRINT name$;" ";remark$ '
530 ENDPROC
540 :
550 :
560 DEF PROC_clear
570 PRINT "Record empty"
580 name$=""
590 remark$=""
600 ENDPROC
610 :
620 :
630 DEF PROC_modify
640 PRINT '"(Enter <Enter> for no change or DELETE to delete)"'
650 INPUT "Name ",temp$
660 temp$=LEFT$(temp$,30)
670 IF temp$<>"" name$=temp$
680 INPUT "Remark ",temp$
690 temp$=LEFT$(temp$,50)
700 IF temp$<>"" remark$=temp$
710 INPUT '"Confirm update record",ans$
720 IF NOT FN_test(ans$) ENDPROC
730 IF name$="DELETE" BPUT#fnum,0:ENDPROC
740 BPUT#fnum,255
750 PRINT#fnum,name$,remark$
760 ENDPROC
770 :
780 :
790 DEF FN_setup(fname$)
800 PRINT "Setting up the database file"
810 fnum=OPENOUT(fname$)
820 FOR record=1 TO maxrec
830   PTR#fnum=FN_ptr(record)
840   BPUT#fnum,0
```

```
850 NEXT
860 =fnum
```

Ex 10 - Simple Random Access Database

The second program in this sub-section expands the previous program into a simple, but quite versatile, database program. A setup procedure has been added which allows you to specify the file name. If it is a new file, you are then allowed to specify the number of records and the number, name and size of the fields you wish to use. This information is stored at the start of the file. If the file already exists this data is read from the records at the beginning of the file. The function for calculating the start position of each record is modified to take into account the room used at the front of the file to store information about the database.

```
 10 REM F-RAN
 20 REM SIMPLE DATABASE PROGRAM
 30 REM Written by R T Russell Jan 1983
 40 REM Mod for BBC BASIC(Z80): D Mounter Dec 1985
 50 :
 60 REM This is a simple database program.  You
 70 REM are asked for the name of the file you
 80 REM wish to use.  If the file does not
 90 REM already exist, you are asked to enter
100 REM the number and format of the records.
110 REM If the file does already exist, the file
120 REM specification is read from the file.
130 :
140 @%=&90A
150 bell$=CHR$(7)
160 CLS
170 WIDTH 0
180 INPUT '"Enter the filename of the data file: "filename$
190 fnum=OPENUP(filename$)
200 IF fnum=0 fnum=FN_setup(filename$) ELSE PROC_readgen
210 PRINT
220 :
230 REPEAT
240   REPEAT
250     INPUT '"Enter record number: "ans$
260     IF ans$="0" CLOSE#fnum:CLS:END
270     IF ans$="" record=record+1 ELSE record=VAL(ans$)
280     IF record<1 OR record>maxrec PRINT bell$;
290   UNTIL record>0 AND record<=maxrec
300   PTR#fnum=FN_ptr(record)
310   PROC_display
320   INPUT '"Do you wish to change this record" ,ans$
330   PTR#fnum=FN_ptr(record)
```

```
340   IF FN_test(ans$) PROC_modify
350 UNTIL FALSE
360 END
370 :
380 :
390 DEF FN_test(A$) =LEFT$(A$,1)="Y" OR LEFT$(A$,1)="y"
400 :
410 :
420 DEF FN_ptr(record)=base+rec_len*(record-1)
430 :
440 :
450 DEF FN_setup(filename$)
460 PRINT "New file."
470 fnum=OPENOUT(filename$)
480 REPEAT
490   INPUT "Enter the number of records (max 1000): "maxrec
500 UNTIL maxrec>0 AND maxrec<1001
510 REPEAT
520   INPUT "Enter number of fields per record (max 20): "fields
530 UNTIL fields>0 AND fields<21
540 DIM title$(fields),size(fields),A$(fields)
550 FOR field=1 TO fields
560   PRINT '"Enter title of field number ";field;": ";
570   INPUT ""title$(field)
580   PRINT
590     REPEAT
600     INPUT "Max size of field (characters)",size(field)
610   UNTIL size(field)>0 AND size(field)<256
620 NEXT field
630 rec_len=1
640 PRINT#fnum,maxrec,fields
650 FOR field=1 TO fields
660   PRINT#fnum,title$(field),size(field)
670   rec_len=rec_len+size(field)+1
680 NEXT field
690 base=PTR#fnum
700 :
710 FOR record=1 TO maxrec
720   PTR#fnum=FN_ptr(record)
730   BPUT#fnum,0
740 NEXT
750 =fnum
760 :
770 :
780 DEF PROC_readgen
790 rec_len=1
800 INPUT#fnum,maxrec,fields
810 DIM title$(fields),size(fields),A$(fields)
```

```
820 FOR field=1 TO fields
830   INPUT#fnum,title$(field),size(field)
 840   rec_len=rec_len+size(field)+1
 850 NEXT field
 860 base=PTR#fnum
 870 ENDPROC
 880 :
 890 :
 900 DEF PROC_display
 910 PRINT '"Record number ";record'
 920 flag=BGET#fnum
 930 IF flag=0 PROC_clear:ENDPROC
 940 FOR field=1 TO fields
 950   INPUT#fnum,A$(field)
 960   PRINT title$(field);" ";A$(field)
 970 NEXT field
 980 ENDPROC
 990 :
1000 :
1010 DEF PROC_clear
1020 FOR field=1 TO fields
1030   A$(field)=""
1040 NEXT
1050 ENDPROC
1060 :
1070 :
1080 DEF PROC_modify
1090 PRINT '"(Enter <Enter> for no change)"'
1100 FOR field=1 TO fields
1110   REPEAT
1120     PRINT title$(field);" ";
1130     INPUT LINE ""A$
1140     IF A$="" PRINT TAB(POS,VPOS-1)title$(field);" ";A$(field)
1150     REM TAB(POS,VPOS-1) moves the cursor up 1 line
1160   UNTIL LEN(A$)<=size(field)
1170   IF A$<>"" A$(field)=A$
1180 NEXT field
1190 INPUT '"Confirm update record",ans$
1200 IF NOT FN_test(ans$) ENDPROC
1210 IF A$(1)="DELETE" BPUT#fnum,0:ENDPROC
1220 BPUT#fnum,255
1230 FOR field=1 TO fields
1240   PRINT#fnum,A$(field)
1250 NEXT field
1260 ENDPROC
```

Ex 11 - Random Access Inventory Program

The final example in this sub-section is a full-blown inventory program. Rather than go through all its aspects at the start, they are discussed at the appropriate point in the listing. (These comments do not have line numbers and are not, of course, part of the program.)

```
 10 REM F-RAND
 20 :
 30 REM Written by Doug Mounter Jan 1982
 40 REM Modified for BBC BASIC(Z80) Dec 1985
 50 :
 60 REM EXAMPLE OF A RANDOM ACCESS FILE
 70 :
 80 REM This is a simple inventory program.  It
 90 REM uses the item's part number as the key
 92 REM and stores:
100 REM  The item description - char max len 30
110 REM  The quantity in stock - numeric
120 REM  The re-order level - numeric
130 REM  The unit price - numeric
140 REM In addition, the first byte of the rec
150 REM is used as a valid data flag.  Set to 0
160 REM if empty, D if the record has been
170 REM deleted or V if the record is valid.
180 REM This gives a MAX record len of 47 bytes
190 REM (Don't forget the CR after the string)
200 :
210 PROC_initialise
220 inventry=FN_open("INVENTRY")
```

The following section of code is the command loop. You are offered a choice of functions until you eventually select function 0. The more traditional ON GOSUB statement has been used for menu selection processing. The newer ON PROC statement is illustrated in the indexed file example which follows. There are some forward jumps within procedures, etc to overcome the lack of a multi line IF statement. It would have been possible to have used further procedures, but the whole thing would have become rather laboured.

```
230 REPEAT
240   CLS
250   PRINT TAB(5,3);"If you want to:-"'
260   PRINT TAB(10);"End This Session";TAB(55);"Type 0"
270   PRINT TAB(10);"Amend or Create an Entry";TAB(55);"Type 1"
280   PRINT TAB(10);"Disp Inventory for One Part";TAB(55);"Type 2"
290   PRINT TAB(10);"Alter Stock  of One Part";TAB(55);"Type 3"
300   PRINT TAB(10);"Disp Items to Reorder";TAB(55);"Type 4"
```

```
310   PRINT TAB(10);"Recover a Deleted Item";TAB(55);"Type 5"
320   PRINT TAB(10);"List Deleted Items";TAB(55);"Type 6"
330   PRINT TAB(10);"Set Up a New Inventory";TAB(55);"Type 9"
340   REPEAT
350     PRINT TAB(5,15);bell$;
360     PRINT "Please enter selection (0 to 6 or 9) ";
370     function$=GET$
380   UNTIL function$>"/" AND function$<"8" OR function$="9"
390   function=VAL(function$)
400   ON function GOSUB 500,670,810,1100,1350,1540,1770,1790,1840 ELSE
410 UNTIL function=0
420 CLS
430 PRINT "Inventory File Closed" ''
440 CLOSE#inventry
450 END
460 :
470 :
```

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.

```
480 REM AMEND/CREATE AN ENTRY
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 :
```

This subroutine allows you to look at a record without the ability to change or delete it.

```
650 REM DISPLAY AN ENTRY
660 :
670 REPEAT
680   CLS
690   PRINT "DISPLAY"
700   partno=FN_getpartno
710   flag=FN_getrec(partno,ASC"V")
720   PROC_display(flag)
730   PRINT '
740   PRINT "Do you wish to view another part?";
750 UNTIL GET$="N"
760 RETURN
770 :
780 :
```

The purpose of this subroutine is to allow you to update the stock level without having to amend the rest of the record.

```
 790 REM CHANGE THE STOCK LEVEL FOR ONE PART
 800 :
 810 REPEAT
 820   CLS
 830   PRINT "CHANGE STOCK"
 840   partno=FN_getpartno
 850   flag=FN_getrec(partno,ASC"V")
 860   REPEAT
 870     PROC_display(flag)
 880     PROC_cteos
 890     REPEAT
 900       PRINT TAB(0,12);:PROC_cteol
 910       INPUT "What is the change ? " temp$
 920       change=VAL(temp$)
 930     UNTIL INT(change)=change AND stock+change>=0
 940     IF temp$="" flag=FALSE:GOTO 1000
 950     stock=stock+change
 960     PROC_display(flag)
 970     PRINT'"Is this correct ? ";
 980     temp$=GET$
 990 :
1000   UNTIL NOT flag OR temp$="Y"
1010   PROC_write(partno,flag,ASC"V")
1020   PRINT return$;bell$;
1030   PRINT "Do you want any more updates ? ";
1040 UNTIL GET$="N"
1050 RETURN
```

```
1060 :
1070 :
```

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.

```
1080 REM DISPLAY ITEMS BELOW REORDER LEVEL
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 :
```

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.

```
1330 REM RECOVER A DELETED ENTRY
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 :
```

This subroutine lists all the deleted entries so you can check you really don't want the data.

```
1520 REM LIST DELETED ENTRIES
1530 :
1540 partno=1
1550 REPEAT
1560   CLS
1570   PRINT "DELETED ITEMS"'
1580   line_count=2
1590   REPEAT
1600     flag=FN_getrec(partno,ASC"D")
1610     IF NOT flag THEN 1660
1620     PRINT "Part Number ";partno
1630     PRINT "Description ";desc$'
1640     line_count=line_count+3
1650 :
1660     partno=partno+1
1670     temp$=INKEY$(0)
1680   UNTIL partno>maxpartno OR line_count>20 OR temp$<>""
1690   PRINT TAB(0,23);bell$;"Push any key to continue or E to end ";
1700 UNTIL partno>maxpartno OR GET$="E"
1710 partno=0
1720 RETURN
1730 :
1740 :
1750 REM DUMMY RETURNS FOR INVALID FUNCTION NUMs
1760 :
1770 RETURN
1780 :
1790 RETURN
1800 :
1810 :
1820 REM REINITIALISE THE INVENTORY DATA FILE
1830 :
1840 CLS
1850 PRINT TAB(0,3);bell$;"Are you sure you want to set up a new
```

```
inventory?"
1860 PRINT "You will DESTROY ALL THE DATA YOU HAVE ACCUMULATED so far."
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$="YES" PROC_setup(inventry)
1930 IF temp$="NEW" function=0
1940 RETURN
1950 :
1960 :
```

This is where all the variables that you usually write as CHR$(#) go. Then you can find them if you want to change them.

```
1970 REM INITIALISE ALL THE VARIOUS PRESETS ETC
1980 :
1990 DEF PROC_initialise
2010 bell$=CHR$(7)
2020 return$=CHR$(13)
2030 rec_length=47
2040 partno=0
```

If you initially set strings to the maximum length you will ever use, you will save prevent the generation of 'garbage'.

```
2050 desc$=STRING$(30," ")
2060 temp$=STRING$(40," ")
2070 WIDTH 0

2130 REM OPEN FILE AND RETURN THE FILE HANDLE
2140 :
2150 REM If the file already exists, the largest permitted
2160 REM part number is read into maxpartno.
2170 REM If it is a new file, the file is
2180 REM initialised and the largest part
2190 REM number is written as the first record.
2200 :
2210 DEF FN_open(name$)
2220 fnum=OPENUP(name$)
2230 IF fnum>0 INPUT#fnum,maxpartno: =fnum
2240 fnum=OPENOUT(name$)
2250 CLS
```

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

```
2260 PROC_setup(fnum)
2270 =fnum
2280 :
2290 REM SET UP THE FILE
2300 :
2310 REM Ask for maximum part number required,
2320 REM write it as the first record and then
2330 REM write 0 in to first byte of each rec.
2340 :
2350 DEF PROC_setup(fnum)
2360 REPEAT
2370   PRINT TAB(0,12);bell$;:PROC_cteos
```

```
2380   INPUT "What is the highest part number required (Max
5000)",maxpartno
2390 UNTIL maxpartno>0 AND maxpartno<5000 AND INT(maxpartno)=maxpartno
2400 PTR#fnum=0
2410 PRINT#fnum,maxpartno
2420 FOR partno=1 TO maxpartno
2430   PTR#fnum=FN_ptr(partno)
2440   BPUT#fnum,0
2450 NEXT
2460 partno=0
2470 ENDPROC
2480 :
2490 :
```

Ask for the required part number. If a null is entered, make the next part number one more than the last.

```
2500 REM GET AND RETURN THE REQUIRED PART NUMBER
2510 :
2520 DEF FN_getpartno
2530 REPEAT
2540   PRINT TAB(0,5);bell$;:PROC_cteos
2550   PRINT "Enter a Part Number Between 1 and ";maxpartno '
2560   IF partno=maxpartno THEN 2590
2570   PRINT "The Next Part Number is ";partno+1;
2580   PRINT " Just hit RETURN to get this"'
2590 :
2600   INPUT "What is the Part Number You Want ", partno$
2610   IF partno$<>"" partno=VAL(partno$):GOTO 2630
2620   IF partno=maxpartno partno=0 ELSE partno=partno+1
2630 :
2640   PRINT TAB(35,9);partno;:PROC_cteol
2650 UNTIL partno>0 AND partno<maxpartno+1 AND INT(partno)=partno
2660 =partno
2670 :
2680 :
2690 REM GET THE RECORD FOR THE PART NUMBER
2700 :
2710 REM Return TRUE if the record exists and
2720 REM FALSE if not  If the record does not
2730 REM exist, load desc$ with "No Record" The
2740 REM remainder of the record is set to 0.
2742 :
2750 DEF FN_getrec(partno,type)
2760 stock=0
2770 reord=0
2780 price=0
2790 PTR#inventry=FN_ptr(partno)
```

```
2800 test=BGET#inventry
2810 IF test=0 desc$="No Record": =FALSE
2820 IF test=type THEN 2850
2830 IF type=86 desc$="Record Deleted" ELSE desc$="Record Exists"
2840 =FALSE
2850 :
2860 INPUT#inventry,desc$
2870 INPUT#inventry,stock,reord,price
2880 =TRUE
2890 :
2900 :
```

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.

```
2910 REM CALCULATE THE VALUE OF PTR FOR THIS REC
2920 :
2930 DEF FN_ptr(partno)=partno*rec_length+5-rec_length
2940 :
2950 :
```

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.

```
2960 REM AMEND THE RECORD
2970 :
2980 DEF FN_amend(partno)
2990 PRINT return$;:PROC_cteol:PRINT TAB(0,4);
3000 PRINT "Please Complete the Details for Part Number ";partno
3010 PRINT "Just hit Return to leave the entry as it is"'
3020 flag=FALSE
3030 type=ASC"V"
3040 INPUT "Description - Max 30 Chars " temp$
3050 IF temp$="DELETE" type=ASC"D": =TRUE
3060 temp$=LEFT$(temp$,30)
3070 IF temp$<>"" desc$=temp$:flag=TRUE
3080 IF desc$="No Record" OR desc$="Record Deleted" =FALSE
3090 INPUT "Current Stock Level " temp$
3100 IF temp$<>"" stock=VAL(temp$):flag=TRUE
3110 INPUT "Reorder Level " temp$
3120 IF temp$<>"" reord=VAL(temp$):flag=TRUE
3130 INPUT "Unit Price " temp$
3140 IF temp$<>"" price=VAL(temp$):flag=TRUE
3150 =flag
3160 :
3170 :
```

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

```
3180 REM WRITE THE RECORD
3190 :
3200 DEF PROC_write(partno,flag,type)
3210 IF NOT flag ENDPROC
3220 PTR#inventry=FN_ptr(partno)
3230 BPUT#inventry,type
3240 PRINT#inventry,desc$,stock,reord,price
3250 ENDPROC
```

```
3260 :
3270 :
```

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.

```
3280 REM DISPLAY THE RECORD DETAILS
3290 :
3300 DEF PROC_display(flag)
3310 PRINT TAB(0,5);:PROC_cteos
3320 PRINT "Part Number ";partno'
3330 PRINT "Description ";desc$
3340 IF NOT flag ENDPROC
3350 PRINT "Current Stock Level ";stock
3360 PRINT "Reorder Level ";reord
3370 PRINT "Unit Price ";price
3380 ENDPROC
3390 :
3400 :
```

The two following procedures rely on the screen width being 80 characters:

```
3410 REM There are no 'native' clear to end of
3420 REM line/screen vdu procedures.  The
3430 REM following two procedures clear to the
3440 REM end of the line/screen.
3450 DEF PROC_cteol
3460 LOCAL x,y
3470 x=POS:y=VPOS
3480 IF y=31 PRINT SPC(79-x); ELSE PRINT SPC(80-x);
3490 PRINT TAB(x,y);
3500 ENDPROC
3510 :
3520 :
3530 DEF PROC_cteos
3540 LOCAL I,x,y
3550 x=POS:y=VPOS
3560 IF y<31 FOR I=y TO 30:PRINT SPC(80);:NEXT
3570 PRINT SPC(79-x);TAB(x,y);
3580 ENDPROC
```

Indexed Data Files

Deficiencies of Random Access Files

As you will see if you dump a random file, a lot of space is wasted. This is because all the records must be allocated the same amount of space, otherwise you could not calculate where the record started. For large data files, over 50% of the space can be wasted. Under these circumstances it is possible to save space by using two files, one as an index to the other. In order for this to work efficiently, you must have complete control over the file pointer. Not many versions of BASIC allow this control, but it is quite simple with BBC BASIC.

The Address Book Program

The final program is an example of an indexed file. It is a computer implementation of the address book discussed way back at the beginning of these notes. Two files are used, one as an index to the other. Both are serial and no space is wasted between records.

File Organisation

The files are organised as shown below:

NAME.NDX (index file)

maxrec 5 bytes	length 5 bytes	index$(1) 1 to 31 bytes	index(1) 5 bytes	index$(2) 1 to 31 bytes	index(2) 5 bytes	etc. →

Where index(n) points to a record in the data file as follows:

ADDRESS.DTA (data file)

Phone Num 1 to 31 bytes	Address 1 1 to 31 bytes	Address 2 1 to 31 bytes	Address 3 1 to 31 bytes	Address 4 1 to 31 bytes	Post Code 1 to 31 bytes

`maxrec`	Is the maximum number of records permitted in this file. The practical limit is governed by the amount of memory available for the index arrays which are held in memory. 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?
`length`	Is the number of entries in the index.
`index(n)`	Is the value of PTR#datanum just prior to the first byte of the data for this entry being written to it. In the Random File examples this value was calculated and it increased by a constant amount for every record.

Program Organisation

The example looks horribly long and complicated. However the actual file handling bits are quite simple. The rest is, as usual, required for tidy input and output of data. The meat of the program is in the procedures and functions for putting and deleting index entries and finding the right place in the index. The latter uses a routine called a 'binary chop' (you could get arrested for that). This looks simple, and it is - when it works. If you are interested there is a flow chart and a brief explanation of how it works at the end of these notes. For the faithful, just use it. It takes considerably less time than any other method to search an ordered list.

The Index

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

Ex 12 (the LAST)

```
 10 REM F-INDEX
 20 REM EXAMPLE OF AN INDEXED FILE
 30 :
 40 REM Written by Doug Mounter - Feb 1982
 50 REM Modified for BBC BASIC(Z80) - Dec 1985
 60 :
 70 REM This is a simple address book filing
 80 REM system.  It will accept names, telephone
 90 REM numbers and addresses and store them in a
100 REM file called ADDRESS.DTA.  The index is in
110 REM name order and is kept in a file called
120 REM NAME.NDX.  All the fields are character
122 REM and the maximum length  of any field
124 REM is 30.
130 :
140 PROC_initialise
150 PROC_open_files
160 ON ERROR IF ERR<>17 PRINT:REPORT:PRINT" At line ";ERL:END
170 REPEAT
180   CLS
190   PRINT TAB(5,3);"If you want to:-" '
200   PRINT TAB(10);"End This Session";TAB(55);"Type 0"
210   PRINT TAB(10);"Enter Data";TAB(55);"Type 1"
220   PRINT TAB(10);"Search For/Delete an Entry";TAB(55);"Type 2"
230   PRINT TAB(10);"List in Alphabetical Order";TAB(55);"Type 3"
240   PRINT TAB(10);"Reorg data File and Index";TAB(55);"Type 4";
250   REPEAT
260     PRINT TAB(5,11);
270     PRINT "Please enter the appropriate number (0 to 4) ";
280     function$=GET$
290     PRINT return$;:PROC_cteol
300   UNTIL function$>"/" AND function$<"5"
```

```
310   function=VAL(function$)
320   PRINT TAB(54,function+5);"<====<<";
330   ON function PROC_enter,PROC_search,PROC_list,PROC_reorg,ELSE
340 UNTIL function=0
350 CLS
360 PROC_close_files
370 *ESC ON
380 PRINT "Address Book Files Closed"''
390 END
400 :
410 :
420 REM ENTER DATA
430 :
440 DEF PROC_enter
450 flag=TRUE
460 temp$=""
470 i=1
480 REPEAT
490   REPEAT
500     IF temp$="N" PROC_message("Data NOT Accepted")
510     PROC_get_data
520     IF length=maxrec OR data$(1)="" flag=FALSE:GOTO 590
530     IF data$(1)="+" OR data$(1)="-" PROC_message("Bad Data"):GOTO
590
540     i=FN_find_place(0,data$(1))
550     IF i>0 PROC_message("Duplicate Record")
560     PRINT '"Is this data correct ? ";
570     temp$=FN_yesno
580     :
590   UNTIL NOT flag OR temp$<>"N"
600   PROC_cteos
610   IF NOT flag THEN 670
620   PROC_put_index(i,data$(1),PTR#datanum)
630   FOR i=2 TO 7
640     PRINT#datanum,data$(i)
650   NEXT
660   :
670 UNTIL NOT flag
680 ENDPROC
690 :
700 :
710 REM SEARCH FOR AN ENTRY
720 :
730 DEF PROC_search
740 i=0
750 REPEAT
760   PRINT TAB(0,11);:PROC_cteol
770   INPUT "What name do you want to look for ",name$
```

```
780   IF name$="" THEN 800
790   IF name$<>""IF name$="DELETE" PROC_delete(i) ELSE
i=FN_display(i,name$)
800 UNTIL name$=""
810 ENDPROC
820 :
830 :
840 REM LIST IN ALPHABETICAL ORDER
850 :
860 DEF PROC_list
870 entry=1
880 REPEAT
890   CLS
900   line_count=0
910   REPEAT
920     PRINT TAB(0,line_count);
930     PROC_read_data(entry)
940     PROC_print_data
950     entry=entry+1
960     line_count=line_count+8
970     temp$=INKEY$(0)
980   UNTIL entry>length OR line_count>16 OR temp$<>""
990   PROC_message("Push any key to continue or E to end ")
1000 UNTIL entry>length OR GET$="E"
1010 ENDPROC
1020 :
1030 :
1040 REM REORGANISE THE DATA FILE AND INDEX
1050 :
1060 DEF PROC_reorg
1070 entry=1
1080 PRINT TAB(0,13);"Reorganising the Data File" '
1090 newdata=OPENOUT"ADDRESS.BAK"
1100 REPEAT
1110   PROC_read_data(entry)
1120   index(entry)=PTR#newdata
1130   FOR i=2 TO 7
1140     PRINT#newdata,data$(i)
1150   NEXT
1160   entry=entry+1
1170 UNTIL entry>length
1180 CLOSE#newdata
```

The time taken to rename a file can be considerable.

```
1190 PRINT "Re-naming the Data File" '
1200 *REN ADDRESS.$$$=ADDRESS.BAK
```

```
1210 PRINT "*";
1220 *REN ADDRESS.BAK=ADDRESS.DTA
1230 PRINT "*";
1240 *REN ADDRESS.DTA=ADDRESS.$$$
1250 PRINT "*";
1260 datanum=OPENUP "ADDRESS.DTA"
1270 ENDPROC
1280 :
1290 :
1300 REM INITIALISE VARIABLES AND ARRAYS
1310 :
1320 DEF PROC_initialise
1340 *ESC OFF
1350 esc$=CHR$(27)
1360 bell$=CHR$(7)
1370 return$=CHR$(13)
1380 maxrec=100
1390 :
1400 REM The maximum record number, maxrec, is
1402 REM read in
1410 REM PROC_read_index if the file already exists.
1420 :
1430 DIM message$(7)
1440 FOR i=1 TO 7
1450   READ message$(i)
1460 NEXT
1470 DATA Name,Phone Number,Address,-- " --,-- "--,-- " --,Post Code
1480 :
1490 DIM data$(7)
1500 FOR i=1 TO 7
1510   data$(i)=STRING$(30," ")
1520 NEXT
1530 temp$=STRING$(255," ")
1540 temp$=""
1550 :
1610 REM OPEN THE FILES
1620 :
1630 DEF PROC_open_files
1640 indexnum=OPENUP"NAME.NDX"
1650 datanum=OPENUP"ADDRESS.DTA"
1660 IF indexnum=0 OR datanum=0 PROC_setup ELSE PROC_read_index
1670 PTR#datanum=EXT#datanum
1680 ENDPROC
1690 :
1700 :
1710 REM SET UP NEW INDEX AND DATA FILES
1720 :
1730 DEF PROC_setup
```

```
1740 CLS
1750 PRINT TAB(0,13);"Setting Up Address Book"
1760 indexnum=OPENOUT"NAME.NDX"
1770 datanum=OPENOUT"ADDRESS.DTA"
1780 length=0
1790 PRINT#indexnum,maxrec,length
1800 CLOSE#indexnum
1810 DIM index$(maxrec+1),index(maxrec+1)
1820 index$(0)=""
1830 index(0)=0
1840 index$(1)=CHR$(&FF)
1850 index(1)=0
1860 ENDPROC
1870 :
1880 :
1890 REM READ INDEX AND LENGTH OF DATA FILE
1900 :
1910 DEF PROC_read_index
1920 CLS
1930 INPUT#indexnum,maxrec,length
1940 DIM index$(maxrec+1), index(maxrec+1)
1950 index$(0)=""
1960 index(0)=0
1970 FOR i=1 TO length
1980   INPUT#indexnum,index$(i),index(i)
1990 NEXT
2000 CLOSE#indexnum
2010 index$(length+1)=CHR$(&FF)
2020 index(length+1)=0
2030 ENDPROC
2040 :
2050 :
2060 REM WRITE INDEX AND CLOSE FILES
2070 :
2080 DEF PROC_close_files
2090 indexnum=OPENOUT"NAME.NDX"
2100 PRINT#indexnum,maxrec,length
2110 FOR i=1 TO length
2120   PRINT#indexnum,index$(i),index(i)
2130 NEXT
2140 CLOSE#0
2150 ENDPROC
2160 :
2170 :
2180 REM WRITE A MESSAGE AT LINE 23
2190 :
2200 DEF PROC_message(line$)
2210 LOCAL x,y
```

```
2220 x=POS
2230 y=VPOS
2240 PRINT TAB(0,23);:PROC_cteol:PRINT bell$;line$;
2250 PRINT TAB(x,y);
2260 ENDPROC
2270 :
2280 :
2290 REM GET A Y/N ANSWER
2300 :
2310 DEF FN_yesno
2320 LOCAL temp$
2330 temp$=GET$
2340 IF temp$="y" OR temp$="Y" ="Y"
2350 IF temp$="n" OR temp$="N" ="N"
2360 =""
2370 :
2380 :
```

This procedure makes use of the machine code routine at the end of the program. It works in a similar fashion to the clear-to-end-of-line and clear-to-end-of-screen procedures defined towards the end of the program.

```
2390 REM CLEAR 9 LINES FROM PRESENT POSITION
2400 :
2410 DEF PROC_clear9
2420 LOCAL x,y,i
2430 PRINT return$;
2440 A%=&A20:B%=0:C%=720:D%=0
2450 CALL int10
2460 ENDPROC
2470 :
2480 :
2490 REM GET INPUT DATA - LIMIT TO 30 CHAR
2500 :
2510 DEF PROC_get_data
2520 LOCAL i
2530 PRINT TAB(0,13);
2540 PROC_clear9
2550 IF length=maxrec PROC_message("Add Book Full")
2560 FOR i=1 TO 7
2570   PRINT TAB(10);message$(i);TAB(25);
2580   INPUT temp$
2590   data$(i)=LEFT$(temp$,30)
2600   IF data$(1)="" i=7
2610 NEXT
2620 ENDPROC
2630 :
2640 :
```

```
2650 REM FIND AND DISPLAY THE REQUESTED DATA
2660 :
2670 DEF FN_display(i,name$)
2680 PRINT TAB(0,12);:PROC_cteos
2690 i=FN_find_place(i,name$)
2700 IF i<0 PROC_message("Name Not Known - Next Highest Given")
2710 PROC_read_data(i)
2720 PRINT
2730 PROC_print_data
2740 =i
2750 :
2760 :
```

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

```
2770 REM DELETE THE ENTRY FROM THE INDEX
2780 :
2790 DEF PROC_delete(i)
2800 INPUT "Are you SURE ",temp$
2810 PRINT TAB(0,VPOS-1);:PROC_cteos
2820 IF temp$<>"YES" ENDPROC
2830 IF i<0 i=-i
2840 FOR i=i TO length
2850   index$(i)=index$(i+1)
2860   index(i)=index(i+1)
2870 NEXT
2880 length=length-1
2890 ENDPROC
2900 :
2910 :
```

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.

```
2920 REM READ DATA FOR ENTRY i
2930 :
2940 DEF PROC_read_data(i)
2950 PTRdata=PTR#datanum
2960 IF i<0 i=-i
2970 PTR#datanum=index(i)
2980 data$(1)=index$(i)
2990 FOR i=2 TO 7
3000   INPUT#datanum,data$(i)
3010 NEXT
3020 PTR#datanum=PTRdata
3030 ENDPROC
```

```
3040 :
3050 :
3060 REM PRINT data$() ON VDU
3070 :
3080 DEF PROC_print_data
3090 LOCAL i
3100 FOR i=1 TO 7
3110   IF data$(i)<>"" PRINT TAB(10);message$(i);TAB(25);data$(i)
3120   IF data$(1)=CHR$(&FF) i=7
3130 NEXT
3140 ENDPROC
3150 :
3160 :
```

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 the gap made at position i and add 1 to the length.

```
3170 REM PUT A NEW ENTRY IN INDEX AT POSITION i
3180 :
3190 DEF PROC_put_index(i,entry$,ptr)
3200 LOCAL j
3210 IF i<0 i=-i
3220 FOR j=length+1 TO i STEP -1
3230   index$(j+1)=index$(j)
3240   index(j+1)=index(j)
3250 NEXT
3260 index$(i)=entry$
3270 index(i)=ptr
3280 length=length+1
3290 ENDPROC
3300 :
3310 :
```

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 new entry). Thus if a part of the index looked like:

(34)	BERT
(35)	FRED
(36)	JOHN

and you entered with FRED, it would return 35. However if you entered with GEORGE, it would return -36.

The function consists of two parts. The first looks at the entry$ to see if it should just up or down the entry number by 1, taking account of wrap-around at the start and end of the index. The second part is the binary chop advertised with such telling wit in the introduction to indexed files. Since we enter this function with the entry pointer i set to its previous value, we must cater for a negative value.

```
3320 REM FIND ENTRY IN INDEX OR PLACE TO PUT IT
3330 :
3340 DEF FN_find_place(i,entry$)
3350 LOCAL top,bottom
3360 IF i<0 i=-i
3370 IF entry$="+" AND i<length =i+1
3380 IF entry$="+" AND i=length =1
3390 IF entry$="-" AND i>1 =i-1
3400 IF entry$="-" AND i<2 =length
```

Here, at last, **T H E B I N A R Y C H O P**
This bit moves the pointer up the index to the first of any duplicate entries.

```
3410 top=length+1
3420 bottom=0
3430 i=(top+1) DIV 2
3440 IF entry$<>index$(i) i=FN_search(entry$)
3450 REPEAT
3460   IF entry$=index$(i-1) i=i-1
3470 UNTIL entry$<>index$(i-1)
3480 IF entry$=index$(i) =i ELSE =-i
3490 :
3500 :
3510 REM DO THE SEARCHING FOR FN_find_place
3520 :
3530 DEF FN_search(entry$)
3540 REPEAT
3550   IF entry$>index$(i) bottom=i ELSE top=i
3560   i=(top+bottom+1) DIV 2: REM round
3570 UNTIL entry$=index$(i) OR top=bottom+1
3580 =i
3590 :
3600 :
```

the two following procedures rely on the screen width being 80 characters:

```
3410 REM There are no 'native' clear to end of
3420 REM line/screen vdu procedures.  The
3430 REM following two procedures clear to the
3440 REM end of the line/screen.
3450 DEF PROC_cteol
3460 LOCAL x,y
3470 x=POS:y=VPOS
3480 IF y=31 PRINT SPC(79-x); ELSE PRINT SPC(80-x);
3490 PRINT TAB(x,y);
3500 ENDPROC
3510 :
3520 :
3530 DEF PROC_cteos
3540 LOCAL I,x,y
3550 x=POS:y=VPOS
3560 IF y<31 FOR I=y TO 30:PRINT SPC(80);:NEXT
3570 PRINT SPC(79-x);TAB(x,y);
3580 ENDPROC
```

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

The Binary Chop

Explanation

The quickest way to find an entry in an ORDERED list is not to search through it from start to end, but to continue splitting the list in two until you reach the entry you are looking for. You begin by setting one pointer to the bottom of the list, another to the top, and a third to mid-way between bottom and top. Then you compare the entry pointed to by this third pointer with the number you are searching for. If your number is bigger you make the bottom equal the pointer, if not make the top equal to it. Then you repeat the process.

Let's try searching the list of numbers below for the number 14.

bottom>	`(1)`	`3`	Set bottom to the lowest position in the list, and top to the highest. Set the pointer to `(top+bottom)/2`. Is that entry 14? No it's more, so set top to the current value of pointer and repeat the process.
	`(2)`	`6`	
	`(3)`	`8`	
	`(4)`	`14`	
pointer>	`(5)`	`19`	
	`(6)`	`23`	
	`(7)`	`34`	
	`(8)`	`45`	
top>	`(9)`	`61`	

bottom>	`(1)`	`3`	Set the pointer to `(top+bottom)/2`. Is that entry 14? No it's less, so set bottom to the current value of pointer and try again.
	`(2)`	`6`	
pointer>	`(3)`	`8`	
	`(4)`	`14`	
top>	`(5)`	`19`	
	`(6)`	`23`	
	`(7)`	`34`	
	`(8)`	`45`	
	`(9)`	`61`	

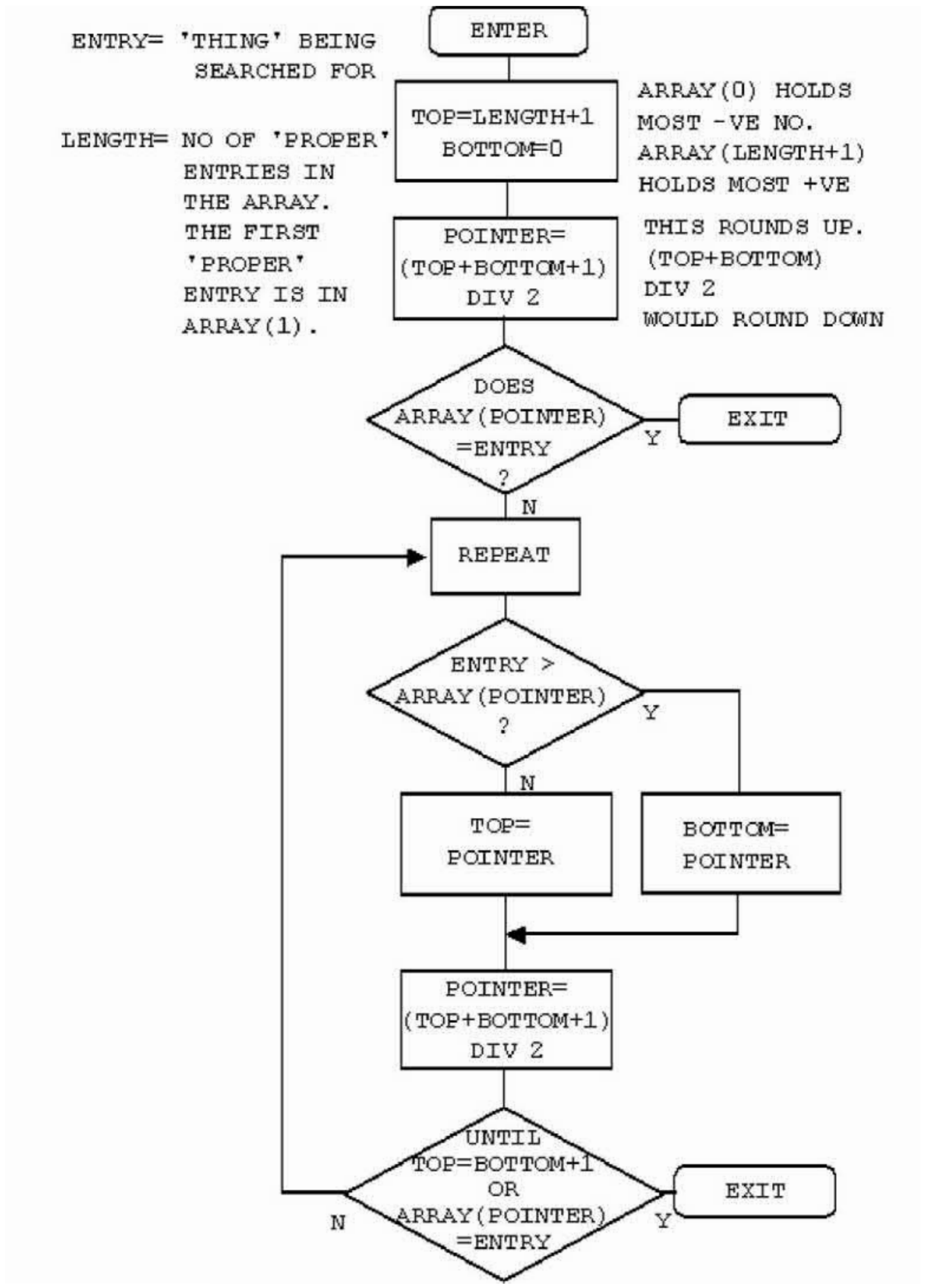
	`(1)`	`3`
	`(2)`	`6`
bottom>	`(3)`	`8`
pointer>	`(4)`	`14`
top>	`(5)`	`19`
	`(6)`	`23`
	`(7)`	`34`
	`(8)`	`45`
	`(9)`	`61`

Set the pointer to `(top+bottom)/2`. Is that entry 14? Yes, so exit with the pointer set to the position in the list of the number you are looking for.

As you can imagine, things are not always as simple as this carefully chosen example. You have to cater for the number not being there, and for the list being empty. There are a number of ways of doing this, but the easiest is to add two numbers of your choice to the list. Make the first entry the most negative number the computer can hold, and the last entry the most positive. This will prevent you from ever trying to search outside the list. Preventing a perpetual loop when the number you want is not in the list is quite simple, just exit when 'top' is equal to 'bottom'+1. If you have not found the number by then, it's not in the list.

You can use this routine to add numbers to the list in order. If you can't find the number, you exit with the position it should go to in the list. Just move all the numbers under it down one slot and put the new number in. This works just as well when the list is empty except for your two 'end markers'.

Have a look at the flowchart on the next 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 7 goes to find the number. The AVERAGE number of comparisons required for a linear search would be 50.


ENTRY= 'THING' BEING SEARCHED FOR
LENGTH= NO OF 'PROPER' ENTRIES IN THE ARRAY. THE FIRST 'PROPER' ENTRY IS IN ARRAY(1).
ENTER
TOP=LENGTH+1
BOTTOM=0
ARRAY(0) HOLDS MOST -VE NO. ARRAY(LENGTH+1) HOLDS MOST +VE
POINTER= (TOP+BOTTOM+1) DIV 2
THIS ROUNDS UP. (TOP+BOTTOM) DIV 2 WOULD ROUND DOWN
DOES ARRAY(POINTER) =ENTRY ?
Y
EXIT
N
REPEAT
ENTRY > ARRAY(POINTER) ?
Y
N
TOP= POINTER
BOTTOM= POINTER
POINTER= (TOP+BOTTOM+1) DIV 2
UNTIL TOP=BOTTOM+1 OR ARRAY(POINTER) =ENTRY
N
Y
EXIT

Annex A: Table of ASCII Codes

Binary	**Hex**	**Dec**	**Char**	
00000000	00	0	◊=	NUL
00000001	01	1	◊A	SOH Start of Heading
00000010	02	2	◊B	STX Start of Text
00000011	03	3	◊C	ETX End of Text
00000100	04	4	◊D	EOT End of Transmit
00000101	05	5	◊E	ENQ Enquiry
00000110	06	6	◊F	ACK Acknowledge
00000111	07	7	◊G	BEL Bell - Audible Signal
00001000	08	8	◊H	BS Back Space
00001001	09	9	◊I	HT Horizontal Tab
00001010	0A	10	◊J	LF Line Feed
00001011	0B	11	◊K	VT Vertical Tab
00001100	0C	12	◊L	FF Form Feed
00001101	0D	13	◊M	CR Carriage Return
00001110	0E	14	◊N	SO Shift Out
00001111	0F	15	◊O	SI Shift In
00010000	10	16	◊P	DLE Data Link Escape
00010001	11	17	◊Q	DC1 X On
00010010	12	18	◊R	DC2 Aux On
00010011	13	19	◊S	DC3 X Off
00010100	14	20	◊T	DC4 Aux Off
00010101	15	21	◊U	NAK Negative Acknowledge
00010110	16	22	◊V	SYN Synchronous File
00010111	17	23	◊W	ETB End of Transmitted Block
00011000	18	24	◊X	CAN Cancel

00011001	19	25	◊Y	EM End of Medium
00011010	1A	26	◊Z	SUB Substitute
00011011	1B	27	◊[	ESC Escape
00011100	1C	28	◊\	FS File Separator
00011101	1D	29	◊]	GS Group Separator
00011110	1E	30	◊£	RS Record Separator
00011111	1F	31	◊-	US Unit Separator

Binary	**Hex**	**Dec**	**Char**
00100000	20	32	Space
00100001	21	33	!
00100010	22	34	"
00100011	23	35	#
00100100	24	36	$
00100101	25	37	%
00100110	26	38	&
00100111	27	39	'
00101000	28	40	(
00101001	29	41	)
00101010	2A	42	*
00101011	2B	43	+
00101100	2C	44	,
00101101	2D	45	-
00101110	2E	46	.
00101111	2F	47	/
00110000	30	48	0
00110001	31	49	1
00110010	32	50	2
00110011	33	51	3

00110100	34	52	4
00110101	35	53	5
00110110	36	54	6
00110111	37	55	7
00111000	38	56	8
00111001	39	57	9
00111010	3A	58	:
00111011	3B	59	;
00111100	3C	60	<
00111101	3D	61	=
00111110	3E	62	>
00111111	3F	63	?

Binary	Hex	Dec	Char	Binary	Hex	Dec	Char
01000000	40	64	@	01100000	60	96	` (◊’)
01000001	41	65	A	01100001	61	97	a
01000010	42	66	B	01100010	62	98	b
01000011	43	67	C	01100011	63	99	c
01000100	44	68	D	01100100	64	100	d
01000101	45	69	E	01100101	65	101	e
01000110	46	70	F	01100110	66	102	f
01000111	47	71	G	01100111	67	103	g
01001000	48	72	H	01101000	68	104	h
01001001	49	73	I	01101001	69	105	i
01001010	4A	74	J	01101010	6A	106	j
01001011	4B	75	K	01101011	6B	107	k
01001100	4C	76	L	01101100	6C	108	l
01001101	4D	77	M	01101101	6D	109	m
01001110	4E	78	N	01101110	6E	110	n

01001111	4F	79	O
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
01011011	5B	91	[
01011100	5C	92	\
01011101	5D	93	]
01011110	5E	94	^
01011111	5F	95	_
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: Mathematical Functions

BBC BASIC(Z80) has more intrinsic mathematical functions than many other versions of BASIC. Those that are not provided may be calculated as shown below.

Function	Calculation
SECANT	`SEC(X)=1/COS(X)`
COSECANT	`CSC(X)=1/SIN(X)`
COTANGENT	`COT(X)=1/TAN(X)`
Inverse SECANT	`ARCSEC(X)=ACS(1/X)`
Inverse COSECANT	`ARCCSC(X)=ASN(1/X)`

Inverse COTANGENT	`ARCCOT(X)=ATN(1/X)`
	 `=PI/2-ATN(X)`

Hyperbolic SINE	`SINH(X)=(EXP(X)-EXP(-X))/2`
Hyperbolic COSINE	`COSH(X)=(EXP(X)+EXP(-X))/2`
Hyperbolic TANGENT	`TANH(X)=EXP(-X)/(EXP(X)+EXP(-X))*2+1`
Hyperbolic SECANT	`SECH(X)=2/(EXP(X)+EXP(-X))`
Hyperbolic COSECANT	`CSCH(X)=2/(EXP(X)-EXP(-X))`

Hyperbolic COTANGENT

```
COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
```

Inverse Hyperbolic SIN

```
ARCSINH(X)=LN(X+SQR(X*X+1))
```

Inverse Hyperbolic COSINE

```
ARCCOSH(X)=LN(X+SQR(X*X-1))
```

Inverse Hyperbolic TANGENT

```
ARCTANH(X)=LN((1+X)/(1-X))/2
```

Inverse Hyperbolic SECANT

```
ARCSECH(X)=LN((SQR(-X*X+1)+1)/X)
```

Inverse Hyperbolic COSECANT

```
ARCCSCH(X)=LN((SGN(X)*SQR(X*X+1)+1)/X
```

Inverse Hyperbolic COTANGENT

```
ARCCOTH(X)=LN((X+1)/(X-1))/2
```

LOGn(X)

```
LOGn(X)=LN(X)/LN(n)
=LOG(X)/LOG(n)
```

Annex C: Error Messages and Codes

Summary
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 #		

* Not applicable to BBC BASIC(Z80)

Trappable - Operating System

No	Error	
192	Too many open files	
220	Bad syntax	
222	Channel	
252	File Access errors	Reported as one of the following:
		Already exists
		Bad filename
		End of file
		File Not found
		File type mismatch
		In use
		Read Protected
		Suspended
		Write protected
253	Sorry, not implemented	
254	Bad command	

Untrappable - Error Code 0

No room	RENUMBER space
Silly	LINE space
Sorry	Bad program
Failed at nnn	

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

Details

BBC BASIC(Z80)'s error messages and codes are briefly explained below in alphabetical order.

Accuracy lost (23)

Before BBC BASIC(Z80) calculates trigonometric functions (sin, cos, etc) of very large angles the angles are reduced to +/- PI radians. The larger the angle, the greater the inaccuracy of the reduction, and hence the result. When this inaccuracy becomes unacceptable, BBC BASIC(Z80) will issue an 'Accuracy lost' error message.

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 BBC BASIC(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 a command name is not recognized as a valid BBC BASIC(Z80) command. Star commands which are unknown to BBC BASIC(Z80) are passed to CP/M-80. If the command is unrecognised by CP/M-80, an untrappable 'Bad command or file name' error occurs.

Bad DIM (10)

Arrays must be positively dimensioned. In other words, the numbers within the brackets must not be negative. This error would be produced by the following example.

```
    DIM table(20,-10)
```

Bad 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 name (204)

This error is generated if a path name exceeds 64 characters in length.

Bad program

From time to time BBC BASIC(Z80) checks to see that the program in memory is of the correct format (See Annex E). If it is unable to follow the program from the start to the 'program end marker' it will report this untrappable error. The error can be caused by a read error, by only loading part of the program or by overwriting part of the program in some way. (Machine code programmers beware.) Without a full understanding of how a program is stored in memory, there is little you can do to recover a bad program.

Can't match FOR (33)

BBC BASIC(Z80) has been unable to find a FOR statement corresponding to the NEXT statement.

Channel (222)

This error is generated by the filing system. It occurs if you try to use a channel which has not been opened, possibly because you are using the wrong channel number.

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

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...
```

End of file (252)

This is one of the ‘catch all’ filing system errors. It will occur if you attempt to read beyond the end of a file.

Escape (17)

This error is generated by pressing the `[Esc]` key. You can trap this, and other errors, by using the ON ERROR GOTO statement.

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, BBC BASIC(Z80) tries to resolve all line numbers referred to by GOTO and GOSUB statements. Should it fail, it will generate a 'Failed at nnn' error, where nnn is the RENUMBERED line which contains the unresolved reference.

The following example:

```
100 REM Demonstration renumber fail program
110 GOTO 250
120 END
```

would renumber as:

```
10 REM Demonstration renumber fail program
20 GOTO 250
30 END
```

and generate the error message 'Failed at 20'.

File not found (252)

This is one of the ‘catch all’ filing system errors. This error will occur if you try to LOAD, *LOAD or CHAIN a file which does not exist.

File type mismatch (252)

This is one of the ‘catch all’ filing system errors. It will occur if you try to access a file in :ROM.0 or open a RAM device or directory.

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
```

In use (252)

This is one of the ‘catch all’ filing system errors. It will occur if you try to delete or rename a file that is currently open. It will also occur if you try to OPENOUT a file already opened with OPENOUT.

LINE space

A program line is too long to be represented in BBC BASIC(Z80)'s internal format.

Log range (22)

Logarithms for zero and negative numbers do not exist. This error message will be generated if you try to calculate the log of zero or a negative number or raise a negative number to a non-integer power.

Missing , (5)

This error message is generated if BBC BASIC(Z80) was unable to find a comma where one was expected. The following example would give rise to this error.

```
20 PRINT TAB(10 5)
```

Missing " (9)

This error message is generated if BBC BASIC(Z80) was unable to find a double-quote where one was expected. The following example would give rise to this error.
10 name$="Douglas

Missing) (27)

This error message is generated if BBC BASIC(Z80) was unable to find a closing bracket where one was expected. The following example would give rise to this error.
10 PRINT SQR(num

Missing # (45)

This error will occur if BBC BASIC(Z80) is unable to find a hash symbol (a pound symbol on some computers) where one was expected. The following example would cause this error.

```
CLOSE 7
```

Mistake (4)

This error will be generated if BBC BASIC(Z80) is unable to make any sense at all of the input line.

-ve root (21)

This error message will occur if BBC BASIC(Z80) attempted to calculate the square root of a negative number. It is possible for this error to occur with ASN and ACS as well as SQR.

```
90 num=-20
100 root=SQR(num)
```

No GOSUB (38)

This error message will be generated if BBC BASIC(Z80) finds a RETURN statement without first encountering a GOSUB statement. (See the sub-section on Program Flow Control.)

No FN (7)

If BBC BASIC(Z80) encounters an end of function without calling a function definition, this error message will be issued. If you forget to put multi-line function definitions out of harm's way at the end of the program you are very likely to get this error message. (See the sub-section on Procedures and Functions.)

No FOR (32)

This error message indicates that BBC BASIC(Z80) has found a NEXT statement without first encountering a FOR statement.

No PROC (13)

If BBC BASIC(Z80) encounters an ENDPROC without performing (calling) a procedure definition, this error message will be issued. If you forget to put multi-line procedure definitions out of harm's way at the end of the program you are very likely to get this error message. (See the sub-section on Procedures and Functions.)

No REPEAT (43)

This error message indicates that BBC BASIC(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.

Although it is most unlikely, you will also get a ‘No Room’ error if you use up all the operating system’s file handles.

No such FN/PROC (29)

When BBC BASIC(Z80) encounters a name beginning with FN or PROC it expects to be able to find a corresponding function or procedure definition. This error will occur if such a definition does not exist.

No such line (41)

This error will occur if BBC BASIC(Z80) tries to GOTO, GOSUB, TRACE or RESTORE to a non-existent line number.

No such variable (26)

Variables are brought into existence by assigning a value to them or making them LOCAL in a function or procedure definition. This error message will be generated if you try to use a variable on the right-hand side of an assignment or access it in a PRINT statement before it has been created. As shown below, you can create variables very simply.

```
10 count=0
20 name$=""
```

No TO (36)

This error message will be generated if BBC BASIC(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.)

ON range (40)

This error will be generated if, in a simple ON GOTO/GOSUB/PROC statement, the control variable was less than 1 or greater than the number of entries in the ON list. These exceptions can be trapped in ON GOTO/GOSUB/PROC statements by using the ELSE option. The first example below will generate an 'ON range' error, whilst the second is correct.

```
10 num=4
20 ON num GOTO 100,200,300

10 num=4
20 ON num GOTO 100,200,300 ELSE 1000
```

ON syntax (39)

This error will be reported if the ON...GOTO statement was malformed. 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 (JR or DJNZ) or used and Index register offset (IX,IY) of more than +127 or -128 bytes or you used a 16 bit port address when only an 8 bit address is allowed.

Read protected (252)

This is one of the ‘catch all’ filing system errors. It will occur if you try to read from a device that is only capable of output. For example:

```
>F=OPENIN(“:SCR”)   : REM Handle is allocated successfully
>PRINT BGET#F       : REM Error occurs when reading from device

>Read protected
```

RENUMBER space

When BBC BASIC RENUMBERs a program it has to build a cross-reference table of line numbers. If there is insufficient memory to hold this table, the 'RENUMBER space' error results. In this case you can still renumber the program using the RENUMBER.COM utility program supplied.

Sorry, not implemented

The Z88 version of BBC BASIC(Z80) does not have any sound or colour commands. Equally, if the Z88 Patch has not been applied and you are using a ROM version up until V4.0, you don’t have any graphics commands either. This error occurs if you try to use one of the valid keywords which are not implemented on the Z88.

String too long (19)

You will get this error 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
```

Suspended (252)

This is one of the ‘catch all’ filing system errors. It will occur if Z88 was switched off by pressing both the [SHIFT] keys (or a battery low interrupt occurred) whilst data was being read from the comms port using BBC BASIC(Z80).

Syntax error (16)

A command was terminated incorrectly. In other words, the first part of the command was recognized, but the rest of it was meaningless or incomplete. Unlike Mistake, BBC BASIC(Z80) was able to recognise the start of the command.

Too big (20)

This error will occur if a number is entered or calculated which is too big for BBC BASIC(Z80) to cope with.

Too many open files (192)

This error will occur if you try to open more than 10 files at any one time inside BBC BASIC(Z80).

Type mismatch (6)

This error indicates that a number was encountered when a string was expected and vice-versa. Don't forget that this can occur if the actual parameters and the formal parameters for a function or procedure do not correspond. (See sub-section on Procedures and Functions for details of parameter passing to functions and procedures.)

Write protected (252)

This is one of the ‘catch all’ filing system errors. It will occur if you try to write to a file opened with OPENIN. It will also occur if you try to write to a device that is only capable of input. For example:

```
>F=OPENIN(“:INP”)        : REM Handle is allocated successfully
>BPUT#F,1                : REM Error occurs when writing to device

>Write protected
```

Annex D: Format of Program and Variables in Memory

Memory Map

BBC BASIC(Z80) runs under the Z88's operating system. When a Z88 has 128Kbytes or more in slot 0 or slot 1, it becomes an expanded machine. The difference between an expanded and non-expanded Z88 are listed below.

	Non-expanded	**Expanded**
Memory available for variables and program	&1D00	&9D00
Maximum map width	80 pixels	256 pixels
User characters	16 (see below)	64
Value of EOF#-1	0 (FALSE)	-1 (TRUE)

For Z88 ROMs V2.2 - V4.0: If you wish to add 128Kbytes or more memory to your Z88 without turning it into an expanded machine, put the memory card in slot 2.

An unexpanded Z88 can use 64 user defined characters, but if an 80 pixel map is used, the last 48 of these will be overwritten by map information when PipeDream is used. Reducing the map width to 64 pixels, or not using the map at all, allows for free use of all 64-characters.

The PTR#, EXT#, and EOF# file attributes functions may be used to discover information about your Z88.

Attribute	**Information**
PTR#-1	High word = No. of free handles in the system. Low word = ROM version code Because 2 values are packed into one number, you need to display the value returned by PTR#-1 in hexadecimal. For example: `PRINT ~PTR#-1`
EXT#-1	Estimate of free memory. See later under 'Memory for Files and Applications'
EOF#-1	TRUE (-1) = expanded FALSE (0) = unexpanded

Memory for Files and Applications

The memory in each card slot is available to the Filer and to applications (and the special device `:RAM.-`) and this memory is allocated on a ‘first come, first served’ basis. However, because of the differences in the way memory is managed by the Filer and applications, the Filer generally has access to slightly less memory than either the applications or the device `:RAM.-`.

The free memory value returned by EXT#-1 is the memory available to applications or the device `:RAM.-`. Consequently, the memory available for files (other than `:RAM.-`) is generally somewhat less than the value returned by EXT#-1.

A file is limited by the size of the device in which it is held (`:RAM.1`, `:RAM.2`, etc). The device `:RAM.-` can use all the free memory anywhere in the system. Consequently, it can be larger than any one physical device and it is useful for very large files. However, there is a bug in versions of the Z88 operating system up to V4.0 which will cause the system to fail if you perform a soft reset whilst any files exist in `:RAM.-`. Ensure that you delete all files in `:RAM.-` as soon as you have finished with them and do not perform a soft reset whilst any files exist in this device.

Auto-boot CLI File and Z88 ROM’s up to V4.0

If there is an EPROM or FLASH card which contains a file called ‘boot.cli’ in slot 3 when the Z88 is reset, the file will be loaded into `:RAM.-` and executed as a CLI command file. Make sure you have deleted this file from `:RAM.-` when you have finished with it.

BBC BASIC(Z80) Program

By default, your program will start on the page boundary immediately following the interpreter's data area and the 'dynamic data structures' will immediately follow your program. The total group of the dynamic data structures is called the 'heap'. The base of the program control stack is located at HIMEM. HIMEM is at &C000 on an expanded Z88 and at &4000 if the Z88 is unexpanded.

As your program runs, the heap expands upwards towards the stack and the stack expands downwards towards the heap. If the two should meet, you get a 'No room' error. Fortunately, there is a limit to the amount by which the stack and the heap expands.

In general, the heap only expands whilst new variables are being declared. However, bad management of string variables can also cause the heap to expand.

In addition to running your program, the stack is also used 'internally' by the BBC BASIC(Z80) interpreter. Its size fluctuates but, in general, it expands every time you increase the depth of nesting of your program structure and every time you increase the number of local variables in use.

The Memory Map

In its Z88 incarnation, BASIC occupies a memory map of the following form:

`&C000 - &FFFF`	BBC BASIC interpreter application (16K)
`&4000 - &BFFF`	(additional 32K of program/workspace, expanded Z88)
`&2000 - &3FFF`	BASIC program/workspace
`&0000 - &1FFF`	Operating system use (and application stack)

BASIC's program/workspace is arranged in the following manner:

```
          +-------------------+   &FFFF
          | BBC BASIC(Z80)    |
          .                   .
          .                   .
HIMEM     +-------------------+   &BFFF or &3FFF
          | Stack             |
          +-------------------+   Current limit of the stack
          .                   .   (Stack expands downwards)
          . Unused memory     .
          .                   .
          +-------------------+   Current limit of the heap
          | Heap              |   (Heap expands upwards)
LOMEM     +-------------------+
          .                   .
          .                   .
TOP       +-------------------+
          | Program           |
PAGE      +-------------------+   &2300
          | Workspace for     |
          | interpreter       |
          +-------------------+   &2000
          | Operating system  |
          | system usage      |
          +-------------------+   &0000
```

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. You will probably only need to change HIMEM if you want to put some machine code sub-routines at the top of memory.

HIMEM	The first address at the top of memory which is not available for use by BBC BASIC(Z80). The base of the program stack is set at HIMEM. (The first 'thing' stored on the stack goes at HIMEM-1.)
LOMEM	The start address for the heap. The first of the dynamic data structures starts at LOMEM.

TOP	The first free location after the end of your program. Unless you have set LOMEM yourself, LOMEM=TOP. You cannot directly set TOP. It alters as you enter your program. The current length of your program is given by:

```
PRINT TOP-PAGE
```

PAGE	The address of the start of your program. You can place several programs in memory and switch between them by using PAGE. Don't forget to control LOMEM as well. If you don't, the heap for one program might overwrite another program.

Memory Management

There is little you can do to control the growth of the stack. However, with care, you can control the growth of the heap. You can do this by limiting the number of variables you use and by good string variable management.

Limiting the Number of Variables

Each new variable occupies room on the heap. Restricting the length of the names of variables and limiting the number of variables used will limit the size of the heap. However, of the techniques available to you, this is the least rewarding. In addition, it leads to incomprehensible programs because your variable names become meaningless. You should keep this technique in the back of your mind whilst you are programming, but only apply it rigorously if you are really stuck for space.

String Management

Garbage Generation

Unlike numeric variables, string variables do not have a fixed length. When you create a string variable it is added to the heap and sufficient memory is allocated for the initial value of the string. If you subsequently assign a longer string to the variable there will be insufficient room for it in its original position and the string will have to be relocated with its new value at the top of the heap. The initial area will then become 'dead' and the heap will have grown by the new length of the string. The areas of 'dead' memory are called garbage. As more and more re-assignments take place, the heap grows and eventually there is no more room. Thus, it is possible to run out of room for variables even though there should be enough space.

Memory Allocation for String Variables

You can overcome the problem of 'garbage' by reserving enough memory for the longest string you will ever put into a variable before you use it. You do this simply by assigning a string of spaces to the variable. If your program needs to find an empty string the first time it is used, you can subsequently assign a null string to it. The same technique can be used for string arrays. The example below sets up a single dimensional string array with room for 20 characters in each entry, and then empties it.

```
10 DIM names$(10)
20 FOR i=0 TO 10
30   name$(i)=STRING$(20," ")
40 NEXT
50 stop$=""
60 FOR i=0 TO 10
70   name$(i)=""
80 NEXT
```

Assigning a null string to stop$ prevents the space for the last entry in the array being recovered when it is emptied.

Program Storage in Memory

The program is stored in memory in the format shown below. The first program line commences at PAGE.

L.length	LSB	MSB	token				:	token			&0D
	Line No		Reserved Word Tokens								
			Program Line								CR

Line Length

The line length includes the line length byte. The address of the start of the next line is found by adding the line length to the address of the start of the current line. The end of the program is indicated by a line length of zero and a line number of &FFFF.

Line Number

The line number is stored in two bytes, LSB first. The end of the program is indicated by a line number of &FFFF and a line length of zero.

Statements

With the exception of the symbols '*', '=' and '[' and the optional reserved word LET, each statement in the line commences with the appropriate reserved word token. Reserved words are tokenised wherever they occur. A token is indicated by bit 7 of the byte being set. Statements within a line are separated by colons.

Line Terminator

Each program line (except the last) is terminated by a carriage-return (&0D).

Variable Storage in Memory

Variables are held within memory as linked lists (chains). The first variable in each chain is accessed via an index which is maintained by BBC BASIC(Z80). There is an entry in the index for each of the characters permitted as the first letter of a variable name. Each entry in the index has a word (two bytes) address field which points to the first variable in the linked list with a name starting with its associated character. If there are no variables with this character as the first character in the name, the pointer word is zero. The first word of all variables holds the address of the next variable in the chain. The address word of the last variable in the chain is zero. All addresses are held in the standard Z80 format - LSB first.

The first variable created for each starting character is accessed via the index and subsequently created variables are accessed via the index and the chain. Consequently, there is some speed advantage to be gained by arranging for all your variables to start with a different character. Unfortunately, this can lead to some pretty unreadable names and programs.

Integer Variables

Integers are held in two's complement format. They occupy 4 bytes, with the LSB first. Bit 7 of the MSB is the sign bit. To make up the complete variable, the address word, the name and a separator (zero) byte are added to the number. The format of the memory occupied by an integer variable called 'NUMBER%' is shown below. Note that since the first character of the name is found via the index, it is not stored with the variable.

LSB	MSB	U	M	B	E	R	%	0	LSB			MSB
Address of next var. starting with the same letter		Rest of Name						Zero	Number			

The smallest amount of space is taken up by a variable with a single letter name. The static integer variables, which are not included in the variable chains, use the names A% to Z%. Thus, the only single character names available for dynamic integer variables are a% to z% plus _% and `% (CHR$(96)). As shown below, integer variables with these names will occupy 8 bytes.

LSB	MSB	%	0	LSB			MSB
Addr. of next variable..				Number			

Real Variables

Real numbers are held in binary floating point format. The mantissa is held as a 4 byte binary fraction in sign and magnitude format. Bit 7 of the MSB of the mantissa is the sign bit. When working out the value of the mantissa, this bit is assumed to be 1 (a decimal value of 0.5). The exponent is held as a single byte in 'excess 127' format. In other words, if the actual exponent is zero, the value of stored in the exponent byte is 127. To make up the complete variable, the address word, the name and a separator (zero) byte are added to the number. The format of the memory occupied by a real variable called 'NUMBER' is shown below.

LSB	MSB	U	M	B	E	R	0	LSB			MSB	EXP
Addr. of...		Rest of Name						Mantissa (number)				

As with integer variables, variables with single character names occupy the least memory. (However, the names A to Z are available for dynamic real variables.) Whilst a real variable requires an extra byte to store the number, the '%' character is not needed in the name. Thus, integer and real variables with the same name occupy the same amount of memory. However, this does not hold for arrays, since the name is only stored once.

In the following examples, the bytes are shown in the more human-readable manner with the MSB on the left.

The value 5.5 would be stored as shown below.

Mantissa				Exponent
.0011 0000	0000 0000	0000 0000	0000 0000	1000 0010
Sign Bit				
&30	00	00	00	&82

Because the sign bit is assumed to be 1, this would become:

Mantissa				Exponent
.1011 0000	0000 0000	0000 0000	0000 0000	1000 0010
&B0	00	00	00	&82

The equivalent in decimal is:

```
         (0.5+0.125+0.0625) * 2^(130-127)
    =    0.6875 * 2^3
    =    0.6875 * 8
    =    5.5
```

BBC BASIC(Z80) stores integer values in real variables in a special way which allows the faster integer arithmetic routines to be used if appropriate. The presence of an integer value in a real variable is indicated by the stored exponent being zero. Thus, if the stored exponent is zero, the real variable is being used to hold an integer and the 4 byte mantissa holds the number in normal integer format.

Depending on how it is put there, an integer value can be stored in a real variable in one of two ways. For example,

```
        number=5
```

will set the exponent to zero and store the integer &00 00 00 05 in the mantissa. On the other hand,

```
        number=5.0
```

will set the exponent to &82 and the mantissa to &20 00 00 00.

The two ways of storing an integer value are illustrated in the following four examples.

Example 1

number=5	`& 00  00  00  00  05`	Integer 5

Example 2

number=5.0	`& 82  20  00  00  00`	Real 5.0

This is treated as

```
                    & 82  A0  00  00  00

=                   (0.5+0.125)*2^(130-127)
=                   0.625*8
=                   5
```

because the sign bit is assumed to be 1.

Example 3

number=-5	`& 00  FF  FF  FF  FB`

The 2's complement gives

```
                    & 00  00  00  00  05    Integer -5
```

Example 4

number=-5.0	`& 82  A0  00  00  00`	Real -5.0

(The sign bit is already 1)

=	`(0.5+0.125)*2^(130-127)`
=	`0.625*8`
Magnitude =	`5`

If all this seems a little complicated, try using the program 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 section for details.)

The layout of the variable 'NMBR' in memory is shown below.

LSB	MSB	M	B	R	0	LSB			MSB	EXP	|

A%-5 points here ⇧⇧⇧

A%-2 points here ⇧⇧⇧

A%-1 points here ⇧⇧⇧

A% points here ⇧⇧⇧

```
 10 NUMBER=0
 20 DIM A% -1
 30 REPEAT
 40   INPUT"NUMBER PLEASE "NUMBER
 50   PRINT "& ";
 60   :
 70   REM Step through mantissa from MSB to LSB
 80   FOR I%=2 TO 5
 90     REM Look at value at address A%-I%
100     NUM$=STR$~(A%?-I%)
110     IF LEN(NUM$)=1 NUM$="0"+NUM$
120     PRINT NUM$;" ";
130   NEXT
140   :
150   REM Look at exponent at address A%-1
160   N%=A%?-1
170   NUM$=STR$~(N%)
180   IF LEN(NUM$)=1 NUM$="0"+NUM$
190   PRINT " & "+NUM$''
200 UNTIL NUMBER=0
```

String Variables

String variables are stored as a string of characters. Since the current length of the string is stored in memory an explicit terminator for the string in unnecessary. As with numeric variables, the first word of the complete variable is the address of the next variable starting with the same character. However, since BBC BASIC(Z80) needs information about the length of the string and the address in memory where it starts, the overheads for a string are more than for a numeric. The format of a string variable called 'NAME$' is shown below.

LSB	MSB	A	M	E	$	0	len	max	LSB	MSB			
Address of next var. starting with the same letter		Rest of Name				Zero	Cur. len. of str.	Max. (org) len. of str.	Addr. of start of string		String		

When a string variable is first created in memory, the characters of the string follow immediately after the two bytes containing the start address of the string and the current and maximum lengths are the same. While the current length of the string does not exceed its length when created, the characters of the string will follow the address bytes. When the string variable is set to a string which is longer than its original length, there will be insufficient room in the original position for the characters of the string. When this happens, the string will be placed on the top of the heap and the new start address will be loaded into the two address bytes. The original string space will remain, but it will be unusable. This unusable string space is called 'garbage'. See the Variables sub-section for ways to avoid creating garbage.

Because the original length and the current length of the string are each stored in a single byte in memory, the maximum length of a string held in a string variable is 255 characters.

Fixed Strings

You can place a string starting at a given location in memory using the indirection operator '$'. For example,

```
$&8000="This is a string"
```

would place &54 (T) at address &8000, &68 (h) at address &8001, etc. Because the string is placed at a predetermined location in memory it is called a 'fixed' string. Fixed strings are not included in the variable chains and they do not have the overheads associated with a string variable. However, since the length of the string is not stored, an explicit terminator (&0D) is used. Consequently, in the above example, byte &8010 would be set to &0D.