REFERENCE MANUAL
PART TWO



BRITISH BROADCASTING CORPORATION
MASTER SERIES MICROCOMPUTER

Acorn

The choice of experience.

Acorn Computers Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
England

The BBC Microcomputer System

Master Series

REFERENCE MANUAL – Part 2

Part number 0443,002
Issue 1
January 1986

Within this publication the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

© Copyright Acorn Computers Limited 1986

Neither the whole or any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material form except with the prior written approval of Acorn Computers Limited (Acorn Computers).

The product decscribed in this manual and products for use with it, are subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by Acorn Computers in good faith. However, it is acknowledged that there may be errors or omissions in this manual. A list of details of any amendments or revisions to this manual can be obtained from Acorn Computers Technical Enquiries. Acorn Computers welcome comments and suggestions relating to the product and this manual.

All correspondence should be addressed to:

Technical Enquiries
Acorn Computers Limited
Cambridge Technopark
Newmarket Road
CAMBRIDGE CB5 8PD

All maintenance and service on the product must be carried out by Acorn Computers' authorised dealers. Acorn Computers can accept no liability whatsoever for any loss or damage caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of this product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any omissions from, this manual, or any incorrect use of the product.

Acorn is a trademark of Acorn Computers Limited
VIEW and ViewSheet are trademarks of Acornsoft Limited
Econet and Tube are registered trademarks of Acorn Computers Limited

This book is part of the BBC Computer Literacy Project.
Typeset by Interaction Systems Ltd, Cambridge.
Cover design concept by Carrods Graphic Design.

First published 1986
Published by Acorn Computers Limited.

Contents

Introduction to Part 2

K BBC BASIC

K.1 Introduction to BBC BASIC
K.2 Variables and operators
K.3 Using filing systems from BASIC

L BASIC Keywords

L.1 The syntax of BASIC
L.2 Syntax and usage of BASIC keywords

M BASIC Error messages

M.1 Errors encountered in BASIC
M.2 Errors in numerical order

N BASIC technical information

N.1 Introduction
N.2 BASIC tokens in keyword order
N.3 BASIC keywords in numerical order of tokens
N.4 The memory map under BASIC
N.5 How BASIC uses the MOS and Filing Systems

O The BBC BASIC Assembler

O.1 Introduction to machine code and assembly language
O.2 Information for previous users of BBC BASIC
O.3 Information for users of other 6502 Assemblers
O.4 The 65C12 microprocessor
O.5 Using the assembler
O.6 The BASIC elements of an assembly language program
O.7 The assembly language elements of an assembly language program
O.8 The machine code environment
O.9 Using machine code programs

P Assembler Keywords

P.1 Introduction to Assembler keywords
P.2 Table of operation codes
P.3 65C12 addressing modes
P.4 Assembler keywords

Q Assembler errors

Q.1 Introduction
Q.2 Assembler error messages and symptoms
Q.3 Coding errors
Q.4 Run-time errors

R The system editor - EDIT

R.1 Introduction
R.2 Selecting EDIT
R.3 The screen layout
R.4 EDIT commands: using the function keys
R.5 Typing text, Insert and Over modes, simple deleting
R.6 Moving the cursor around the screen
R.7 Moving around the text buffer, scroll margins
R.8 Changing existing text
R.9 Moving, copying and deleting text
R.10 BREAK, clearing and restoring text
R.11 Finding and replacing text (elementary)
R.12 Printing text
R.13 Saving and Loading text
R.14 Using filing and operating system commands
R.15 Finding and replacing text (advanced)
R.16 Automatic editing: editor command files (advanced)
R.17 The editor environment

S The EDIT text formatter

S.1 Introduction
S.2 Printing to the screen
S.3 Printing on a printer
S.4 The formatter commands
S.5 Command parameters
S.6 Defining the style of the page
S.7 Controlling pagination

S.8 Defining the style of paragraphs
S.9 Defining the style of single lines
S.10 Underlined and bold text
S.11 Using tabs
S.12 Ignoring text and commands
S.13 Linking files for printing
S.14 Printing special codes and translating characters
S.15 Storing and manipulating text and numbers
S.16 Numeric registers
S.17 Macros
S.18 Generating contents lists and indices

T System editor/formatter error messages

T.1 Introduction
T.2 Editor errors
T.3 Text formatting errors
T.4 Dealing with other problems

U The TERMINAL Emulator

U.1 Introduction
U.2 Using the TERMINAL emulator
U.3 Using the function keys with TERMINAL
U.4 The ANSI compatible mode
U.5 ANSI Mode escape sequences
U.6 The BBC VDU driver modes
U.7 Summary of ESC sequences in ASCII order

Index

Introduction to Part 2

Part 1 of this Reference Manual describes the use of the central core of the machine: the hardware and the software provided to control it. This includes information such as how to alter keyboard characteristics, produce sounds, use disc drives and connect peripherals such as disc drives and printers.

This volume describes the software which you will probably use for most of the time. i.e. the BASIC language, the built-in Assembly language, the System Editor and the TERMINAL emulator. Instructions for using the VIEW word-processor and the ViewSheet spreadsheet are given in the VIEW and ViewSheet Guides respectively.

Like Part 1, this manual has been designed as a reference manual and you should not expect to read it straight through from the first to the last page. Different users will be interested in different components of the system and hence will read different sections to varying depths.

An overview of the machine as a whole is provided in the Welcome Guide supplied with the computer.

If you want details on a particular subject, you are strongly advised to use the index to locate entries of interest. Note that where a facility is described in both Parts 1 and 2, Part 1 will generally cover the more fundamental aspects and do so to a greater depth; entries in this volume will generally provide details specific to a particular language or subsystem.

Conventions used in this volume

[ ]	denotes the single key on the keyboard with the corresponding legend, for example [RETURN] means the RETURN key.
<description>	the < and > enclose text which should not be taken literally (eg typed) but which should be interpreted according to the context in which it appears. For example, `LOAD` <expression> indicates that the word `LOAD` must be followed by a sequence of characters obeying the rules defined by <expression>. Generally, definitions of such items will be found in the introduction to the appropriate section.
[entry]	the [and] enclose an entry which is optional, the entry being either a specific sequence of characters or a description enclosed within < and > (as above). For example, `*EDIT` [<filename>] means that either `*EDIT` or `*EDIT` <filename> are acceptable.
`text like this`	is used for examples of commands, programs and output that you might see on the screen.
bold text	is used for emphasis
italic text	is used for emphasis

K BBC BASIC

K.1 Introduction to BBC BASIC

There have been four versions of BBC BASIC to date, numbered I, II, III and IV. BASIC I and II were used on earlier versions of the BBC microcomputer. BASIC III was used on the American version. The version of BBC BASIC provided with this computer is BASIC IV.

This section briefly outlines the differences between BBC BASIC IV and earlier versions of BBC BASIC (I, II and III), and the otherwise similar Microsoft BASICs. It is assumed that the reader is already familiar with one or other of these versions of BASIC.

Introduction for previous users of BBC BASIC

BASIC IV is very similar to BASIC II, but has been rewritten to make it faster and provide a number of new facilities. Some of these facilities are simply extensions to existing commands, others are entirely new to take advantage of new facilities in the computer.

Enhancement to the ON... statement

It is now possible to call procedures in an ON statement, as well as subroutines. For example:

```
120 ON in%-129 PROCleft, PROCright, PROCdown, PROCup ELSE PROCerr
```

The PROC calls have exactly the same syntax as normal procedure calls, ie they may have parameters separated by commas between brackets:

```
1240 ON menu% PROCadd(name$,age%),PROCdel(name$),PROCshow(name$)
```

The LIST IF construction

The LIST command has been extended to provide a 'cross-reference' facility. The standard LIST command (eg LIST or LIST 100,2000) may be followed by the key word IF, which is in turn followed by a string of characters. Only program lines containing these characters will be listed. The characters following the IF are tokenised as usual, so keywords may be found. Examples are:

LIST IF IF	list IF statements
LIST IF TIME	list lines with TIME as a function

```
LIST IF name$(          list lines using the array name$()
LIST 100,200 IF var=    list assignments to var in lines 100-200
```

The EDIT command

You can now use the powerful facilities of a text editor to edit a BASIC program without having to *SPOOL it first, by using the new command EDIT. EDIT on its own edits the whole program, but can be used with exactly the same arguments as LIST (including the IF part) to edit only parts of the program.

EDIT converts the program into plain text (effectively by LISTing it into memory, with a LISTO option of 0) then calls the system editor with that text in its buffer. Because both the plain text and tokenised versions of the program are required during this conversion, there is a limit on the size of programs that can be edited in this way. The largest program that can be edited in this way is between 9 Kbytes and 15 Kbytes. For details of using the system editor, see Section R.

Once the program text has been edited to the user's satisfaction, it can be converted back to a tokenised BASIC program by using the editor's 'return to language' command and specifying BASIC as the language. This command calls BASIC and returns the text in the editor to BASIC as if the text were being typed in. After this, the edited program is ready to RUN, LIST or SAVE immediately.

The TIME$ pseudo-variable

This is related to TIME, and provides access to the system's real-time battery-backed clock/calendar. It may be used as a function, eg PRINT TIME$, or as a statement, eg TIME$="....". For the format of the string used by TIME$, see the entry in the BASIC keyword section.

The EXT#= statement

In BASIC II, the function EXT# is used to return the length of an open sequential file. In BASIC IV, you can also assign to EXT#, to change the length of an open file. This facility can only be used with suitable filing systems (eg ADFS, Network, but not cassette).

The COLOR statement

In BASIC IV, the COLOUR statement may be spelt alternatively as COLOR and still be accepted. It will, however, be listed as COLOUR.

Extensions to the assembler

The microprocessor used in this BBC microcomputer is the 65C12, a CMOS version of the 6502 processor with an enhanced instruction set. To cater for

this, the assembler built into BASIC IV has been extended to accept the new instructions. The new mnemonics and addressing modes are described in Section P.

Assembly listings are now formatted in a more readable manner than in earlier BASICs, so that labels of up to nine characters (including the initial .) may be used without disturbing the formatting.

You can use lower case letters throughout the source program if you choose, such as `lda &70,x` and `equs "fred"`.

The assembler no longer gets confused by the accumulator addressing mode. For example, in previous versions `ASL ALFRED` would be treated as `ASL A` followed by the 'comment' `LFRED`. This no longer happens (the same applies to `LSR`, `ROL`, `ROR`, `DEC` and `INC`).

BASIC sets up some addresses pointing to floating point routines. These may be called from user's machine code programs. The addresses provided and their uses are described under the `CALL` statement.

Improved `LISTO` formatting

After a `LISTO7` command, `LIST` indents `REPEAT`s and `NEXT`s more neatly, so that `UNTIL`s lie under the corresponding `REPEAT`s and `NEXT`s line up with their `FOR`s.

BASIC also now strips all trailing spaces from input program lines, and if the current `LISTO` option is non-zero will also strip leading spaces (between the line number and the first non-space character on the line). A side-effect of this is that blank lines such as:

```
1000 [RETURN]
```

cannot be entered if `LISTO` is non-zero. (Use `1000:` [RETURN] instead.)

Other minor changes

`SAVE` can take any string expression as its argument such as `SAVE a$+b$`, and the indirection operators ! and ? may be used as formal parameters eg `DEF FN fred(!&70)`.

You may now use ASCII code 141 in comments and strings, for example to produce double-height characters in teletext display modes:

```
100 REM<&8D>Big comment
110 REM<&8D>Big comment
```

(Previously, the `RENUMBER` and `LIST` commands could be confused by this code.)

In addition `LIST` will now list comments that include teletext colour codes as

coloured lines in teletext modes; you no longer need to include a " just after the REM.

The AUTO command no longer prints a space after the line number.

Introduction for users of Microsoft BASIC

BBC BASIC IV is an enhanced version of the widely-acclaimed BBC BASIC. The following describes the main features that make it superior to the Microsoft and Microsoft-compatible BASICs found on many other popular machines.

Aids to structured programming

BBC BASIC provides several features that help you write well-structured and readable programs. The structuring features are:

- The IF... THEN... ELSE... statement
- The REPEAT... UNTIL... construct
- The ON... PROC... statement
- Multi-line user-defined procedures
- Multi-line user-defined functions
- Procedure and function parameters and local variables

The enhancements to aid the writing of readable programs are:

- Long, meaningful variable names
- Automatic indentation of loop structures, if required
- A powerful RENUMBER command

In addition, the usual BASIC constructs such as GOTO, GOSUB, ON... GOTO/GOSUB and FOR... NEXT are provided.

Access to the operating system

The computer has a powerful machine operating system (MOS) wh h controls the the machine's hardware, such as the screen, keyboard, analog to digital convertors, printer etc. The MOS also supports the filing systems used to store and retrieve information on floppy discs, tapes, cartridges etc. BBC BASIC provides full access to the MOS and filing systems through statements such as PLOT, INKEY, ADVAL, OSCLI etc. These are much easier to use and understand than controlling the machine through obscure 'PEEK' and 'POKE' instructions.

Access to machine code

Although BBC BASIC IV is currently the fastest eight-bit version of BASIC available, there are occasions when the extra speed of machine code is needed, eg when very large arrays have to be sorted quickly. To this end, BASIC

provides a powerful 65C12 assembler which can be used to develop small to medium-sized machine code applications.

The CALL statement can be used to call machine code routines and pass them BASIC parameters of any type.

The USR function can be used to call machine code routines, initialising the 6502 registers from BASIC variables on entry and returning the register contents on exit.

CALL and USR also provide easy access to operating system routines that are not directly supported by BASIC functions.

Versatile data handling

BBC BASIC supports four-byte integers (rather than the usual two-byte ones) and five-byte floating point numbers. Strings may be assigned dynamically and can be up to 255 characters in length. Multi-dimensional arrays of integers, reals and strings are available, as are arrays of single bytes.

Memory access is provided through the indirection operators ? ! and $. ? has similar uses to PEEK and POKE (which are neither necessary nor supported in BBC BASIC). ! is similar but acts on four bytes at once instead of a single byte. $ acts on strings of characters (this is provided in addition to normal string variables).

A rich set of numeric and string functions is provided such as trigonometric and logarithmic functions, string splitting and searching functions.

Print formatting

Numbers may be printed in any of three formats (general, exponent and fixed) with variable field widths and numbers of significant digits. Integers may be printed in hexadecimal, and the STR$ function can be used to convert numbers to hexadecimal strings, or print-formatted strings.

The PRINT statement has several positional modifiers: ' to print a newline, SPC to print a number of spaces and TAB to move to a given tab position on the current line or any given place on the screen.

Error trapping

An ON ERROR facility is provided so that non-fatal errors may be trapped and dealt with by the program. The facility is supported by the ERR and ERL functions, which give the error number and error line of the latest error, and the REPORT statement which prints the latest error message. It is a simple task to include user error messages which behave in every way like BASIC or MOS messages.

The BASIC command mode

BASIC's prompt is > and its presence at the start of a line indicates that BASIC is ready to accept input. The user may type commands such as `LIST` and `AUTO`, statements to be executed immediately such as `PRINT LOG(12)`, or BASIC lines to be entered into the program. This last should be preceded by a line number in the range 0-32767. Single program lines may be deleted typing the line number followed immediately by [RETURN].

Below is a list of commands and their meanings. Note that a command may not be executed from within a program, and may not be preceded by another command or statement. If followed by a statement, that statement will be ignored.

Command	Meaning
`AUTO`	Generate line numbers automatically
`DELETE`	Delete a range of line numbers from the current program
`EDIT`	Call the sytem editor to edit the BASIC program
`LIST`	List the program
`LISTO`	Set `LIST` indentation option
`LOAD`	Load a BASIC program
`NEW`	Erase the current program
`OLD`	Retrieve a `NEW`ed program
`RENUMBER`	Resequence the program line numbers
`SAVE`	Save the current BASIC program
	in addition the following statements can be used within a program but are most often used in command mode:
`CHAIN`	Load and run a BASIC program
`RUN`	Execute the current BASIC program

K.2 Variables and operators

This section describes the types of data that may be used in BBC BASIC IV programs and the operators and functions provided to deal with them.

Variable types in BASIC

There are three fundamental types of data in BBC BASIC: real numbers, integer numbers and strings. For each there is a corresponding variable type: real, integer and string. There is also a facility for declaring multi-dimensional arrays of these types. Each variable type has a set of functions and operators which may operate on it. These are described in this chapter. Note that reals and integers are largely interchangeable; BASIC automatically converts real values to integer values and vice versa when required.

Reals

Variables of type real are simply identifiers. An identfier in BASIC is a sequence of one or more characters in the set {A,B.. Y,Z, a,b,.. y,z, 0,1.. 8,9, _,£}. The first character may not be a digit. Examples of real variables are:

```
£amount
numVars
A_Long_Variable_Name
var1234_21z
_
```

It can be seen that the special characters _ (underline) and £ (pound) act as extra letters. A restriction on identifiers is that they may not begin with most reserved words. For example, `GETADDR` is illegal as `GET` is a reserved word. However, reserved words may be embedded in an identifier, as in `NAMELIST`. Since reserved words have to be in upper case, identifiers such as `getaddr` and even `list` are legal. The reserved words which are permitted at the start of an identifier are those which are not normally followed by anything else: `CLEAR`, `CLG`, `CLS`, `COUNT`, `END`, `ENDPROC`, `ERL`, `ERR`, `FALSE`, `HIMEM`, `LOMEM`, `NEW`, `OLD`, `PAGE`, `PI`, `POS`, `REPORT`, `RETURN`, `RUN`, `STOP`, `TIME`, `TRUE`, `VPOS`. Thus, variables such as `COUNTER` and `POST` are quite legal and distinct from `COUNT` and `POS`.

Arrays of reals are simply identifiers followed by subscript(s) in brackets:

```
counts(char-128)
mat(i%,j%)
numVars(0)
```

Note that the array `numVars()` and the simple variable `numVars` may both be used in the same program and are entirely separate entities.

A real constant is a sequence of digits with an optional decimal part and an optional exponent part. Examples are:

```
1224
12.34
-0.000001
.123
10293.1234
+0.1E10
-112.32E-4
```

The sign of the number is taken to be positive if not given explicitly. The number <n> after the E means 'multiply the number by 10 to the power of n.'. The largest real magnitude that can be represented by BASIC is 1.701411834E+38. An attempt to generate a value larger than this will result in a 'Too big' error. The smallest real magnitude (ie the number closest to zero, except for zero itself) is 1.469367939E-39. An attempt to generate a number smaller than this will yield zero.

Integers

Integer variables are identifiers followed by a percent sign, %. Examples are:

```
check_sum%
£pennies%
cx%
YCOORD1%
```

Integer array elements are indicated in the same way as real ones – by following the variable with a subscript list. Examples are:

```
lookup%(i%)
board%(row,col)
```

Integer constants are written as sequences of between 1 and 10 significant digits, optionally preceded by a sign. Examples are:

```
-99
234134112
-532354
+42
```

Integer constants may also be written in hexadecimal by preceding the constant part with an ampersand, &. The number itself is a sequence of 1 to 8 significant hexadecimal digits (0,1.. 8,9,A,B.. E,F). Examples are:

```
&0D
&FF7FF80
-&55AA1
&80000000
```

There should be no spaces beteen the & and the first digit. Integers are stored as four bytes in two's complement form by BASIC. This gives them a range of −2147483648 to +2147483647, or in hexadecimal &80000000 to &7FFFFFFF.

System integer variables

There are 27 integer variables which are permanently defined, called the 'system integer variables'. They are A%-Z% and @%.

@% is used to control print formatting (see the PRINT statement). O% and P% have special meanings when assembling machine code programs. A%, C%, X% and Y% have special meanings when using machine code programs. The other system integer variables are completely free for use by the programmer, as are A%, C%, O%, P%, X% and Y% if not using machine code.

The main advantage of the system integer variables is that they are not cleared by CLEAR, NEW, OLD, RUN, LOAD and CHAIN, so they can be used to pass information between programs. Another advantage is that BASIC accesses them very quickly, because it knows exactly where they are. It is therefore a good idea to use these variables when speed is important (such as within a FOR ... NEXT loop).

Other variables have to be created using an assignment or INPUT statement and are destroyed by the statements mentioned above (or changing the program in any way). They are refered to as 'dynamic' variables.

Indirect integers

There is another form of integer variable, accessed using the word indirection operator, ! (pronounced 'pling'). The format of a such a variable is !<factor>, where <factor> is a number, a variable, a function call or an expression in brackets. Thus some typical indirect integers are:

```
!&70
!ptr%
!words%(i%)
```

The value of an indirect integer is the value of the four bytes at the address specified. Thus !&70 has a value determined by the four-byte, two's complement number stored at locations &70-&73. Similarly, !ptr% takes ptr% as an address of a four-byte value at ptr% to ptr%+3, and gives the value of this integer.

There is an extension to the pling notation. It has the form <variable>!<factor>. This time, <variable> is a numeric (real or integer) variable. <factor> is as described above, and the value obtained is of the four-byte integer at location <variable>+<factor> to <variable>+<factor>+3. It can be seen from this that <variable>!<factor>

is just another way of writing !(<variable>+<factor>). Here are some examples:

```
table!10
vals%!i%
address%!(2*index%)
```

written in a similar way to arrays of reals and integers, viz:

```
address1$(i%)
table$(row,col,1)
```

Strings

String variables may hold strings of up to 255 characters. The shortest length string is the null string which has a length of zero. The address of a string variable (as supplied in a `CALL` parameter block, for example) is actually the address of its string information block (SIB). This is four bytes long and has the format:

Address of string text SIB + 0

Bytes allocated to the string SIB + 2

Current length of string SIB + 3

To get the best out of BASIC, it is advantageous to know how strings are stored. This is particularly important if you are writing large programs that use many strings, because BBC BASIC does not perform any string 'garbage collection'; by declaring strings in a particular way, you can greatly reduce the risk of running out of storage space.

When a string variable is created (eg by an assignment statement), BASIC sets aside storage for it. If the initial length of the string is less than eight characters, this storage (the 'bytes allocated' value) is the same as the length. If the initial length is eight or greater, the bytes allocated will be eight characters longer than the initial length, up to a limit of 255 characters. So, for example:

```
a$="123456"
b$="123456790"
```

will allocate 6 bytes for `a$` (the same as its length), and 18 bytes for `b$` (eight greater than its length). By allocating more characters than the initial length, BASIC can allow the string to 'grow' before a new area of storage has to be found for it. This is important as once the string outgrows its first location and is moved to a larger area, the storage it occupied previously is no longer accessible and can't be re-used.

Consider what happens when a$ and b$ above are both lengthened by two bytes, eg by the statements:

```
a$=a$+"AB"
b$=b$+"AB"
```

a$ will now occupy eight bytes. As only six bytes were allocated for it when first declared, the string must be moved to a larger area. The new area will reserve 8+8 bytes for it, as the new length is in the range 8-255. The six bytes previously occupied by a$ will become inaccessible.

When b$ is re-assigned, its new length is 10. This fits into the 18 bytes originally allocated to it, so no extra storage is required, and the string does not have to be moved around.

The only exception is that if a string variable is the latest dynamic variable to be created, it may grow up to the maximum string length allowed (255 characters) without needing a new space. This is because BASIC knows that as there is nothing after the string variable, it can be extended without having to move it. Thus if you have a string whose length will vary widely, it is sensible to make this the last variable created by the program. Be careful though – LOCAL variables and procedure/function parameters can create variables if they don't already exist.

Summarising the above paragraphs, for maximum efficiency the first time you assign to a string variable you should make its length equal to the maximum length to which it will grow. You can reset it to null immediately after, of course. For example:

```
a$=STRING$(30,"*")
a$=""
```

Adopting this policy will:

- speed up program execution as string movement is minimised;
- prevent 'No room' errors cause by the constant re-allocation of strings.

String constants are sequences of between 0 and 255 characters, enclosed in double quotes ("). The upper length limit is also limited by the maximum length input of line that BASIC will allow: 238 characters.

To include the double quote character in a string, you must type it twice.

Example string constants are:

```
"A string constant"
"A string "" with a quote in"
"" : REM  a null string
"""" : REM one double quote
```

Indirect strings

There is an operator which is analogous to the pling (!) operator. The unary operator $ has the form $<factor>, and stands for the string at the address given by factor. These strings consist of zero to 255 characters terminated by a carriage return (&0D). There is no equivalent form to <variable>!<factor>, as it would be impossible distinguish between <real variable>$<factor> and <string array access>, eg `a$(i%+2)` would be ambiguous.

Indirect strings have no string inforamtion blocks (SIBs); the address of the string is the address of first character in the string. Some examples of indirect strings in use:

```
$buffer = "some text"
INPUT "Type some text :"$&2000
IF $mesg%="" THEN PRINT "No messages"
```

When used on the left hand of an assignment statement, $ inserts the required carriage return after the text. When used to access strings, if there is no delimiting carriage return in the first 256 characters after the address, $ will give a null string. This provides a simple way of ensuring that there is a valid $ string at the address, viz

```
$addr=$addr
```

Note that you cannot place indirect strings in zero page (ie addresses &00-FF).

Arrays of $ strings are effectively arrays of integers or reals preceded by $, eg:

```
$string%(10)
$vars(i%-8)
```

To make sure that an indirect string is not using space already used by another part of the system (such as BASIC variables or program lines), you will usually reserve space for them with the byte form of `DIM`. It is the programmer's reponsibility to ensure that indirect strings do not change the wrong locations. For example:

```
DIM buff 5
$buff="1234567890"
```

would have undesired effects as more characters are assigned to the buffer than were allocated to it.

Operators in BASIC

The types of variable described above may be combined to form expressions. Expressions consist of variables, constants, operators and functions. Some of the operators (!, ? and $) have already been described above.

When BASIC evaluates expressions it uses a series of rules (precedence rules) to determine which part of the expression to do first. Operator precedence is simply the way in which, for example, multiplication and division are performed before addition. This is sometimes learned as BODMAS or some other acronym at school.

The precedence rules for BASIC expressions are more complex than BODMAS, because BASIC has many operators. There are seven groups of precedence, summarised in the following table within which:

- r denotes a real operand;
- i denotes an integer operand;
- s denotes a string operand.

Group 1	**Accepts**	**Yields**	**Meaning**
–	ri	ri	Unary minus
+	ri	ri	Unary plus
NOT	ri	i	Logical NOT
Function calls	sri	sri	Built-in and user
()	sri	sri	Brackets
! ? $	ri	si	Indirection operators
Group 2			
^	ri	ri	Exponentiation
Group 3			
*	ri	ri	Muliplication
/	ri	ri	Division
DIV	ri	i	Integer division
MOD	ri	i	Integer remainder
Group 4			
+	sri	sri	Addition (concatenation)
–	ri	ri	Subtraction
Group 5			
=	sri	sri	Equal
<>	sri	sri	Not equal
<	sri	sri	Less than
>	sri	sri	Greater than
<=	sri	sri	Less than or equal
>=	sri	sri	Greater than or equal

Group 6

AND	ri	i	Logical AND

Group 7

OR	ri	i	Logical OR
EOR	ri	i	Logical EOR

Operators in the lower-numbered groups have higher precedence (are acted upon first), and operators in the higher-numbered groups have lower precedence. The 'types' are the type of operand that the operator (or function) can accept. 's' is string, 'r' is real and 'i' is integer. Where 'i' appears on its own, 'r' will be accepted and converted to an integer (by truncation) if possible.

Operators in the same group are evaluated left to right, so a−b+c is treated as (a−b)+c rather than a−(b+c). If you learn to apply the precedence rules given in the table, you will rarely need to use brackets. For example, a long conditional expression such as:

```
length>=1 AND length<=10 OR NOT checking
```

has the intuitive bracketed meaning of:

```
((length>=1) AND (length<=10)) OR (NOT checking)
```

There are occasions, however, when brackets are needed to give the desired result. An example is when checking the contents of a two-byte value using !:

```
IF (!&20E AND &FFFF) = mywrch% THEN ....
```

The brackets are needed as otherwise the expression would be treated as:

```
!20E AND (&FFFF = mywrch%)
```

which is not quite the same thing!

Below is a brief description of what each operator does for the various operand types.

Group 1

−	Negates the value of the numeric object following it.
+	No action.
NOT	Inverts the state of each bit in its integer operand.
Function calls	See keywords section.
( )	Give the enclosed expression a high precedence.
!, $, ?	See the descriptions above.

Note: Expressions that only use operators in group 1 are known as factors. Such expressions can be used as the arguments to any of the BBC BASIC functions that take a single parameter.

Group 2

^	Raises its left operand to the power of its right operand.

Group 3

*	Produces the product of its numeric operands.
/	Produces the quotient of its numeric operands.
DIV	Produces the integer quotient of its numeric operands.
MOD	Produces the remainder after its left operand has been divided by its right one. `A% MOD B% = A% - B%*(A% DIV B%)`.

Group 4

+	Produces the sum of its numeric operands.
+	Produces the right operand joined to its left operand, eg `"ABCD"+"1234" = "ABCD1234"`.
–	Produces the difference of its numeric operands.

Group 5

=	Produces `TRUE` if its operands are equal, `FALSE` otherwise.
<>	Produces `TRUE` if its operands are not equal, `FALSE` otherwise.
<	Produces `TRUE` if its left operand is less than the right one, `FALSE` otherwise.
>	Produces `TRUE` if its left operand is greater than the right one, `FALSE` otherwise.
<=	Produces `TRUE` if its left operand is less than or equal to the right one, `FALSE` otherwise.
>=	Produces `TRUE` if its left operand is greater than or equal to the right one, `FALSE` otherwise.

Note: String comparisons are done on a character by character basis, using the ASCII code of the characters for comparison. For example, these conditions are TRUE:

```
"a" > ""
"ASC"="ASC"
"ABC" < "abc"
"z" > "ZZZZ"
```

Groups 6 and 7

AND	Combines the two integer operands by logically ANDing corresponding bits.
OR	Combines the two integer operands by logically ORing corresponding bits.
EOR	Combines the two integer operands by logically EORing corresponding bits.

K.3 Using filing systems from BASIC

BBC BASIC provides commands, statements and functions which access the filing system directly. Because the filing systems have been designed to present a uniform software interface, the same program can normally work with, say, disc, cassette or ROM filing systems. The main differences in operation are those forced by the restrictions of the medium used. For example, cassette files are 'serial' and cannot support random access, while ROMs are 'read-only'.

Filing systems are used by executing two distinctly different types of command: filing system commands in the current language, and filing system * commands. Language commands can only be used while the language is selected as the current language and are designed to make it easy to perform language-oriented tasks such as saving and loading programs and data. Filing system * commands (which may be obeyed directly by the filing system or indirectly, via the MOS) can be used with any language, but are not so good at performing language-oriented tasks.

In this chapter, the language commands only are described in detail. Some of the more common filing system * commands such as `*LOAD` are mentioned briefly, but for detailed information you should refer to Sections G-J.

Whole-file operations

These operations act on whole files, whether they are BASIC programs or machine code files.

Loading and saving BASIC programs

The BASIC `SAVE` command uses a filing system call, OSFILE &00, to save the BASIC program currently in memory onto the medium. An example is:

```
SAVE "stats"
```

which will save the current program under the name 'stats'. The start address is set to `PAGE`, the end address (ie the location after the last byte to be saved) is set to `TOP`. BASIC sets the high-order bits of the load address to the high-order address of the processor it is running on. This enables you to tell if a file was saved from the I/O processor or a co-processor. For example, if there was a BASIC file called `prog1`, its information might look like this:

```
prog1     FFFF0E00 FFFF8023 00000777 000023
```

This indicates that `prog1` was saved on an I/O processor-only machine with

PAGE set to &E00. The execution address (FFFF8023) is not significant for BASIC programs. The length (00000777) is in hexadecimal, and is equal to TOP–PAGE.

Filing systems do not distinguish between types of files, apart from requiring directories to have a certain format. BASIC files are no exception, so they can be treated as any other; for instance they can be opened for reading or writing, *SAVEd or *LOADed. This latter property can prove very useful when developing programs, as illustrated below in 'Merging BASIC programs'.

The BASIC LOAD command uses OSFILE &FF. The load address is supplied as the current value of PAGE: the file's own load address is ignored. An example is:

```
LOAD "stats"
```

which will load the program called stats at PAGE. After the program has loaded, BASIC checks that it is a valid BASIC program (as it does when END is executed), and if it finds an error, reports it as a 'Bad program'. Note that you may be able to RUN a 'bad' program, even if you can't LIST it.

BASIC does not check the length of a program before loading it (as this cannot be done on the cassette filing system) so a program may be 'bad' because it is too large to fit into memory. This is most likely to happen when loading a program that was written on a co-processor using HIBASIC into the I/O processor, or loading a program written in shadow mode when in a non-shadow mode. A program may also be 'bad' because it is not a BASIC program.

LOAD is like NEW in that it removes all of the current program's variables, apart from the system integer variables. If the LOAD is unsuccessful (ie the program cannot be found), the old program and its variables remain intact.

The CHAIN statement (which can also be used in a program) acts exactly as LOAD followed by RUN, except that if the program is 'bad', BASIC will not RUN it. Under ADFS, both LOAD and CHAIN allow 'wildcards' in the filename, for example:

```
CHAIN "BM*"
```

will load and run the first program in the current directory that begins with the letters BM. (The 'first' program being the first one in alphabetical order.)

SAVE, LOAD and CHAIN all allow general string expressions as their arguments. This can be used to ensure that a program is always stored using the same name, by placing a line in the program such as:

```
10000 DEF FNNM="sort1"
```

and always saving the program while you develop it with the command:

```
SA.FNNM
```

Merging BASIC programs

One of the strengths of BBC BASIC is the way it enables you to build up libraries of procedures, functions and other small programs to use as 'building blocks' when writing programs. To use programs from such a library, you need to be able to add them to a BASIC program in memory. There are two ways of doing this.

The first method involves *LOADing a program file directly onto the end of the program in memory. This is a pure 'append', as the second program is literally attached to the end of the first one.

The second method involves using text versions of programs and facilities such as EDIT, *SPOOL and *EXEC. This is a true 'merge', as the second program is added to the first as if it were typed at the keyboard with the first program already loaded.

Suppose you want to append a file called shellSort to the current program.

```
PRINT~TOP-2
```

displays the address of the end of the last line of the current program, for example:

```
1A36
```

This tells you where to place the new program.

```
*LOAD shellSort 1A36
END
```

puts the second program file at the end of the current program. The END statement ensures that BASIC updates its version of TOP to the end of the merged program. (LIST will also do this.)

To simplify this task, you can program a function key to do it all for you:

```
*KEY0 INPUT'"File name: "f$:OSCLI"LOAD "+f$+" "+STR$~(TOP-2):END!M
```

This technique also works when there is no program in the machine already, so it can be used to load as well as append.

One small problem with this appending technique is that if the appended program has line numbers that overlap those in the original one, you get programs looking like this:

```
2100 REPEAT
2110 UNTIL INKEY-99
 800 DEF PROCshellSort(first,last)
 810 REM....
```

If there are no GOTOs, GOSUBs or other statements that use line numbers in the

program, the simple solution is to RENUMBER it. If there are any of these statements, RENUMBER will probably get confused; the best solution is to make sure that files which you append have very high line numbers (remember that the upper limit is 32767).

An alternative technique, which is a true 'merge' rather than the 'append' of the first method, is to convert library files into text rather than tokenised programs, then merge them into programs you develop using the system text editor or *EXEC.

To create a text version of a BASIC program, load it into BASIC then use EDIT. For example, to convert a BASIC program called quickprog into a text file called quicktext:

```
*BASIC (if not already in BASIC)
LOAD "quickprog"
EDIT
```

The EDIT command converts the BASIC program into a text file and enters the system editor with this text in its buffer. Now use **f3** (SAVE FILE) to save the contents of the buffer as file quicktext.

If you get the 'No room at line nnn' message, change to a shadow display mode and try again. If you still get the error message, the library program is too large to convert to text using EDIT; use *SPOOL instead, as follows:

```
*SPOOL quicktext
LIST
*SPOOL
```

The first line opens a *SPOOL file named quicktext; any characters sent to the screen after it will be put into the text file called quicktext. The LIST command works as usual, but the listing is not just displayed but also sent to quicktext. The second *SPOOL closes quicktext and stops spooling so that displayed output will only go to the screen again.

When you have converted the library program to text, load the 'main' program that you want to merge it into, for example:

```
LOAD "dBase"
```

To merge in the text file, type:

```
*EXEC quicktext
```

(If you are using the cassette filing system, remember to rewind the tape first.)

*EXEC reads the contents of the named file and 'types' them just as if you were typing the same characters yourself. The screen displays the contents of quicktext and BASIC thinks it is getting lines from the user, so it inserts them

into the BASIC program as usual. At the end of `quicktext`, `*EXEC` stops and input reverts to the keyboard.

To avoid having to do the first stage (converting the library program to text) each time, you could save all library files as text (from `EDIT`) rather than as BASIC programs (using `SAVE`).

Machine code files

When a BASIC program calls a machine code routine, it only needs the object program in memory, not the source program. The usual way of doing this is to assemble the routine with another program, using the 'assemble at P% for O%' assembly option (see Section O) and save the object program in a file with the appropriate execution and re-load addresses. When the routine is needed, the file containing it is loaded into the appropriate place.

For example, suppose there is a machine code routine which is about two pages (512) bytes long. A convenient place to put it is at the top of RAM in shadow mode (&7E00). `HIMEM` should be moved down first to accommodate and protect the code. The body of the program to assemble the code will look like this:

```
 980 *SHADOW
 990 MODE 3
1000 DIM code &200 : object=&7E00 : REM  Assuming a shadow mode
1010 FOR pass=4 TO 6 STEP 2
1020  P%=object : O%=code
1030  [ OPT pass
1040  \The source program
1050  .entry
1055  \The rest of the source
1060  ]
1070 NEXT pass
1080 OSCLI"SAVE objProg "+STR$~code+" "+STR$~O%+" "+STR$~entry+""+S
TR$~object
```

This uses the long form of the `*SAVE` command:

`*SAVE` <name> <start addr> <end addr> <execution addr) <reload addr>

To use the routine in another program, use a sequence of instructions like this:

```
300 HIMEM=&7E00
310 *LOAD objProg 7E00
320 CALL &7E00,dataptr,in%
330 REM and so on
```

The load address doesn't have to be specified as it was set to &7E00 by the

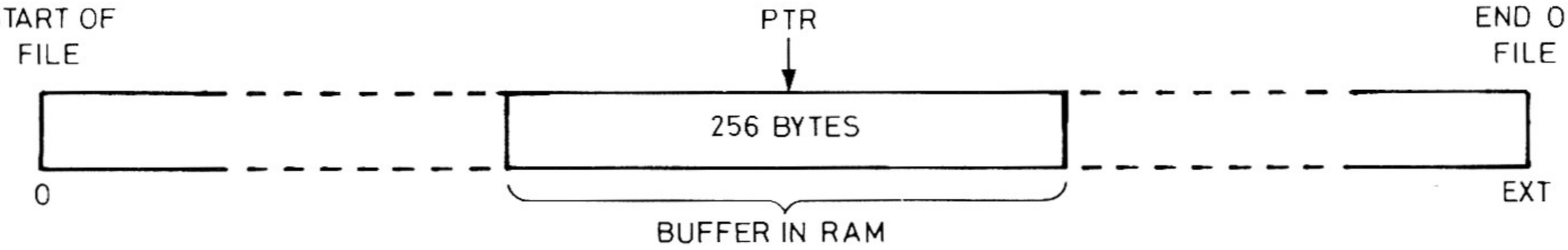
assembly program, but it is safer to make it explicit. If the object had to called only once, and with no parameters, the following three lines would suffice:

```
300 HIMEM=&7E00
310 */objProg
320 HIMEM=&8000
```

Sequential files

Data used in BASIC programs is lost when the machine is turned off. To provide a permanent record of variables, use the 'sequential files' provided by filing systems.

A sequential file can be regarded as an array of bytes, similar to the arrays dimensioned by the byte form of the DIM statement. The difference is that files lie on a medium such as disc or tape, while arrays are in the computer's main memory. To read data from a sequential file you must 'open' it, which reads a small section of the file into an area of the computer's main memory, known as a buffer. This provides rapid access to, usually, 256 bytes of the file. An open sequential file may be pictured as below:



PTR is the sequential pointer which marks the 'current location' in the file. It is set to 0 on opening a file, so read and write operations start at the begining of the file. PTR is discussed in detail later. EXT means 'extent' – the length of the file. There is a BASIC function to return this value for an open file. Suitable filing systems (not the CFS or RFS) may alter the length of a file by assigning to EXT or PTR.

BASIC supports sequential files with various statements and functions. Before it can use a file, the file must be 'opened'. When this is done, the user may read characters from the file, write characters to the file, or change parts of an existing file. There are three functions in BASIC used to open files:

OPENIN – Open a file for input only. The file must exist already
OPENUP – Open a file for input and output. The file must exist already
OPENOUT – Open a file for input and output. The file need not exist already

All three take a filename as an argument and return a 'channel number'. This is used in all subsequent dealings with the file. The value of the channel number depends on the filing system, but is always in the range 1-255. If an OPEN function returns a value of 0, the file could not be opened because, for example, a file specified to OPENIN does not exist already. Here are some examples of files being opened:

```
c%=OPENIN("data1")
outChan%=OPENOUT("$.dat.newData")
file=OPENUPdata$ : IF file=0 THEN PRINT "Can't find ";data$:END
```

The result of OPENIN, OPENUP and OPENOUT functions should always be assigned to a variable for later use. Note that OPENOUT immediately overwrites any existing file with the specified name, so it should be used with care.

Single-byte file operations

Once a file has been opened, you can write to or read from it, using the BPUT statement and the BGET function. For example, the following program prints the contents of a text file:

```
100 REPEAT
110   INPUT "File name: "file$
120   file=OPENIN(file$)
130 UNTIL file
140 REPEAT
150   char=BGET#file
160   IF char=&0D THEN PRINT ELSE IF char<&20 THEN PRINT"."; ELSE PR
INT CHR$(char);
170 UNTIL FALSE
```

This program emulates *PRINT. The first repeat loop asks for a filename from the user, until it finds a file with the name given. The second loop repeatedly gets characters from the file and prints them out. Control characters are printed as . except for carriage return (&0D) which does a newline. Other characters are printed as themselves.

The BGET function acts in a similar way to the GET function; it returns a character code between 0 and 255, but instead of using the keyboard, it uses the file whose channel number is given in its argument. In common with the other filing system keywords that take a channel number, BGET is always followed by a hash, #.

This program is not very well-designed; when it reads to the end of the file (all of the characters in it have been read and printed), it carries on trying to read characters and makes the filing system generate an error. To help you overcome this type of problem, BASIC provides a function called EOF. This

returns TRUE if the last character of the file has been read, and FALSE otherwise. Thus to make the above program end correctly (without an error), line 170 should be:

```
170 UNTIL EOF#file
```

In addition, after a program has finished with an open file it should close it. This frees the area of memory set aside for its buffer, so that another file may be opened. All filing systems have a maximum number of files that may be open at once, such as two for the CFS and 10 for the ADFS. Another line should be added to the 'PRINT' program:

```
180 CLOSE#file
```

Closing files after use is particularly important if they have been written to rather than just read from, as closing them ensures that the buffer is copied onto the file medium so that the file is kept up to date.

The BPUT statement writes a single byte to a file. The file must have been opened for update or output. For example, the program below emulates the *BUILD command provided by the operating system:

```
1000 REPEAT
1010   INPUT "File name: "file$
1020   file=OPENOUTfile$
1030 UNTIL file
1040 ON ERROR CLOSE#file: PRINT'"Escape": END
1050 line=1
1060 REPEAT
1070   PRINT RIGHT$("000"+STR$(line),4)" ";
1080   INPUT LINE ""in$
1090   in$=in$+CHR$(&0D)
1100   FOR i%=1 TO LEN(in$)
1110     BPUT#file,ASC(MID$(in$,i%))
1120   NEXT i%
1130   line=line+1
1140 UNTIL FALSE
```

The first four lines open the file for output. As mentioned above, this will delete any file of the same name already present. Line1040 sets the ON ERROR action for when the user presses [ESCAPE] to exit the program. Line1050 initialises the line number printed at the start of each input line. The second REPEAT loop prints the line number, gets a line of text, adds a carriage return to the end, and writes the lines one character at a time to the file. This continues until [ESCAPE] is pressed. As shown, BPUT takes two arguments; the channel number, followed by the code of the character to be written to the file.

BPUT, in common with the other sequential file statements and functions, has two errors associated with it. Missing # means that the # character after the keyword was omitted, and Channel means a channel number has been specified that does not correspond to a file opened for input.

Writing and reading BASIC variables

BGET and BPUT are useful when processing text files, but are not so good for dealing with the information that BASIC handles – variables and constants. To handle these, there are special forms of the PRINT and INPUT commands: PRINT# and INPUT#. These allow you to store constants and variables of any type in files, then read them back later.

As an example of using these commands, the following program takes a list of ten names and ages from the user and places them in a file called ages:

```
1000 ages=OPENOUT("ages")
1010 FOR i%=1 TO 10
1020   PRINT "Name, age number ";i%": ";
1030   INPUT ""name$,age%
1040   PRINT#ages,name$,age%
1050 NEXT i%
1060 CLOSE#ages
```

Using PRINT#, expressions that would normally be printed on the screen are sent to the file instead (BASIC actually uses the same routine as BPUT to do this). PRINT# may only be followed by a list of expressions; you cannot use print formatters such as TAB with it.

After ten names and ages have been saved, the example program closes the file it has written to. To get the information back from the file, a program such as the following would suffice:

```
2000 ages=OPENIN("ages")
2010 FOR i%=1 TO 10
2020   INPUT#ages,name$,age%
2030   PRINT "Name: "name$;TAB(20)"Age: ";age%
2040 NEXT i%
2050 CLOSE#ages
```

INPUT# causes the variables to be read from the file specified rather than from the keyboard. (BASIC uses the same routine as for BGET to do this). The usual INPUT prompts and print formatters are not allowed with INPUT#.

Although the examples above use the same variables in the PRINT# and INPUT# statments, there is no need for this. Indeed, as PRINT# can store constants, such a requirement would be impossible.

BASIC automatically converts reals to integers and vice versa where required, so data written with the line:

```
PRINT#file,1.11,324,&123,1.212E2
```

may be read back using:

```
INPUT#file,a%,real,fred%,int%
```

Note however that it is illegal to read a string into a numeric variable, or vice versa, so:

```
PRINT#file,"A string followed by ",anumber%
```

may not be subsequently read back in with:

```
INPUT#file,anumber%,astring$
```

This would produce a `Type mismatch` error.

Variable formats in files

When BASIC puts data into a file using `PRINT#`, it does not simply write the characters that would appear on the screen if the same data were `PRINT`ed. (If you want to do this, use a `*SPOOL` command followed by normal `PRINT` statements.) Instead, a more compact format is used, in which integers, reals and strings are written as a 'type byte', followed by the data proper. The formats are:

Integer

Type byte of &40 (64) followed by the four bytes of the integer, most significant byte first.

Real

Type byte of &FF (255) followed by the four bytes of the mantissa (least significant byte first) followed by the exponent. The internal representation of real variables is discussed in chapter L.2 under `CALL`.

String

Type byte of &00 followed by the length of the string (one byte) followed by the characters of the string in reverse order.

The lengths of the items are therefore 5 for integers, 6 for reals and `LEN(string$)+2` for a string. This information is mainly important when using direct access files.

The program below reads through a file in the format created by `PRINT#` and prints the types of the data it finds. For integers and strings (but not reals), it also prints their values:

```
1000 file=OPENIN"printFile"
1010 REPEAT
1020   type=BGET#file
1030   IF type=0 THEN PROCstr ELSE IF type=&40 THEN PROCint ELSE PRO
Creal
1040 UNTIL EOF#file
1050 CLOSE#file
1060 END
1070
2000 DEF PROCstr
2010  len=BGET#file
2020  str$=""
2030  IF len=0 THEN 2070
2040  FOR i%=1 TO len
2050    str$=CHR$(BGET#file)+str$
2060  NEXT
2070  PRINT "String (len=";len")"TAB(20)str$
2080 ENDPROC
2090
3000 DEF PROCint
3010  FOR i%=3 TO 0 STEP -1
3020    i%?&70=BGET#file
3030  NEXT
3040  PRINT "Integer"TAB(20);!&70
3050 ENDPROC
3060
4000 DEF PROCreal
4010  FOR i%=1 TO 5
4020    dummy = BGET#file
4030  NEXT
4040 PRINT "Real"
4050 ENDPROC
```

The sequential pointer

The file handling discussed until now has been entirely 'serial' in nature: the `*PRINT` type program reads characters from the start of the file and carries on until it reaches the end, while the `*BUILD` type program creates a new file and adds characters to it until the user presses [ESCAPE]. Similarly, the `PRINT#` statements shown until now have simply added data to the file. This method of file access is supported by all filing systems (with the obvious exception of writing to the read-only filing systems such as ROM).

The ADFS and (optional) Network filing systems support a more sophisticated access method known as 'direct' or 'random' access. Associated with each open

file is an entity known as its sequential pointer. If a file is regarded as the array of bytes mentioned above, then the sequential pointer is the subscript of the current element. The sequential pointer is accessed in BASIC through the PTR pseudo-variable. When a file is first opened, its pointer is set to zero, and the line:

```
PRINT PTR#file
```

will print 0. Every time a byte is written to or read from a file, PTR for the file is increased by one. Thus after:

```
FOR i%=1 TO 10 : BPUT#file,i% : NEXT
```

PTR#file will be set to 10. It is also possible to assign to PTR, so that reading or writing occurs at a particular position.

The sequential pointer is most useful when accessing a file using PRINT# or INPUT#. It is often convenient to treat the file as a sequence of records, where a record is a collection of related fields. A field is a value, such as a string or number. Consider a very simple stock control situation. The stock file would be made from records, each of which might consist of a part number, a description, a quantity and a price. To allow you to access any record quickly by its part number, you must have created the file using a fixed length for each record. To read the appropriate record, multiply the record number (which is the same as the part number) by the fixed record length, then set PTR to this value, and you will be ready to read or write that record.

If we have a maximum description length of 15 characters, the record length will be (15+2) + 5 + 6, remembering that strings take up their length plus 2 bytes, integers (the quantity) take 5 bytes and reals (the price) take 6. The record length is therefore 28 bytes, and to access record (part number) n, PTR must be set to 28*n. The program below uses these figures to give a very simple stock control database:

```
1000 REM "Stock Control"
1010 recLen=28:maxRec=100
1020 MODE 135
1030 file=OPENUP("stock")
1040 IF file=0 THEN PROCcreate("stock",maxRec*recLen):file=OPENUP("st
ock")
1060 REPEAT
1070   CLS
1080   PRINT''''"1. Enter a record"''''"2. Examine a record"''''"3. Quit"
1090   PRINT''''"Which one (1-3) "
1100   REPEAT
1110     INPUT TAB(16,10),choice
1120   UNTIL choice>=1 AND choice<=3
```

```
1130   IF choice=1 THEN PROCenter ELSE IF choice=2 THEN PROCexamine
1140 UNTIL choice=3
1150 CLOSE#file
1160 END
1170:
1180 DEF PROCenter
1190  REPEAT
1200    CLS
1210    REPEAT
1220      INPUT TAB(0,3)"Product number",pn%
1230    UNTIL pn%>=0 AND pn%<=maxRec
1240    IF pn%=0 THEN1310
1250    INPUT TAB(0,5)"Description",ds$
1260    ds$=LEFT$(ds$,15)
1270    INPUT TAB(0,7)"Quantity",qn%
1280    INPUT TAB(0,9)"Price",pr
1290    PTR#file=pn%*recLen
1300    PRINT#file,qn%,ds$,pr
1310  UNTIL pn%=0
1320 ENDPROC
1330:
1340 DEF PROCexamine
1350  REPEAT
1360    CLS
1370    REPEAT
1380      INPUT TAB(0,3)"Product number",pn%
1390    UNTIL pn%>=0 AND pn%<=maxRec
1400    IF pn%=0 THEN1500
1410    PTR#file=pn%*recLen
1420    type=BGET#file
1430    IF type<>&40 THEN ds$="Undefined":qn%=0:pr=0:GOTO1500
1440    PTR#file=pn%*recLen
1450    INPUT#file,qn%,ds$,pr
1460    PRINTTAB(0,5)"Description:"TAB(20)ds$
1470    PRINT'"Quantity:"TAB(20);qn%
1480    PRINT'"Price:"TAB(20);pr
1490    wait=GET
1500  UNTIL pn%=0
1510 ENDPROC
1520:
1530 DEF PROCcreate(file$,length)
1540  file=OPENOUT(file$)
1550  PTR#file=length
```

```
1560  CLOSE#file
1570 ENDPROC
```

The call to PROCcreate is only made if a file called stock does not exist already. If it does, it is simply opened for update. The program's main loop prints a menu and performs one of three tasks; enter a record, examine a record or quit. The examine procedure detects if a valid item is at the record specified by testing if the first byte of the record is &40 (the integer type byte). If it is, the record is valid and its contents are printed, otherwise a dummy value is assigned and that is printed. PROCcreate simply opens for output the file named in file$, sets its pointer to length (which will fill the file from position 0 to length–1 with zeros), then closes it again.

File access summary

OPENIN

Open for input. In general, this returns a channel number for the file specified, or 0 if no such file can be found. The position of the character to be read may be set by PTR#=<expression>. The PTR may not be set at or past EXT, and the file's length may not be changed using EXT#=<expression>.

In the CFS and RFS, PTR and EXT are not valid. The CFS never returns 0 as it waits for the file to be found before OPENIN returns. The RFS may return zero.

OPENUP

Open for update. Returns channel number or zero. The file may be lengthened by setting PTR or EXT, and shortened by setting EXT. BPUT and BGET may be used.

In the CFS and RFS, OPENUP acts exactly as OPENIN.

OPENOUT

Open for output. Returns channel number or zero. The file is created if it doesn't exist already, or will be overwritten if it does. The ADFS allocates 64K (using the same mechanism as *CREATE) when a file is opened for output. The file may be lengthened by setting PTR or EXT, and shortened by setting EXT. BPUT and BGET may be used.

The CFS never returns zero. PTR and EXT are not valid. In the RFS, OPENOUT may not be used.

L BASIC Keywords

L.1 The syntax of BASIC

This section describes each of the BASIC IV keywords in detail. It gives the format of keyword (its syntax) any arguments it needs and what result it returns (for functions). Also given are examples and any other points which are relevant to the use of the keyword.

Introduction to BASIC syntax

A BASIC program is a set of BASIC lines. A line consists of an (integer) line number in the range 0-32767 followed by one or more statements. When a line is being entered, it may be up to 248 characters long. A line without a line number is executed immediately; it may be a command (eg `LIST`) or statement(s). Statements on a single line are separated by colons.

Here are some examples of legal BASIC syntax:

```
10 PRINT "Hello, world"
FOR i%=32 TO 126:VDU i%:NEXT
LIST ,100
```

Here are some examples of illegal BASIC syntax:

```
10 LIST          (A command in a program line)
20 PRUNT 1       (An incorrectly spelled keyword)
```

Note that the syntax of program lines is not checked until they are executed. Thus the illegal lines above would be accepted when typed in, but would produce a 'Syntax error' when the program was `RUN`.

Most BASIC keywords may be abbreviated to the first few letters followed by a dot. For example, `ENDPROC` may be typed as `E.` . When the program is `LIST`ed, the full spelling is produced. The minimum abbreviation for each keyword is given in the next chapter, and these are summarised in chapter N.2.

Spaces are optional in most circumstances, but are often used to make the program easier to read. One place where spaces are required is when a keyword follows an identifier, eg `DEF PROCfred PRINT...`, not `DEF FNfredPRINT`.

The 'marker' keywords `DEF` and `DATA` must be the first keywords on the line.

Where possible, plain English is used to describe the actions of the keywords.

However, formal syntax descriptions are also used because they are more precise and compact. The notation used in these syntax descriptions is as follows.

Words in upper case are the keywords themselves. Most punctuation, such as , and = also stands for itself. The exceptions are square brackets and angled brackets. Square brackets, [and], enclose items that are optional, eg:

```
STR$[~]<factor>
LOCAL [<variable>] [,<variable>] etc
```

In the first example, the tilde may be omitted. In the second example, the first variable and the variables following it are optional.

When an item is followed by etc, the item may be repeated an arbitrary number of times (including zero).

Angled brackets < and > are used to delimit parts of the syntax which are not to be taken literally, but stand for a class of objects. An example is <relational> which means 'any relational expression'. The classes used are as follows:

<factor>

This means an expression which is a single unit, ie a variable reference, a constant, a function call (user-defined or built-in), or a general expression enclosed in round brackets. The argument of a single-parameter function is always a <factor>, so examples of function calls are:

```
SINRAD45
LENA$
STR$1234
EXP-A%
TAN(ANG%+45)
```

It can be seen that brackets are only needed around arguments if they contain more than one 'element' or, as in the case of RND, if the argument is optional.

<expression>

This is any valid numeric or string expression. The type of the expression required is always given in the explanation of the keyword. Where more than one expression is required, <expression1>, <expression2> is sometimes used to make the identification of each expression easier.

In general, wherever an integer is required, BASIC will accept a real, as long as it can be converted into an integer. This means the real must be in the range −2147483648 to +2147483647 to be used as an integer. Moreover, if an integer in the range, say, 0-255 is required, BASIC will accept any integer and only use

the least significant byte. This is mostly the case when BASIC uses the operating system. Only enough bytes will be taken from an expression to satisfy the requirements of the MOS. A full list of the ways in which BASIC uses the operating system is given at the end of Section N.

<relational>

This is an expression that contains operators of higher precedence than the logical operator under discussion. Thus in the syntax:

<relational>`AND`<relational>

the relational expression may contain any operator except `AND`, `OR` and `EOR`, as these are the lowest precedence operators. To use one of these operators (eg `OR`) in an operand, you will need brackets:

```
(a>b OR  c<d) AND f<>1
```

For a full description of operator precedence, see Section K.

<operand>

This is used to stand for the operands of `*`, `/`, `DIV` and `MOD`. It may be any expression containing operators of higher precedence than these operators. This means <factor> or <factor>`^`<factor>. Thus to divide `2^power` by `LOG(1Ø)`, we could simply say:

```
2^power DIV LOG(1Ø)
```

but to divide `2+disp` by `1Ø+len`, brackets are needed:

```
(2+disp) DIV (1Ø+len)
```

<integer>

This is used in the `AUTO`, `RENUMBER` and `DELETE` commands and means an integer constant between 0 and 32767 (a line number).

<step>

This is also used by `RENUMBER` and `AUTO` and means an integer between 1 and 255.

<line range>

This is the range of line numbers to be used by `LIST` and `EDIT`. It is zero, one or two <integers>, separated by a comma. If the first number is absent, 0 is assumed. The second line defaults to 32767. Thus:

```
LIST          means LIST Ø,32767
LIST 1ØØ,     means LIST 1ØØ,32767
```

```
LIST ,300        means LIST 0,300
LIST 200,1000    means LIST 200,1000
```

<variable>

This is a reference to any variable. Examples are `a`, `a%`, `a$`, `a(23)`, `a%(32)`, `a$(i+1)`, `$a`, `?a`, `!a`, `a?1`, `a!1` etc. Sometimes, it is qualified, eg <numeric variable>. There is a special case of <numeric variable> in the second form of the `DIM` statement, which is only a or `a%` types, not `!a`, `a(1)` etc

<identifier>

This is a sequence of one or more alphanumeric characters, starting with an alphabetic character, ie a real variable name. In BBC BASIC, alphabetic characters are `A-Z`, `a-z`, `£` and `_`. Numerics are the digits `0-9`. There is also the special identifier, `@%` (see `PRINT`). User-defined procedures and functions may have names that start with a digit, and contain `@`. Other identifiers may not.

<string>

This means zero or more ASCII characters. For example, after a `*` statement, there is a <string> which is passed to the operating system. A <string> is terminated by the carriage return at the end of the line.

<space>

This is simply the ASCII character " ", with code 32, or &20. There are few occasions where spaces are necessary in BBC BASIC, although they can be used to good effect to format a program so it is easy to understand.

<statements>

This means zero or more statements, separated by colons, eg `PRINT "Hello": GOTO 100`. Null statements are permitted in BBC BASIC, so lines such as:

```
IF GET
```

are valid. In this example, the optional `THEN` is omitted, and there is a null `THEN` part. The keywords `REPEAT`, `THEN`, `ELSE` and the assembly language introducer `[` do not require colons to separate them from the next statement. Also, there is no need for a colon after the `PROC` or `FN` part of a `DEF` statement.

<proc part>

This is the text that follows the `PROC` or `FN` part of a user-defined procedure or function. It has the format:

<identifier>[<parameter list>]

where <identifier> is described above and the optional <parameter list> has a

format dependent on the context. If the PROC/FN is in a DEF statement, it has the format:

(<variable> [,<variable>] etc)

that is, one or more variables (the formal parameters) between round brackets. When used in a PROC/FN call, the parameter list has the format:

(<expression> [,<expression>] etc)

where the expressions correspond in type to the formal parameters.

L.2 Syntax and usage of BASIC keywords

Below are the complete descriptions of the statements, commands and functions that BASIC understands, in alphabetical order. None of the assembler mnemonics or pseudo-operators are mentioned, as these are described in Section P.

ABS

Function giving magnitude of its numeric argument

Syntax

`ABS`<factor>

Argument

Any numeric.

Result

Same as the argument if this is positive, or −(the argument) if it is negative.

Example

```
diff=ABS(length1-length2)
```

Note

The largest negative integer does not have a legal positive value, so that if `a%=-2147483648`, `ABS(a%)` yields `-2147483648`.

ACS

Function giving the arc-cosine of its numeric argument

Syntax

`ACS`<factor>

Argument

Real or integer between −1 and 1 inclusive.

Result

Real in the range 0 to PI (radians).

Examples

```
angle=DEG(ACS(cos1))
PRINT ACS(Ø.5)
```

ADVAL AD.

Function reading data from an analogue port or giving buffer data

Syntax

(1) ADVAL<factor>
(2) ADVAL<factor>

(1)

Argument

Negative integer −n, where n is a buffer number between 1 and 9.

Result

The number of spaces or entries in the buffer is given in the table below

−n	Buffer name	Result	
−1	Keyboard (input)	Number of characters used	(0-31)
−2	RS-423 (input)	Number of characters used	(0-255)
−3	RS-423 (output)	Number of characters free	(0-191)
−4	Printer (output)	Number of characters free	(0-63)
−5	Sound 0 (output)	Number of bytes free	(0-15, step 3)
−6	Sound 1 (output)	Number of bytes free	(0-15, step 3)
−7	Sound 2 (output)	Number of bytes free	(0-15, step 3)
−8	Sound 3 (output)	Number of bytes free	(0-15, step 3)

In the table 'step 3' means that one entry in the buffer uses three bytes.

(2)

Argument

Zero

Result

ADVAL(Ø) gives information on the joystick 'fire' buttons and the last analogue channel to complete conversion

```
ADVAL(Ø) AND 1      is non-zero if the 'left' button is being pressed
ADVAL(Ø) AND 2      is non-zero if the 'right' fire button is being pressed
ADVAL(Ø) DIV &1ØØ   gives the number of the last A/D channel (1-4) to complete
                    a conversion, or 0 if no channel has completed since the
                    last *FX16 or *FX17.
```

(3)

Argument

Positive n, where n is the number of an analogue channel in the range 1-4.

Result

Integer in the range 0-65520 increasing in steps of 16. A voltage of zero on channel n gives a result of 0, increasing linearly to approximately 1.8 volts on channel n, which gives a result of 65520.

Examples

```
IF ADVAL(Ø)=3 THEN PROCboth_firing
REPEAT UNTIL ADVAL(1) < 32768
```

AND

A.

Operator giving logical or bitwise AND

Syntax

<relational>`AND`<relational>

Operands

Relational expressions, or 'bit' values to be `AND`ed

Result

The logical bitwise AND of the operands. Corresponding bits in the integer operands are `AND`ed to produce the result. If used to combine relational values, `AND`'s operands should be true logical values, ie `TRUE` (−1) or `FALSE` (0), otherwise unexpected results may occur. For example, 2 and 4 are both 'true' (ie non-zero), but `2 AND 4` yields `FALSE`, ie zero.

Examples

```
a= x AND y:REM a is set to binary and of x and y
PRINT variable AND 3:REM print lowest 2 bits of variable
IF day=7 AND month$="March" THEN PRINT "Happy birthday, Rob"
IF temp>5Ø AND NOT windy THEN PROCgo_out ELSE PROCstay_in
REPEAT a=a+1
b=b-1
UNTIL a>1Ø AND b<Ø
isadog= feet=4 AND tails=1 AND hairy:REM set isadog to logical
true if all conditions are met
```

ASC

Function giving the ASCII code of the first character in string

Syntax

`ASC`<factor>

Argument

String of 0-255 characters.

Result

ASCII code of the first character of the argument in the range 0-255, or −1 if the argument is a null string.

Examples

```
x2=ASC(name$)
IF code >= ASC("a") AND code <= ASC("z") THEN PRINT "Lower case"
```

ASN

Function giving the arc-sine of its numeric argument

Syntax

`ASN`<factor>

Argument

Numeric between −1 and 1 inclusive.

Result

Real in the range −PI/2 to +PI/2 radians.

Examples

```
PRINT ASN(opposite/hypotenuse)
angle=DEG(ASN(0.2213))
```

ATN

Function giving the arc-tangent of its numeric argument

Syntax

`ATN`<factor>

Argument

Any numeric

Result

Real in the range −PI/2 to +PI/2 radians.

Examples

```
PRINT "The slope is ";ATN(opposite/adjacent)
```

AUTO AU.

Command initiating automatic line numbering

Syntax

AUTO [<integer>] [,<step>]

Parameters

<integer> is an integer constant 0-32767 and is the first line to be automatically generated. <step> is an integer constant 1-255 and is the amount by which line numbers increase when **RETURN** is pressed.

Purpose

AUTO is used when entering program lines to produce a line number automatically, so you do not have type them yourself. To turn off AUTO, use **ESCAPE**. AUTO will stop if the line number becomes greater than 32767.

Examples

```
AUTO
AUTO 1000
AUTO 12,2
```

BGET# B.#

Function returning the next byte from a file

Syntax

BGET#<factor>

Argument

A channel number returned by an OPEN function in the range 1-255 (depending on the filing system and how many files are open).

Result

The ASCII code of the character read (at position PTR#) from the file in the range 0-255.

Note

PTR# is updated to 'point to' the next character in the file. If the last character in the file has been read, EOF# for the channel will be TRUE. The next BGET# will return 254 and the one after that will produce an 'EOF' error.

Examples

```
char%=BGET#(channel)
char$=CHR$(BGET#fileno)
```

BPUT# BP.#

Statement to write a byte to a file

Syntax

BPUT#<factor>,<expression>

Arguments

<factor> is a channel number in the range 1-255, as returned by an OPEN function. The <expression> is an integer 0-255, and is the ASCII code of the character to be sent to the file.

Note

PTR# is updated to point to the next character to be written. If the end of the file is reached, the length (EXT#) will increase too. It is only possible to use BPUT# with OPENUP and OPENOUT files, not OPENIN ones.

Examples

```
BPUT#outfil,byte%
BPUT#2,ASC(mid$(name$,pos,1))
```

CALL CA.

Statement to execute a machine code subroutine

Syntax

CALL<expression> [,<variable>] etc

Arguments

<expression> is the address in the range 0-&FFFF (65535) of the routine to be

called. The zero or more parameter variables may be of any type, and must exist when the CALL statement is executed. They are accessed through a parameter block that BASIC sets up at location &600. The format of this parameter block and of the variables accessed through it are described below.

Purpose

Use CALL to enter a machine code program from BASIC. Before the routine is called, the least significant bytes of A%, X% and Y% are transferred to the A, X and Y 65C12 registers and the least significant bit of the C% variable is transferred into the C bit of the status register.

Format of the CALL parameter block

The addresses of the variables given after the machine code address in the CALL statement are stored in a table which may be accessed by the machine code routine. This table, called the parameter block, starts at address &600, and has the following format

Address	Contents
&600	Number of parameters
&601	Address of parameter 0
&603	Type of parameter 0
&604	Address of parameter 1
&606	Type of parameter 1
&607	Address of parameter 2
&609	Type of parameter 2
&60A	and so on...

In summary, &600 contains the number of parameters, &601+3*n and &602+3*n contain the address of parameter n (in normal low-byte, high-byte order), and &603+3*n contains the type number of parameter n. There may be 0 to 85 parameters.

The type numbers of variables are

Type	Description
&00	A one-byte integer, eg ?X%
&04	A four-byte integer, eg !X%, X%
&05	A five-byte real number, eg X
&80	A string at an address, eg $X%
&81	A string variable, eg X$

These types have the following formats:

?X%

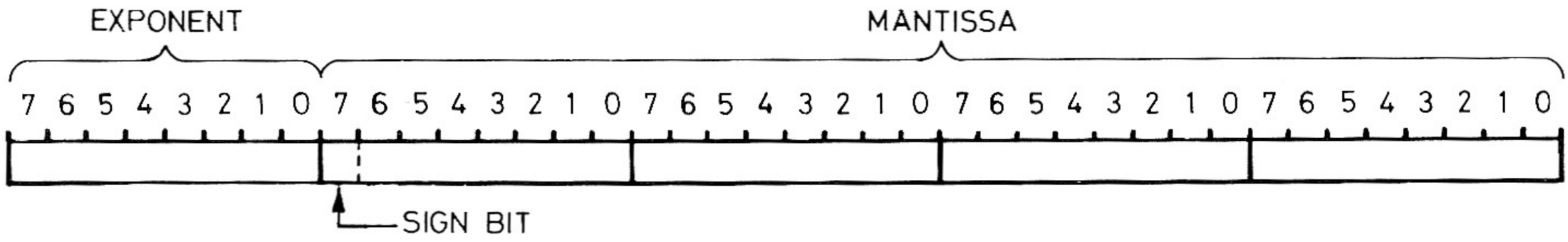
The address points to a single-byte value in the range 0-255, or −128 to +127 if it is being interpreted as a signed quantity.

!X%, X%

The address points to a four-byte integer, with the least significant byte at the lowest address. Integers are stored in two's complement form, and so can have values in the range −2147483648 (&80000000) to +2147483647 (&7FFFFFFF).

X

The address points to a five-byte real number. The first byte is the exponent in excess−128 format. The next four bytes are the mantissa, most significant byte first. The most significant bit of the first mantissa byte is the sign of the whole number. The mantissa is always normalised so that there is an assumed 0.1 before the 31 bits of the stored mantissa. This format is summarised in the diagram below:



$X%

The address points to a string of up to 255 characters which is terminated by a carriage-return (CHR$(13)). Any character except carriage-return may appear in the string. If the address points directly at the carriage return, the string is a null string, ie has a length of zero.

X$

The address points at the string information block of the variable. This is a four-byte table giving the following information

Address+0	Start address of the string	(2 bytes)
Address+2	Number of bytes allocated	(1 byte)
Address+3	Current length of string	(1 byte)

In order to access the contents of the string, the address given in the CALL parameter block must be used to access the string information block. The first two bytes of this structure give the address of the text of the string. The third byte of the SIB gives the size to which the string may grow before a new area of memory is allocated for it. The last byte gives the current length of the string. Both of these bytes are in the range 0-255.

Accessing arrays with CALL

The variables passed to a CALLed routine may be array elements. As arrays are

always stored contiguously, the address of an array may be found by passing its first element. Similarly, the addresses of a range of elements may be found by passing the first and last elements in the range.

An example would be a sort program, which may be called with `CALL sort,name$(0),name$(100)`. The `CALL` parameter block would have two entries; one for the address of the SIB of `name$(0)`, and one for the address of the SIB of `name$(100)`. The SIB for `name$(1)` would be found at the address of the SIB of `name$(0)` plus four.

The same technique may be applied for arrays of reals and integers, though in these cases the `CALL` parameter block addresses point to the elements themselves, rather than information blocks.

CHAIN CH.

Statement to `LOAD` and `RUN` a BASIC program

Syntax

`CHAIN`<expression>

Argument

<expression> should evaluate to a filename (and therefore a string) which is valid for the filing system in use, eg up to ten characters long for the cassette filing system.

Notes

A filing system error may be produced if, for example, the file specified can't be found. When the program is loaded, all existing variables are lost (except the system integer variables). It is possible to `LOAD` and then `RUN` a 'bad' program, but not to `CHAIN` one.

Examples

```
CHAIN "partB"
CHAIN a$+"2"
```

CHR$

Function giving the character corresponding to an ASCII code

Syntax

`CHR$`<factor>

Argument

An integer in the range 0-255.

Result

A single-character string whose ASCII code is the argument.

Examples

```
PRINT CHR$(code);
lower$=CHR$(ASC(upper$) OR &20)
```

CLEAR CL.

Statement to remove all program variables

Syntax

```
CLEAR
```

Purpose

When this statement is executed, all variables except the system integer variables are removed, and so become undefined. In addition, any procedures, subroutines, loops etc that are active will be forgotten.

CLG

Statement to clear the graphics window to the graphics background colour

Syntax

```
CLG
```

CLOSE# CLO.#

Statement to close an open file

Syntax

```
CLOSE#<factor>
```

Argument

An integer in the range 0-255. If 0 is used, all open files will be closed, otherwise only the file with the channel number specified will be closed. The channel number should have been assigned by an `OPEN` function.

Purpose

Closing a file ensures that its contents will be updated on whatever medium is being used. This is necessary as a certain amount of buffering is used to make the transfer of data between computer and mass-storage device more efficient. Closing a file therefore releases a buffer for use by another file (all filing

systems have a limit to the number of files that may be open at once, eg 2 for the CFS and 10 for the ADFS).

Examples

```
CLOSE#indexFile
CLOSE#0:REM This closes all open files.
```

CLS

Statement to clear the text window to the text background colour

Syntax

```
CLS
```

Note

`CLS` also resets `COUNT` to zero.

COLOUR (COLOR)

C.

Statement to set the text colours

Syntax

`COLOUR`<expression>

Argument

<expression> is an integer in the range 0-255. The range 0-15 sets the text forground colour; adding 128 to this (ie 128-143) sets the text background colour. The colour is treated `MOD` the number of colours in the current mode. The argument is the 'logical' colour. For a list of the default logical colours, see the VDU drivers section. The actual colours produced by the logical colours may be altered using `VDU 19`. The table on the next page shows the default colours for two, four and sixteen-colour modes

Note that only colours 0 and 1 are 'valid' in two-colour modes. After that the cycle repeats. Similarly, only colours 0, 1, 2 and 3 are distinct in the four-colour modes. Colours 8-15 in the 16-colour modes are the flashing colours; the screen alternates between the two colours shown at a rate set by `*FX9` and `*FX10`.

Examples

```
COLOUR fore+1
COLOUR 128+back
```

Colour	2-colour modes	4-colour modes	16-colour modes
0	Black	Black	Black
1	White	Red	Red
2	Black	Yellow	Green
3	White	White	Yellow
4	Black	Black	Blue
5	White	Red	Magenta
6	Black	Yellow	Cyan
7	White	White	White
8	Black	Black	Black/white
9	White	Red	Red/cyan
10	Black	Yellow	Green/magenta
11	White	White	Yellow/blue
12	Black	Black	Blue/yellow
13	White	Red	Magenta/green
14	Black	Yellow	Cyan/red
15	White	White	White/black

COS

Function giving the cosine of its numeric argument.

Syntax

`COS`<factor>

Argument

<factor> is an angle in radians. The angle is scaled down so that it is between `-PI` and `PI` radians.

Result

Real between −1 and +1.

Notes

If the argument is outside the range −8388608 to 8388608 radians it may not be scaled down correctly and the error `Accuracy lost` will be given.

Examples

```
PRINT COS(RAD(45))
adjacent = hypotenuse*COS(angle)
```

COUNT COU.

Function giving the number of characters printed since the last newline

Syntax

```
COUNT
```

Result

Integer between 0 and 255, giving the number of characters output since the last newline was generated by BASIC.

Notes

`COUNT` is reset to zero every time a carriage return is printed (which may happen automatically if `WIDTH` is being used). It is incremented every time a character is output by `PRINT`, `INPUT` or `REPORT`, but not by `VDU` or any of the graphics commands. `COUNT` is also rest to zero by `CLS` and `MODE`.

Examples

```
REPEAT PRINT " ";
UNTIL COUNT=20
chars = COUNT
```

DATA D.

Passive statement marking the position of data in the program

Syntax

`DATA` [<expression>] [`,`<expression>] etc

Arguments

The expressions may be of any type and range, and are only evaluated when a `READ` statement requires them.

Notes

The way in which `DATA` is interpreted depends on the type of variable in the `READ` statement. A numeric `READ` will evaluate the data as an expression, whereas a string `READ` will treat the data as a literal string. Leading spaces in the data item are ignored but trailing spaces (except for the last data item on the line) are counted. If it is necessary to have a comma or quote in the data item, it must be put within quotes, eg

```
DATA "A,B", """"ABCD"
```

If an attempt is made to execute a `DATA` statement, BASIC treats it as a `REM`.

The DATA statement, like the other passive statements, should be the first on a line in order to be recognised by BASIC.

Examples

```
DATA Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
DATA 3.26,SE-3,"WATTS"
```

DEF

Passive statement defining a function or procedure.

Syntax

`DEF FN`<proc part>
or
`DEF PROC`<proc part>

where <proc part> has the form <identifier>[<parameter list>]

Parameters

The optional parameters, which must be enclosed between round brackets and separated by commas, may be of any type, eg parm, parm%, parm$, !parm, $parm etc

Purpose

The DEF statement marks the first line of a user-defined function or procedure, and also indicates which parameters are required and their types. The parameters are local to the function or procedure, and are used within it to stand for the values of the actual parameters used when it was called.

Notes

Function and procedure definitions should be placed at the end of the program, so that they cannot be executed except when called by the appropriate PROC statement or FN function. The DEF statement should be the first on its line.

Examples

```
DEF FNmean(a,b)
DEF PROCinit
DEF PROCthrow_dice(d%,tries,mesg$)
```

DEG

Function returning the numbers of degrees of its radian argument.

Syntax

DEG<factor>

Argument

Any numeric value.

Result

A real equal to 180*n/PI, where n is the argument's value.

Example

```
angle=DEG(ATN(a))
PRINT DEG(PI/4)
```

DELETE DEL.

Command section of the program

Syntax

DELETE<integer>,<integer>

Arguments

Integer constants in the range 0-32767. They give the first and last line to be deleted respectively. If the first line number is greater than the second, only the first line specified is deleted.

Examples

```
DELETE 110,150
DELETE 0,10000
```

DIM

Statement declaring arrays or reserving storage.

Syntax

DIM <dim part> [,<dim part>] etc

where <dim part> is

(1) <identifier>[% or $](<expression> [,<expression>] etc) or
(2) <numeric variable><space><expression>

(1)
The <identifier> can be any real, integer or string variable name. The expressions are integers which should evaluate to 0-32767. They declare the upper bound of the subscript; the lower bound is always 0.

This is the usual way of declaring arrays in BASIC. They may be multi-dimensional, and the upper bound is limited only by the amount of memory in the computer. Numeric arrays are initialised to zeros and string arrays to null strings.

(2)
The <numeric variable> is any integer or real name. The <expression> gives the number of bytes of storage required minus 1 and should be in the range −1 to 65535.

The use of this form of `DIM` is to reserve a given number of bytes, for example in which to put machine code. The address of the first byte reserved is placed in the <numeric variable>. The byte array is uninitialised.

Examples

```
DIM name$(num_names%)
DIM sin(90)
DIM vartop -1:REM sets vartop to the current top of variables
DIM bytes% size*10+overhead
```

DIV

Integer operator giving the quotient of its operands

Syntax

<operand>`DIV`<operand>

Operands

Integer-range numerics. Reals are converted to integers before the divide operation is carried out. The righthand side must not evaluate to zero.

Result

The (integer) quotient of the operands, always rounded towards zero.

Examples

```
PRINT (first-last) DIV 10
a%=space% DIV &100
```

DRAW DR.

Statement to draw a line to specified co-ordinates

Syntax

`DRAW`<expression>`,`<expression>

Arguments

The <expressions> are integer numerics in the range −32768 to +32767. They are the coordinates to which a line is drawn in the current graphics foreground colour. The graphics cursor position is updated to these co-ordinates. DRAW is equivalent to PLOT 5.

Examples

```
DRAW 640,512:REM Draw a line to the middle of the screen
DRAW x%*16, y%=4
```

EDIT ED.

Command calling the text editor from within BASIC

Syntax

EDIT [<line range>] [IF <string>]

Arguments

See the section on LIST for a description of the <line range> and IF part.

Purpose

The EDIT command converts the lines specified (the whole program by default) into a text file in memory. The screen editor, which is usually accessed by the *EDIT command, is then entered. This enables the BASIC program to be edited as a text file, with all the usual search and replace facilities that are available in the screen editor. When the program has been edited to the user's satisfaction, the 'Return to language' command may be used to re-enter BASIC. As this is done, BASIC re-tokenises the program so that it may be executed as usual.

Notes

When the program is being converted for the Screen Editor, there has to be enough room in memory for both the tokenised and text versions. If there isn't, a No room error will be given, along with the line number at which the conversion process failed. At this stage, CLEAR should be typed. It can be seen that the largest program that may be edited occupies about two thirds of the memory between PAGE and HIMEM. If this limitation proves to be too severe, it is suggested that the program be maintained as an EDIT text file, and only tokenised into BASIC's internal form when it has to be RUN, by issuing a 'Return to language' command from the editor. As the tokenisation process overwrites the text version of the program present, very large files may be tokenised.

When BASIC calls the editor, a special form of the *EDIT command is used *EDIT <ptr1>,<ptr2>. <ptr1> is the address in hex of a pointer to the start of the text area. <ptr2> is the address in hex of a pointer to the byte after the end of the text area.

ELSE EL.

Part of the IF ... THEN ... ELSE construct

Syntax

ELSE<statements>

Notes

ELSE may occur anywhere in the program, but is only meaningful after an IF or ON ... GOSUB/GOTO statement. If the expression after the IF evaluated to FALSE (zero), or the expression after the ON is not in the correct range, then the statements following the ELSE will be executed. Elsewhere, ELSE is treated as a REM statement.

Examples

```
IF a=b THEN PRINT "hello" ELSE PRINT "goodbye"
IF ok ELSE PRINT "Error"
ON choice GOSUB 100,200,300,400 ELSE PRINT"Bad choice"
```

END

Statement terminating the execution of a program

Syntax

END

Note

END is not always necessary in programs; execution will stop when the line at the end of the program is executed. However, END (or STOP) must be included if execution is to end at a point other than at the last program line. This prevents control 'falling through' into a procedure, function or subroutine.

ENDPROC E.

Statement marking the end of a user-defined procedure

Syntax

ENDPROC

Purpose

When executed, an ENDPROC statement causes BASIC to terminate the execution of the current procedure, and restore local variables and actual parameters. Control is passed to the statement after the PROC which called the procedure. ENDPROC should only be used in a procedure, otherwise a No PROC error will be given when it is encountered.

Examples

```
ENDPROC
IF a<=0 THEN ENDPROC ELSE PROCrecurse(a-1)
```

ENVELOPE ENV.

Statement defining a sound envelope

Syntax

ENVELOPE <expression1>, ... <expression14>

Arguments

The 14 expressions are all treated as one-byte quantities. They control various aspects of the way in which sound generated by the SOUND statement changes in pitch and loudness with time. The meanings of the expressions are as follows:

(1) N – Envelope number.

This is in the range 1-16. If RS-423 output is being used, or a cassette output file (ie an OPENOUT file) is open, then only envelope numbers 1-4 are available.

(2) T – Time interval.

This is length of a single time step in the envelope, and it controls the rate at which the pitch and loudness of the sound change. It is in the range 0-127 (in centi-seconds). An interval of zero is taken to mean an interval of one centi-second. Usually the pitch envelope (see below) 'auto-repeats' at the end of its cycle. By adding 128 to the time inerval, so that it lies in the range 128-255, this auto-repeat is disabled.

(3)-(5) PI1, PI2 and PI3 – Pitch increments.

The pitch envelope is divided into three sections. These parameters control the amount by which the pitch varies on each envelope step. They are treated as signed one-byte quantities in the range −128 to +127.

(6)-(8) PN1, PN2 and PN3 – Pitch steps.

These three parameters give the number of steps in each section of the pitch

envelope, in the range 0-255. The pitch varies by PI1 for PN1 steps, then by PI2 for PN2 steps and finally by PI3 for PN3 steps. Pitch chnages occur every T centi-seconds. The amount by which the pitch varies throughout the envelope is thus given by PI1*PN2+PI2*PN2+PI3*P3. After time T*(PN1+PN2+PN3) the pitch envelope ends. If T is in the range 0-127, the pitch is reset to its initial value and the pitch envelope repeats, otherwise the pitch remains at its final value.

(9) AA – Attack phase amplitude change.

Whereas in the pitch of an envelope-controlled note starts at the value given by the SOUND statement causing it, the amplitude (loudness) always starts at zero. It increases every T centi-seconds by the amount specified in AA. This is a signed integer in the range −128 to +127, though as the initial amplitude is zero, only positive steps give predictable results.

(10) AD – Decay phase amplitude change.

Once the amplitude has reached the value given by ALA (see below), the decay phase commences. During this phase, the value in AD is added to the amplitude until this reaches ALD (see below). AD may be in the range −128 to +127.

(11) AS – Sustain phase amplitude change.

When the final decay amplitude is reached (ALD), the volume varies by the step given in AS. As its name implies, this is often zero, so there is no change in amplitude, though like the other amplitude change parameters, it may be in the range −128 to 127. The note changes by AS until it has been sounding for the time given in the SOUND statement that started it.

(12) AR – Release phase amplitude change.

At the end of the sustain phase, the amplitude envelope enters its final stage, the release. During this phase, the amplitude is varied every T centi-seconds by the amount given in AR, which, as usual, is in the range −128 to +127. The release phase terminates when the amplitude reaches zero. By making AR zero, the note may be sounded at the level set during the sustain phase indefinitely.

(13) ALA – Attack phase target level.

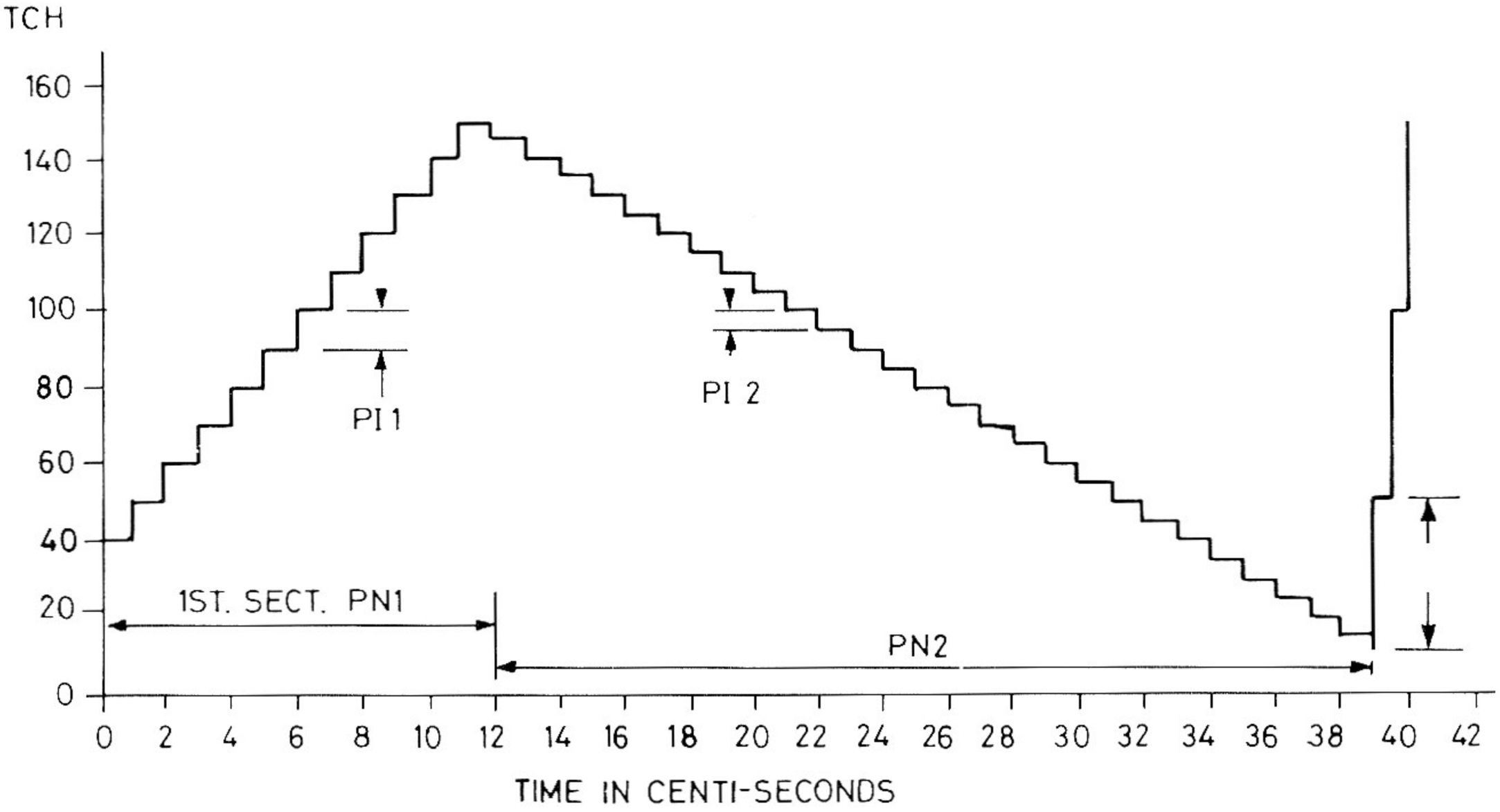
The volume of the sound increments by AA every T centi-seconds until the amplitude ALA is reached. ALA should be in the range 0-126.

(14) ALD – Decay phase target level.

The volume of the sound increments by AD every T centi-seconds until the amplitude ALD is reached. ALD should be in the range 0-126.

Pitch envelope example

Below is a diagram of a typical pitch envelope. It shows the time in centi-seconds along the horizontal axis and the pitch value along the vertical axis.



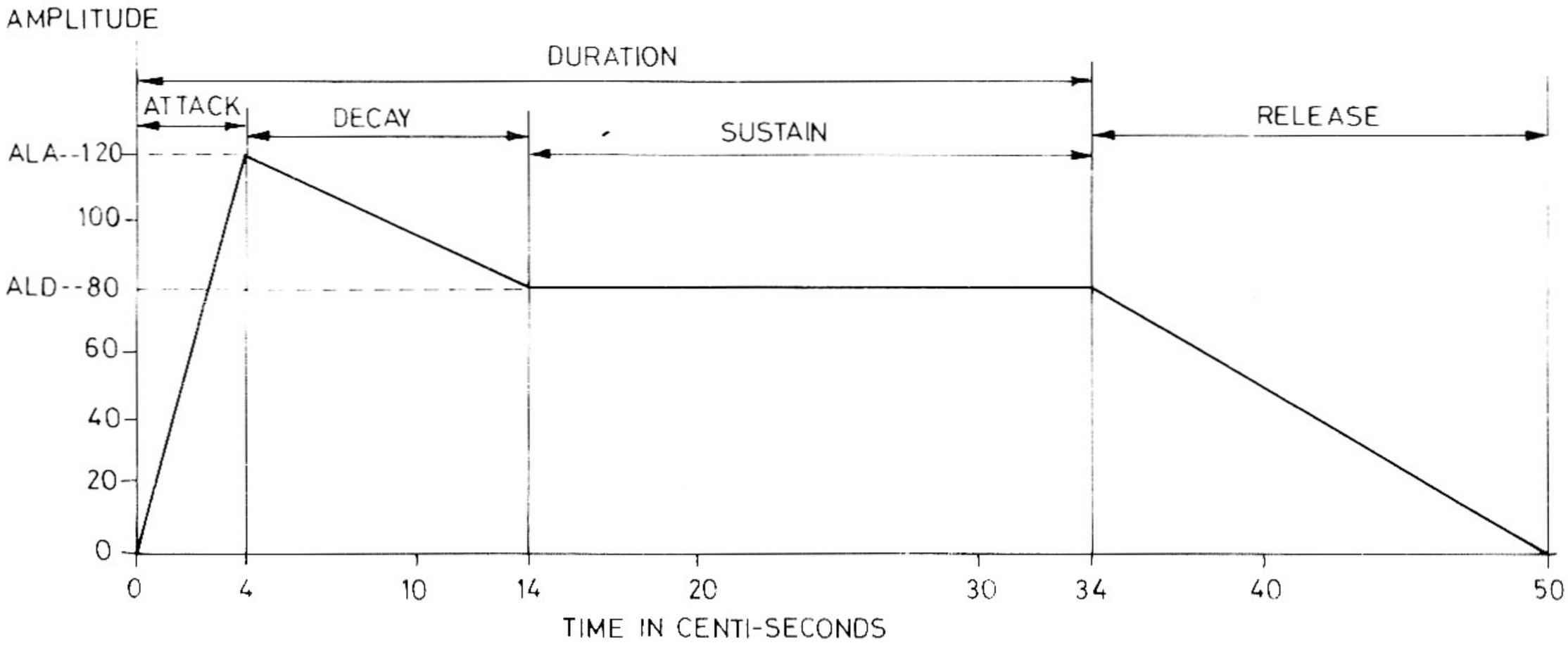
The ENVELOPE statement to produce the pitch variation shown in the diagram is

```
ENVELOPE 1,1,10,-5,50,12,27,3,127,0,0,-127,127,0
```

The SOUND command used would be something like `SOUND 1,1,40,10`. The amplitude parameters are irrelevant in this example, the important part of the envelope being the step T of one centi-second, and the six pitch parameters. It can be seen that the pitch varies in steps of 10 for 12 centi-seconds, then decreases in step of 5 for 27 centi-seconds, and finally increases in steps of 50 for three centi-seconds. As T is less than 128, the pitch envelope auto-repeats. Note that the pitch parameters 'wrap around', so if an attempt is made to increase the pitch above 255, the new value will be treated `MOD 256`.

Amplitude envelope example

The diagram below shows how the volume of a note may vary with time. The horizontal axis shows times in centi-seconds and the vertical axis shows the volume.



The command used to produce this envelope is

```
ENVELOPE 1,1,0,0,0,0,0,0,30,-4,0,-5,120,80
```

The SOUND command producing the duration shown in the diagram would be `SOUND 1,1,40,7`.

EOF#

Function giving end of file status of a file

Syntax

EOF#<factor>

Argument

A channel number returned by an OPEN function in the range 1-255 (depending on the filing system and how many files are open).

Result

TRUE if the last character in the specified file has been read, FALSE otherwise.

Examples

```
REPEAT VDU BGET#file
UNTIL EOF#file
IF EOF#invoices PRINT "No more invoices"
```

EOR

Operator giving the logical or bitwise excusive-OR

Syntax

<relational>EOR<relational>

Operands

Relational expressions, or 'bit' values to be exclusive-ORed

Result

The logical bitwise exclusive-OR of the operands. Corresponding bits in the operands are ex-ORed to produce the result. The result is zero if the operands are equal, non-zero otherwise.

Examples

```
PRINT height>10 EOR weight<20
bits = mask EOR value1
```

ERL

Function returning the last error line

Syntax

```
ERL
```

Result

Integer between 0 and 32767. This is the line number of the last error to occur. An error line of zero implies that the error happened in 'immediate mode'.

Examples

```
REPORT
IF ERL<>0 THEN PRINT " at line ";ERL
IF ERL=3245 PRINT "Bad function, try again"
```

ERR

Function returning the last error number

Syntax

```
ERR
```

Result

An integer between 0 and 255. Errors produced by BASIC are in the range 1-127 and Operating System errors are greater than 127.

Notes

The error number 0 is classed as a 'fatal' error and cannot be trapped by the ON ERROR statement. An example of a fatal error is that produced when a BASIC STOP statement is executed.

Examples

```
IF ERR=18 THEN PRINT "Can't use zero; try again!!"
IF ERR=17 THEN PRINT "Sure?"
IF GET$="Y" THEN STOP
```

ERROR ERR.

Part of the ON ERROR statement

Syntax

ON ERROR<statements> or
ON ERROR OFF

Notes

The ERROR reserved word only occurs in an ON ERROR statement which is described below.

EVAL EV.

Function causing its argument to be evaluated

Syntax

EVAL<factor>

Argument

A string which EVAL tries to evaluate as a BASIC expression.

Result

EVAL can return anything that could appear on the right hand side of an assignment statement. It can also produce the same errors that occur during assignment, eg Type mismatch and No such FN/PROC.

Examples

```
INPUT hex$
PRINT EVAL("&"+hex$)
f$="MID$(":e$=EVAL(f$+""""+a$+""",10)")
```

EXP

Function returning the exponential of its argument

Syntax

EXP<factor>

Argument

Numeric from the largest negative real (about −1E38) to approximately +88.

Result

Positive real in the range 0 to the largest positive real (about 1E38). The result could be expressed as e^(argument), where e is the constant 2.718281828.

Example

```
DEF FNcosh(x)=(EXP(x)+EXP(-x))/2
```

EXT#

Pseudo-variable controlling the length (extent) of an open file

Syntax

(1) EXT#<factor>
(2) EXT#<factor>=<expression>

(1)

Argument

Channel number, as allocated by one of the OPEN functions.

Result

Integer giving the current length of the file, from zero to, in theory 2147483648, though in practice the extent is limited by the file medium in use.

(2)

Argument

Channel number, as allocated by one of the OPEN functions.

The <expression> is the desired extent of the file, whose upper limit depends on the filing system. The lower limit is zero. The main use of the statement is to shorten a file, eg EXT#file=EXT#file-&1000. A file may be lengthened by using PTR#.

Examples

```
IF EXT#file>90000 THEN PRINT "File full":CLOSE#file
EXT#op=EXT#op+&2000
```

FALSE FA.

Function returning the logical value 'false'

Syntax

```
FALSE
```

Result

The constant zero. The function is used mnemonically in logical expressions.

Examples

```
flag=FALSE
REPEAT .... UNTIL FALSE
```

FN

Word introducing or calling a user-defined function

Syntax

(1) `DEF FN`<proc part>
(2) `FN`<proc part>

(1)

For the format of <proc part>, see `DEF` above. It gives the names and types of the parameters of the function, if any. For example:

```
1000 DEF FNmin(a%,b%) IF a%<b% THEN =a% ELSE =b%
```

`a%` and `b%` are the 'formal parameters'. They stand for the expressions passed to the function (the 'actual parameters') when `FNmin` is called. The result of a user-defined function is given by a statement starting with `=`. As the example above shows, there may be more than one `=` in a function; the first one encountered terminates the function.

User defined-functions may span several program lines, and contain all of the normal BASIC statements, eg `FOR` loops, `IF` statements etc. It is also possible to declare `LOCAL` variables which will preserve the value of a variable for the duration of the function.

(2)

<proc part> is an identifier followed by a list of expressions corresponding to the formal parameters in the DEF statement for the function. The result depends on the 'assignment' that terminated the function, and so can be of any type and range. An example function call is:

```
PRINT FNmin(2*bananas%, 3*apples%+1)
```

Examples

```
DEF FNfact(n%) IF n%<1 THEN =1 ELSE =n%*FNfact(n%-1)
DEF FNhex4(n%)=RIGHT$("000"+STR$(n%),4)
REPEAT PRINT FNhex4(GET): UNTIL FALSE
```

FOR F.

Part of the FOR... NEXT statement

Syntax

FOR <numeric variable>=<expression> TO <expression> [STEP <expression>]

Arguments

The <numeric variable> can be any numeric variable reference, eg I, len%, !&70. The <expressions> can be any numeric expressions, though they must lie in the integer range if the <numeric variable> is an integer one. It is recommended that integer looping variables are used as (a) the loops go faster and (b) rounding errors are avoided. If the STEP part is omitted, the step is taken to be +1.

Notes

The statements between a FOR and its corresponding NEXT are executed at least once, ie the test for loop termination is performed at the NEXT rather than the FOR. Thus a loop started with: FOR I=1 TO 0 ... will execute once with I set to 1 in the body of the loop. The value of the looping variable when the loop has finished should be treated as 'undefined', and shouldn't be used before being reset by an assignment.

Examples

```
FOR addr%=HIMEM TO &8000 STEP 4
FOR I=1 TO LEN(a$)
```

GCOL GC.

Statement to set the graphics colours and actions

Syntax

GCOL <expression1>,<expression2>

Arguments

<expression1> is the plot action in the range 0-255. It determines the effect of future PLOT commands on the screen. Currently defined values are:

Plot action	Meaning
0	Store the colour <expression2> on the screen
1	OR the colour <expression2> with the screen
2	AND the colour <expression2> with the screen
3	EOR the colour <expression2> with the screen
4	Invert the current colour, disregarding <expression2>
5	Don't affect the screen at all

<expression2> determines the colour that will combined with the screen for PLOT actions 0-5. It is in the range 0-127, and is treated MOD (the number of colours in the present mode). Adding 128 to the expression causes the background colour to be changed instead of the foreground.

If 16 is added to the values of <expression1> above, the first extended colour fill pattern is used instead of <expression2>. Adding 32 uses the second ECF pattern, adding 48 uses the third ECF pattern, and adding 64 causes the fourth ECF pattern to be used instead of <expression2>. VDU 23,2 to VDU 23,5 are used to set the ECF patterns. Not that when an ECF pattern is re-defined, a GCOL must be executed to tell the VDU drivers about it, otherwise the the old pattern will still be used. See the section on the VDU drivers for more information.

Examples

```
GCOL 4,128:CLG:REM Invert the graphics window
GCOL 1,2: REM  OR the screen with colour 2
```

GET

Function returning a character code from the input stream

Syntax

GET

Result

An integer between 0 and 255. This is the code of the next character in the

buffer of the currently selected input stream (keyboard or RS-423). The function will not return until a character is available, so can be used to halt the program temporarily.

Examples

```
PRINT "Press space to continue":REPEAT UNTIL GET=32
ON GET-127 GOSUB 1000,2000,3000 ELSE PRINT"Illegal entry"
```

GET$ GE.

Function returning a character from the input stream

Syntax

```
GET$
```

Result

A one-character string whose value is `CHR$(GET)`, if `GET` had been called instead. It is provided so you can use statements like `IF GET$="*"...` rather than `IF CHR$(GET)="*"...`

GOSUB GOS.

Statement to call a subroutine

Syntax

`GOSUB` <expression>

Argument

<expression> should evaluate to an integer between 0 and 32767, ie a line number. If the expression isn't a simple <integer> (eg 1030) then it should be between round brackets. The line given will be jumped to, and control will be returned to the statement after the `GOSUB` by the next `RETURN` statement.

Examples

```
GOSUB 2000
GOSUB (2300+20*opt)
```

Notes

The `RENUMBER` command will only work correctly if all `GOSUB`, `GOTO` and `RESTORE` line numbers are <integer>s. Line numbers that are expressions can't be renumbered, so the program will stop working correctly. Thus, the `ON ... GOSUB` construct is recommended over the `GOSUB` (<expression>) one.

GOTO G.

Statement to transfer control to another line

Syntax

GOTO <expression>

Argument

See GOSUB above, though control may not be returned to the next statement by executing a RETURN statement.

Examples

```
GOTO 230
IF TIME<1000 THEN GOTO 1000
```

HIMEM H.

Pseudo-variable holding address of the BASIC stack

Syntax

(1) HIMEM
(2) HIMEM=<expression>

(1)

Result

An integer giving the location of the BASIC stack, where local variables etc are stored. The stack grows down towards LOMEM, so the expression HIMEM – LOMEM gives an idea of how much free space is available for the program.

(2)

Argument

<expression> should be an integer between LOMEM and top of useable memory, as given by OSBYTE &84.

Examples

```
a%=HIMEM-&200 : HIMEM=a% : REM Reserve 2 pages
IF HIMEM<&8000 THEN PRINT "Compatible mode??"
```

Notes

If HIMEM is set carelessly, running the program may produce the 'No room' error. See the section on memory maps under BASIC for more guidance about the setting of HIMEM. The expression HIMEM–LOMEM gives the number of bytes available for variable storage and procedure return-address storage.

IF

Statement to conditionally execute statements

Syntax

`IF` <expression> [`THEN`] [<statements>] [`ELSE` [<statements>]]

Arguments

<expression> is treated as a truth value. If it is non-zero, it is counted as `TRUE` and any <statements> in the `THEN` part are executed. If the expression evaluates to zero, then the `ELSE` part <statements> are executed.

<statements> is either a list of zero or more statements separated by colons, or a line number. In the latter case, there is an implied `GOTO` after the `THEN`, which has to be present.

Examples

```
IF french THEN PROCkiss ELSE PROChandshake
IF temp<=10 PROClow_temp
IF a%>b% THEN t%=a%:a%=b%:b%=t% ELSE PRINT "No swap"
IF GET : REM  wait for a keypress
```

Notes

The `THEN` is optional before <statements> unless any of the statements is an assignment to a pseudo-variable, so `IF a THEN HIMEM=...` rather than `IF a HIMEM=...`. The `ELSE` part matches any `IF`, so be wary of nesting `IF`s on a line: constructs of the form

```
IF a THEN... IF b THEN... ELSE...
```

should be avoided. However, the form:

```
IF a THEN... ELSE IF b THEN...
```

can be used.

INKEY

Function returning a character code from the input stream or keyboard

Syntax

(1) `INKEY`<factor>
(2) `INKEY`<factor>
(3) `INKEY`<factor>

(1)

Argument

A positive integer in the range 0-32767, which is a time limit in centi-seconds.

Result

The ASCII code of the next character in the current input buffer, if one appears in the time limit set by the argument, or −1 when the 'timeout' occurs.

(2)

Argument

A negative integer in the range −255 to −1, which is the 'negative INKEY code' of the key being interrogated (see section D).

Result

TRUE if the key is being pressed at the time of the call, FALSE if it isn't.

(3)

Argument

−256.

Result

A number indicating what version of the operating system is in use. Possible values are:

INKEY(−256)	MOS type
−1	BBC MOS 1.00/1.20
0	BBC MOS 0.10
1	Acorn Electron MOS 1.00
250	Acorn ABC MOS
251	BBC MOS 2.00
252	German BBC
253	Current UK BBC (ie this machine)
254	BBC US MOS 1.00 or 1.1

Examples

```
DEF PROCwait(secs%) dummy=INKEY(100*secs%):ENDPROC
IF INKEY(-99) THEN REPEAT UNTIL NOT INKEY(-99)
```

INKEY$ INK.

Function returning a character from the input stream

Syntax

INKEY$<factor>

Argument

As INKEY

Result

Where INKEY would return −1, INKEY$ returns the null string. In other situations it returns CHR$(INKEY<argument>).

Example

```
REPEAT UNTIL INKEY$(500)=" " : REM wait for space key for five secon
ds
```

INPUT I.

Statement obtaining a value or values from the input stream

Syntax

The INPUT statement is too complex to summarise using the simple conventions adopted in this section. In words, INPUT is folowed by an optional prompt, which, if present, may be followed by a semi-colon or comma, which causes a ? to be printed out after the prompt. These are followed by a list of variable names of any type, separated by commas. After the last variable, the whole sequence may be repeated, separated from the first by a comma. In addition the position of prompts may be controlled by the SPC, TAB(and ' print formatters (see PRINT).

Examples

```
INPUT a$:REM  Print a simple "?" as a prompt
INPUT "How many ",num% : REM prompt is "How many ?"
INPUT "Address &"hex$  : REM prompt is "Address &" (no "?" as no ,)
INPUT TAB(10)"Name ",n$,TAB(10)"Address ",a$
INPUT a,b,c,d,'"More ",yn$
```

Notes

An additional modifier for INPUT is LINE. If this follows INPUT immediately, and the input variable is a string, all of the user's input is read into the variable, including leading and trailing spaces and commas. Usually leading spaces are skipped and commas mark the end of input for the current item. If the input variable is numeric, only a single value will be selected from the input line.

Example

```
INPUT LINE ">" basic$
```

INPUT# I.#

Statement obtaining a value or values from a file

Syntax

INPUT#<factor> [,<variable>] etc

Arguments

<factor> is the channel number of the file from which the information is to be read in the range 1-255. The list of zero or more <variable>s may be of any type. The separators may be semi-colons.

Examples

```
INPUT#data,name$,addr1$,addr2$,addr3$,age%
INPUT#data,$buffer,len
```

Notes

For the format of variables read using INPUT#, see Chapter K.3

INSTR(INS.

Function to find the position of a substring in a string

Syntax

INSTR(<expression1>,<expression2> [,<expression3>])

Arguments

<expression1> is any string which is to be searched for a substring. <expression2> is the substring required. <expression3> is a numeric in the range 0-255 and determines the position in the main string at which the search for the substring will start. This defaults to 1.

Result

An integer in the range 0-255. If zero is returned, the substring could not be found in the main string. A result of 1 means that the substring was found at the first character of the main string, and so on. The position of the first occurence only is returned.

Notes

If the substring is longer than the main string, zero will always be returned. If the substring is the null string, the result will always be equal to <expression3>, or 1 if this is omitted.

Examples

```
IF INSTR(any$,"") <> 1 PRINT "Bug in BASIC!!"
REPEAT a$=GET$:UNTIL INSTR("YyNn",a$) > 0
pos%=INSTR(com$,"*FX",10)
```

INT

Function giving the integer part of a number

Syntax

INT<factor>

Argument

Any integer-range numeric.

Result

Nearest integer less than or equal to the argument.

Examples

```
DEF FNround(n)=INT(n+0.5)
size=len%*INT((top-bottom)/100)
```

LEFT$(LE.

Function returning the left part of a string

Syntax

LEFT$(<expression1>,<expression2>)

Arguments

<expression1> is a string of between zero and 255 characters. <expression2> is a numeric in the range 0-255.

Result

A string taken from the leftmost <expression2> characters of <expression1>. If <expression2> is greater than LEN(<expression1>) then all of the string is returned.

Examples

```
left_bit$=LEFT$(input$,LEN(input$) DIV 2)
REM LEFT$(any$,256)=LEFT$(any$,0) as n is treated MOD 256
```

LEN

Function returning the length of a string

Syntax

LEN<factor>

Argument

Any string of zero to 255 characters.

Result

The number of characters in the argument string, from 0-255.

Examples

```
REPEAT INPUT a$: UNTIL LEN(a$)<=10
IF LENin$ > 12 THEN PRINT "Invalid filename"
```

LET

Statement assigning a value to a variable

Syntax

LET <variable>=<expression>

Arguments

The <variable> is any 'addressable object', eg `a`, `a$`, `a%`, `!a`, `a?10`, `$a` etc <expression> is any expression of the range and type allowed by the variable: for reals, any numeric; for integers, any integer-range numeric; for strings, any string of 0 to 255 characters and for bytes any integer in the range 0-255 (though an integer-range number will be treated `MOD 256`).

Notes

The `LET` keyword is always optional in an assignment, and must not be used in the assignment to a pseudo-variable, eg `LET TIME=100` is illegal.

Examples

```
LET starttime=TIME
LET a$=LEFT$(addr$,10)
LET table?i=127*SIN(RAD(i))
```

LINE

Modifier to the INPUT statement

Syntax

See INPUT

Example

```
INPUT LINE "Type message: "'a$
```

LIST

Command to list the program

Syntax

LIST [<line range>][IF <string>]

Arguments

<line range> gives the start and end lines to be listed. Both values are optional and should be separated by a comma. The first value defaults to zero and the last to 32767. The IF, when present, is followed by a string of ASCII characters. Only lines which contain this string will be listed.

Notes

Because the string after IF is tokenised, only one version of the pseudo-variables (each of which has two tokens) may be found. This is the one acting as a function (as in PRINT TIME), rather than the statement version (as in TIME=<expression>).

There is a small problem with LIST IF. Occasionally lines are listed that do not appear to match the IF part string. The reason is that BASIC line numbers contain a token &8D followed by three bytes of encoded line number. These three bytes will sometimes match the desired string, hence the spurious listing.

Examples

LIST	list the whole program
LIST 1000,	list from line 1000 to the end
LIST ,50	list from the start to line 50
LIST 10,40	list from line 10 to 40 inclusive
LIST IF DEF	list all lines containing a DEF
LIST ,100 IF fred%=	list all lines up to line 100 containing fred%=

LISTO L.O

Command to set the LIST indentation options

Syntax

`LISTO` <expression>

Argument

<expression> should be in the range 0 to 7. It is treated as a three-bit number, the meanings of the bits being:

Bit 0 : A space will be printed after the line number.
Bit 1 : `FOR` loops will be indented by two spaces.
Bit 2 : `REPEAT` loops will then be indented by two spaces.

Notes

BASIC strips trailing spaces from program lines at all times. If the current `LISTO` option is non-zero, it also strips leading spaces. To enter blank lines (eg '1000 '), either execute `LISTO Ø` first, or include a colon on the 'blank' line (eg '1000:').

Examples

```
LISTO Ø     no indentation
LISTO 7     all types of indentation
```

LN

Function returning the natural logarithm of its argument

Syntax

`LN`<factor>

Argument

Numeric in the range 0 to about 1E38, with the exception of zero itself.

Result

Real in the range −89 to +88, which is the log to base 'e' (2.718281828) of the argument.

Examples

```
DEF FNlog2(n)=LN(n)/LN(2)
PRINT LN(1Ø)
```

LOAD LO.

Command to load a BASIC program at PAGE

Syntax

LOAD <expression>

Argument

<expression> is a string which should evaluate to a filename that is valid for the filing system in use.

Examples

```
LOAD ":2.DEMO"
LOAD FNnextFile
```

LOCAL LOC.

Statement to declare a local variable in a procedure or function

Syntax

LOCAL [<variable>] [,<variable>] etc

Arguments

<variable>s that follow the LOCAL may be of any type, eg a, a%, a$, !&70, $buffer etc The statement causes the current value of the variables cited to be stored on BASIC's stack, for retrieval at the end of the procedure or function. This means the value inside the procedure may be altered without fear of corrupting a variable of the same name outside the procedure. At the end of the procedure, the old value of the variable is restored.

Notes

Local numerics are initialised to zero and local strings are initialised to the null string.

Examples

```
LOCAL dx,dy
LOCAL a$,len%,price
```

LOG

Function returning the logarithm to base ten of its argument

Syntax

LOG<factor>

Argument

Any numeric between 0 and approximately 1E38, excluding 0 itself.

Result

Real in the range −38 to +38, which is the log to base ten of the argument.

Examples

```
DEF FNdiff(n)=ABS(LOG(n)-LN(n)/LN(10))
PRINT LOG(2.4323)
```

LOMEM LOM.

Pseudo-variable holding the address of BASIC variables

Syntax

(1) LOMEM
(2) LOMEM=<expression>

(1)

Result

The address of the start of the BASIC variables. This is usually the same as TOP.

(2)

Arguments

<expression> is the address at which BASIC variables start. The expression should be in the range TOP to HIMEM to avoid corruption of the program and/or 'No room' errors. In any case the expression is taken as a two-byte integer.

Notes

LOMEM should not be changed after any assignments in a program. If it is, variables assigned before the change will be lost. LOMEM is reset to TOP by CLEAR (and thus by RUN).

Examples

```
LOMEM=TOP+&100 : REM  reserve a page above TOP
PRINT~LOMEM
```

MID$(M.

Function returning a substring of a string

Syntax

MID$(<expression1>, <expression2> [,<expression3>])

Arguments

<expression1> is a string of zero to 255 characters. <expression2> is the position within the string of the first character required. <expression3>, if present, gives the number of characters in the substring. The default value is 255 (or to the end of the source string).

Result

The substring of the source string, starting from the position specified and of a length given in the third argument. The result string can never be of greater length than the source string.

Examples

```
PRINT MID$("ABCDEFG",2,3);" should say BCD"
PRINT MID$(any$,LEN(any$)+1,any%);" gives a null string"
right_half$=MID$(any$,LEN(any$) DIV 2)
```

MOD

Operator giving the integer remainder of its operands

Syntax

<operand>MOD<operand>

Arguments

The <operand>s are integer-range numerics.

Result

Remainder when the left-hand operand is divided by the right-hand one using integer division.

Examples

```
INPUT i%: i%=i% MOD max_num%
count%=count% MOD max% + 1
PRINT result% MOD 100
```

MODE MO.

Statement changing the display mode

Syntax

MODE <expression>

Argument

<expression> should be in the range 0-255. This byte is passed to the operating system through the VDU drivers. HIMEM may also be changed. It is set to the highest free RAM address, as returned by OSBYTE &84. For the format of the various modes, see the section on the VDU drivers. It is not possible to change mode inside a user-defined function or procedure unless BASIC is running on a co-processor.

Examples

```
MODE 0
MODE m%+128
```

Note

For descriptions of the various modes, see the section on the VDU drivers.

MOVE

Statement to set the position of the graphics cursor

Syntax

MOVE <expression>,<expression>

Arguments

The <expression>s are x- and y-coordinates in the range −32768 to 32767, ie two-byte integers. The usual range for the x-coordinate is 0-1279, and for the y-coordinate it is 0-1023, though this changes if a graphics origin has been defined. MOVE is equivalent to PLOT 4.

Examples

```
MOVE 0,0 : REM Goto the origin
MOVE 4*x%,4*y% : REM Scale coordinates.
```

NEW

Command to remove the current program

Syntax

NEW

Purpose

The NEW command does not destroy the program, but merely sets a few pointers as if there is no program in the memory. The effect of NEW may be undone using the OLD command, providing no program lines have been typed in, or variables created, between the two commands. BASIC does an automatic NEW whenever it is entered.

NEXT N.

Part of the FOR .. TO .. NEXT structure

Syntax

NEXT [<variable>] [,[<variable>]] etc

Arguments

The <variable>s are of any numeric type, and if present should correspond to the variable used to open the loop.

Notes

The variables after the NEXT should always be specified while the program is in development, as this enables BASIC to detect improperly nested loops. If the loop variable given after a NEXT does not correspond to the innermost open loop, BASIC will close the inner loops until a matching looping variable is found.

Examples

```
NEXT a%
NEXT        : REM close one loop
NEXT j%,i% : REM close two loops
NEXT ,,,    : REM close four loops
```

NOT

Function returning the logical "NOT" of its argument

Syntax

NOT <factor>

Argument

An integer-range numeric.

Result

An integer in which all the bits of the argument have been inverted, ie 1s changed to 0s and 0s changed to 1s. If the argument is a truth value, NOT can be

used in logical statements to invert the condition. In this case, the truth value should only be one of the values −1 (TRUE) or 0 (FALSE)

Examples

```
IF NOT ok THEN PRINT "Error in input"
inv%=NOT mask%
REPEAT UNTIL NOT INKEY(-99)
```

OFF

Part of the ON ERROR and TRACE statements.

Syntax

(1) ON ERROR OFF
(2) TRACE OFF

(1)

Purpose

ON ERROR OFF disables error trapping so that when an error occurs the default error action of printing the error message (and line number) and terminating the program takes place.

(2)

Purpose

TRACE OFF turns off the tracing of the current program. This is done automatically when an error occurs. See TRACE.

OLD O.

Command to retrieve a NEWed program

Syntax

```
OLD
```

Purpose

The OLD command retrieves a program lost by NEW or BREAK providing no new program lines or variables have been entered.

ON

Part of the ON... GOTO/GOSUB/PROC and ON ERROR statements.

Syntax

(1) ON <expression0> GOTO <expression1> [,<expression n>] etc [ELSE <statement>]
(2) ON <expression0> GOSUB <expression1> [,<expression n>] etc [ELSE <statement>]
(3) ON<expression0><proc>[,<proc>] etc [ELSE<statement>]
(4) ON ERROR[<statements>]

(1) and (2)

Arguments

<expression0> following the ON is an integer between 1 and n, where n is the number of expressions following the GOTO/GOSUB. The <expression1> to <expression n> are line numbers (see GOSUB for the rules of line numbers). The optional ELSE part follows the last line number, and is followed by a statement.

Purpose

The ON... GOTO/GOSUB statement provides a multi-way branch facility. <expression0> is evaluated. If its value is m, then the mth line number following the GOSUB/GOTO is jumped to. The m is less than 1 or greater than n, the number of line numbers, the ELSE part is executed. If there is no ELSE part, an 'ON range' error is generated. Note that only a single statement may come after ELSE. Any other following statements, separated by colon, will be executed unconditionally. For example:

```
10 ON a GOSUB 100,200,300 ELSE P."error":PROCerrStuff
```

The call to PROCerrStuff will made whether the value of a is outside of the range 1-3 or not. The consequence of this is that the ELSE part of an ON statement usually contains a GOTO statement.

The difference between the GOTO and GOSUB versions is that in the latter case control will be returned back to the statement following the ON when a RETURN is executed.

(3)

Arguments

<expression0> following the ON is an integer between 1 and n, where n is the number of <proc> parts. The <proc> parts are normal calls to procedures, with or without parameters. The optional ELSE part follows the last line number, and is followed by a statement.

Purpose

The ON... PROC statement is very similar to ON... GOSUB, the difference being

that a call is made to a procedure instead of a subroutine. This provides greater flexibility (and speed – calling a procedure is faster than calling a subroutine), but is less 'standard' than `ON... GOSUB`. The note about `ELSE` made above applies here too.

(4)

Arguments

The <statements> following `ERROR` are zero or more legal BASIC statements separated by colons.

Purpose

When an `ON ERROR` is executed, BASIC remembers the position of the statements following the word `ERROR`. When an error is subsequently encountered, BASIC jumps to and executes these statements. Note that after an error, all loops, procedures etc are closed, so `ON ERROR NEXT` is not sensible.

Examples

```
ON choice% GOSUB 1000,2000,3000,4000 ELSE PRINT"Bad choice"
On ASC(c$) - 127 GOTO 100,120,10,200
ON addrMode PROCacc, PROCimp, PROCabs, PROCabx, PROCaby ELSE PROCothe
rMode
ON choice%+1 PROCload(prog$), PROCsave(prog$), PROCinsert(prog$) ELSE
PROCerr
ON ERROR GOTO 10000
ON ERROR PROCerr
ON ERROR IF ERR=17 THEN RUN ELSE REPORT:PRINT "at line ";ERL:END
ON ERROR OFF : REM  Disable the error trapping
```

OPENIN OP.

Function opening a file for input only

Syntax

`OPENIN`<factor>

Argument

A string which evaluates to a valid filename for the filing system in use.

Result

A single byte integer acting as a channel number for the file. The exact value depends upon the filing system and how many files are already open, but will always lie in the range 1-255, or zero if the file wasn't found.

Examples

```
in_file%=OPENIN("Invoices")
data%=OPENIN(":1."+data$)
```

OPENOUT OPENO.

Function opening a file for output only

Syntax

OPENOUT<factor>

Argument and result

See OPENIN above.

Examples

```
out_file%=OPENOUT("Customers")
data%=OPENOUT(":2."+data$)
```

OPENUP

Function opening a file for input and output

Syntax

OPENUP<factor>

Argument

See OPENIN above.

Example

```
random_file%=OPENUP("records")
```

OR

Operator giving the logical 'OR' of its operands

Syntax

<relational>OR<relational>

Arguments

<relational>s can be any integer-range numerics.

Result

An integer obtained by ORing together the corresponding bits in the operands. The operands may be interpreted as bit-patterns or logical values.

Examples

```
PRINT a% OR &AA55
IF a<1 OR a>10 THEN PRINT "Bad range"
```

OSCLI OS.

Statement to pass a string to the operating system

Syntax

OSCLI <expression>

Argument

<expression> should be a string of between 0 and 255 characters. It is passed to the operating system OSCLI routine.

Notes

The difference between passing a string to the OS via a star command and OSCLI is that the former make no attempt to process the text following it, whereas the latter interprets the text as a BASIC expression.

Examples

```
OSCLI "LOAD "+file$+" "+STR$buff% : REM get the file in buffer
OSCLI "FX138,0,"+STR$(ASC(c$))    : REM insert a character into
buffer
```

PAGE PA.

Pseudo-variable holding the address of the program

Syntax

(1) PAGE
(2) PAGE=<expression>

(1)

Result

An address which is a two-byte unsigned number. The lower byte is always 0 (ie PAGE is always on a &100 byte boundary). PAGE is the location at which the current BASIC program starts.

(2)

Argument

<expression> is an integer in the range OSHWM to HIMEM. The lower byte will be set to zero automatically. By changing PAGE, several BASIC programs may reside in the machine at once. PAGE defaults to OSHWM, as returned by OSBYTE &83

Example

```
IF PAGE<>&2200 THEN PAGE=&2200: CHAIN"LOADER"
```

PI

Function returning the value of pi

Syntax

```
PI
```

Result

The constant 3.141592653

Examples

```
DEF FNarea(r)=2*PI*r
```

PLOT PL.

Statement peforming an an operating system PLOT function

Syntax

PLOT<expression1>,<expression2>,<expression3>

Arguments

<expression1> is the plot number in the range 0 to 255, eg 4 for MOVE and 85 for an absolute triangle plot in the current foreground colour. See the section on the VDU drivers for a full description of the plot numbers. The second and third expressions are the x- and y-coordinates respectively, in the range −32768 to +32767.

Examples

```
PLOT 85,100,100 : REM Draw a triangle
PLOT 69,x,y : REM Plot a single point
```

POINT(PO.

Finds the logical colour of a graphics pixel

Syntax

```
POINT(<expression1>,<expression2>)
```

Arguments

<expression1> is the x co-ordinate of the pixel and <expression2> is the y co-ordinate. These are integers in the range −32768 to +32767.

Result

This is an integer in the range −1 to n, where n is 1 less than the number of logical colours in the current mode. For example, n is 15 in the 16 colour `MODE 2`. If −1 is returned, the point specified lies outside the current graphics window, otherwise it is the logical colour of the point. For an explanation of the co-ordinate system and the logical colours, see the section on the VDU drivers.

Example

```
REPEAT Y%=Y%+4:UNTIL POINT(640,Y%)<>0
```

POS

Function returning the x-coordinate of the text cursor

Syntax

```
POS
```

Result

An integer between 0 and n, where n is the width of the current text window minus 1. This is the position of the text cursor relative to the left edge of the text window.

Examples

```
old_x%=POS
IF POS<>0 THEN PRINT
```

PRINT P.

Print information on the output stream(s)

Syntax

The items following `PRINT` may be string expressions, numeric expressions, and

the print formatters, ; SPC TAB(' and <space>. By default, numerics are printed in decimal, right justified in the print field given by @% (see below). Strings are printed left justified in the print field. The print separators have the following effects when printing numbers:

;

Don't right justify (print leading spaces before) numbers in the print field. Set numeric printing to decimal. Semi-colon stays in effect until a comma is encountered. Don't print a new line at the end if this is the last character of the PRINT statement.

,

Right justify numbers in the print field. Set numeric printing to decimal. This is the default print mode. Comma stays in effect until a semi-colon is encountered. If the cursor isn't at the start of the print field, print spaces to reach the next one.

~

Print numbers as hexadecimal integers, using the current right justify mode. Tilde stays in effect until a comma or semi-colon is encountered.

'

Print a new line. Retain current right justify and hexadecimal/decimal modes.

TAB(

If there is one argument, eg TAB(n), print (n−COUNT) spaces. If the cursor is initially past position n (ie COUNT > n), print a newline first. If there are two arguments, eg TAB(10,20), move directly to that tab position. Right justify and hexadecimal/decimal modes are retained.

SPC(

Print the given number of spaces, eg SPC(5) will output five spaces. Right justify and hex/decimal modes are retained.

<space>

Print the next item, retaining right justify and hex/decimal modes.

When strings are printed, the descriptions above apply except that hexadecimal mode does not affect strings. Also, no trailing spaces are printed after a string unless it is followed by a comma. This prints enough spaces to move to the start of the next print field.

The format in which numbers are printed, and the width of print fields are determined by the value of the special system integer variable, @%. Each byte in the variable has a special meaning. These are:

Byte 4.

This determines whether the STR$ function will use the print format determined by @% when converting its argument to a string, or whether it will

use a default 'general' format. If the byte is zero (the default), STR$ will use a general format; if it is non-zero, STR$ will use the format determined by @%.

Byte 3.
This selects the format to be used. The legal values are:

0 – General format. Numbers have the form nnn.nnn, the maximum number of digits printed being given in byte 2. This is the default format.

1 – Exponent format. Numbers have the form n.nnnEnn, then number of digits printed being given in byte 2

2 – Fixed format. Numbers have the form nnn.nnn, the number of digits after the decimal point being given in byte 2.

Byte 2.
This determines the number of digits printed. In General format, this is the number of digits which may be printed before reverting to Exponent format (1-10); in Exponent format it gives the number of significant to be printed after the decimal point (1-10), and in fixed format it gives the number of digits (exactly) that follow the decimal point.

Byte 1.
This gives the print field width for tabulating using commas, and is in the range 0-255.

Examples of @%

@%=&0102020A will use Fixed format with two decimal places in a tab field width of ten. In addition, STR$ will use this format instead of its default (which is &0A0A). Numbers will be printed out in the form 1.23, 923.10 etc

@%=&00010408 will use Exponent format. Four significant digits will be printed, in a field of eight characters. Thus numbers will look like 1.234E0, 1.100E−3, −9.980E10 etc

@%=&0000090A will use General format with up to nine significant digits in a field width of ten characters. Note that General format reverts to Exponent format when the number is less than 0.1. This is the default setting of @%.

Notes

Setting byte 2 to 10, eg &0A0A will show the innaccuracies which arise when trying to store certain numbers in binary. For example:

```
PRINT 7.7
```

will print 7.666666669 when @%=&0A0A.

The print formatters ', TAB(and SPC may also be used in INPUT statements.

Examples

```
PRINT "How many eggs ";
PRINT a,SIN(RAD(a)),x,y''p,q
PRINT TAB(10,3)"Profits "SPC(10);profits;
```

PRINT# P.#

Print information to an open file

Syntax

PRINT#<factor> [,<expression>] etc

Arguments

The <factor> is the channel number of a file opened for output or update. The expressions, if present, are any BASIC integer, real or string expressions. They are evaluated and sent to the file specified. For the internal format of expressions sent by PRINT#, see section K.3

Example

```
PRINT#file, name$+":",INT(100*price+.5),qnty%
```

PROC

Word introducing or calling a user-defined procedure

Syntax

(1) DEF PROC<proc part>
(2) PROC<proc part>

(1)

<proc part> has the form <identifier>[(<parameter list>)]. It gives the name of the procedure (the <identifier>) and the names and types of the optional parameters, which must be enclosed in brackets and separated by commas.

(2)

The second form is used when the procedure is actually invoked, and this time the parameter list comprises expressions of types corresponding to the parameters declared in the DEF PROC statement. The expressions are evaluated and assigned (locally) to the parameter variables. Control returns to the calling program when an ENDPROC is executed.

Examples

```
DEF PROCdelay(n) TIME=0:REPEAT UNTIL TIME=n*100:ENDPROC
IF ?flag=0 THEN REPEAT PROCdelay(0.1): UNTIL ?flag
```

PTR#

Pseudo variable accessing the pointer of a file

Syntax

(1) PTR#<factor>
(2) PTR#<factor>=<expression>

(1)

Argument

<factor> is a one-byte file channel number in the range 1-255, as returned from an OPEN function.

Result

An integer giving the position relative to the start of the file of the next byte to be read or written. The minimum value is 0 and the maximum value depends on the filing sytem in use.

(2)

Argument

<factor> is as (1). The <expression> is an integer giving the desired position of the sequential pointer in the file.

Note

PTR# is not valid when the cassette filing system is in use.

Examples

```
PRINT PTR#file;" bytes processed"
PTR#chan%=rec%*rec_len%
```

RAD

Function returning the radian value of its argument

Syntax

RAD<factor>

Argument

A number representing an angle in degrees.

Result

A real giving the corresponding value in radians, ie <argument>*PI/180.

Examples

```
sin%?i%=SIN(RAD(i%))
PRINT RAD(90)-PI/2;" isn't quite zero!!"
```

READ

Statement reading information from a DATA statement

Syntax

READ [<variable>] [,<variable>] etc

Arguments

The zero or more variables should correspond in type to the items in the DATA statement being read. In fact, a string READ item will be able to read any type of DATA and interprets it as a string constant after stripping leading spaces. A numeric READ item tries to evaluate its DATA, so in the latter case the DATA expression should yield a suitable number.

Examples

```
READ a$,fred%
READ !&70, $addr%
```

REM

Statement indicating a remark

Syntax

REM<string>

Argument

<string> can be absolutely anything; it is ignored by BASIC. The purpose of REMs is to annotate the program in English.

Examples

```
REM  find the next prime
REMARKABLE COMMENT, THIS
```

RENUMBER REN.

Command to resequence the program line numbers

Syntax

```
RENUMBER [<integer>] [,<step>]
```

Arguments

See AUTO for a description.

Purpose

RENUMBER resequences the lines in the program so that the first line is <integer> and the line numbers increase in steps of <step>. It also changes line numbers within the program, eg after GOTOs, so that they match the new line numbers. If a GOTO line cannot be found, the message 'Failed at ....' is given. RENUMBER needs some workspace, and if there is not enough room to successfully change the line numbers, a RENUMBER space error is generated.

Examples

```
RENUMBER
RENUMBER 1000,20
```

REPEAT REP.

Statement marking start of a REPEAT ... UNTIL loop

Syntax

```
REPEAT
```

Purpose

The statements following REPEAT will be repeatedly executed until the condition following the matching UNTIL evaluates to FALSE. The statements may occur over several program lines, or may all be on the same line separated by colons. The second approach is useful in 'immediate' statements.

Examples

```
REPEAT UNTIL NOT INKEY-99 : REM wait for SPACE to be released
REPEAT a%=a%+1:c%=c% DIV 2:UNTIL c%=0
```

REPORT REPO.

Statement printing the message of the last error encountered

Syntax

```
REPORT
```

Examples

```
REPORT:PRINT "at line ";ERL:END
REPORT:PRINT " error!!""':END
```

RESTORE RES.

Statement setting the DATA pointer

Syntax

```
RESTORE [<expression>]
```

Argument

<expression> is a line number. If it is absent, the DATA pointer is reset to the first DATA statement in the program, and the next item READ will come from there. If the line number is present, the DATA pointer will be set to the first item on the line specified, so that subsequent READs access that line (and those which follow).

Note

See the comments on RENUMBERing line numbers in the GOSUB arguments section.

Examples

```
RESTORE
RESTORE 1000
```

RETURN R.

Statement returning control from a subroutine

Syntax

```
RETURN
```

Purpose

RETURN returns control to the statement following the most recent GOSUB. If there are no GOSUBs currently active, a No GOSUB error occurs.

RIGHT$(RI.

Function returning the rightmost characters of a string

Syntax

```
RIGHT$(<expression1>,<expression2>)
```

Arguments

<expression1> should be a string of zero to 255 characters. <expression2> should be a numeric, n, in the range 0-255.

Result

A string consisting of the rightmost n characters from the source string (<expression1>). If n is greater than the length of the source string, the whole source string is returned.

Examples

```
PRINT RIGHT$(any$,4)
year$=RIGHT$(date$,2)
```

RND

Function returning a random number

Syntax

(1) `RND`
(2) `RND(`<expression>`)`

(1)

Result

A four-byte signed random integer between −2147483648 and +2147483647.

(2)

Result

<expression> < 0
This 'reseeds' the random number generator, and the function returns its argument as a result. Reseeding the generator with a given seed value will always produce the same sequence of random numbers.

<expression> = 0
This returns the same number as the last `RND(1)`

<expression> = 1
This returns a random real number between 0 and .999999999

<expression> > 1
The expression, n, should be an integer. The result is an integer between 1 and n inclusive.

Examples

```
dummy=RND(-TIME) : REM reseed the generator 'randomly'
ON RND(4) GOSUB 1000,1200,1400,1600
x%=RND(1280) : y%=RND(1024)
```

RUN

Statement to execute the current program

Syntax

```
RUN
```

Purpose

RUN executes the program in memory, if one is present, after clearing all variables and resetting LOMEM.

SAVE SA.

Command to save a program in a file

Syntax

SAVE<expression>

Argument

<expression> should evaluate to a string which is a valid filename under the filing system in use. The current BASIC program will be stored (without variables etc) on the medium under this name.

Examples

```
SAVE "Version1"
SAVE FNprogName
```

SGN

Function returning the sign of its argument

Syntax

SGN<factor>

Argument

Any numeric.

Result

−1 for negative arguments, 0 for zero-valued arguments and +1 for positive arguments.

Examples

```
DEF FNsquare(th)=SGN(SIN(th))
ON SGN(arg)+2 GOSUB 1000,2000,3000
```

SIN

Function returning the sine of its argument

Syntax

SIN<factor>

Argument

A numeric representing an angle in radians, and is scaled down to between −PI and PI radians.

Result

A real in the range −1 to 1, being the sine of the argument.

Notes

If the argument is outside the range −8388608 to +8388607 radians, an Accuracy lost error will be given.

Examples

```
PRINT SIN(RAD(135))
opp=hyp*SIN(theta)
```

SOUND SO.

Statement generating a sound

Syntax

SOUND <expression1>,<expression2>,<expression3>,<expression4>

Arguments

<expression1> is the channel number
<expression2> is the amplitude
<expression3> is the pitch
<expression4> is the duration
as described below.

Channel number

A two-byte integer giving the channel number to be used. Simply regarded, it has the range 0 to 3:

0 Noise channel
1 Note channel 1
2 Note channel 2
3 Note channel 3

C can also be regarded as a four-digit hex number, which can be written as &HSFC. The meanings of these four parts are:

C – *The sound channel.*
In the range 0-3, and mentioned above.

F – *The 'flush' part.*
It can have the value zero or one. If it is zero, the note is added to the sound queue for the appropriate channel as normal. If it is 1, then the sound queue for channel C is 'flushed' before sounding the note. This means that any note sounding on channel C, and any waiting in the queue are removed, and the sound generated with the 'flush' bit is sounded immediately. Thus the flush part provides a way of over-riding the sound queue.

S – *The synchronise part.*
This can be in the range 0 to 2. A value of zero means start playing the note specified as soon as the previous note in the queue has been playing for the time specified in its duration part (see below). This is the normal state. However, setting S to one indicates that we do not want this note to play until there is another note ready (on another channel's queue) with its synchronise part set to one. When such a note becomes available, both notes will be sounded together, hence the term 'synchronise'. Similarly, if the synchronise part has a value of 2, the note will not be sounded until two other notes, also with synchronisation values of two, are present on other queues. The S part is obviously useful for playing two- and three-note chords.

H – *This part can be set to 0 or 1.*
A zero value lets the note start playing as soon as the current note on the queue has been playing for the time specified in its duration part. However, if this note is under the control of an ENVELOPE it may have a gentle release part which would be abrubtly cut-off by the new note. Setting the H bit causes the new note to be queued in the normal way, but not sounded, ie the pitch and amplitude parts are ignored. In addition, it won't truncate the release part of a note already playing.

Examples of the channel specification

C = &0101 Use channel one, and wait for one more note before sounding
C = &1002 Use channel two, and don't sound if there's already a note playing on that channel
C = &0013 Use channel three, and remove any other sounds in that queue.

A – Amplitude

This is an integer between −15 and +15. The range −15 to 0 is a simple volume (amplitude), −15 being the loudest and 0 being the quietest (ie no sound). Positive values of A are treated as an envelope number. Envelopes 5-15 are only available if cassette `OPENOUT` files are not used. For a description of sounds under `ENVELOPE`, see that section.

P – Pitch

This is treated as an integer in the range 0-255. The note A above middle C has a pitch value of 89 and differences in P of 48 correspond to differences in pitch of one octave, ie there are 4 pitch values per semi-tone. If an envelope is in use, P specifies the initial pitch.

When the noise channel (zero) is used, the P parameter controls the type of noise produced. It has the following possible values:

P – Effect

0	Produces high frequency periodic noise
1	Produces medium frequency periodic noise
2	Produces low frequency periodic noise
3	Produces periodic noise whose frequency is determined by P on channel 1
4	Produces high frequency 'white' noise
5	Produces medium frequency 'white' noise
6	Produces low frequency 'white' noise
7	Produces 'white' noise whose frequency is determined by P on channel 1

D – Duration

The last `SOUND` parameter is also treated as a one-byte integer. It gives the duration of the note in twentieths of a second. A value of 255 gives a note with an 'infinite' duration, ie one that doesn't stop unless the sound queue is flushed in some way. When an envelope is in use, D gives the time for which the note sounds before the release phase is entered.

Examples

```
SOUND 1,-15,255,10
SOUND &102,-15,100,200
```

SPC

Print modifier to generate spaces

Syntax

SPC<factor>

Argument

A one-byte integer between 0 and 255. It gives the number of spaces to printed.

Examples

```
PRINT a$;SPC(10);b$
INPUT SPC(7)"How many ",a$
```

SQR

Function returning the square-root of its argument

Syntax

SQR<factor>

Argument

Any positive numeric.

Result

A real which is the argument's square-root.

Examples

```
DEF FNlen(x1,y1, x2,y2)=SQR((x2-x1)^2+(y2-y1)^2)
disc=SQR(b*b-4*a*c)
```

STEP

Part of the FOR... TO... STEP statement

Syntax

FOR... TO... [STEP <expression>]

Argument

<expression> is any numeric (preferably an integer) giving the step by which the FOR variable is to be incremented when a NEXT is encountered. If the STEP part is omitted, it is assumed to be 1.

Examples

```
FOR i%=32 TO 128 STEP 32
FOR addr%=HIMEM TO HIMEM-&2000 STEP -4
```

STOP

Statement producing the fatal error STOP to terminate the program

Syntax

```
STOP
```

Purpose

The STOP statement causes the fatal (untrappable) error message STOP to be produced. It differs from END, as the latter produces no message.

Example

```
IF NOT ok THEN PRINT"Bad data":STOP
```

STR$

Function producing the string representation of its argument

Syntax

STR$[~]<factor>

Argument

Any numeric for decimal conversion, any integer for hexadecimal conversion. Decimal conversion is used when the tilde (~) is absent, hex conversion is used when it is present.

Result

Decimal or hex string representation of the argument, depending upon the absence or presence of the tilde.

Notes

The string returned by STR$ is usually formatted in the same way as the argument would be printed with @% set to &A0A. However, if the most significant byte of @% is non-zero, STR$ will return the result in exactly the

same format as it would be printed, taking the current value of @% into account. See also PRINT.

Examples

```
DEF FNhex4(a%)=RIGHT$("000"+STR$~(a%),4)
DEF FNdigits(a%)=LEN(STR$(a$))
dp=INSTR(STR$(any_val),".")
```

STRING$(STRI.

Function returning multiple copies of a string

Syntax

STRING$(<expression1>,<expression2>)

Arguments

<expression1> is an integer, n, in the range 0 to 255. <expression2> should be a string of length 0 to (255 DIV n).

Result

A string comprising n concatenated copies of the source string, of length 0 to 255.

Examples

```
PRINT STRING$(40,"_"); :REM underline across the screen
pattern$=STRING$(20,<-->")
```

TAB(

Print modifier to position text cursor

Syntax

(1) TAB(<expression>)
(2) TAB(<expression1>,<expression>)

(1)

Argument

A numeric in the range 0-255. It expresses the desired x-coordinate of the cursor. This position is obtained by printing spaces. A newline is generated first if the current position is at or to the right of the required one.

(2)

Arguments

<expression1> is the desired x-coordinate; <expression2> is the desired y-coordinate. The position is reached by using the VDU 31 command. Both coordinates must lie within the current text window, otherwise no cursor movement will take place.

Examples

```
PRINT TAB(10);"Product";TAB(20);"Price"
INPUT TAB(0,10)"How many eggs ",eggs%
```

TAN T.

Function giving the tangent of its argument

Syntax

TAN<factor>

Argument

A real number and is interpreted as an angle in radians. It is reduced to be in the range –PI/2 to +PI/2.

Result

A real giving the tangent of the angle, in the range –1E38 to +1E38

Notes

If the argument is outside the range –8388608 to +8388607 radians, an Accuracy lost error will be given.

Example

```
opp=adj*TAN(RAD(theta))
```

THEN TH.

Part of the IF... THEN... ELSE statement

Syntax

See IF

Notes

THEN is optional except in the case where a pseudo-variable is being assigned. It must also be included in statements of the form IF <expression> THEN <expression>.

Example

```
IF a>3 THEN PRINT "Too large"
```

TIME TI.

Pseudo-variable accessing the value of the centi-second clock

Syntax

(1) TIME
(2) TIME=<expression>

(1)

Result

A four-byte integer giving the number of centi-seconds that have elapsed since the last time the clock was set to zero.

(2)

Arguments

<expression> is used to set the lower four-bytes of the five-byte system clock, and should be in the integer range. The most significant byte is set to zero automatically.

Example

```
DEF PROCdelay(n) TIME=0:REPEAT UNTIL TIME>=n*100
```

TIME$

Pseudo-variable accessing the real-time clock.

Syntax

(1) TIME$
(2) TIME$=<expression>

(1)

Result

TIME$ returns a 24 character string of the format:

'Fri,24 May 1984.17:40:59'. The date and time part are separated by a period (.).

(2)

The expression should be a string specifying the date, the time, or both.

Punctuation and spacing are crucial and should be as shown in the examples below.

Examples

```
PRINT TIME$
TIME$="Tue, 1 Jan 1972"
TIME$="21:12:00"
TIME$="Tue, 1 Jan 1972.21:12:00"
```

TO

Part of the FOR... TO... STEP statement

Syntax

See FOR

Example

```
FOR i%=1 TO 100
```

TOP

Function returning the address of the end of the program

Syntax

```
TOP
```

Result

TOP gives the address of the first byte after the BASIC program. The length of the program is TOP – PAGE. HIMEM is usually set to TOP, so this is where variables start.

Example

```
PRINT "Load at "; ~TOP-2
```

TRACE TR.

Statement to initiate statement tracing

Syntax

TRACE <expression> or TRACE ON or TRACE OFF

Argument

<expression> is a line number. All line numbers below this will be printed out when they are encountered during the execution of the program.

TRACE ON is the same as TRACE 32767, ie all line numbers are printed as they are met.

TRACE OFF disables tracing.

Examples

```
IF debug THEN TRACE 9000
IF debug THEN TRACE OFF
```

TRUE

Function returning the constant −1

Syntax

```
TRUE
```

Result

TRUE always returns −1, which is the number yielded by the relational operators when they succeed, eg 1+1 < 3 gives TRUE as its result.

Examples

```
debug=TRUE
REM IF anyvar=TRUE ... should be written IF anyvar ...
```

UNTIL U.

Statement to terminate a REPEAT loop

Syntax

UNTIL <expression>

Argument

<expression> can be any numeric or string condition which can be evaluated to give a truth value. If it is zero (FALSE), control will pass back to the statement immediately after the corresponding REPEAT. If the expression is non-zero, control continues to the statement after the UNTIL.

Examples

```
DEF PROCirritate REPEAT VDU 7:UNTIL FALSE:ENDPROC
REPEAT PROCmove:UNTIL gameOver
REPEAT GOSUB 1000:UNTIL TIME>2000
```

USR

Function returning 65C12 registers after executing a routine

Syntax

USR<factor>

Argument

A two-byte integer, interpreted as the address of the machine code to be called.

Result

A four-byte integer of the form &PYXA, where P is the 65C12 processor status, Y is the Y register, X is the X register and A is the A register (accumulator) on exiting the routine. Parameters may be passed to the registers through the A%, X%, Y% and C% system integer variables, as described under CALL.

Examples

```
DEF FNoshwm A%=&83: =(USR(&FFF4) AND &FFFF00) DIV &100
DEF FNromByte(!&F6,Y%)=USR&FFB9 AND &FF
```

VAL

Function returning the numeric value of a decimal string

Syntax

VAL<factor>

Argument

A string of zero to 255 characters.

Result

The number that would have been read if the string had been typed in response to a numeric INPUT statement. The string is interpreted up to the first character that is not a legal numeric one (0-9, E and .).

Example

```
date=VAL(date$)
```

VDU

V.

Statement sending bytes to the VDU drivers

Syntax

VDU [<expression>] [, or ; or ¦ <expression>] etc [; or ¦]

Arguments

The zero or more <expression>s may be followed by a comma, a semi-colon, a vertical bar, or nothing in the case of the <expression> at the end of the line. Expressions followed by a semi-colon are sent as two bytes (low byte first) to the operating system VDU drivers (ie the OSWRCH routine). Expressions followed by a comma (or nothing in the case of the last expression) are sent to the VDU drivers as one byte, taken from the least significant byte of the expression. The vertical bar means ',0,0,0,0,0,0,0,0,0,' so sends the <expression> before it as a byte followed by 9 zero bytes. Thus the cursor may be turned off with `VDU 23,1|`.

Notes

For the meanings of the VDU codes, see the section on the VDU drivers

Examples

```
VDU 23,224,&AA55;&AA55;&AA55;&AA55; : REM cross-hatch pattern
VDU 19,0,8| : REM Flashing background
```

VPOS VP.

Function returning the Y coordinate of the text cursor

Syntax

```
VPOS
```

Result

The vertical position of the text cursor relative to the top of the text window, in the range 0-255.

Examples

```
DEF FNmyTab(x%) PRINT TAB(x%,VPOS);: =""
IF VPOS>10 THEN PRINT TAB(0,10);
```

WIDTH W.

Statement seting the line width in BASIC

Syntax

`WIDTH`<expression>

Arguments

<expression> should be an integer between zero and 255. Expressions in the range 1-255 will cause BASIC to print a newline and reset COUNT to zero every time COUNT exceeds that number. If the expression is zero-valued, BASIC will stop generating auto-newlines, which is the default.

Examples

```
WIDTH 0:REM  'infinite width'
WIDTH 40: REM newline every 40 characters
```

M BASIC Error messages

M.1 Errors encountered in BASIC

When an error is encountered during the execution of a BASIC program, the program stops and all loops, subroutines and procedures that were active are 'forgotten'. A message of the form:

```
Syntax error at line 1234
```

is printed, and the BASIC prompt > re-appears. If an error is encountered during an immediate statement (ie one typed without a line number), just the error message is printed, with no `at line` part, since the error is not in a line.

Most errors may be 'trapped' – stopped from terminating the program. The statement:

`ON ERROR` <statements>

causes execution to be transferred to the <statements> after the `ON ERROR` when an error is encountered. Loops etc are still exited automatically. After the <statements> have been executed, the program continues at the line after the `ON ERROR`.

An example of an `ON ERROR` statement is:

```
100 ON ERROR PROCerror
```

`PROCerror` might be defined as follows:

```
2000 DEF PROCerror
2010  IF ERR=17 THEN REPORT:PRINT:END
2020  IF ERL<>1240 THEN REPORT : PRINT " at line ";ERL
2030  PRINT "Bad expression, please try again":ENDPROC
2040 ENDPROC
```

This procedure performs the actions required for various classes of error. It illustrates the use of three other keywords that are relevant to error handling: `ERR`, `ERL` and `REPORT`.

`ERR` is a function that returns the number of the last error encountered. These numbers provide an easy way of uniquely indentifying errors. This is used at line 2010 to check for an escape condition (see below).

`ERL` returns the line number of the last error encountered, or 0 for errors in an immediate statement.

REPORT is a statement that prints a newline, followed by the error message of the latest error. This is used in line 2020 to print a standard form of error message.

Some errors have an error number of 0. These are called 'fatal' errors and cannot be trapped by an ON ERROR statement. Examples of such errors are the STOP statement and No room error.

Error trapping may be turned off with the ON ERROR OFF statement. After this, BASIC handles errors as normal.

Note that error trapping does not just work for errors with BASIC statements – you can trap errors in the execution of other commands such as *LOAD, too.

Error messages produced by BBC BASIC

Below are explanations of the error messages that BASIC can give, in alphabetical order. The error numbers of these messages lie in the range 0-45. Errors generated by the MOS, CFS and RFS lie in the range 128-255 and are discussed separately.

Accuracy lost **(Error 23)**

The trigonometric functions SIN, COS and TAN require angles in the range −PI to +PI radians. If the argument is outside this range, BASIC will scale it down without error. However, only angles in the range −8388608 to +8388608 can be scaled correctly: arguments outside this range give Accuracy lost errors.

Arguments **(Error 31)**

This error is given if the number of parameters specified in a call to a user-defined function or procedure is different to the number given in the definition. For example:

```
100 PROC fred(1)
110 ....

  ........
  ........

1000 DEF PROCfred(a,b)
```

would give an Arguments error at line 100.

Array **(Error 14)**

This is usually caused when an array which hasn't been declared is accessed, or when a function has been misspelt:

```
PRINT a$(1) : REM a$ hasn't been DIMed
b$=RGHT$(a$,10) : REM RIGHT$ mis-spelt
```

In BASIC you must declare all arrays using DIM before referencing them. BASIC treats any identifier followed by a bracket as an array reference, which is why the second example would give an Array error.

Bad call **(Error 30)**

When PROC or FN is used without being followed by a valid identifier, a Bad call error is given. A valid FN/PROC name is any string of one or more of a-z, A-Z, 0-9, _, £ or @. So PROC12 and PROC@ff are valid, but PROC$ and PROC?Q are not.

Bad DIM **(Error 10)**

This error results when a DIM statement attempts to dimension an array to either a negative number of elements, or greater than 16383 elements. The error can also occur when trying to dimension an array which has already been declared, or when dimensioning a byte array of fewer than zero bytes (ie DIM a% –2 or less).

Bad hex **(Error 28)**

The only valid characters that may follow an & in a hexadecimal constant are 0-9 and A-F. An attempt to use any other character (eg &a) will produce a Bad hex error.

Bad MODE **(Error 25)**

This error message will result if you attempt to change display mode using MODE to a mode in which screen memory would overlap the user's program or variables. It will also result if you use MODE within a user-defined function or procedure, except in a co-processor. If you wish to change modes in a procedure or function, as long as it is to a shadow mode (128-135), you can do so using VDU 22.

Bad program **(No error number)**

This is not a normal type of error, which can be printed using REPORT, but one that BASIC displays when it detects that the program in memory is corrupt. This can happen as a result of changing PAGE to a location where there is no program, overwriting a program with screen memory by changing modes with VDU 22, or calling a machine code program that 'writes over' the BASIC program area. The best way to get rid of Bad program error is to execute a NEW and reload the program. Note that you may be able to run a program which is 'Bad', although this is neither recommended nor guaranteed.

Byte **(Error 2)**

This is an error produced by the 65C12 assembler when it encounters a number

greater than 255 (&FF) used in a situation where only a single byte is valid. Examples are:

```
100 LDA #&200
230 STA (table,X)
```

In the second example it is assumed that table has been assigned a value of greater than &FF. 'Byte' can also occur in the first pass of an assembly when an undefined label is used, even though that label may later be set to a valid (single byte) value. This is because when an operand expression cannot be evaluated during the first pass, the value of the program counter (P%) is substituted. The example below will give a Byte error:

```
1000 P%=&2000
1010 [ OPT 0
1020   LDA #val
1030 ]
1040 val=&34
```

as the unknown value of val will be taken as &2000.

Can't match FOR **(Error 33)**

This error is produced when a NEXT <variable> statement is encountered for which there is no corresponding FOR <variable> ... statement.

DIM space **(Error 11)**

Integer array elements occupy 4 bytes, real elements take 5 bytes, and string elements need 4+(length of string) bytes. If the subscripts in a DIM are such that the space needed for the array exceeds the memory available, this error is given.

Division by zero **(Error 18)**

It is illegal, of course, to divide by zero. This error is encountered when the right hand side of a /, DIV or MOD operator is zero. Remember that LOCAL numeric variables and numeric array elements are initialised to zero, so this error will be generated if you forget to assign a value to them (rather than No such variable).

$ range **(Error 8)**

Indirect strings may be addressed anywhere in memory other than zero page (addresses &0000-&00FF). An attempt to access an indirect string in zero page produces this message.

Escape **(Error 17)**

This message is produced by BASIC whenever the [ESCAPE] key is pressed, as long as [ESCAPE] is enabled. If a BASIC program calls a machine code routine which executes for a long time, the routine should also check for [ESCAPE].

Exp range **(Error 24)**

If the argument to EXP would produce a result larger than the biggest real number (about 1E38), this error is produced. The upper limit on EXP's argument is about 88.

Failed at <line> **(No error number)**

This error is produced by a RENUMBER command. RENUMBER tries to change all GOTO, GOSUB, ON and RESTORE statements so that the line numbers they use correspond to the changed line numbers in the renumbered version of the program. If a line number is referenced which doesn't exist, then RENUMBER can't find it and produces a 'Failed at <line>' message. <line> is the number of the line containing the line number reference, which is left unaltered. Note that the RENUMBER command does not stop when it encounters this error, so all erroneous line are reported, with the rest of the program renumbered correctly.

FOR variable **(Error 34)**

After the FOR in a FOR statement, BASIC expects to find a valid numeric variable name. If it can't, this error will be given.

Index **(Error 3)**

This is an assembler error produced if an index is missing when one is required, or an illegal index register is given. Examples are:

```
120 LDX &10,X
340 LDA table,Z
```

LINE space **(Error 0)**

This error is given when typing in a BASIC line for which there is insufficient room in memory. It does not occur often, as you would usually have RUN a version of the program before this, producing a No room error.

Log range **(Error 22)**

This error is given when executing a LOG or LN statement with a negative argument. It would also be produced when attempting to raise a negative real number to a power.

Missing , (Error 5)

Lists in BASIC are frequently separated by commas (,) and this error is given if such a separator cannot be found. An example would be when MID$(A$) is executed.

Missing " (Error 9)

String constants should start and end with a double-quote ("). If the second quote cannot be found by the end of the line, BASIC gives this error message. A common cause of the error is incorrect specification of embedded quotes in strings. These should be written as two adjacent double-quotes (eg "a double quote "" is a wondrous thing").

Missing) (Error 27)

When an expression starts with a bracket, it should end with one too. If BASIC reaches the end of an expression that starts with a (and the next character is not a), it gives this error message. This message is also produced if a function call such as TAB(10,10) has the closing bracket omitted.

Missing # (Error 45)

The file access keywords such as CLOSE and BGET must be followed immediately by a #. Failure to do this will result in a Missing # error.

Mistake (Error 4)

This error can occur when a keyword is spelt wrong, as in:

```
PRUNT SIN(10)
```

It means that BASIC cannot make any sense of the command. Another reason for this error is when an unknown variable is used with an indirection operator assignment. Instead of No such variable being generated, Mistake would occur. Examples are:

```
fred?1=32
jim!I%=&8000
```

In both of these lines, Mistake would be given if fred or jim had not been assigned.

-ve root (Error 21)

This stands for Negative root and reflects the fact that it is not legal to take the square root of a number less than zero. Thus, PRINT SQR(-10) will give this error. It also crops up when ASN and ACS are used with arguments outside the range −1 to +1.

No GOSUB **(Error 38)**

This is produced when a RETURN statement is executed with no GOSUB currently active.

No FN **(Error 7)**

This is given when the 'end of function' sequence =<expression> is encountered and there are no user-defined functions active, or if there are, when a user-defined procedure has been called from within the most recent function.

No FOR **(Error 32)**

This error is given when a NEXT is executed and there are no FOR loops currently active.

No PROC **(Error 13)**

This is caused by an ENDPROC being encountered when there are no user-defined procedures active, or if there are, when a user-defined function has been called from within the most recent procedure.

No REPEAT **(Error 43)**

If an UNTIL statement is encountered and there are no active REPEATs, this error will be reported.

No room **(Error 0)**

This error is caused when BASIC runs out of space to put its variables. It is more likely to occur if a lot of string variables or arrays are declared. This error may also be produced by even quite small programs if they include a recursive procedure or function which is called to a very deep level of nesting.

No such FN/PROC **(Error 29)**

If a user-defined procedure or function is invoked (eg PROCfred or PRINT FNjim), BASIC tries to find a corresponding DEF FN or DEF PROC line. If it can't, this error will be generated. BASIC can fail to find the DEF because it isn't the first statement on the line, because the name of the FN/PROC is spelt differently to the one cited in the invocation, or simply because you forgot to define the FN/PROC.

No such line **(Error 41)**

This can occur when a line number is referenced but can't be found. Line numbers are found in GOTO, GOSUB, RESTORE and ON... GOTO/GOSUB statements. One way of detecting all of the potential No such line errors is to

RENUMBER the program. Un-matched line number references will cause Failed at... errors.

No such variable **(Error 26)**

In BBC BASIC, all variables must be assigned before they can be used. For example, the statement PRINT i will result in a No such variable error if i had not previously been created with an assignment or an INPUT statement. This applies to all variables except the system integer variables @% and A%–Z%, which are always present and retain their values after NEW, CLEAR, RUN etc, and LOCAL variables.

No such variable is also produced when BASIC expects to find a variable but cannot, eg A=B+ will give the error, as there is no second operand to the +.

No TO **(Error 36)**

The format of the FOR statement is:

FOR <var>=<expression> TO <expression>

If BASIC manages to evaluate the first <expression>, but can't find a TO after it, this error message is given.

Not LOCAL **(Error 12)**

A LOCAL statement may only appear when a procedure or function is active. If one is encountered at any other time, this error is generated.

ON range **(Error 40)**

The expression after ON should be in the range 1 to <n>, where <n> is the number of line numbers following the GOTO or GOSUB. For example, in:

```
ON key%+1 GOSUB 100,200,300
```

The expression key%+1 should evaluate to a numeric between 1 and 3. If it does not, an ON range error will be given. This error may be trapped as usual, or may be prevented altogther by adding an ELSE clause to the statement.

ON syntax **(Error 39)**

If the ON statement does not conform to the syntax illustrated in the example above (and isn't an ON ERROR statement), this message will be given.

Out of DATA **(Error 42)**

For every variable in a READ statement, there must be an expression in a DATA statement to be read. If there are more READs than DATA expressions, this error will be given. DATA may be re-read by using RESTORE.

Out of range **(Error 1)**

This is an assembly error. The 65C12 branch instructions are followed by a one-byte displacement. The destination of the branch can thus lie between −126 and +129 bytes from the branch instruction. If the destination is outside this range, `Out of range` will be given. This error may be caused by having too much code between the instruction and its destination, by failing to reset `P%` on each pass, or by using the same label twice by accident.

STOP **(Error 0)**

This fatal error is given when a BASIC `STOP` statement is encountered.

String too long **(Error 19)**

A string expression must at no time be longer than 255 characters. The error may be caused by using the `STRING$(` function or by the string concatentation operator, `+`.

Subscript **(Error 15)**

All arrays in BBC BASIC have a lower bound of zero and an upper bound determined by the `DIM` statement in which the array was declared. An attempt to access an array element outside this range will produce a subscript error.

Syntax error **(Error 16)**

This is caused by ending a statement incorrectly, eg

```
WIDTH 80a
```

or by trying to execute a command from within a program, or by giving a line number outside the range 0–32768 when typing in a line.

Too big **(Error 20)**

If an integer is required by BASIC, a real may be used as long as it is in the integer range of −2^31 to +2^31−1. If you use a real outside this range, the result will be too big. The error will also be given if any number outside the real range of approximately +/−1E38 is generated.

Too many FORs **(Error 35)**

There may only be 10 `FOR` loops active at one time. An attempt to start more loops that this will result in a `Too many FORs` error. The error may be generated by an unwise `GOTO`, eg jumping to the `FOR` line of a loop before the `NEXT` has been encountered, or by a deeply recursive procedure or function that contains a recursive call inside a `FOR` loop.

Too many GOSUBs **(Error 37)**

The limit of nesting of subroutines is 26. An attempt to call more than 26 subroutines without executing a RETURN will cause this error. This error can usually be avoided by using procedures, as these have a nesting limit which is determined solely by the amount of free memory. For example, with LOMEM at &0F00 and HIMEM at &8000, procedures and functions may be nested over 2000 deep.

Too many REPEATs **(Error 44)**

REPEAT statements may be nested up to 20 deep. If this limit is exceeded, a Too many REPEATs error will be given. This may be caused by similar situations as described for the Too many FORs error.

Type mismatch **(Error 6)**

If BASIC expects a numeric value but finds a string instead, this error will be produced. It is also given if a string is expected and a numeric value is encountered. Common causes are omitting the $ from string variables, and putting function arguments in the wrong positions, eg:

```
A="HELLO, WORLD": REM  Missing $
PRINT STRING$("-",20): REM  transposed arguments
```

M.2 Errors in numerical order

The table below lists the error messages described above in numerical order.

Fatal errors

```
Ø  LINE space
Ø  No room
Ø  STOP
```

Non-fatal (trappable) errors

```
1   Out of range
2   Byte
3   Index
4   Mistake
5   Missing ,
6   Type mismatch
7   No FN
8   $ range
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  Bad MODE
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  Too many FORs
36  No TO
37  Too many GOSUBs
38  No GOSUB
39  ON syntax
40  ON range
41  No such line
42  Out of DATA
43  No REPEAT
44  Too many REPEATs
45  Missing #
```

N BASIC technical information

N.1 Introduction

This section contains detailed information about the 'internals' of BBC BASIC, which will mainly be of interest to advanced users. This serves two purposes: it enables BASIC programmers to get more out of the language by increasing their understanding of how BASIC works and it shows assembly language programmers how BASIC uses the operating system facilities in the correct manner.

N.2 BASIC tokens in keyword order

The table below lists the BASIC reserved words along with the minimum abbreviations and tokens used. (rhs) following a keyword indicates the token when the keyword is used or the right-hand side of an expression (eg `PRINT HIMEM`); (lhs) indicates the token used when the keyword is used on the left-hand side of an expression (eg `HIMEM=&2800`).

Keyword	Abbreviation	Token			
ABS	ABS	&94	ELSE	EL.	&8B
ACS	ACS	&95	END	END	&E0
ADVAL	AD.	&96	ENDPROC	E.	&E1
AND	A.	&80	ENVELOPE	ENV.	&E2
ASC	ASC	&97	EOF	EOF	&C5
ASN	ASN	&98	EOR	EOR	&82
ATN	ATN	&99	ERL	ERL	&9E
AUTO	AU.	&C6	ERR	ERR.	&9F
BGET	B.	&9A	ERROR	ERR.	&85
BPUT	BP.	&D5	EVAL	EV.	&A0
CALL	CA.	&D6	EXP	EXP	&A1
CHAIN	CH.	&D7	EXT	EXT	&A2
CHR$	CHR$	&BD	FALSE	FA.	&A3
CLEAR	CL.	&D8	FN	FN	&A4
CLG	CLG	&DA	FOR	F.	&E3
CLOSE	CLO.	&D9	GCOL	GC.	&E6
CLS	CLS	&DB	GET	GET	&A5
COLOUR	C.	&FB	GET$	GE.	&BE
COS	COS	&9B	GOSUB	GOS.	&E4
COUNT	COU.	&9C	GOTO	G.	&E5
DATA	D.	&DC	HIMEM (rhs)	H.	&93
DEF	DEF	&DD	HIMEM (lhs)	H.	&D3
DEG	DEG	&9D	IF	IF	&E7
DELETE	DEL.	&C7	INKEY	INKEY	&A6
DIM	DIM	&DE	INKEY$	INK.	&BF
DIV	DIV	&81	INPUT	I.	&E8
DRAW	DR.	&DF	INSTR(	INS.	&A7
EDIT	ED.	&CE	INT	INT	&A8

LEFT$	LE.	&C0	RAD	RAD	&B2
LEN	LEN	&A9	READ	READ	&F3
LET	LET	&E9	REM	REM	&F4
LINE	LINE	&86	RENUMBER	REN.	&CC
LIST{O}	L.	&C9	REPEAT	REP.	&F5
LN	LN	&AA	REPORT	REPO.	&F6
LOAD	LO.	&C8	RESTORE	RES.	&F7
LOCAL	LOC.	&EA	RETURN	R.	&F8
LOG	LOG	&AB	RIGHT$(	RI.	&C2
LOMEM	LOM.	&92	RND	RND	&B3
LOMEM	LOM.	&D2	RUN	RUN	&F9
MID$(	M.	&C1	SAVE	SA.	&CD
MOD	MOD	&83	SGN	SGN	&B4
MODE	MO.	&EB	SIN	SIN	&B5
MOVE	MOVE	&EC	SOUND	SO.	&D4
NEW	NEW	&CA	SPC	SPC	&89
NEXT	N.	&ED	SQR	SQR	&B6
NOT	NOT	&AC	STEP	S.	&88
OFF	OFF	&87	STOP	STOP	&FA
OLD	O.	&CB	STR$	STR$	&C3
ON	ON	&EE	STRING$(	STRI.	&C4
OPENIN	OP.	&8E	TAB(	TAB(	&8A
OPENOUT	OPENO.	&AE	TAN	T.	&B7
OPENUP	OPENUP	&AD	THEN	TH.	&8C
OR	OR	&84	TIME (rhs)	TI.	&91
OSCLI	OS.	&FF	TIME (lhs)	TI.	&D1
PAGE (rhs)	PA.	&90	TO{P}	TO	&B8
PAGE (lhs)	PA.	&D0	TRACE	TR.	&FC
PI	PI	&AF	TRUE	TRUE	&B9
PLOT	PL.	&F0	UNTIL	U.	&FD
POINT(	PO.	&B0	USR	USR	&BA
POS	POS	&B1	VAL	VAL	&BB
PRINT	P.	&F1	VDU	V.	&EF
PROC	PROC	&F2	VPOS	VP.	&BC
PTR (rhs)	PTR	&8F	WIDTH	W.	&FE
PTR (lhs)	PTR	&CF			

N.3 BASIC keywords in numerical order of tokens

The table below lists the tokens used by BASIC. All possible values in the range 128–255 are used. They are grouped logically, eg operators are in a contiguous list, as are statements.

Token	keyword	Token	keyword	Token	keyword
&80	AND	&9F	ERR	&BE	GET$
&81	DIV	&A0	EVAL	&BF	INKEY$
&82	EOR	&A1	EXP	&C0	LEFT$(
&83	MOD	&A2	EXT	&C1	MID$(
&84	OR	&A3	FALSE	&C2	RIGHT$(
&85	ERROR	&A4	FN	&C3	STR$
&86	LINE	&A5	GET	&C5	EOF
&87	OFF	&A6	INKEY	&C4	STRING$(
&88	STEP	&A7	INSTR(	&C6	AUTO
&89	SPC	&A8	INT	&C7	DELETE
&8A	TAB(	&A9	LEN	&C8	LOAD
&8B	ELSE	&AA	LN	&C9	LIST{O}
&8C	THEN	&AB	LOG	&CA	NEW
&8D	(see below)	&AC	NOT	&CB	OLD
&8E	OPENIN	&AD	OPENUP	&CC	RENUMBER
&8F	PTR	&AE	OPENOUT	&CD	SAVE
&90	PAGE	&AF	PI	&CE	EDIT
&91	TIME{$}	&B0	POINT(	&CF	PTR
&92	LOMEM	&B1	POS	&D0	PAGE
&93	HIMEM	&B2	RAD	&D1	TIME{$}
&94	ABS	&B3	RND	&D2	LOMEM
&95	ACS	&B4	SGN	&D3	HIMEM
&96	ADVAL	&B5	SIN	&D4	SOUND
&97	ASC	&B6	SQR	&D5	BPUT
&98	ASN	&B7	TAN	&D6	CALL
&99	ATN	&B8	TO{P}	&D7	CHAIN
&9A	BGET	&B9	TRUE	&D8	CLEAR
&9B	COS	&BA	USR	&D9	CLOSE
&9C	COUNT	&BB	VAL	&DA	CLG
&9D	DEG	&BC	VPOS	&DB	CLS
&9E	ERL	&BD	CHR$	&DC	DATA

&DD	DEF	&E9	LET	&F5	REPEAT
&DE	DIM	&EA	LOCAL	&F6	REPORT
&DF	DRAW	&EB	MODE	&F7	RESTORE
&E0	END	&EC	MOVE	&F8	RETURN
&E1	ENDPROC	&ED	NEXT	&F9	RUN
&E2	ENVELOPE	&EE	ON	&FA	STOP
&E3	FOR	&EF	VDU	&FB	COLOUR
&E4	GOSUB	&F0	PLOT	&FC	TRACE
&E5	GOTO	&F1	PRINT	&FD	UNTIL
&E6	GCOL	&F2	PROC	&FE	WIDTH
&E7	IF	&F3	READ	&FF	OSCLI
&E8	INPUT	&F4	REM		

Token &8D is used for line number targets and for compressing error messages.

N.4 The memory map under BASIC

This chapter provides information for 'advanced' users. Read it to increase your understanding of how BASIC uses the memory of the computer and co-processors. The 'internal' information is given for interest only, and should not be taken to be true for future (or indeed past) versions of BASIC.

The diagram on the next page shows how the random access memory (RAM) of the computer is segmented when BASIC is being used. The order of the parts is always the same, but their actual addresses depend on factors such as the machine configuration and program size.

The addresses marked as 'Typical' may change depending on which filing systems are present etc. The values given above are for a BASIC program in a machine with no co-processor active or extra filing systems installed, with some dynamic variables declared and some procedures active. The contents of the regions are described on the following pages.

Address	Contents of region	Value
HIMEM		&8000 (typical)
	BASIC stack	
STACK_TOP		&7932 (Example)
	Free space	
VAR_TOP		&3457 (Example)
	Dynamic variables	
LOMEM/TOP		&3184 (Example)
	BASIC Program	
PAGE/OSHWM		&0E00 (Typical)
	MOS and FS workspace	
WORK_END		&0800
	Static variables etc	
WORK_START		&0400
	Stack, OS and FS workspace	
PAGE_1		&0100
	OS and FS workspace	
USER_ZP_END		&0090
	User's zero-page locations	
USER_ZP_START		&0070
	BASIC's zero-page locations	
BASIC_ZP_START		&0000

BASIC_ZP_START to USER_ZP_END–1

The MOS reserves locations &0000 to &008F in the page zero for the current language. BASIC uses &0000 to &006F for its own information, such as system pointers, accumulators etc and leaves &0070 to &008F free for the user. Zero page locations between &0090 and &00FF are 'out of bounds' to the BASIC programmer, although it may be useful to note that the error pointer (which contains the address of the error number byte of the last error) is at &FD and &FE, and &FF contains the escape condition flag.

PAGE_1 to WORK_START–1

Page 1 (addresses &0100 to &01FF) is used by the 65C12 microprocessor for its hardware stack, although when it reports an error, the ADFS (and other filing systems) put the error message text starting at &0100 (in the I/O processor). In general, page one should only be accessed from machine code using stack-addressing instructions such as `PHA`, `PLX` etc.

Pages two and three contain operating system workspace such as the two-byte indirection vectors. Again, they should only be changed by the BASIC user in exceptional cases, when machine code is used.

WORK_START to WORK_END–1

The current language uses locations &400-&7FF as workspace. BASIC reserves all of this for its own use and the user should not access it directly. BASIC uses the area for its resident integer variables (`@%`, `A%-Z%`); pointers to dynamic variable lists and user-defined procedures/functions; `FOR`, `REPEAT` and `GOSUB` stacks; the input buffer and the string workspace.

WORK_END to OSHWM–1

This region contains more operating system and filing system workspace and should not be tampered with.

OSHWM/PAGE to TOP–1/LOMEM–1

This is where the user's program is stored. When BASIC is first entered, it sets `PAGE` to the operating system high-water mark. Typical values for OSHWM are:

Value	Machine
&0800	65C12 co-processor
&0E00	standard computer with built-in filing systems
&1200	standard computer with an extra filing system that claims 3 pages of workspace from the main memory

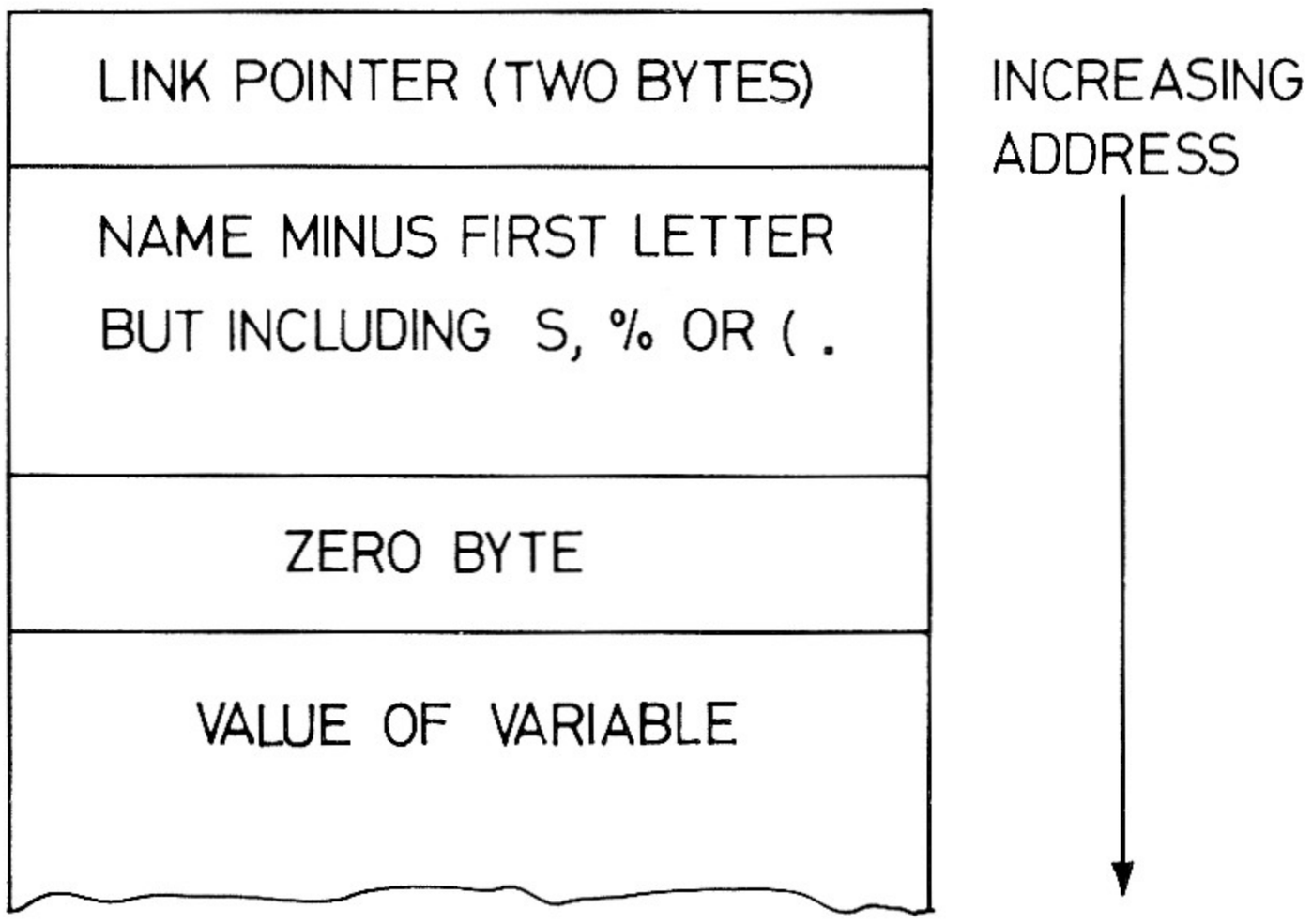
In addition, the user may reserve space (eg for machine code) by increasing the BASIC pseudovariable PAGE. As its name implies, PAGE is always set to a page boundary; when changing it, the new value is effectively ANDed with &FF00 to ensure this. After changing PAGE, if there is no program at the new address, type NEW before entering a program (unless the program is LOADed or CHAINed, of course). By manipulating PAGE carefully, you can have several separate BASIC programs in the computer at once.

The user's program extends to the address given by TOP−1, ie TOP holds the address of the first byte after the program. TOP is a function and you cannot change it with an assignment. TOP is however changed each time a program line is entered or deleted. It is clear that the size of the program is given by TOP−PAGE.

LOMEM is the address of the start of BASIC variables (except for @% and A%-Z%). It is set to TOP when the program is RUN. As LOMEM is a BASIC pseudovariable, it may be altered by the user. For instance, it is possible to *SAVE a BASIC program with machine code tagged on the end. This program should have as its first line a statement to increase LOMEM to past the end of the machine code, so that when it is RUN, the machine code is protected from overwriting by BASIC variables. The machine code would have to be totally relocatable and would need to be called relative to TOP, eg CALL TOP, CALL TOP+9.

LOMEM to VAR_TOP−1

The BASIC program's dynamic variables lie in this region. They are stored as 54 linked lists. The pointer to the head of each list is stored at &400+2*ASC(c$) and &401+2*ASC(c$), where c$ is the first character of the variable name. The format for a variable entry is:



The end of the list is marked by a zero high-byte in the link pointer (or the head pointer in page four). The size of the value of a variable depends on its type, eg

four bytes for integers, five bytes for reals, a four-byte string information block (SIB) for strings etc.

Procedures and functions have their own linked lists. The value part gives the address of the DEF of the procedure/function in the program.

VAR_TOP to STACK_TOP–1

This is free space. It is decreased by VAR_TOP increasing (dynamic variables being created, new string space being allocated and the byte form of DIM), and by STACK_TOP decreasing. VAR_TOP can never decrease. STACK_TOP moves down when user-defined procedures and functions are entered, when LOCAL variables are declared and during the evaluation of expressions. It moves up again when procedures and functions are exited and at the end of expression evaluation. Outside of procedures and expressions, STACK_TOP is set to HIMEM (which could be called STACK_BOT).

To see exactly how much free memory there is, STACK_TOP – VAR_TOP must be calculated. As STACK_TOP is not accessible 'legally', some nefarious peeking has to take place. To find the value of VAR_TOP, the statement:

```
DIM V% -1
```

is used. This sets the variable V% to VAR_TOP. The reason for using a system integer variable is to avoid increasing VAR_TOP by creating a new variable in the DIM. STACK_TOP is held at locations 4 and 5 in zero-page. Thus the amount of free memory may be found like this:

```
DIM V% -1
F%=(!4 AND &FFFF) - V% + 4
```

The extra +4 in the calculation compensates for the four bytes of the stack used during the assignment. Immediately after a NEW or CLEAR, F% will have the same value as HIMEM – LOMEM.

STACK_TOP to HIMEM–1

This is the BASIC stack, mainly used to store parameters, local variables and return addresses when user-defined procedures are active. Before a formal parameter variable (the one in the DEF part of the procedure) is set to the actual value given in the call, its current value is stored on the stack. If there is currently no dynamic variable with the name of the formal parameter, one is created and set to zero (or the null string), and this value is stored on the stack. The new variable is then assigned the value of the actual parameter. At the end of the procedure, the saved values are restored from the stack.

LOCAL variables are treated in exactly the same way, except that after the current value is saved, they are set to zero (or the null string). The return

address of the procedure is also saved; this is the place in the program immediately after where the procedure was called.

Function calls work in exactly the same way. In addition, before the function returns its result is stored in the appropriate BASIC accumulator in zero page (or in the string buffer, for strings).

The value of `HIMEM` varies. When BASIC is called it is set to the 'top of free RAM', as given by OSBYTE 132. It may be altered by assigning to the pseudovariable HIMEM. Given the uses of `HIMEM` noted above, it is obviously not a good idea to change it within a procedure or function.

Some typical examples of `HIMEM` are:

Value	**Machine**
&8000	Standard computer in shadow mode, any screen mode
&7C00	Standard computer in non-shadow mode, `MODE 7`
&3000	Standard computer in non-shadow mode, `MODE 0`, 1 or 2
&8000	65C12 co-processor running BASIC
&B800	65C12 co-processor running HIBASIC

N.5 How BASIC uses the MOS and filing systems

Many of the powerful features of BASIC are made possible by the facilities offered by the machine operating system and filing system. This chapter explains the ways in which keywords use these facilities.

For a description of the routines used, see Sections D (Using MOS Routines) and F (Filing systems).

In the descriptions, A, X and Y are the names of the 65C12 registers. XY is used when the two registers contain an address (low byte in X). A call such as OSWORD 1 means a call to the operating system routine OSWORD, with the A register (accumulator) set to 1.

Note that all character output from BASIC keywords (such as `PRINT` and `INPUT`), uses OSWRCH unless otherwise mentioned.

ADVAL

This function uses the call OSBYTE 128. The lower two bytes of the argument are placed in X and Y before the call, and the result is a sign-extended version of XY on exit.

BGET and BPUT

These use the two routines OSBGET and OSBPUT respectively. `BGET` passes its argument (the channel number) in Y, and returns the value in the accumulator on exit. `BPUT` places its first argument in Y (the channel number), and the second in A (the character to be written) before calling OSBPUT.

CHAIN

This statement uses OSFILE with A=&FF to load the program. It sets up the parameter block so that the name pointer points to the program filename, and the load address is set to `PAGE`.

CLG

This uses the OSWRCH routine with A=16 (clear graphics VDU command).

CLOSE

This uses the OSFIND routine. A is set to 0 (close) and Y is set to the channel number given by the argument.

CLS

This uses the OSWRCH routine with A set to 12 (clear text VDU command).

COLOUR

This calls OSWRCH with A first set to 17 (define text colour VDU command) then with A set to the value given after `COLOUR`.

DRAW

This statement calls OSWRCH with A set first to 25 (VDU plot command) then 5 (draw absolute plot command). Then it calls OSWRCH four times with the low and high bytes of the x co-ordinate specified and the y co-ordinate specified.

ENVELOPE

This uses OSWORD 8. The 14 parameters after `ENVELOPE` are stored in a 14-byte parameter block pointed to by XY.

EOF

This function uses OSBYTE 127. X is set to the channel number following `EOF#`, and the result is a sign-extended version of the X register on exit from OSBYTE.

ERR

This uses the operating system pointer at &00FD and &00FE to read the error number. Its action can be expressed in BASIC as `ERR=?!&FD`

EXT

This uses the call OSARGS 2. Y is set to the channel number given after `EXT#`. X is set to point to a four-byte area of zero page. The result is returned in those four bytes. When using `EXT#` as a statement OSARGS 3 is used, with X pointing to the four-byte area holding the new length (taken from the expression after the `EXT#chan=`) and Y holds the channel number given after the `EXT#`.

GCOL

This uses OSWRCH with A=18 (graphics colour VDU command) followed by the two parameters of `GCOL`, sent as one byte each.

GET

This function uses the OSRDCH routine. The result (returned in A) is the result of the function.

GET$

As GET, but BASIC converts the result to a string.

HIMEM

The default value of HIMEM is read from the operating using OSBYTE 132. The 2-byte value is returned as XY. HIMEM is read when BASIC is entered and when a MODE change is affected.

INKEY

This uses the call OSBYTE 129. The argument to INKEY is treated as a two-byte number, passed to OSBYTE in XY. On exit, if Y=0, the result is in the X register. If Y is not 0, the result in X is −1.

INKEY$

As INKEY, but BASIC converts the result to a string. BASIC converts an INKEY result of −1 to the null string.

INPUT

This uses OSWORD 0 to input a line. The parameter block used has the following values: Length=238, minimum ASCII=32 (space), maximum ASCII=255.

LOAD

This uses OSFILE, in a similar way to CHAIN.

MODE

This calls OSWRCH twice, with A=22 (screen mode VDU command) then A=<the expression after MODE>. BASIC checks that changing mode will not corrupt the program or its variables by calling OSBYTE 133 with X=<the expression after MODE>. Because of this, the screen MODE should not be changed using VDU 22 unless you are in *SHADOW mode or are running BASIC in a co-processor.

MOVE

This uses OSWRCH 25 (plot) followed by OSWRCH 4 (move absolute), followed by four calls to OSWRCH. These calls use the low byte of the x co-ordinate given, the high byte of x, then the low and high bytes of the y co-ordinate.

OPENIN, OPENOUT and OPENUP

These functions use OSFIND. XY is set to point to the string specfified as the

parameter. A is set to &40, &80 or &C0 respectively. The result is the the value of A on exit.

OSCLI

This passes its string argument to the operating sytem by setting XY to the address of the string and calling OSCLI.

PAGE

The default value of PAGE, set when BASIC is entered, is read from the operating system using OSBYTE 131 (read OS high-water mark). The page number is read from Y on exit. The low byte (in X) is ignored.

PLOT

This uses OSWRCH. First OSWRCH 25 (plot) is called, then the first argument to PLOT is sent as a single byte (the plot option), and finally the two co-ordinates are sent as four bytes in the order low x, high x, low y, high y.

POINT(

This uses OSWORD 9. A parameter block is set up, the first four bytes of which are the x and y co-ordinates specified after POINT(. XY is set to point to the parameter block and OSWORD is called. On exit, the fifth byte of the parameter block is returned. If this has the top bit set (ie is greater then &7F), −1 is returned.

POS

This function calls OSBYTE 134 (read text cursor position) and returns the contents of the X register on exit.

PTR

This uses OSARGS. When PTR# is used as a function, BASIC sets A to 0, Y to the channel number and X to the address of a four-byte area of zero page. The result of the function is returned in this four-byte area. The four bytes are interpreted as an integer value. When PTR# is used as a statement, BASIC sets A to 1, and X and Y as for the function, and the four bytes in zero page are set to the integer expression assigned to PTR#.

REPORT

This statement uses the MOS error pointer at locations &FD and &FE. The BASIC code equivalent to report would be:

```
10 PRINT
20 addr%=(!&FD AND &FFFF)+1
```

```
30 REPEAT
40   PRINT CHR$(?addr%);
50   addr%=addr%+1
60 UNTIL ?addr%=0
```

SAVE

This uses OSFILE 0. BASIC sets XY to point to a 17-byte parameter block, containing the address of the filename specified with SAVE, the start address of the file as PAGE and the end address as TOP.

SOUND

This uses OSWORD 7. XY points to an eight-byte parameter block, derived from the four parameters of SOUND, each of which occupies two bytes.

TAB(

When used with one parameter, this simply prints spaces with OSWRCH (perhaps preceded by a call to OSNEWL to print a newline). When used with two parameters, OSWRCH is called with A=31 (move cursor to co-ordinates VDU command), followed by the lowest byte of the x co-ordinate and the lowest byte of the y co-ordinate.

TIME

When used as a function, this uses OSWORD 1 (read system clock) with XY pointing to a five byte parameter block. The result is the integer formed from the lower four bytes of this block returned from OSWORD. When used as a statement, TIME calls OSWORD 2 (write system clock). Again, XY points to the five-byte parameter block. The expression after TIME= is stored in the lower four bytes and the fifth byte is set to zero.

TIME$

This is similar to TIME, but uses OSWORD 14 and 15. When used as a function, BASIC sets XY to point to a 25-byte parameter block, of which the first byte is set to zero (to make OSWORD return a time and date string) and calls OSWORD 14. OSWORD then stores the time and date string in the last 24 bytes of the parameter block, which is the string returned by TIME$. (See TIME$ in Section L for the exact format.) When TIME$ is used as a statement, BASIC evaluates the expression assigned to TIME$, puts this into the parameter block pointed to by XY, after a byte giving the length of the string (legal strings are 8, 15 or 24 bytes long).

VDU

This statement passes each of its arguments in turn to the OSWRCH routine. If the expression was followed by a comma, or is the last one in the statement, it is passed as a single byte. If it was followed by a semi-colon, it is sent as two bytes, low byte first.

VPOS

This function calls OSBYTE 134 (read text cursor position) and returns the contents of the Y register (horizontal position) on exit.

O The BBC BASIC Assembler

O.1 Introduction to machine code and assembly language

This Section provides instruction in the use of the 65C12 assembler provided in the BASIC IV interpreter. For a description of the 65C12 keywords (instruction mnemonics and assembler directives), see Section P (Assembler Keywords). For information on assembler errors see Section Q (Assembler Errors).

This section will mainly be of interest if you want to write machine code programs or routines.

Assembly language

Assembly language is a programming language, whose statements translate directly into single machine code instructions, data or 'assembler switches' (see OPT, chapter O.6). It is known as a 'low level' language, because it allows you to control the computer at a low level. This translation is accomplished by use of a program known as an assembler.

Machine code is the native language of the central workhorse of the computer, its 65C12 microprocessor. The MOS, languages, filing systems and even the assembler itself are all written in machine code, executed by the 65C12 microprocessor. Although 65C12 machine code is a primitive programming language that can only perform a range of fairly elementary data manipulation functions, it executes them at great speed.

The computer provides you with a range of facilities to enable you to create and use your own machine code programs, as described within this section.

The advantages of assembly language

Most programs that can be written in a high level language such as BASIC can also be written in a low level language such as machine code, via an assembler. Machine code programs have three main advantages: processing speed, low level access to the MOS and filing systems and language-independence.

Speed

Although it is quicker and easier to write programs in a high level language, programs written in the interpreted high level languages used in the computer (and the majority of other modern microcomputers) will run much more slowly than machine code programs. This is because a program in an interpreted language is run by interpreting each language statement in turn into the required machine code calls, *at the time of running the program*. Although assembly language programs go through a comparable process to turn them into machine code, this is only done once (during assembly) – when the machine code program is run, its instructions are executed directly ie without interpretation.

Low level access

High level languages provide neatly packaged access to filing systems and the MOS through commands such as `LOAD`, `PRINT#`, `SOUND` and `OSCLI`. Filing systems and the MOS also provide access through commands such as `*INFO` and `*KEY`. These are very convenient if they allow you to achieve what you want, but if they do not, you can get far greater flexibility by using assembly language. The MOS and filing systems are unusual in providing easy access to a wide range of facilities from machine code, as described elsewhere in this guide.

The MOS also allows you to supply routines to replace or supplement some of its processing. These routines cannot be written in a high level language; they must be written in machine code.

Language independence

Programs written in high level languages can only be used while the language interpreter is selected as the current language. Machine code programs must be assembled with BASIC selected, but once that has been done they can, in principle, be used with any language. Writing utility programs in machine code thus makes them more flexible.

The disadvantages of assembly language

Writing machine code programs using assembly language has several disadvantages compared to using the high level interpreted languages available with the BBC Microcomputer system: the need to assemble, program complexity and error handling.

Assembling

Unlike high level programs, you cannot simply run an assembly language program – after writing it or changing it, you must assemble the program before you can use it. This only results in a small delay if there is room in

memory for the program that does the assembling and the machine code produced from it, but if the program is large this is not possible and the re-assembling becomes a problem.

Program complexity

Assembly language programs are usually more complex to design and understand than corresponding programs written in high level languages. This is because individual machine code instructions can only perform such elementary functions as adding, subtracting, storing or retrieving eight bit numbers – you will have to write your own routines to perform many of the functions provided by single statements in high level languages. For example to print the message "Hi there!":

In PASCAL:
```
writeln("Hi there!")
```

In BASIC:
```
PRINT"Hi there!"
```

In assembly language:
```
LDX #0
.loop LDA message,X
INX
JSR &FFE3
CMP #13
BNE loop
BRA .carryon
.message EQUS "Hi there!":EQUB 13
.carryon ...
```

Error handling

Assembly language programs are more difficult to check and correct, since there is no error handling provided when running machine code programs (you must supply your own), programs are more complex and hence more likely to be incorrect, machine code registers and variables cannot be examined easily and any changes have to be made to the assembly language which must then be assembled again before the program can be tested.

Combining high level languages and machine code

Writing a complex program entirely in machine code is a major task which should not be underestimated, although the computer reduces the effort needed considerably by providing simple access to a wide range of MOS and filing system routines.

Wherever possible, it is best to use the high level language for the more complex part of a task and use machine code programs as a kind of subroutine

for repetitive simple tasks. In this way, with a little extra effort you can usually achieve impressive improvements in speed.

The only types of programs that are usually written entirely in machine code are programs that must either run with any language (such as a filing system) or must run as quickly as possible (such as a fast-moving graphics game).

Preparing machine code programs

Machine code is simply a sequence of numbers stored in memory that the microprocessor steps through, interpreting these numbers as instructions or data. It is entirely possible to construct a machine code program directly from these numbers using the facilities of a language such as BASIC, but this involves a great deal of repetitive and exacting work.

For anything but the smallest program, the best method is not to write machine code directly, but to write the program in a low level language known as assembly language and then use the assembler provided in BASIC to translate that program into machine code. This is quite unlike using a high level language, since the translation process occurs once only – the program that you run subsequently is true machine code.

It is easier to write programs using assembly language than machine code because it uses more meaningful (mnemonic) names for the machine code instructions and provides other facilities (labels, assembler directives, expression evaluation and error checking) to reduce the amount of housekeeping and calculation you will have to do.

Assembly language programs are prepared in the same ways as BASIC programs ie using the system editor or selecting BASIC and using the screen editor. The program is then assembled into machine code by RUNning it in BASIC. Machine code generated by the assembler is stored in memory ready for you to execute, move or save in a filing system.

The assembler has been designed as a well-integrated part of the BASIC ROM, so that you can use many of the facilities of BASIC in your assembly language, or switch between assembly language and BASIC at will to implement sophisticated facilities such as macros and conditional assembly (see below).

O.2 Information for previous users of BBC BASIC

The assembler provided is very similar to that provided in previous versions of BBC BASIC. The main enhancements are the additions of new instructions and addressing modes in the 65C12 instruction set. Programs which assembled correctly using previous versions of BBC BASIC should assemble without needing to make any changes.

Although existing programs may assemble correctly, the resulting program may not run correctly, but this is only likely if it uses one of the few MOS commands or routines that have changed (see Section C), or relies on improper techniques such as direct screen addressing. If this is the case you will need to amend your program before assembling it.

Note that the way the computer memory is organised (the system memory map) is somewhat different from previous BBC microcomputers.

To summarise, the changes are:

- support for the extra 65C12 mnemonics and addressing modes (see chapter O.4, below).

- register names and the assembler directives `EQUS`, `EQUB`, `EQUD`, `EQUW` may be in lower case.

- `DEA` may be used instead of `DEC A`, `INA` instead of `INC A`, `CLR` instead of `STZ`.

- operands which begin with 'A' will no longer be misinterpreted when used with instructions that can allow accumulator addressing.

For previous users of BASIC I, the following features (which were added in BASIC II) will also be new:

- provision of the `EQUS`, `EQUB`, `EQUW` and `EQUD` assembler directives (instead of using BASIC's indirection operators – `$address=`, `?address=` etc).

- provision of an extra assembly switch (via OPT) to allow assembling of machine code at a different address from that at which it will be run.

- more than one assembler statement may be used on a line, by separating them with `:`

O.3 Information for users of other 6502 assemblers

The assembler recognises the instruction mnemonics used by various other manufacturers, including M.O.S. Technology, Commodore Business Machines, Synertek and Rockwell International. For a description of the conventions used for addressing modes and assembler directives, see the next Section.

The assembler always assembles to memory; object programs must be saved to the filing system explicitly.

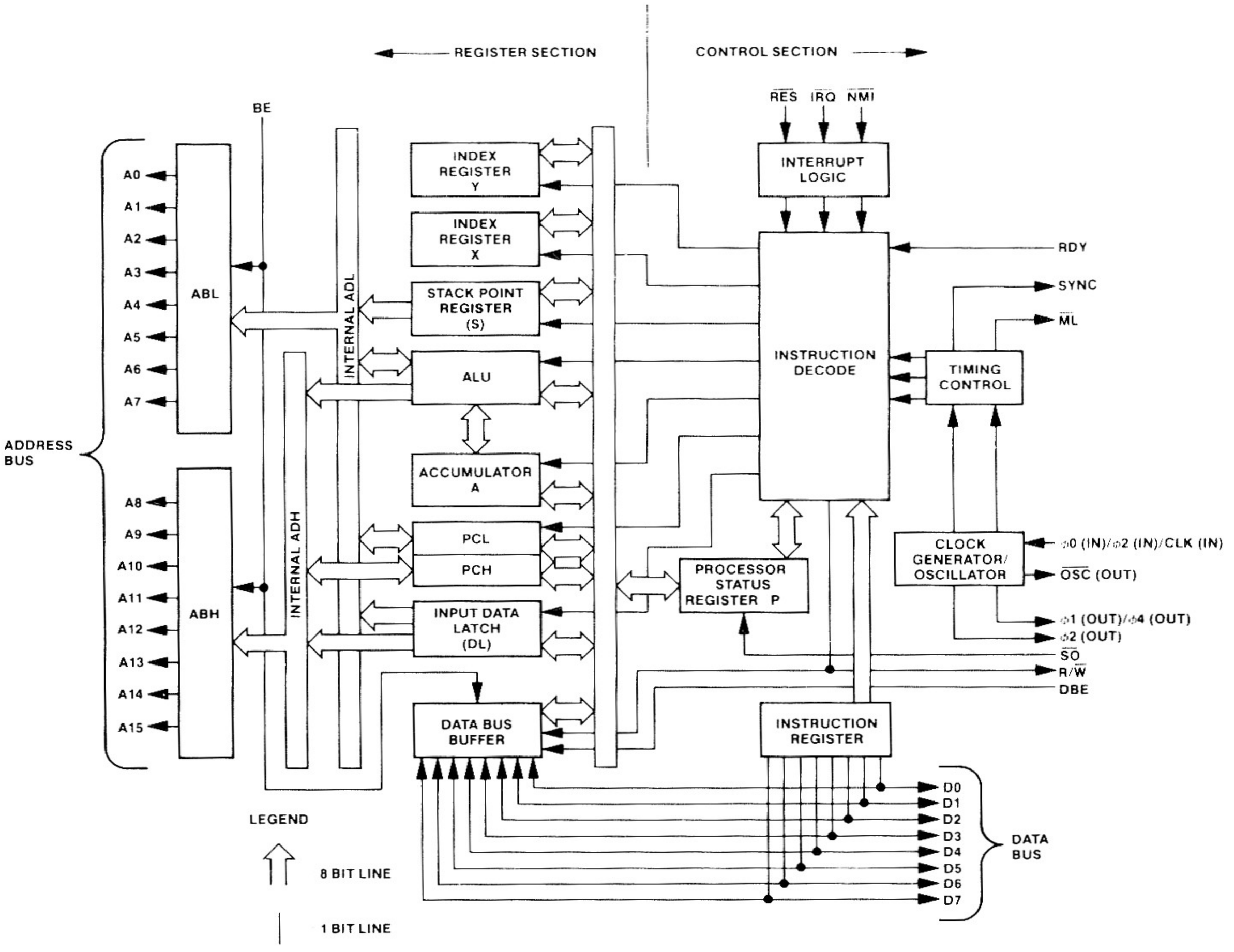
The assembler does not automatically perform multi-pass assembly; this must be coded by the user.

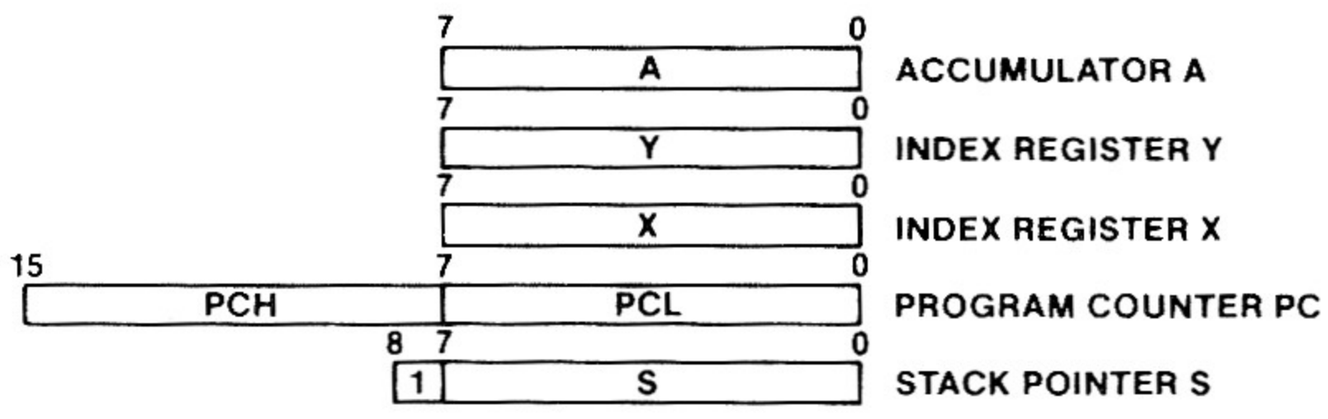
Expression evaluation is performed by BASIC.

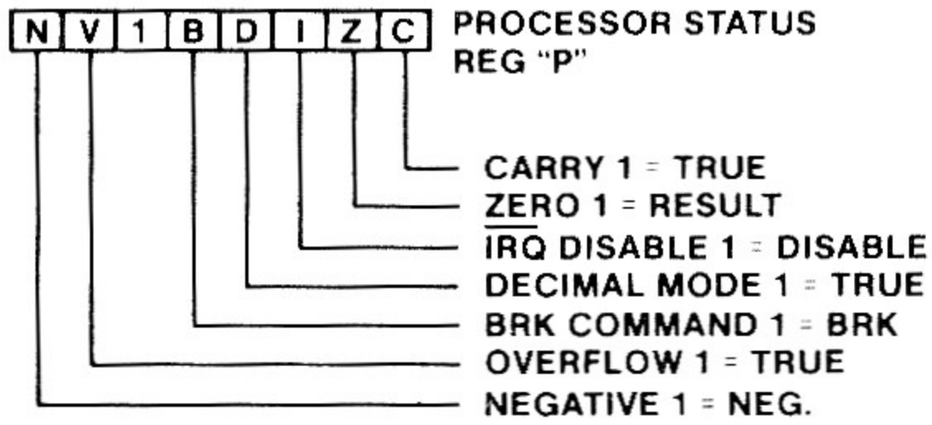
Labels are treated as BASIC variables.

O.4 The 65C12 microprocessor

The 65C12 is an enhanced version of the 6502 microprocessor that has previously been used in a wide variety of microcomputers, including the BBC Model B microcomputer. The 65C12 will execute all documented 6502 instructions correctly; the main enhancements are extra instructions and addressing modes, marked by a blob in the table on the next page. The 65C12 also corrects certain minor 'bugs' in the 6502.







	ADC	Add Memory to Accumulator with Carry
	AND	"AND" Memory with Accumulator
	ASL	Shift One Bit Left
	BCC	Branch on Carry Clear
	BCS	Branch on Carry Set
	BEQ	Branch on Result Zero
	BIT	Test Memory Bits with Accumulator
	BMI	Branch on Result Minus
	BNE	Branch on Result Not Zero
	BPL	Branch on Result Plus
●	BRA	Branch Always
	BRK	Force Break
	BVC	Branch on Overflow Clear
	BVS	Branch on Overflow Set
	CLC	Clear Carry Flag
	CLD	Clear Decimal Mode
	CLI	Clear Interrupt Disable Bit
	CLV	Clear Overflow Flag
	CMP	Compare Memory and Accumulator
	CPX	Compare Memory and Index X
	CPY	Compare Memory and Index Y
	DEC	Decrement by One
	DEX	Decrement Index X by One
	DEY	Decrement Index Y by One
	EOR	"Exclusive-or" Memory with Accumulator
	INC	Increment by One
	INX	Increment Index X by One
	INY	Increment Index Y by One
	JMP	Jump to New Location
	JSR	Jump to New Location Saving Return Address
	LDA	Load Accumulator with Memory
	LDX	Load Index X with Memory
	LDY	Load Index Y with Memory
	LSR	Shift One Bit Right
	NOP	No Operation
	ORA	"OR" Memory with Accumulator
	PHA	Push Accumulator on Stack
	PHP	Push Processor Status on Stack
●	PHX	Push Index X on Stack
●	PHY	Push Index Y on Stack
	PLA	Pull Accumulator from Stack
	PLP	Pull Processor Status from Stack
●	PLX	Pull Index X from Stack
●	PLY	Pull Index Y from Stack
	ROL	Rotate One Bit Left
	ROR	Rotate One Bit Right
	RTI	Return from Interrupt
	RTS	Return from Subroutine
	SBC	Subtract Memory from Accumulator with Borrow
	SEC	Set Carry Flag
	SED	Set Decimal Mode
	SEI	Set Interrupt Disable Bit
	STA	Store Accumulator in Memory
	STX	Store Index X in Memory
	STY	Store Index Y in Memory
●	STZ	Store Zero in Memory
	TAX	Transfer Accumulator to Index X
	TAY	Transfer Accumulator to Index Y
●	TRB	Test and Reset Memory Bits with Accumulator
●	TSB	Test and Set Memory Bits with Accumulator
	TSX	Transfer Stack Pointer to Index X
	TXA	Transfer Index X to Accumulator
	TXS	Transfer Index X to Stack Pointer
	TYA	Transfer Index Y to Accumulator

Note: ● = New Instruction

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK	ORA ind, X			**TSB zpg**	ORA zpg	ASL zpg		PHP	ORA imm	ASL A		**TSB abs**	ORA abs	ASL abs		0
1	BPL rel	ORA ind, Y	**ORA ind**		**TRB zpg**	ORA zpg, X	ASL zpg, X		CLC	ORA abs, Y	**INC A**		**TRB abs**	ORA abs, X	ASL abs, X		1
2	JSR abs	AND ind, X			BIT zpg	AND zpg	ROL zpg		PLP	AND imm	ROL A		BIT abs	AND abs	ROL abs		2
3	BMI rel	AND ind, Y	**AND ind**		**BIT zpg, X**	AND zpg, X	ROL zpg, X		SEC	AND abs, Y	**DEC A**		**BIT abs, X**	AND abs, X	ROL abs, X		3
4	RTI	EOR ind, X				EOR zpg	LSR zpg		PHA	EOR imm	LSR A		JMP abs	EOR abs	LSR abs		4
5	BVC rel	EOR ind, Y	**EOR ind**			EOR zpg, X	LSR zpg, X		CLI	EOR abs, Y	**PHY**			EOR abs, X	LSR abs, X		5
6	RTS	ADC ind, X			**STZ zpg**	ADC zpg	ROR zpg		PLA	ADC imm	ROR A		JMP ind	ADC abs	ROR abs		6
7	BVS rel	ADC ind, Y	**ADC ind**		**STZ zpg, X**	ADC zpg, X	ROR zpg, X		SEI	ADC abs, Y	**PLY**		**JMP ind, X**	ADC abs, X	ROR abs, X		7
8	**BRA rel**	STA ind, X			STY zpg	STA zpg	STX zpg		DEY	**BIT imm**	TXA		STY abs	STA abs	STX abs		8
9	BCC rel	STA ind, Y	**STA ind**		STY zpg, X	STA zpg, X	STX zpg, Y		TYA	STA abs, Y	TXS		**STZ abs**	STA abs, X	**STZ abs, X**		9
A	LDY imm	LDA ind, X	LDX imm		LDY zpg	LDA zpg	LDX zpg		TAY	LDA imm	TAX		LDY abs	LDA abs	LDX abs		A
B	BCS rel	LDA ind, Y	**LDA ind**		LDY zpg, X	LDA zpg, X	LDX zpg, Y		CLV	LDA abs, Y	TSX		LDY abs, X	LDA abs, X	LDX abs, Y		B
C	CPY imm	CMP ind, X			CPY zpg	CMP zpg	DEC zpg		INY	CMP imm	DEX		CPY abs	CMP abs	DEC abs		C
D	BNE rel	CMP ind, Y	**CMP ind**			CMP zpg, X	DEC zpg, X		CLD	CMP abs, Y	**PHX**			CMP abs, X	DEC abs, X		D
E	CPX imm	SBC ind, X			CPX zpg	SBC zpg	INC zpg		INX	SBC imm	NOP		CPX abs	SBC abs	INC abs		E
F	BEQ rel	SBC ind, Y	**SBC ind**			SBC zpg, X	INC zpg, X		SED	SBC abs, Y	**PLX**			SBC abs, X	INC abs, X		F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

See Section P.3 for information on 65C12 addressing modes.

O.5 Using the assembler

The assembler is part of the BBC BASIC language and is used by entering the BASIC language and RUNning a program consisting of BASIC and assembly language statements. A complete program to assemble assembly language into machine code and then run the machine code program may consist of the following stages:

– reserve memory space for machine code (`DIM`)	
– initialise external variables (labels)*	BASIC
– implement passes required (`FOR...`)*	
– set memory pointers (`P%` or `P%` and `O%`)	
– enter assembler	
– set assembler switch (`OPT`)*	ASSEMBLY
– assembly language statements	LANGUAGE
– leave assembler	
– (`...NEXT`)*	BASIC
– run/save/move object program*	

*These stages are optional

Example program

```
10 REM initialise
20 message$="This bit comes from BASIC"
30 DIM Q% 100
40 osasci=&FFE3
50 REM assemble
60 FOR C=0 TO 2 STEP 2
70 P%=Q%
80 [
90 .entry LDY #0
100 .loop LDA buffer, Y
110 CMP #13
120 BEQ exit\end of message
130 JSR
140 INY
150 BRA loop
160 .exit RTS
170 ]
180 P%=P%+10:REM leave room for patching object code
```

```
190 [
200 OPTC
210 .buffer EQUS message$+" but that's all for now"
220 EQUB 13\message terminator
230 ]
240 NEXT
250 REM test
260 CALL entry
```

Each BASIC element will now be described in turn, followed by a summary of the assembly language elements.

O.6 The BASIC elements of an assembly language program

Reserving memory space for machine code

The machine code generated by the assembly process is stored in memory, at locations indicated by P% (see below). The assembler does not set memory aside for this purpose automatically, so you must reserve sufficient memory yourself. To do this, put DIM <variable> <bytes> near the start of your program.

The variable used should be numeric, but not P% or O%. This variable will be used to set P% (or O% if needed) as described below.

<bytes> specifies the space to reserve for your machine code program, including memory locations used for data storage (EQUB etc). As an approximate guide, allocate 3 bytes for each assembler statement and directive.

Note that although the machine code will be stored at the location specified, it can be assembled to execute at a different location – see below.

Other methods can be used to reserve space, such as moving the HIMEM or PAGE pseudovariables, but these methods have no advantages and several disadvantages for the assembly process. These methods are best used to reserve space for machine code programs to be loaded into BASIC programs – see chapter O.8.

Initialising external variables

The assembler allows the use of BASIC variables as addresses or data in instructions and assembler directives. By initialising variables in BASIC, you can pass information such as the addresses of MOS or filing system entry points to the assembly process in a simple and self-evident manner. Using external variables in this way simplifies making changes to the program and improves its documentation.

For example, if your machine code calls OSASCI (&FFE3), instead of using JSR &FFE3 in your assembly language, you can put osasci=&FFE3 in the BASIC code then use JSR osasci in the assembly language – the machine code produced in each case will be the same.

This is the only method of setting assembler 'labels' to specific values – the assembler does not itself provide such a mechanism. (Although labels declared within the assembly language as .<label> are assigned the current value of P% automatically, this is not directly under your control.)

Implementing passes required

In programs that use labels, the assembler will normally need to process the source program twice to find the value of labels which are defined later in the program, known as forward references. This is most conveniently done using a FOR...NEXT loop, as this also allows you to set the assembler switch OPT automatically. On the first pass, bits 0 and 1 of OPT should be set to 0 to allow the assembler to ignore 'errors' caused by forward references. On the second pass, bits 0 and 1 should be set to 1 so that the assembly process can be checked. eg

```
FORZ%=Ø TO 3 STEP 3
[
OPT Z%
...
]
NEXT
```

Note that P% (and O% if used) must be initialised before each pass, in BASIC ie before the [that activates the assembler.

Setting memory pointers

Before entering the assembler, you should set P% to the first location to be used to store the object program, reserved using DIM <variable> <bytes> (see above). If you are using multiple passes of the assembler, you will have to set P% to this value before the start of each pass as P% is incremented after each byte of machine code is written to memory.

Machine code instructions will normally be assembled to run at the address which is the current value of P% and also be stored at that location in memory. To store the program at one location but assemble it so that it will run at another, set P% to the address the program will start at when run, set O% as described for P% above, then set bit 2 of the assembler switch OPT after entering the assembler. This instructs the assembler to use O% as the start address for storing the generated machine code, but P% *as the start address for generating label values*. Again, O% and P% must be set to these values before the start of each assembly pass.

If you want to run the object program in a variety of environments (ie not just with the program that assembles it), you will need to assemble it outside the main memory used by languages, editors etc for workspace. For further details, see chapter O.8.

Running, saving or moving the object program

Once the program has assembled correctly, it can be run, saved or moved as

required. Labels declared within the assembly code (.label) are BASIC numeric variables obeying normal rules of scope – they may be read or even written (eg label=2) as if they had been declared as BASIC variables, without affecting the machine code. (For further details, see below.)

Setting bit 2 of the assembler switch OPT alters the significance of P% and the label variables (see below), so running, saving and moving the machine code is described separately for the two settings.

Bit 2 of OPT clear

P% indicates the next free location in memory after the end of the machine code and the labels indicate the addresses at which the subsequent instruction was assembled.

To execute the program from BASIC, use CALL <entry> or <numeric>=USR (<entry>), where <entry> is a numeric variable (usually a label declared within the assembly language program) or constant. See Section L for further details. You may also execute the program with *GO <entry> where <entry> is a hexadecimal constant.

To save the program, use the filing system command *SAVE. You will usually need the OSCLI statement to specify the parameters to *SAVE as they will be held as BASIC variables eg

```
OSCLI "SAVE prog2 "+STR$~start+" "+STR$~(P%-start)+" "+STR$~entry
```

See Section L (BASIC Syntax) for details of OSCLI and Section H for details of *SAVE.

The program should not be moved as it will only execute at a different address if it is entirely relocatable.

Bit 2 of OPT set

P% and the labels refer to locations that the code was assembled for, not to the locations it resides at in memory. O% indicates the next location in memory free after the end of the machine code.

If the program has been assembled with O% and P% set to different addresses, it should be moved from its current start address (the initial value of O%) to the correct address (the initial value of P%) before you attempt to run it. The most convenient method to use is to *SAVE it from the current address and then *LOAD it to the correct address. This will not be necessary if the code is entirely relocatable ie does not use any instructions that address locations within the code, other than branch instructions (BRA, BNE etc).

To save the program, use the filing system command *SAVE with a reload address so that it will run at the intended address when loaded. You will

usually need the OSCLI statement to specify the parameters to *SAVE as they will be held as BASIC variables. For example, if the program was assembled to run correctly if loaded at run% but stored in memory starting at memory%, to save it so it will run at the intended location:

```
OSCLI "SAVE prog "+STR$~memory%+" "+STR$~O%+" "+STR$~entry+" "+STR$~
run%
```

Example BASIC code

```
10REM fast alpha sort
20DIM data$(100), ptr%(100)
30DIM code% 250:REM reserve memory space for machine code
40osasci=&FFE3:OSBGET=&FFD7:OSBPUT=&FFD4:REM initialise external lab
els
50tab%=9:ret%=13:REM constants
60start%=&E00:REM reload address at PAGE
100FOR opt%=0 TO 3 STEP 3:REM implement passes required
110  O%=code%:P%=start%:REM set memory pointers
```

enter assembler
set assembler switch (OPT)*
assembly language statements
leave assembler

} ASSEMBLY LANGUAGE

```
540NEXT
550OSCLI "SAVE sort " + STR$~code% + STR$~O% + STR$~entry + STR$ ~sta
rt%:REM save program to run at reload
560END
```

O.7 The assembly language elements of an assembly language program

Entering the assembler

The assembler is entered when the BASIC interpreter executes a statement which has [as the first character. The assembler then assembles subsequent assembly language statements until it reads a] as the first character in an assembly language statement, when the BASIC interpreter is re-entered and then executes subsequent statements as normal.

On entering the assembler, `OPT` is set to 3.

Setting the assembler switch – listing, error handling and memory usage

Certain facilities of the assembler are controlled by a 3 bit switch, set using the assembler directive `OPT`. `OPT` defaults to a setting of 3 each time the assembler is entered – to change the setting, place an `OPT`<n> statement at the start of each segment of assembly language source. Setting individual bits in <n> has the following effect:

Listing

With bit 0 of `OPT` set (i.e. `OPT`=1, 3, 5 or 7), each statement is listed in full as it is assembled as:

<run time location> <opcode> <address> <statement>

eg

```
2C00 A2 FF    try LDX #-1 \prepare loop counter
```

Error handling

The majority of errors will cause the assembler to abort and report the error, but with bit 1 of `OPT` reset (i.e. `OPT`=0, 1, 4 or 5), the `Branch out of range` and `No such variable` errors will be ignored. This should be used on the first pass of an assembly using forward referenced labels to avoid spurious error reporting. For a full description of error handling and the assembler, see Section Q.

Memory usage

Instructions are normally assembled to run correctly if stored at the current

setting of P% and are stored at this location. With bit 2 of OPT set (i.e. OPT=4, 5, 6 or 7), instructions are assembled to run at P%, but actually loaded into memory at O%.

An OPT statement is valid anywhere within an assembly language program, affecting the assembly of subsequent statements. In particular, OPT can be used to list only certain parts of the program.

The syntax of assembler programs

The syntax of the components that you may use in assembly language programs is summarised below. For further details of mnemonics and assembler directives, see Section P.

An assembly language program consists of:

```
[
<lines>
]
```

Where <lines> consist of:

<line number>|<statement>[|:<statement> etc|

Where <line number> is as defined for BASIC and <statement> is of the form:

[.<label> etc|<spaces><instruction>[<spaces><ignored>||

<label> is any legal BASIC numeric variable (eg start%, print, arr(4,2))
<spaces> is one or more spaces
<ignored> is any text excluding :
<instruction> is one of the following:

<mnemonic>[<address>][\<comment>]
<directive><data>[\<comment>]
OPT<switch>[\<comment>]

<mnemonic> is one of the 3-character names used for the 65C12 instructions, listed in Section P. These names may be in upper or lower case.

<directive> is one of EQUB, EQUW, EQUD, EQUS (see Section P)

<data> is a constant or BASIC expression which resolves to an integer result for EQUB, EQUD and EQUW or to a string for EQUS. The expression may use any BASIC variables including A, X, Y, a, x or y.

<switch> is a constant or BASIC expression that resolves to an integer result in the range 0-7.

<address> is described in full in Section P. If the address includes a numeric

part, this is a constant or BASIC expression which resolves to an integer result. This expression may use variables named `A`, `X`, `Y`, `a`, `x` or `y` as long as they are not in a position where they might be confused with the registers. The entire expression should not be enclosed in brackets, as these have a special significance to the assembler (indicating indirect addressing).

Note that BASIC statements are not accepted by the assembler, but BASIC expressions are accepted as data or address fields, being evaluated at *the time of assembly*. If multiple passes are used, the value used in the object program will be that calculated on the last pass.

Labels and address evaluation

Assembly language labels are normal BASIC numeric variables (floating point, or integer if they end with %) generated by the assembler when it reads .<name> as the first non-space text in an assembly language statement. The label (variable) is assigned the current value of `P%` (the program location counter) ie the (runtime) address of the instruction or assembler directive that followed it.

Labels can be declared multiply within an assembly language program, but this is rarely desirable or what is intended.

Since labels are BASIC variables, they follow the usual rules of scope – unless they are declared as `LOCAL` within an assembly procedure or function, the label value will replace any previous value for that variable name, and the labels will be stored permanently, taking up memory space.

The address field of assembler statements may include a numeric part, which is evaluated as a normal BASIC expression, so that addresses may include any of the following:

- numeric constants (eg &3000 or 4) – note that these are taken as decimal unless preceeded by &
- assembler labels
- BASIC numeric variables (external labels)
- numeric expressions (eg `P%+5`, `(data1-data2)*2`)
- numeric functions (eg `FNcond(flag)`)

Using variables as addresses allows you to specify constant addresses outside the source program (such as MOS routine entry points) in the supporting BASIC code – effectively defining external labels. This method can also be used to link separately assembled machine code modules together so that they can call each other.

For a description of the rules governing BASIC variables and expressions, see Section K.

Conditional assembly

Conditional assembly is a method of making the machine code produced depend on the results of a test made at the time of assembling. In this way a single assembly language program can generate machine code to cater for different requirements (such as inserting error checking code or not), under the control of BASIC variables. The test can either be made within the assembly language program (Method 1), or outside it in the supporting BASIC code (Method 2):

Method 1

```
80 check=1:REM insert error checking code
90 FOR C=0 TO 2 STEP 2
100 [ OPT C: <first code>
110 EQUS FNcheck(check)
120 <next code>]
130 NEXT
.
.
200 END
210 DEF FNcheck(switch)
220 IF switch THEN [OPT C : <check code>]
230 =" "
```

Method 2

```
80 check=1:REM insert error checking code
90 FOR C=0 TO 2 STEP 2
100 [ OPT C: <first code]
110 IF check THEN PROCcheck
120 [ OPT C: <next code>]
130 NEXT
.
.
200 END
210 DEF PROCcheck
220 [OPT C : <check code>]
230 ENDPROC
```

Note carefully the differences in the two methods. Method 1 uses an unnecessary `EQUS` assembler directive to introduce the conditional code using `FNcheck`. Both methods bracket the conditionally assembled code with `[OPT C ...]` since at the point the procedure/function is entered, it is the BASIC interpreter which is reading the program, not the assembler.

Assembler macros

An assembler macro is a named group of assembler statements, which once defined can be included in the assembly language program as many times as wanted by using the macro's name. The macro name is effectively an abbreviation for the statements in the macro definition.

Macros can be implemented in two ways, similarly to conditional assembly:

Method 1

```
90 FOR C=0 TO 2 STEP 2
100 [ OPT C: <code>
110 EQUS FNerrmess(1,"Error one")
120 <code>
130 EQUS FNerrmess(2,"Error two")
140 <code>]
130 NEXT
.
.
200 END
210 DEF FNerrmess(number,text$)
220 [OPT C : BRK:EQUB number:EQUS text$:EQUB 0]
230 =" "
```

Method 2

```
90 FOR C=0 TO 2 STEP 2
100 [ OPT C: <code>]
110 PROCerrmess(1,"Error one")
120 [ OPT C: <code>]
130 PROCerrmess(2,"Error two")
140 [ OPT C: <code>]
150 NEXT
.
.
200 END
210 DEF PROCerrmess(number,text$)
220 [OPT C:BRK:EQUB number:EQUS text$:EQUB 0]
230 ENDPROC
```

O.8 The machine code environment

Programs written in a high level language run in an environment that checks for runtime errors, automatically allocates and de-allocates memory needed for programs and data and provides simple interfaces with filing systems and the MOS. In this way, programmers using high level languages need not concern themselves with low level aspects of the system.

The environment in which machine code programs run is much less controlled. The following information and suggestions should help you to write programs that are not only logically correct, but will also run correctly in the environment of the BBC Microcomputer.

Reserving memory

Machine code programs use memory for three main reasons – to store the program, to store data and variables used by the program and to implement the microprocessor's stack. Each will now be described in turn:

For the program:

If the machine code is being used only with the BASIC program that generates it, it can be executed at the locations the assembler uses to store it in memory.

If the program will not always be run with the program that generates it (eg utilities), it must be assembled to run in an area of memory that will always be free when you want to use it.

If the program is allowed to corrupt the current memory contents (eg a game, language compiler, spelling checker), it can simply be assembled to run at some convenient location in the language workspace (such as OSHWM, usually &0E00) and always loaded and run there.

If the program is not allowed to corrupt the current memory contents (eg a compressor for BASIC programs, editor utility) it can be run in &DD00-&DEFF, the transient program area. The transient program area is quite small (512 bytes), will be overwritten if you use utilities such as `*AFORM` and `*VERIFY` on the ADFS utilities disk and requires use of extended vectors if you want to use it for interrupt service routines, but it is the best area to use for small utility programs.

If the program can be made entirely relocatable, it can be *LOADed to whatever area of memory is free then run with `*GO` <entry>.

Otherwise, the MOS memory pointers can be changed to keep an area of memory away from the language and the program must then always be loaded and run at that location. The BASIC language provides the pseudo-variables PAGE and HIMEM which can be changed to reserve space at the bottom or top of the language workspace (eg PAGE=PAGE+&100 reserves 256 bytes, usually starting at &0E00). If the language does not provide such a facility, use OSBYTE 131 to increase OSHWM (see Section D) then initialise the desired language (by **CTRL BREAK** or selecting it with *name).

For data and variables:

Reserving space for data and variables which reside in locations other than page zero (&0000-&0100) is accomplished in exactly the same way as space for the program itself. This can usually be done during the assembly by increasing P% (and O% if used) or using the EQUB, EQUW, EQUD and EQUS assembler directives.

Locations &0070-&008F are reserved for use by assembly language programs so they can be used freely in your programs. Locations &0000-&006F are allocated to the current language in a single processor (or the language processor of a co-processor system), but are available if the language is not being used. If you are using a co-processor, these locations in the I/O processor are used by the Tube.

Locations &0090-9F are allocated to Econet, but are available if Econet is not being used. Other locations are used by filing systems and the MOS and should not be interfered with. Page zero locations outside &70-&8F may be altered by a hard reset (**CTRL BREAK**). For details of the co-processor memory map, refer to the co-processor user guide.

The stack:

The stack is a portion of memory (&100-&1FF) used by the 65C12 for the temporary storage of data (such as return addresses from subroutines). No special action is needed to initialise or protect the stack. Return from a machine code program to the current language (via RTS or BRK) is all that is necessary to restore normal stack usage.

Using the MOS

The BBC Microcomputer system provides a wide range of operating system facilities which can be used easily from machine code. They are used by calling entry points in page &FF (ie JSR &FFnn). For details of using the MOS from machine code, see Sections B to F. The following is a brief summary of the facilities available:

– executing *FX commands;

- executing * commands;
- reading from the input stream;
- reading from a paged ROM;
- reading graphics cursor co-ordinates;
- reading character definitions;
- reading/writing I/O processor memory;
- reading/writing the clock/calendar, system clock and interval timer;
- reading/writing the VDU palette;
- writing bytes to the output stream;
- writing the sound queue;
- defining sound envelopes;
- generating events.

Using filing systems

Filing system routines are used by calling entry points in page &FF (ie JSR &FFnn). For details of using filing systems from machine code, see Sections G to J. The following is a brief summary of the facilities available.

- opening and closing files;
- loading and saving files;
- loading and saving data about files;
- loading and saving blocks of characters;
- loading and saving bytes.

Using BASIC's floating point routines

It is possible to call certain of BASIC's internal routines through addresses held in page &700. The addresses and their contents are

Address	Contents	
&7FO	Address of SQR routine	(facc = SQR(facc))
&7F2	Address of / routine	(facc = [argp] / facc)
&7F4	Address of * routine	(facc = [argp] * facc)
&7F6	Address of + routine	(facc = [argp] + facc)
&7F8	Address of unary -	(facc = -facc)
&7FA	Address of lda routine	(facc = [argp])
&7FC	Address of sta routine	([argp] = facc)
&7FE	Address of argp in zero page	
&7FF	Address of facc in zero page	

where facc is the floating point accumulator and argp is the address of the 'argument pointer', ie the address of a floating point variable to be acted upon.

O.9 Using machine code programs

Machine code programs can be used in a wide variety of ways, as described in other parts of this guide. The following list is a simple survey:

Executing programs in memory from BASIC

`CALL` <entry>[`,`<variable> etc.] – passes optional variables and `A%`, `X%`, `Y%`, and `C%`

<numeric>=`USR`(<entry>) – passes `A%`, `X%`, `Y%` and `C%`; returns A, X, Y and status registers

Executing programs in memory

`*GO` <entry> – run program in single or language processor

`*GOIO` <entry> – run program in the I/O processor

Executing programs stored in a filing system

`*`<name> – load and run the named file, using the locations defined when it was saved

`*RUN` <name> – load and run the named file, using the locations specified when it was saved (or using other locations in the command)

Linking programs in memory to the MOS

The MOS provides a facility for the user to access his own routines through the two `*` commands `*CODE` and `*LINE`. The user routine should be vectored from &0200−1. When a `*CODE` <n1>`,`<n2> command is executed, this routine will be entered with A=0, X=<n1>, Y=<n2>. When a `*LINE` <text> command is executed, this routine will be entered with A=1, X=<string address LSB>, Y=<MSB>; the string will terminate in ASCII 13.

P Assembler keywords

P. 1 Introduction to assembler keywords

The keywords recognised by the assembler are the instruction mnemonics (eg `LDA`, `PLP`, `RTS`) and the assembler directives (`OPT`, `EQUB`, `EQUW`, `EQUD`, `EQUS`). Mnemonics are assembled into machine code instructions; assembler directives are assembled into data or change the behaviour of the assembler.

This section describes the mnemonics and addressing modes, assembler directives and data fields supported by the BASIC IV assembler. The syntax required by the assembler and other details of the assembly process are described in the previous Section.

This information is presented in three parts:

- Tables relating mnemonic+addressing mode to the machine code (opcode) generated, and mnemonics to addressing modes and action
- An explanation of the addressing modes provided
- Individual entries for each mnemonic and directive in alphabetical order, giving its addressing modes and detailing its action

P. 2 Table of operation codes

MNE-MONIC	OPERATION	IMMEDIATE OP	n	#	ABSOLUTE OP	n	#	ZERO PAGE OP	n	#	(4) IMPLIED OP	n	#	(IND. X) OP	n	#	(1) (IND). Y OP	n	#	ZPG.X OP	n	#	(1) ABS.X OP	n	#	(1) ABS.Y OP	n	#	RELATIVE (2) OP	n	#	INDIRECT OP	n	#	ZPG.Y OP	n	#	PROCESSOR STATUS CODE 7 6 4 3 2 1 0 N V B D I Z C	MNE-MONIC
ADC	A + M + C → A (3)	69	2	2	6D	4	3	65	3	2				61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3				72	5	2				N V • • • Z C	ADC
AND	A∧M → A	29	2	2	2D	4	3	25	3	2				21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3				32	5	2				N • • • • Z •	AND
ASL	C ← [7 0] ← 0				0E	6	3	06	5	2	0A	2	1							16	6	2	1E	6	3													N • • • • Z C	ASL
BCC	BRANCH IF C=0																												90	2	2							• • • • • • •	BCC
BCS	BRANCH IF C=1																												B0	2	2							• • • • • • •	BCS
BEQ	BRANCH IF Z=1																												F0	2	2							• • • • • • •	BEQ
BIT	A ∧ M (5)	89	2	2	2C	4	3	24	3	2										34	4	2	3C	4	3													M₇ M₆ • • • Z •	BIT
BMI	BRANCH IF N=1																												30	2	2							• • • • • • •	BMI
BNE	BRANCH IF Z=0																												D0	2	2							• • • • • • •	BNE
BPL	BRANCH IF N=0																												10	2	2							• • • • • • •	BPL
BRA	BRANCH ALWAYS																												80	2	2							• • • • • • •	BRA
BRK	BREAK										00	7	1																									• • 1 0 1 • •	BRK
BVC	BRANCH IF V=0																												50	2	2							• • • • • • •	BVC
BVS	BRANCH IF V=1																												70	2	2							• • • • • • •	BVS
CLC	0 → C										18	2	1																									• • • • • • 0	CLC
CLD	0 → D										D8	2	1																									• • • 0 • • •	CLD
CLI	0 → I										58	2	1																									• • • • 0 • •	CLI
CLV	0 → V										B8	2	1																									• 0 • • • • •	CLV
CMP	A-M	C9	2	2	CD	4	3	C5	3	2				C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3				D2	5	2				N • • • • Z C	CMP
CPX	X-M	E0	2	2	EC	4	3	E4	3	2																												N • • • • Z C	CPX
CPY	Y-M	C0	2	2	CC	4	3	C4	3	2																												N • • • • Z C	CPY
DEC	DECREMENT				CE	6	3	C6	5	2	3A	2	1							D6	6	2	DE	6	3													N • • • • Z •	DEC
DEX	X-1 → X										CA	2	1																									N • • • • Z •	DEX
DEY	Y-1 → Y										88	2	1																									N • • • • Z •	DEY
EOR	A∨M → A	49	2	2	4D	4	3	45	3	2				41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3				52	5	2				N • • • • Z •	EOR
INC	INCREMENT				EE	6	3	E6	5	2	1A	2	1							F6	6	2	FE	6	3													N • • • • Z •	INC
INX	X + 1 → X										E8	2	1																									N • • • • Z •	INX
INY	Y + 1 → Y										C8	2	1																									N • • • • Z •	INY
JMP	JUMP TO NEW LOC				4C	3	3							7C	6	3																6C	6	3				• • • • • • •	JMP
JSR	JUMP SUB				20	6	3																															• • • • • • •	JSR
LDA	M → A	A9	2	2	AD	4	3	A5	3	2				A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3				B2	5	2				N • • • • Z •	LDA
LDX	M → X	A2	2	2	AE	4	3	A6	3	2																BE	4	3							B6	4	2	N • • • • Z •	LDX
LDY	M → Y	A0	2	2	AC	4	3	A4	3	2										B4	4	2	BC	4	3													N • • • • Z •	LDY
LSR	0 → [7 0] → C				4E	6	3	46	5	2	4A	2	1							56	6	2	5E	6	3													0 • • • • Z C	LSR
NOP	NO OPERATION										EA	2	1																									• • • • • • •	NOP
ORA	A∨M → A	09	2	2	0D	4	3	05	3	2				01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3				12	5	2				N • • • • Z •	ORA
PHA	A → Ms S-1 → S										48	3	1																									• • • • • • •	PHA
PHP	P → Ms S-1 → S										08	3	1																									• • • • • • •	PHP
PHX	X → Ms S-1 → S										DA	3	1																									• • • • • • •	PHX
PHY	Y → Ms S-1 → S										5A	3	1																									• • • • • • •	PHY
PLA	S+1 → S Ms → A										68	4	1																									N • • • • Z •	PLA
PLP	S+1 → S Ms → P										28	4	1																									N V 1 D I Z C	PLP
PLX	S+1 → S Ms → X										FA	4	1																									N • • • • Z •	PLX
PLY	S+1 → S Ms → Y										7A	4	1																									N • • • • Z •	PLY
ROL	[C ← [7 0] ← C]				2E	6	3	26	5	2	2A	2	1							36	6	2	3E	6	3													N • • • • Z C	ROL
ROR	[C → [7 0]]				6E	6	3	66	5	2	6A	2	1							76	6	2	7E	6	3													N • • • • Z C	ROR
RTI	RTRN INT										40	6	1																									N V 1 D I Z C	RTI
RTS	RTRN SUB										60	6	1																									• • • • • • •	RTS
SBC	A-M-C̄ → A (3)	E9	2	2	ED	4	3	E5	3	2				E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3				F2	5	2				N V • • • Z C	SBC
SEC	1 → C										38	2	1																									• • • • • • 1	SEC
SED	1 → D										F8	2	1																									• • • 1 • • •	SED
SEI	1 → I										78	2	1																									• • • • 1 • •	SEI
STA	A → M				8D	4	3	85	3	2				81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3				92	5	2				• • • • • • •	STA
STX	X → M				8E	4	3	86	3	2																											96	4	2	• • • • • • •	STX
STY	Y → M				8C	4	3	84	3	2										94	4	2																	• • • • • • •	STY
STZ	00 → M				9C	4	3	64	3	2										74	4	2	9E	5	3													• • • • • • •	STZ
TAX	A → X										AA	2	1																									N • • • • Z •	TAX
TAY	A → Y										A8	2	1																									N • • • • Z •	TAY
TRB	Ā∧M → M (6)				1C	6	3	14	5	2																													• • • • • Z •	TRB
TSB	A∨M → M (6)				0C	6	3	04	5	2																													• • • • • Z •	TSB
TSX	S → X										BA	2	1																									N • • • • Z •	TSX
TXA	X → A										8A	2	1																									N • • • • Z •	TXA
TXS	X → S										9A	2	1																									• • • • • • •	TXS
TYA	Y → A										98	2	1																									N • • • • Z •	TYA

Notes

1. Add 1 to "n" if page boundary is crossed, except STA and STZ.
2. Add 1 to "n" if branch occurs to same page. Add 2 to "n" if branch occurs to different page.
3. Add 1 to "n" if decimal mode.
4. Accumulator address is included in Implied address.
5. "N" and "V" flags are unchanged in immediate mode.
6. "Z" flag indicates A∧M result (same as BIT instruction)

X	Index X	+	Add
Y	Index Y	-	Subtract
A	Accumulator	∧	And
M	Memory per effective address	∨	Or
Ms	Memory per stack pointer	∨-	Exclusive or

n	No. Cycles
#	No. Bytes
M_6	Memory Bit #6
M_7	Memory Bit #7

P. 3 65C12 addressing modes

Instructions in 65C12 assembly language consist of two fields – an instruction mnemonic and an address field which may be blank. (Label and comment fields are not actually part of the instruction and are described in the preceding section.) The mnemonic specifies the action to be performed and must be one of the three-character names listed at the end of this section. The address field indicates the element (register, data or memory location) to use and/or change in performing that action.

The address field may contain a valid numeric expression in BASIC which evaluates to a single 8 bit or 16 bit constant. After replacing the expression with this constant, the resulting address field determines the addressing mode, as shown below. If the address field is not one of the following or if it is one of the following but is not a valid addressing mode for the instruction mnemonic, it will be reported as an error by the assembler.

address field	**addressing mode**
blank	implied
A or a	accumulator
#<8 bit number>	immediate
<16 bit number>	absolute
<8 bit number>	zero page
(<8 or 16 bit number>)	absolute indirect
(<8 bit number>)	zero page indirect*
<16 bit number>,X or ,Y	absolute indexed
<8 bit number>,X or ,Y	zero page indexed
(<8 bit number>,X)	pre-indexed indirect
(<8 bit number>),Y	post-indexed indirect
(<16 bit number>,X)	pre-indexed absolute indirect*
<8 or 16 bit number>	relative

*These addressing modes were not available on the 6502

The addressing modes will now be described in turn, after a brief description of how the processor handles illegal calculated addresses and an explanation of the significance of zero page and absolute addressing.

Illegal addresses

Many of the addressing modes used by the 65C12 only resolve to an effective address at the time of execution, when the microprocessor calculates them by adding variable and constant portions (eg <n>,X means the address <n+X>).

This can result in illegal addresses which seem to extend beyond the address range allowed for that instruction ie beyond &FF for page zero addresses or &FFFF for absolute addresses. These are not flagged by the assembler, because it cannot anticipate the values the variable portion will take when the program is run.

In fact, the microprocessor is completely oblivious to this apparent problem, as it ignores any carry beyond the allowed address range. Thus if an address of &0102 is generated for an instruction which was assembled to address the zero page; &02 is used. Similarly absolute addresses such as &1002E will be used as &002E. This can have particularly confusing results if the generated address is used to read another address from memory (indirect addressing), as the first byte read may be from &FFFF (or &FF) and the second read from &0000 (or &00)!

You should be aware of this and design your programs accordingly.

Absolute and zero page addresses

Any location in the microprocessor's memory map can be specified using a 16 bit integer address (in the range &0000–&FFFF or 0–65535). Many addressing modes use such 16 bit addresses, which are termed absolute addresses since they specify the location without reference to the instruction's location.

Zero page addresses are locations in page zero of memory, or &0000–&00FF, which can be addressed using zero page addressing, specified by an 8 bit integer address (&00–&FF). These locations are treated almost as an extra bank of registers by the microprocessor. Zero page addressing is quicker than absolute addressing (since only a 2 byte address needs to be decoded instead of up to four bytes) and can be used for a wider variety of addressing modes than absolute addressing.

If the assembler has a choice between using zero page addressing or absolute addressing (ie the address constant is in the range &0000–&00FF and the instruction can use either addressing mode), it will choose the former, as the machine code generated is shorter and executes more quickly.

Implied addressing

Implied addressing is indicated by the instruction mnemonic itself. Instructions using implied addressing include an implicit specification of the address in the mnemonic ie there is no address field in the instruction. Examples are `RTS`, `PHP`, `SEC`, `TAX`.

Accumulator addressing

Accumulator addressing is indicated by an address field consisting of A. Instructions using accumulator addressing manipulate the contents of the

accumulator rather than memory locations. The only valid uses of this addressing mode are `ASL A`, `ROL A`, `ROR A`, `LSR A`, `DEC A` and `INC A`.

Immediate addressing

Immediate addressing is indicated by an address field consisting of # followed by a single byte. This byte is the data used by the instruction ie the address used is of the next byte after the instruction. Examples are `LDA #1`, `BIT #&80`, `CMP #ASC"!"`.

Absolute addressing

Absolute addressing is indicated by an address field consisting of a 2 byte integer and an instruction which cannot use relative addressing. This integer specifies the location for the instruction to address, which may be anywhere in memory although the assembler will normally assemble the instruction to use zero page addressing in preference to absolute addressing if the address is in the range &0000–&00FF. Examples are `LDA &1000`, `SBC data1` (where data1 evaluates to >&FF)

Zero page addressing

Zero page addressing is indicated by an address field consisting of a 1 byte integer. This integer specifies the location for the instruction to address, in the zero page of memory (ie &0000–&00FF). Examples are `STA &40`, `SBC data1` (where data1 evaluates to <&100)

Absolute indirect addressing

Absolute indirect addressing is indicated by an address field consisting of a 2 byte integer surrounded by brackets ie (&nnnn). The address used is stored at the address in memory specified by the integer, as <LSB> <MSB>. If the address field is (&1000), &20 is stored at &1000 and &0E at &1001 then the address used is &0E20. The only instruction using this mode is `JMP`, for example: `JMP (400)`, `JMP` (table+offset).

(There was a bug in the 6502 which meant that if the address field was (&nnFF), the <LSB> would be read from &nnFF and the <MSB> from &nn00. This has been corrected in the 65C12.)

Zero page indirect addressing

Zero page indirect addressing is indicated by an address field consisting of a 1 byte integer surrounded by brackets. The address used is that stored at the zero page location specified by the integer, as <LSB> <MSB>. For example, if the address field is (&30), &1E is stored at &30 and &1F at &31 then the

address used is &1F1E. Examples are `LDA (Ø)`, `CMP (&FE)`, `SBC (ptr1)` (where ptr1 is less than &100).

Zero page indirect addressing is a new mode, not available on the 6502.

Absolute indexed addressing

Absolute indexed addressing is indicated by an address field consisting of a 2 byte integer followed by ,X or ,Y. The address used is calculated when the instruction is executed by adding X or Y (as indicated by ,X or ,Y) to the 2 byte integer. For example if the address field is &2000,X and when the instruction is executed X=44&1F, the address used is &201F. This addressing mode allows you to address a sequence of memory locations by including it in a loop with an instruction to increment/decrement X or Y. Examples are `BIT &2ØØ,X`, `SBC buffer,Y` (where buffer is greater than &FF).

Zero page indexed addressing

Zero page indexed addressing is indicated by an address field consisting of a 1 byte integer followed by ,X or ,Y. (The Y indexed instruction is valid only for instructions to load or store X.) The address used is calculated when the instruction is executed by adding X or Y (as indicated by ,X or ,Y) to the 1 byte integer. This addressing mode allows you to address a sequence of memory locations by including it in a loop with an instruction to increment/decrement X or Y. Examples are `ROL Ø,X` and `LDX &FF,Y`.

Pre-indexed indirect addressing

Pre-indexed indirect addressing is indicated by an address field of (<1 byte integer>,X). The address used is that stored in page zero at <integer>+X as <LSB>, <MSB>. For example, if the address field is (&30,X), X=&6F when the instruction is executed, and &10 is stored at &009F and &FF at &00A0 then the address used will be &FF10. This addressing mode allows you to select the address from a table of addresses stored in zero page. Note than the Y register cannot be used instead of X. Examples are `LDA (&ØØ,X)`, `CMP (&FE,X)`.

Post-indexed indirect addressing

Post-indexed indirect addressing is indicated by an address field of (<1 byte integer>),Y. The address used is that stored in page zero at <integer> as <LSB>, <MSB> plus Y. For example, if the address field is (&30),Y, Y=&6F when the instruction is executed, and &13 is stored at &0030 and &FF at &0031 then the address used will be &FF13+&6F. This addressing mode allows you to select the address as a variable offset from a single address stored in page zero eg in processing entries in a buffer. Note than the X register cannot be used instead of Y. Examples are `SBC (Ø),Y`, `AND (&C1),Y`.

Pre-indexed absolute indirect addressing

Pre-indexed absolute indirect addressing is indicated by an address field of (<2 byte integer>,X). The address used is that stored at <integer>+X in memory as <LSB>, <MSB>. For example, if the address field is (&3000,X), X=&6F when the instruction is executed, and &13 is stored at &306F and &FF at &3070 then the address used will be &FF13. This addressing mode is only used with the JMP instruction. It allows you to select the jump address from a table of addresses stored in memory. Note that the Y register cannot be used instead of X. Examples are `JMP (ostable,X)`, `JMP (&FEEE,X)`.

Relative addressing

Relative addressing is indicated by using one of the branch instructions, which should be followed by a 2 byte integer. The integer specifies the location to which a branch should be made, which should be within +129 bytes or −126 bytes of the start of the branch instruction. Examples are `BRA loop`, `BNE P%-4`, `BEQ P%+messlen`.

P. 4 Assembler keywords

The keywords that the assembler recognises are presented in alphabetical order, using the following conventions:

** means a new instruction or addressing mode, not available on the 6502.

++ means an addressing mode that was available on the 6502, but not with the instruction in question.

The addressing modes are explained in the previous chapter.

The letters under the heading 'Flags affected' for each mnemonic indicate the flags in the 65C12 status register affected by the instruction:

Flag	**Status register bit**	**meaning**
N	7	negative flag
V	6	overflow flag
	5	unused
B	4	BRK flag
D	3	decimal flag
I	2	interrupt flag
Z	1	zero flag
C	0	carry flag

ADC add memory and carry to accumulator

Description

Add the contents of the memory location to the accumulator. If the carry bit is set before the instruction, 1 is also added.

Flags affected

NVZC

Addressing modes

Immediate
Absolute
Zero page
Zero page indirect**
Absolute indexed
Zero page indexed
Pre-indexed indirect
Post-indexed indirect

Comments

To add without the carry bit, use `CLC` before the `ADC` instruction.

AND — AND memory with accumulator

Description

Perform a logical AND of the contents of the memory location with the accumulator, storing the results in the accumulator.

Flags affected

NZ

Addressing modes

Immediate
Absolute
Zero page
Zero page indirect**
Absolute indexed
Zero page indexed
Pre-indexed indirect
Post-indexed indirect

Comments

ANDing resets all bits in the accumulator except those that were set both in the accumulator and the data. For example:

```
        00110010 binary
AND     10011101 binary
gives   00010000
```

ASL — arithmetic shift left

Description

Shift the contents of the memory location one bit to the left, placing a zero in bit 0 and bit 7 in the carry.

Flags affected

NZC

Addressing modes

Accumulator
Absolute
Zero page
Absolute indexed (X only)
Zero page indexed (X only)

Comments

This is effectively a multiply by 2 instruction, with any overflow in C.

BCC — branch if carry clear

Description

Branch if the carry flag = 0.

Flags affected

None

Addressing mode

Relative

Comments

The carry flag is changed by the following instructions `ADC`, `ASL`, `CLC`, `CMP`, `CPX`, `CPY`, `LSR`, `PLP`, `ROL`, `ROR`, `RTI`, `SBC` and `SEC`.

BCS — branch if carry set

Description

Branch if the carry flag = 1.

Flags affected

None

Addressing mode

Relative

Comments

See `BCC`

BEQ — branch if equal to zero

Description

Branch if the Z flag = 1.

Flags affected

None

Addressing mode

Relative

Comments

The Z flag is changed by the following instructions ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA and TYA.

BIT — test bits in memory with accumulator

Description

The accumulator is compared with the contents of the memory location and Z set if equal otherwise reset. For Addressing modes other than immediate, Bit 6 of the memory location is stored in the V flag and bit 7 in the N flag. Used with an immediate data, N and V are unchanged. The accumulator is unchanged.

Flags affected

NVZ

Addressing modes

Immediate++
Absolute
Zero page
Absolute indexed (X only)++
Zero page indexed++

Comments

Note the warning about immediate mode in the description.

BMI — branch if minus

Description

Branch if the N flag = 1.

Flags affected

None

Addressing mode

Relative

Comments

The N flag is changed by the following instructions ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, PLX, PLY, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA, TXS, and TYA.

BNE — branch if not equal to zero

Description

Branch if the Z flag is reset.

Flags affected

None

Addressing mode

Relative

Comments

See BEQ

BPL — branch if plus

Description

Branch if the N flag = 0.

Flags affected

None

Addressing mode

Relative

Comments

See BMI

BRA — branch always**

Description

Branch independently of status flag settings.

Flags affected

None

Addressing mode

Relative

Comments

This instruction provides a shorter, relocatable alternative to the JMP instruction when the destination address is within range.

BRK — break

Description

This instruction is a software interrupt. The contents of the program counter+2 are pushed onto the stack, followed by the status register. The program then jumps indirect via the contents of &FFFE (LSB) and &FFFF (MSB).

Flags affected

BI

Addressing mode

Implied

Comments

This instruction is trapped by the MOS and used by approved languages and filing systems to implement error trapping (see Section Q). It can also be used to implement other facilities (such as breakpointing) by intercepting the break vector see reference 1 in Appendix A for details.

BVC — branch if overflow clear

Description

Branch if the V flag = 0.

Flags affected

None

Addressing mode

Relative

Comments

The V flag is changed by the following instructions ADC, BIT, CLV, PLP, RTI and SBC. It can also be changed by a signal on one of the microprocesor pins.

BVS — branch if overflow set

Description

Branch if the V flag = 1.

Flags affected

None

Addressing mode

Relative

Comments

See BVC

CLC — clear carry flag

Description

Resets the carry flag (C) to 0.

Flags affected

C

Addressing mode

Implied

Comments

Mainly used to prepare for ADC or SBC instructions. See also SEC.

CLD — clear decimal flag

Description

Resets the decimal flag (D) to 0.

Flags affected

D

Addressing mode

Implied

Comments

This instruction should be used at the start of any routine which uses binary (rather than binary coded decimal – BCD) arithmetic or before using any MOS or filing system routines. See also SED.

Warning: the 65C12 takes one cycle longer than the 6502 to perform decimal mode arithmetic.

CLI clear interrupt mask

Description

Resets the I flag to 0, to enable maskable (IRQ) interrupts.

Flags affected

I

Addressing mode

Implied

Comments

This instruction is used to re-enable maskable interrupts after they have been disabled using SEI. These interrupts are used extensively by the MOS, to provide background processing and service peripheral devices.

CLI should only be necessary after SEI, which must be used with extreme caution, as it suspends normal functioning of the MOS. Such routines should also preserve the status register (ie PHP at start, PLP at end).

CLR clear memory

Description

This instruction is supported by the assembler as an alternative spelling of STZ. See entry for STZ.

CLV clear overflow flag

Description

Resets the overflow flag (V) to 0.

Flags affected

V

Addressing mode

Implied

Comments

See BVC and BVS.

CMP compare memory with accumulator

Description

Subtracts the contents of the memory location from the accumulator and set

the status flags accordingly. The accumulator is actually unchanged by this instruction.

Flags affected

NZC

Addressing modes

Immediate
Absolute
Zero page
Zero page indirect**
Absolute indexed
Zero page indexed
Pre-indexed indirect
Post-indexed indirect

Comments

Z is set if A=data, C is reset if A<data, C is set if A>=data.

CPX compare memory with X register

Description

Subtracts the contents of the memory location from the X register and set the status flags accordingly. The X register is actually unchanged by this instruction.

Flags affected

NZC

Addressing modes

Immediate
Absolute
Zero page

Comments

Z is set if A=data, C is reset if A<data, C is set if A>=data.

CPY compare memory with Y register

Description

Subtracts the contents of the memory location from the Y register and set the status flags accordingly. The Y register is actually unchanged by this instruction.

Flags affected

NZC

Addressing modes

Immediate
Absolute
Zero page

Comments

Z is set if A=data, C is reset if A<data, C is set if A>=data.

DEA — decrement accumulator**

Description

This instruction is supported by the assembler as an alternative spelling of `DEC A`. See entry for `DEC`.

DEC — decrement memory

Description

Subtracts 1 from the contents of the memory location, setting flags accordingly.

Flags affected

NZ

Addressing modes

Accumulator++
Absolute
Zero page
Absolute indexed (X only)
Zero page indexed (X only)

Comments

Provides a simple way of implementing counters. `DEC A` may also be typed as `DEA`.

DEX — decrement X register

Description

Subtracts 1 from the X register, setting flags accordingly.

Flags affected

NZ

Addressing mode

Implied

Comments

Provides a simple way of implementing counter/pointers.

DEY — decrement Y register

Description

Subtracts 1 from the Y register, setting flags accordingly.

Flags affected

NZ

Addressing mode

Implied

Comments

Provides a simple way of implementing counter/pointers.

EOR — exclusive OR memory with accumulator

Description

Performs a logical exclusive-OR of the contents of the memory location with the accumulator, storing the results in the accumulator.

Flags affected

NZ

Addressing modes

Immediate
Absolute
Zero page
Zero page indirect**
Absolute indexed
Zero page indexed
Pre-indexed indirect
Post-indexed indirect

Comments

Exclusive ORing resets all bits in the accumulator except those that were set in either the accumulator or the data but not in both.

eg	00110010 binary
EOR	10011101 binary
gives	10101111

EQUB — data byte assembler directive

Description

Reserves a single byte of storage, initialised by the data field.

Data field

Single byte integer

Comments

Initialised at time of assembly. Negative integers will be treated as 2's complement (eg −1 is stored as &FF) without generating an assembly error.

EQUD — data double word assembler directive

Description

Reserves 4 bytes of storage, initialised from the data field as <LSB>....<MSB>.

Data field

Four byte integer.

Comments

Initialised at the time of assembly. Negative integers will treated as 2's complement (eg −2 is stored as &FFFFFFFE) without generating an assembly error.

EQUS — data string assembler directive

Description

Reserves 0-255 bytes of storage, initialised from the data field as <first character>...<last character>.

Data field

0-255 character string.

Comments

Initialised at the time of assembly. This can be used for messages or reserving buffer space. Note that a terminator is not included by the assembler to indicate the end of the string and a null string will result in no storage being used.

EQUW — data word assembler directive

Description

Reserves 2 bytes of storage, initialised from the data field as <LSB><MSB>.

Data field

Two byte integer.

Comments

Initialised at the time of assembly. Negative integers will treated as 2's complement (eg −3 is stored as &FFFD) without generating an assembly error.

INA — increment accumulator**

Description

This instruction is supported by the assembler as an alternative spelling of INC A. See entry for `INC`.

INC — increment memory

Description

Adds 1 to the contents of the memory location, setting flags accordingly.

Flags affected

NZ

Addressing modes

Accumulator++
Absolute
Zero page
Absolute indexed (X only)
Zero page indexed (X only)

Comments

Provides a simple way of implementing counters. `INC A` may also be typed as `INA`.

INX — increment X register

Description

Adds 1 to the X register, setting flags accordingly.

Flags affected

NZ

Addressing mode

Implied

Comments

Provides a simple way of implementing counter/pointers.

INY — increment Y register

Description

Adds 1 to the Y register, setting flags accordingly.

Flags affected

NZ

Addressing mode

Implied

Comments

Provides a simple way of implementing counter/pointers.

JMP — jump

Description

Jumps to the specified location ie load the program counter with the address and continue execution.

Flags affected

None

Addressing modes

Absolute
Absolute indirect
Pre-indexed absolute indirect**

Comments

The effective address may be anywhere in memory. This is the only instruction to use absolute indirect and indexed absolute indirect addressing.

JSR — jump to subroutine

Description

The contents of the program counter plus 2 are pushed onto the stack and execution continues at the specified address.

Flags affected

None

Addressing mode

Absolute

Comments

The subroutine returns to the address after the JSR instruction when RTS is executed. All operating system and filing system calls are made using JSR.

LDA — load accumulator

Description

Loads the accumulator with the contents of the memory location.

Flags affected

NZ

Addressing modes

Immediate
Absolute
Zero page
Absolute indexed
Zero page indexed
Zero page indirect**
Pre-indexed indirect
Post-indexed indirect

Comments

Since LDA sets the N and Z flags, it can be used to test as well as load the contents of memory locations.

LDX — load X register

Description

Loads the X register with the contents of the memory location.

Flags affected

NZ

Addressing modes

Immediate
Absolute
Zero page
Absolute indexed (Y only)
Zero page indexed (Y only)

Comments

Since `LDX` sets the N and Z flags, it can be used to test as well as load the contents of memory locations.

LDY — load Y register

Description

Loads the Y register with the contents of the memory location.

Flags affected

NZ

Addressing modes

Immediate
Absolute
Zero page
Absolute indexed (X only)
Zero page indexed (X only)

Comments

Since `LDY` sets the N and Z flags, it can be used to test as well as load the contents of memory locations.

LSR — logical shift right

Description

Shifts the contents of the memory location one bit to the left, placing a zero in bit 7 and bit 0 in the carry.

Flags affected

NZC

Addressing modes

Accumulator
Absolute
Zero page
Absolute indexed (X only)
Zero page indexed (X only)

Comments

This is effectively a divide by 2 instruction (treating the data as an unsigned integer) with the remainder in C.

NOP — no operation

Description

Does nothing for two clock cycles.

Flags affected

None

Addressing mode

Implied

Comments

This instruction is used to adjust program timing, to replace deleted instructions when changing machine code or to leave room for instructions or break points to be added later to the machine code when writing assembly language programs.

OPT — assembler options switch

Description

Determines whether statements are listed as they are assembled, certain assembly errors are suppressed, and whether machine code is stored at P% or 0%, by bit settings in the data field.

Data field

3 bit integer, as follows.

bit 0=0	no assembly listing
bit 0=1	assembly listing

bit 1=0 assembler errors Branch out of range and No such variable suppressed
bit 1=1 assembler errors Branch out of range and No such variable enabled
bit 2=0 machine code placed at P%
bit 2=1 machine code placed at O%

Comments

OPT also provides a convenient way of calling macros or conditionally assembled modules, using OPT FNmacro. See Section O.

ORA — OR memory with accumulator

Description

Performs a logical OR of the contents of the memory location with the accumulator, storing the results in the accumulator.

Flags affected

NZ

Addressing modes

Immediate
Absolute
Zero page
Zero page indirect**
Absolute indexed
Zero page indexed
Pre-indexed indirect
Post-indexed indirect

Comments

ORing resets all bits in the accumulator except those that were set in either the accumulator or the data.

eg 00110010 binary
OR 10011101 binary
gives 10111111

PHA — push accumulator onto stack

Description

Places the contents of the accumulator at the stack location pointed to by the stack pointer then decrement the stack pointer.

Flags affected

None

Addressing mode

Implied

PHP — push flags register onto stack

Description

Places the contents of the flags register at the stack location pointed to by the stack pointer then decrement the stack pointer.

Flags affected

None

Addressing mode

Implied

PHX — push X register onto stack**

Description

Places the contents of the X register at the stack location pointed to by the stack pointer then decrement the stack pointer.

Flags affected

None

Addressing mode

Implied

Comments

**This instruction not available on the 6502

PHY — push Y register onto stack**

Description

Places the contents of the Y register at the stack location pointed to by the stack pointer then decrement the stack pointer.

Flags affected

None

Addressing modes

Implied

PLA pull data from stack into accumulator

Description

Place the contents of the stack location pointed to by the stack pointer in the accumulator then increment the stack pointer.

Flags affected

NZ

Addressing mode

Implied

Comments

Warning – PLA affects the N and Z flags.

PLP pull data from stack into the flags register

Description

Place the contents of the stack location pointed to by the stack pointer in the flags register then increment the stack pointer.

Flags affected

NVBDIZC (all)

Addressing mode

Implied

Comments

Note that this may change the interrupt state.

PLX pull data from stack into the X register**

Description

Place the contents of the stack location pointed to by the stack pointer in the X register then increment the stack pointer.

Flags affected

NZ

Addressing mode

Implied

Comments

Warning – PLX affects the N and Z flags.

PLY pull data from stack into the Y register**

Description

Place the contents of the stack location pointed to by the stack pointer in the Y register then increment the stack pointer.

Flags affected

NZ

Addressing mode

Implied

Comments

Warning – PLY affects the N and Z flags.

ROL rotate left

Description

Rotates the contents of the memory location left one bit through the carry (ie treating the carry as bit 9).

Flags affected

NZC

Addressing modes

Accumulator
Absolute
Zero page
Absolute indexed (X only)
Zero page indexed (X only)

Comments

This instruction is mainly used for extended precision shifts (eg to shift two bytes at &6000-1; `ASL &6000 ROL &6001`).

ROR rotate right

Description

Rotates the contents of the memory location right one bit through the carry (ie treating the carry as bit 9).

Flags affected

NZC

Addressing modes

Accumulator
Absolute
Zero page
Absolute indexed (X only)
Zero page indexed (X only)

Comments

This instruction is mainly used for extended precision shifts (eg to shift two bytes at &6000-1; `LSR &6001 ROR &6000`).

RTI — return from interrupt

Description

Loads the flags register and then the program counter from the stack (LSB then MSB), incrementing the stack pointer accordingly (SP->SP+3).

Flags affected

NVBDIZC (all)

Addressing mode

Implied

Comments

Although this instruction is provided to return from an interrupt driven routine, user-supplied interrupt service routines will normally be entered through IRQ1V or IRQ2V under the control of the MOS and should exit by jumping to the default contents of these vectors.

RTS — return from subroutine

Description

Loads the program counter from the stack (LSB then MSB) then increments it by one (PC->PC+1), incrementing the stack pointer accordingly (SP->SP+2).

Flags affected

None

Addressing mode

Implied

Comments

Used to return from a subroutine entered by JSR, or to return to the environment calling a machine code program (eg BASIC program, command line interpreter mode of the system editor).

SBC subtract from accumulator with carry

Description

Subtracts the contents of the memory location and the borrow bit from the accumulator. The borrow bit is the inverse of the carry flag (ie NOT C).

Flags affected

NVZC

Addressing modes

Immediate
Absolute
Zero page
Zero page indirect**
Absolute indexed
Zero page indexed
Pre-indexed indirect
Post-indexed indirect

Comments

To subtract without the borrow bit, set C with SEC before using the SBC instruction.

SEC set carry flag

Description

Sets the carry flag (C) to 1.

Flags affected

C

Addressing mode

Implied

Comments

Mainly used to prepare for SBC instructions. See also CLC.

SED — set decimal flag

Description

Sets the decimal flag (D) to 1.

Flags affected

D

Addressing mode

Implied

Comments

This instruction should be used at the start of any routine which uses binary coded decimal (BCD) rather than binary arithmetic arithmetic. See also `CLD`.

Warning: the 65C12 takes one cycle longer than the 6502 to perform decimal mode arithmetic.

SEI — set interrupt mask

Description

Sets the I flag to 1, to disable all maskable (IRQ) interrupts.

Flags affected

I

Addressing mode

Implied

Comments

This instruction should only be used to prevent interrupts during brief crucial tasks such as changing a vector, after which interrupts must be re-enabled by `CLI`. Maskable interrupts are used extensively by the MOS, to provide background processing and service peripheral devices. You must not disable interrupts for more than approximately 2 milliseconds if you wish the MOS to continue with its normal housekeeping.

STA — store accumulator in memory

Description

Stores the contents of the accumulator at the specified memory location. The accumulator is unchanged.

Flags affected

None

Addressing modes

Absolute
Zero page
Absolute indexed
Zero page indexed
Zero page indirect**
Pre-indexed indirect
Post-indexed indirect

STX store X register in memory

Description

Stores the X register at the specified memory location. The X register is unchanged.

Flags affected

None

Addressing modes

Absolute
Zero page
Zero page indexed (Y only)

Comments

Note that although the absolute,Y addressing mode is valid with `LDX`, it is not valid with `STX`.

STY store Y register in memory

Description

Stores the Y register at the specified memory location. The Y register is unchanged.

Flags affected

None

Addressing modes

Absolute
Zero page
Zero page indexed (X only)

Comments

Note that although the absolute,X addressing mode is valid with LDY, it is not valid with STY.

STZ — store zero in memory**.

Description

Stores a 0 at the specified memory location.

Flags affected

None

Addressing modes

Absolute
Zero page
Absolute indexed (X only)
Zero page indexed (X only)

Comments

The assembler will accept CLR as an alternative to STZ.

TAX — transfer accumulator to X

Description

Copies the contents of the accumulator into the X register. The accumulator is unchanged.

Flags affected

NZ

Addressing mode

Implied

TAY — transfer accumulator to Y

Description

Copies the contents of the accumulator into the Y register. The accumulator is unchanged.

Flags affected

NZ

Addressing mode

Implied

TRB — test and reset bits**

Description

ANDs the complement of the accumulator with the specified memory location, storing the result in the memory location. The Z flag is set if A AND the memory location is zero.

Flags affected

Z

Addressing modes

Absolute
Zero page

Comments

If A=11001100 binary, memory=11001001 binary
Then complement of A=00110011
result is 00110011 AND 11001001 = 00000001
Z is reset

If A=11111010 binary, memory=11001111 binary
Then complement of A=00000101
result is 000000101 AND 11001111 = 00000101
Z is set

TSB — test and set bits**

Description

ORs the accumulator with the specified memory location, storing the result in the memory location. The Z flag is set if A AND the memory location is zero, otherwise Z is reset.

Flags affected

Z

Addressing modes

Absolute
Zero page

Comments

If A=11110000 binary, memory=00001111 binary
result is 11110000 OR 00001111 = 11111111
Z is set

If A=11110001 binary, memory=00001111 binary
result is 11110001 OR 00001111 = 11111111
Z is reset

TSX transfer stack pointer to X

Description

Copies the contents of the stack pointer into the X register. The stack pointer is unchanged.

Flags affected

NZ

Addressing mode

Implied

Comments

This is the only instruction that allows you to read the stack pointer. See TXS.

TXA transfer X to accumulator

Description

Copies the contents of the X register into the accumulator. The X register is unchanged.

Flags affected

NZ

Addressing mode

Implied

TXS transfer X to stack pointer

Description

Copies the contents of the X register into the stack pointer. The X register is unchanged.

Flags affected

NZ

Addressing mode

Implied

Comments

This is the only instruction that allows you to set the stack pointer. See TSX.

TYA transfer Y to accumulator

Description

Copies the contents of the Y register into the accumulator. The Y register is unchanged.

Flags affected

NZ

Addressing mode

Implied

Q Assembler errors

Q.1 Introduction

This section describes the rather diverse topic of errors in the assembly and machine code environment:

- how to interpret and correct errors detected during the assembly process (assembly errors);

- how to detect, analyse and correct improper operation of the program (coding errors);

- how to design your programs to detect and handle errors which were not generated by it, while the program is running (run-time errors);

Note that the only 'automatic' error handling is during the assembly process. The absence of assembly errors only guarantees that your assembly language program has been translated into the equivalent machine code program. It is the programmer's responsibility to write an assembly language program that performs the functions required and copes with error conditions beyond its control. This task is far more difficult in assembly language than in high level languages, since the error detection, error handling and program debugging facilities available in the machine code environment are rather limited – the programmer must provide his own.

Q.2 Assembler error messages and symptoms

Detecting assembly errors

The detection and reporting of errors in the assembly language program (errors of syntax rather than logic) is the responsibility of the assembler and BASIC interpreter. The supporting BASIC code may also contain errors which may be detected by the BASIC interpreter.

By default, when an error is detected the assembly process aborts and an error message is displayed that states the nature of the error and the line number at which it was detected (eg Syntax error at line 350). The BASIC pseudovariable `ERR` is set to the error number and `ERL` to the line number in error. The error message can be redisplayed with the statement `REPORT` at any time up to the next BASIC error or reset.

There are two mechanisms that can change this error handling: BASIC's `ON ERROR` and the assembler switch `OPT`:

`ON ERROR` is described in full in Section L. If used incorrectly during an assembly, it can stop assembly errors being reported, resulting in apparently successful generation of a machine code program which is in fact incorrect.

`OPT` is described in full in Section O. If the assembler reads OPT <n> with bit 1 of <n> reset to 0, the Branch out of range and No such variable errors will not be detected nor reported. This mechanism is provided to prevent generation of inappropriate error messages during the first pass of a multi-pass assembly. For further details, see Section O.

Analysing assembly errors

The majority of assembly errors will be reported as a message describing the nature of the error and the line number of the BASIC line on which the error was detected (subject to `OPT` and `ON ERROR` – see above). These errors are of two main types: those generated by the assembler and those generated by the BASIC interpreter.

If the line in error is a multistatement line, setting bit 0 of `OPT` can help to decide which statement is in error by providing a listing of the statements as they are assembled.

To check fields reported as illegal, you can also use `P.`<field> to check their syntax and range.

Assembler error messages result from mis-spelt mnemonics, branching out of range, invalid address field format or range.

BASIC error messages result from incorrect address or data field expressions (such as mis-spelt variables or functions) within assembly language statements, or errors in the coding of the BASIC program performing the assembly.

Both types of error message are listed in Section M (BASIC Errors), together with a guide to their resolution.

Note that the line reported in error may in fact be correct, eg `BRA` <label> would be reported as an error even if your error was incorrect spelling of the label declaration, on another line.

Errors in the supporting BASIC code can produce confusing error messages and symptoms. Incorrect setting of `P%` can result in a legal assembly which will not run; incorrect setting of `O%` or `P%` can overwrite the microprocessor stack (system locks up), crucial MOS variables (system locks up), parts of the program (Bad program, Syntax error) or its variables (No such variable), or the BASIC stack (various errors). Reserving insufficient space for the object program (with `DIM`) results in overwriting variables declared subsequently, including assembler labels (No such variable).

More difficult to detect and analyse are those errors in the assembly process which result in an incorrect translation of the program but do not produce error messages. These errors, which are only obvious when you attempt to run the machine program are described in the next chapter – Design errors.

Handling assembly errors

Error handling during the assembly process can be provided using `ON ERROR` as decribed under the previous heading, but will not usually serve a useful purpose.

Acting on assembly errors

Once you have analysed the errors in your assembly language program, you must correct them using the screen or system editor then `RUN` it again to produce a new assembly. You may have to do this several times before the program assembles correctly, since the assembly process stops at the first error reported.

Q.3 Coding errors

Once a program has assembled correctly, you will want to use it. Proceed carefully, making sure that you have saved the assembly language program in the filing system before running the machine code program.

If the program performs exactly as required (which is often a laborious task to prove) then your design was correct.

If the program does not perform correctly even though it seemed to assemble correctly and you only supply it with correct data, it contains errors in the machine code caused by:

- errors in the supporting BASIC code (eg `P%` or `O%` set incorrectly, `OPT` or `ON ERROR` suppressing errors, calling the wrong address)

- incorrect high level design (flow charts, pseudo-code etc) of the program (eg variables not initialised)

- incorrect translation of the high level design in assembly language statements (eg omitting some of the required instructions or using them incorrectly)

Detecting coding errors

Coding errors will usually only become evident when you run the machine code program, as minor or drastically incorrect performance.

Analysing coding errors

Deciding where the source of the error lies depends on first deciding which type of error has been made.

If the program aborts quickly without any resemblance to the intended performance, look for errors in the supporting BASIC code (eg `P%` or `O%` set incorrectly or not set to the same value on each pass, insufficient passes to resolve forward references, external variables such as MOS entry points initialised incorrectly, `OPT` or `ON ERROR` suppressing assembly errors, calling the wrong address). Check the BASIC code carefully; if you can find no errors in it, the fault may be in the design or assembly language, but this is less likely.

If the program performs nearly as expected, the error may be in the high level design or in its translation by you into assembly language. If your program has been designed in a modular fashion, you should be able to trace the malfunction

to either one or a small group of subroutines. Check the high level design first by running the test data through the algorithms used in those subroutines.

If the design seems correct, you must check the assembly language – this is often known as 'debugging'. Try running the assembler routines in isolation, supplying them with correct data and analysing their output. The contents of the A, X, Y and flags registers can be returned to BASIC for examination (by calling with z=USR(<entry>), exiting with RTS and masking z using the AND operator) and the contents of locations used for storage can be examined from BASIC using P.?<address> or P.!<address> (see Section L). If the routines are complex or use instructions you are not too familiar with, try simplifying them or using more familiar instructions.

The machine code environment is not as helpful in tracking down errors as high level language environments. In particular, there are no execution tracing, error reporting or error handling facilities. To trace execution you can either include extra code to display messages at crucial points of your program or use a commercial debug/trace package. By placing BRK instructions at locations in your program you can cause a "software interrupt" when that location is executed; by default that interrupt is used for error handling (see the next chapter) but by changing the BRKV vector it can be used to link other debugging code into your program, such as register dumping, or address reporting.

Your program may have assembled without error messages but still be an incorrect translation of the assembly language program, even though assembly error handling was not disabled using OPT or ON ERROR and the BASIC code was correct. This can happen for four main reasons as described below. Although they are all errors in your assembly language program, they are not reported as such by the assembler and so are not likely to be detected until the program is run.

Multiple declaration

```
100[OPT C
110.label  ...
120  ...
130  ...
140.label  ...
150  ...
160  ...
170]
```

Use of label on lines 110 to 130 assembles as the location of the instruction on line 110; use of label on lines 140 to 160 assemble as the location of the instruction on line 140, which is unlikely to be what was intended. The solution is to declare labels once only.

Q.3-2

Changes in address field length in the second pass

```
100 FOR C=1 TO 3 STEP 2:
110 P%=&2000:[OPT C
120.label1
130   ...
140LDA message,X
150JSR label2
160   ...
170.label2
180   ...
190   ...
200]
210P%=&70
220 [OPT C
230.message EQUS "Program running":EQUB 0
240]
250NEXT
```

The problem arises because on the first pass, in line 140 the assembler uses the current value of P% for the undeclared label message (it has not yet read the declaration in line 230) resulting in an address field of 2 bytes (eg LDA &2010,X). On the second pass, message is now correctly set to &70 so that line 140 results in an address field of 1 byte (ie LDA &70,X). As a result, all subsequent instructions will assemble on the second pass one byte nearer the start of the program than they were on the first pass.

Unfortunately, this does not immediately adjust the label declarations read on the first pass – they are only recalculated to the correct (one byte lower) addresses as the declarations are read again on the second pass. Thus in line 150, JSR label2 assembles using the value of label2 generated during the first pass, one byte past the intended destination.

The solution is to ensure that labels which will result in 1 byte address fields are initialised in the first pass before they are used as addresses, eg by placing lines 210-240 before line 110 in the above example.

Using .<label> in an address field

Although labels are declared using .<label>, the dot is not part of the label name and must not be included in address fields. If you do so, the dot will be taken as a decimal place and the rest of the field ignored so that the address field becomes 0 eg JSR .labelz would assemble without error as JSR &0000.

The solution is simply to use <label> and not .<label> in address fields.

Invalid address fields

If an address field is read as complete by the assembler, any remaining text in that statement is ignored. This means that you can code an (illegal) addressing mode without being aware that it is being assembled to a (legal) instruction with a quite different action eg `CPX &70,Y` would assemble without error as `CPX &70` (followed by a comment of `,Y`).

The solution is to always check that an addressing mode is valid with an instruction before using it.

Handling coding errors

While it is possible to correct errors in program design by adding code to detect those conditions that are handled incorrectly and handle them with ad hoc code, it is far better to redesign the program so that a single coherent program can cater for all valid data.

Acting on coding errors

Coding errors should usually be corrected by changing the assembly program, running it and testing the results again. While it is possible to make minor changes directly to a machine code program (known as patching), this is error-prone and unlikely to be necessary with the types of programs written on microcomputer systems.

Q.4 Run-time errors

Although a program may operate entirely correctly when the data and external routines it uses behave correctly, it is not really reliable until it can handle at least the commonest types of incorrect data and cope sensibly when errors are detected by other parts of the system. These errors are termed run-time errors.

Detection, analysing and handling of run-time errors must all be performed in the program and so should be included in the design process. You should attempt to detect and report all possible errors, as programs which abort without any error diagnostics are very unpleasant to use. Concentrate your efforts on handling the more frequent errors in the most user-friendly manner.

Detecting run-time errors

Any program that reads data from the user, MOS, memory or filing system should be designed to check that the data read can be handled correctly by the program. This will usually require an explicit check on range, format or content for each type of input. The exception is where the data can be guaranteed as valid eg where it is read from a file that was written by a program that has already performed all the checks required.

Programs that use other routines should be prepared for any error conditions that may be signalled by those routines. The MOS, languages and filing systems all use the same mechanism to report errors, as described below. By default this mechanism displays a message and returns control to the current language or the MOS command line interpreter so that user programs need not detect such errors. It is however possible to intercept this mechanism by changing a vector (BRKV) to point to your own error handling code, then whenever that code is entered, an external routine has generated an error (or your own program has executed a BRK instruction).

Analysing run-time errors

Errors in data read by the program are of course analysed by the user code that detected them. The error should be analysed sufficiently well for the error handling that the program is to provide eg well enough so that a helpful error message can be displayed (eg name too long, no name specified, illegal characters in name rather than bad name) or the data can be corrected/skipped/reported as appropriate.

Errors detected by other parts of the system and handled by the program (ie after changing BRKV) can be analysed easily as they are uniquely identified by the error message number stored as described under the next heading. For a

description of error codes and messages, refer to the Section on the appropriate filing system or language.

Handling run-time errors

Once your program has detected and analysed a run-time error, it can handle it in a variety of ways.

Display error message and abort program

Continue with/without error message and:

– Ignore the data
– Correct the data
– Try again for correct data (reprompt, reread)
– Try the call again
– Try the call again with modified parameters

To display an error message, abort the program and return to the current language (or MOS command line interpreter if there is no current language), execute a BRK instruction followed by an error code and the error message text terminated by 0 (an error block). For example:

```
100[OPT C
110   ...
120BNE carryon
130BRK: EQUB errno:EQUS "Division by zero attempted":EQUB 0 \error
block
140.carryon   ...
150   ...
160   ...
170]
```

When a BRK instruction is executed, it causes a 'software interrupt', which is handled by a MOS routine that stores the socket number of the currently active ROM (language or filing system) and places a pointer to the error code in &00FD-&00FE. The MOS routine ends by indirecting through the BRKV vector. Languages initialise BRKV to point to their error handling routines, which read the error code and message using the pointers provided by the MOS, display the text or use the information in other ways, then leave the language selected.

A filing system generates an error message by copying an error message block from the filing system ROM to the stack (&100-) then JMPing to &100, so that the current language's error handling routine can read the error message. A language generates an error message by JMPing to the start of an error message block in the language ROM.

The same error message format is used for all MOS, language and filing system error messages, so that any error messages generated by these ROMs will also abort your program, returning to the current language.

By changing BRKV to point to your own routine, you can take over error handling from the language and allow your program to continue execution after errors. If you do so, displaying error messages is the sole responsibility of your program. To find the location of the error code and message, call OSBYTE 186 for the ROM socket number and load the memory pointer from &00FD-&00FE. Error messages generated by programming languages (eg BASIC) may have been tokenised by the language to save space.

R The system editor

R.1 Introduction

What EDIT is for

EDIT is a text editor and formatter; a tool designed to make preparing, changing and printing text easier and quicker.

It is most useful for writing programs but can also be used to write and print documents (letters, chapters of books). The VIEW word processor provides an alternative way of preparing and printing documents which will probably be preferred by most users, since it formats text on the screen as you type it in, so you can see how it will look when printed. Which of these tools you use to prepare text is really up to you, but EDIT will probably appeal more if you spend more time programming, VIEW if you spend more time preparing documents. Printing with EDIT is very flexible indeed, but is perhaps not as simple to use as the VIEW printing facilities.

Printing facilities (provided by a part of EDIT called a 'formatter') are described in the next Section.

How to deal with errors and error messages (from the editor and formatter) is described in the Section after that (Section T).

What EDIT can do

EDIT provides facilities to let you do the following:

- type, insert, copy, move and delete text;
- quickly find, replace or count text specified in a variety of ways;
- use filing system and operating system commands;
- enter formatting commands;
- print text;
- use text as a program.

The types of things that EDIT cannot do are:

- perform arithmetic;
- perform language processing.

The keyboard

The letter, number and punctuation keys have their normal functions within EDIT, but some of the others have modified functions:

RETURN	type return character, move cursor or complete a command
TAB	move the cursor (2 modes)
SHIFT + TAB	switch TAB modes
f0-f9	invoke edit commands
SHIFT + f0-f9	
COPY	delete the character at the cursor
SHIFT + COPY	switch the cursor to cursor copying mode
cursor keys	move cursor a character/line at a time
SHIFT + cursor key	move cursor a word/screenful at a time
CTRL + cursor key	move cursor to start/end of line/text
ESCAPE	cancel an incomplete command or leave cursor editing mode

Editing concepts

Text within EDIT, whether entered by typing, loading a file, or copying text etc is held in an area of memory called the 'edit buffer'. A portion of this text is always displayed on the screen, as a kind of 'window' onto the buffer. This window can be moved through the text at will.

The editor provides a range of functions that you can use to change the text, called 'editing' functions. Most editing functions affect the text displayed on the screen, at the position indicated by the cursor. You can move the cursor around the screen (and hence the buffer) in a variety of ways.

Changes made on the screen are made simultaneously to the text in the buffer.

Text in the edit buffer is stored in RAM, so it will be lost if you turn the computer off or load another file from a filing system. To preserve the text, you must use the 'save file' command to store it as a file in the current filing system.

Text printing and formatting is described in the next Section.

R.2 Selecting EDIT

There are three main ways to select EDIT:

- to edit an existing *program*, select the language it is written in as the current language (eg `*BASIC`), load the program, then use that language's EDIT command (if available; see the language reference guide). This converts the program into a form suitable for editing, selects EDIT and loads the converted program.
- to edit an existing *text* file, enter the editor with `*EDIT` <filename>. This selects EDIT and loads the file for you (alternatively, you can enter with `*EDIT` and load files using editor commands)
- to prepare a program or text from scratch, enter the editor with `*EDIT`. This selects EDIT with an empty edit buffer

To run an edited program, you will need to use the commands described in 'Saving and loading text', below.

If the language does not provide an EDIT command, you will need to use a different technique:

select the language:	eg `*BASIC` RETURN
load the program:	eg `LOAD "bobot"` RETURN
spool a listing:	eg `*spool bobot!` RETURN `L.` RETURN `*spool` RETURN
edit the spooled listing:	eg `*edit bobot!` RETURN

R.3 The screen layout

Whenever you enter EDIT, the screen clears and divides into a number of regions. The very first time you enter EDIT, the display divides into four areas:

1. command key legends

   briefly describes the edit command provided by each function key

2. description of command selected and other keys

   briefly describes the functions of the [TAB], [COPY] and cursor keys. Also describes the action of the edit commands in greater detail when they are selected

3. text display/entry area

   displays a portion of the text in the buffer, including the text being type

4. status information, messages and command dialogue

   usually indicates the current text entry mode and number of marks set, but is also used for information, warning and error messages, command prompts and replies to prompts

There are other display modes, which may be more suited to your eyesight, screen resolution or personal preference. To change to another display mode, use SET MODE, and specify the mode required by typing the mode name shown below followed by [RETURN]. You can do this at any time, although this has the side-effect of moving the cursor (and the display window) to the start of the text buffer. The modes are:

Mode name	**areas displayed**
D:	full display as above (Descriptive mode)
K:	areas 1, 3 and 4 (Keyword mode)
0,1,3,4,6,7:	areas 3 and 4

The D and K modes provide helpful information for new users of the editor, but have two disadvantages: they leave less room on the screen for text than the corresponding display MODE (128) and scroll more slowly because they use windowing and software scrolling.

All display MODEs use shadow display to leave you the maximum space for editing in.

The numeric display modes select the corresponding shadow MODEs (eg 128 for Ø) and use hardware scrolling so that the screen is updated quicker when you move through the text.

In all MODEs other than 7, control codes are displayed as the corresponding letter in black on a white block and the end of the buffer marker is displayed as a black asterisk on a white block. In MODE 7, these are all displayed as a white block, due to the limitations of this display mode.

When you next select EDIT, it will start with whatever display mode you had selected last time you left EDIT (this information is automatically stored in the CMOS RAM so that it is preserved when you turn the computer off).

R.4 EDIT commands: using the function keys

Once you have selected EDIT, you will use most of its facilities by pressing function keys, either on their own or in conjunction with [SHIFT] and [CTRL]. To help you remember the editing function of each of these keys, a 'function key ruler' is provided for EDIT: insert this under the transparent strip behind the function keys before starting to use EDIT.

The editing function performed by each function key is also displayed at the top of the screen in the 'K' and 'D' modes. This information is also summarised here for your convenience while reading this Section:

No.	Normal	+ [SHIFT]	+ [CTRL]
0	GOTO LINE	DISPLAY RETURNS	
1	COMMAND LINE	INSERT/OVER	
2	LOAD FILE	INSERT FILE	
3	SAVE FILE	REMOVE MARGINS	
4	FIND STRING	RETURN TO LANGUAGE	
5	GLOBAL REPLACE	SET MODE	
6	MARK PLACE	CLEAR MARKS	SET TOP
7	MARKED COPY	MARKED MOVE	SET BOTTOM
8	PRINT TEXT	MARKED DELETE	
9	OLD TEXT	CLEAR TEXT	

R.5 Typing text, INSERT and OVER modes, simple deleting

As soon as you enter EDIT, anything you type with the letter, number or punctuation keys is displayed on the screen and placed in the edit buffer at the position indicated by the cursor.

You can choose for the characters you type to either replace those currently at the cursor position on the screen (OVER mode) or push existing text to the right to make room (INSERT mode). The mode you are in is displayed in the bottom left corner of the screen (the status area). To switch between these two modes, use [SHIFT]+[f1](INSERT/OVER). Certain keys behave differently in the two modes – these differences will be mentioned as the functions of those keys are introduced.

When the text you are typing reaches the right hand side of the screen it overflows onto the next screen line. EDIT imposes no limit on the number of characters that you can type, other than the maximum buffer size, dictated by the available RAM. Note that the end of a line of text on the screen has no special significance for the EDIT editor or print formatter – it is *not* stored in the text buffer in any way.

Pressing [RETURN] in INSERT mode puts a carriage return character (ASCII 13) at the cursor and the cursor moves to the start of the next screen line. The return *character* is important as it marks the end of a paragraph to the editor and the text formatter. Return characters are normally invisible. To make EDIT display return characters (as black M in a white rectangle, or a solid rectangle in MODE 7), use [SHIFT]+[f0] (DISPLAY RETURNS).

Pressing [RETURN] in OVER mode simply moves the cursor to the start of the next screen line.

To correct minor errors as you type, use [DELETE] to erase characters to the left of the cursor, then retype them correctly. In INSERT mode, this removes the character from the screen and buffer; in OVER mode it doesn't remove it, but replaces it with a space.

If you are preparing a BASIC program, you need not type the line numbers at the start of each program line, as long as you can avoid using line numbers in the program (eg in GOTO and GOSUB instructions). At the end of each program line, press [RETURN] but do not type the line number for the next line. Note that to convert a program written in this way into a runnable form, you cannot use [SHIFT]+[f4](RETURN TO LANGUAGE) – see chapter R.13 for details.

R.6 Moving the cursor around the screen

The cursor indicates the position within the text buffer at which typing, single character deleting, marking and many other editing operations will occur. To do anything more than type and correct minor typing errors by 'backspacing' with [DELETE], you will need to move the cursor around the text.

To move the cursor around the screen use the cursor keys marked with arrows, as follows:

To move the cursor a character space or line at a time, use the appropriate cursor key.

To move the cursor a word at a time, use the left and right cursor keys with [SHIFT]

To move the cursor under the first character of the first word on the preceding line, press [SHIFT] + [TAB] until TAB below words is displayed on the status line at the bottom of the screen, then use [TAB]

To move the cursor to the next tab position to the right, press [SHIFT] + [TAB] until TAB columns of 8 is displayed on the status line at the bottom of the screen, then use [TAB]

To move the cursor to the beginning or end of the screen line, use the left or right cursor keys with [CTRL]

To move the cursor to the beginning of the next screen line while in OVER mode, press [RETURN]

To move the cursor several columns or lines, hold the cursor key down so that its action is repeated. If you find that the cursor moves too fast or too slow, alter the rate at which the cursor keys (and all others) repeat, change the key repeat rate by selecting command line mode with [f1] (COMMAND LINE) and using *FX12 (eg *FX12,2 for a very fast repeat rate) or *CONFIGURE REPEAT if you want to make the change permanent.

R.7 Moving around the text buffer, scroll margins

When the text in the text buffer is more than will fit on a screen, the screen 'window' can only show you part of the text at a time. To see another part of the text, use the cursor keys as follows:

To move though the edit buffer a line at a time ('scrolling'), use the up or down cursor keys – the screen will scroll when the cursor comes within a few lines of its top or bottom (this distance is the 'scroll margin', and can be changed with [SHIFT]+[f3](REMOVE MARGINS): see below)

To move through the text buffer a screenful at a time, use the up or down cursor keys with [SHIFT]

To move to the start or end of the text buffer, use the up and down cursor keys with [CTRL]

To move to a particular paragraph in the buffer, use [f0](GOTO LINE) and specify its line number, which is the number of return characters between it and the start of the buffer. GOTO LINE displays the current line number (At line nnn, new line:) to simplify jumping back and forth through the buffer

To move to a known piece of text in the buffer, use [f4](FIND STRING) – see chapter R.15.

The screen will normally scroll when the cursor is moved to within four lines of the top or bottom of the screen (the 'scroll margin') to ensure that you can always see the context of any text you are changing. To change the scroll margins, press [SHIFT]+[f3](REMOVE MARGINS) to set the margins to the top and bottom of the screen, then redefine them (if required) by moving the cursor to the new top margin line and pressing [CTRL]+[f6](SET TOP), then moving to the new bottom margin line and pressing [CTRL]+[f7](SET BOTTOM).

R.8 Changing existing text

To make small changes to text in the buffer, move the cursor to the area to change, then either delete text using [DELETE] or [COPY] or type new text, as follows.

In INSERT mode, [DELETE] deletes characters to the left of the cursor; [COPY] deletes characters to the right of the cursor; typing inserts the new text at the cursor.

In OVER mode, [DELETE] replaces characters to the left of the cursor with spaces and moves the cursor to the left; [COPY] replaces the character at the cursor with a spaces (once only); typing replaces existing text with new text.

To insert text at the cursor, simply type in INSERT mode (if in OVER mode, select INSERT with INSERT/OVER). To type over text at the cursor (ie replace it), simply type in OVER mode (if in INSERT mode, select OVER with INSERT/OVER).

Pressing [RETURN] in INSERT mode will split the text at the cursor into two separate paragraphs; pressing [RETURN] in OVER mode simply moves the cursor to the start of the next screen line.

To remove a blank line or join two paragraphs, you must delete the unwanted [RETURN] character: move the cursor to the [RETURN] character (made visible using DISPLAY RETURNS) and press [COPY], or move the cursor to the start of the next screen line after the [RETURN] and press [DELETE].

R.9 Moving, copying and deleting text

To delete a single character at a time from the screen, either position the cursor immediately to its right and press [DELETE] or position the cursor on the character and press [COPY]. In INSERT mode, the text closes up around the deleted character; in OVER mode, the character is replaced by a space.

To delete a short piece of text, select INSERT mode and either position the cursor at its end and hold down [DELETE] or position the cursor at its beginning and hold down [COPY] until the text has been deleted.

To copy short sequences of text from the screen, position the cursor where you want the text to go and press [SHIFT] + [COPY] to select CURSOR EDITING mode. Now move the cursor to the text you want to copy and hold down [COPY] to duplicate the characters required. You may type text in between using [COPY] and use [DELETE] (but not [COPY]) to delete typed or copied text in the usual way. To return to the previously selected text entry mode (INSERT/OVER), press [ESCAPE].

Cursor editing has certain limitations:

- you cannot use editor commands while cursor editing and cursor motion only affects where characters will be copied from.
- the source and destination text must both be on the screen at the same time, as the screen will not scroll in this mode.
- you cannot copy control characters (including return characters, if displayed).
- if the destination line is before the source line, copying can move the source text away from the copy cursor, if the copied text overflows the destination line.

However, it has certain compensating advantages:

- the function keys [f0]-[f9] may be used as soft keys, to type complete words or phrases with a single keystroke or type ASCII codes above 127 – see the entry for *KEY in Section C.
- the [SHIFT] and [CTRL] function keys place characters with ASCII codes above 128 in the text buffer – this can be used to prepare EDIT command files (see chapter R.16)

Although these techniques are very useful, even more powerful editing commands are provided in INSERT and OVER modes to let you delete, move and copy blocks of text. These blocks are only limited in size by the text in the

buffer. The first step in using any of these commands, is defining the block of text to be used. You do this by moving the cursor to the start and/or end of the block and marking its position with f6 (MARK PLACE).

To delete a block of text, mark the start or end, then move the cursor to the other end of the block and use SHIFT + f8 (MARKED DELETE)

To move a block of text, mark the start and the end (in either order), then move the cursor to the destination and use SHIFT + f7 (MARKED MOVE)

To copy a block of text, mark the start and the end (in either order), then move the cursor to the destination and use f7 (MARKED COPY). From now on, each time you press MARKED COPY a further copy will be placed at the cursor, until you erase the marks using SHIFT + f6 (CLEAR MARKS).

To copy a file from the filing system into existing text in the edit buffer, position your cursor where you want the text to go, then use SHIFT + f2 (INSERT FILE), typing the file name in response to the prompt `Type filename to insert:`.

A single mark is also used to limit the effect of a global replace command to part of the text buffer – see chapter R.11.

EDIT removes marks automatically after the block delete or move commands, changing screen modes or producing the `Bad marking` error message. Marks are not removed automatically after the copy command, to allow you to make multiple copies of marked text easily. You can clear all marks at any time with SHIFT + f6 (CLEAR MARKS). It is best to do this as soon as you are finished with them, as they stop you defining other blocks, modify the action of certain commands (GLOBAL REPLACE and SAVE FILE) and can prevent operation of some commands, causing the error message `Bad marking`. The number of marks currently set is usually displayed in the status line at the bottom of the screen.

R.10 BREAK, clearing and restoring text

To delete the entire contents of the text buffer and reinitialise the editor and filing system, press [BREAK]. This should only be necessary if the editor no longer responds to commands.

To delete the entire contents of the text buffer, use [SHIFT]+[f9](CLEAR TEXT), then press any key except [ESCAPE] in response to the prompt `Text` will be cleared if a key is hit.

To recover the contents of the text buffer immediately after using CLEAR TEXT or pressing [BREAK], use [f9](OLD TEXT).

If you cannot recover your text using OLD TEXT, try using:

```
*SAVE <name> 800 7FFF
```

This saves all of the text buffer apart from the last character. You should now be able to load it into the editor with LOAD FILE. (See Section T for further information.)

R.11 Finding and replacing text (elementary)

The editor provides a very powerful pair of commands (f4 (FIND STRING) and f5 (GLOBAL REPLACE)), which can perform the following types of functions:

- rapidly changing all or selected occurrences in the edit buffer of one string into another string – correcting spelling mistakes, improving consistency, expanding or shortening variable names in programs, expanding abbreviations, removing comment lines from debugged programs to make them shorter etc.
- finding the next time a text string appears in the edit buffer – moving to a line whose number you don't know but whose content you do, checking procedure/function parameters in programs etc.
- counting the number of times a string occurs in the edit buffer – checking the size of a document, guarding against overuse of particular phrases, analysing style, looking for underused variables/procedures/functions in programs etc.

After selecting one of these commands, you are prompted to specify what text to search for in the text buffer, and what to do with any matching text found. Matching text may be deleted, counted, replaced or moved to. The specification of the text to match is the 'target string', occurrences of matching text are 'found strings' and the specification of text to replace found strings with (if supplied) is the 'replacement string'.

The find/replace commands can be used simply, as described immediately below, but to make the best use of these powerful commands you must understand the conventions that can be used to specify target and replacement strings, described in chapter R.15.

When you select one of the find and replace commands, you are prompted with either `Find and replace:` or `Global replace:`. The text that you type in response, up to pressing RETURN (or aborting with ESCAPE) specifies the action required.

The following functions can be achieved easily using FIND STRING and GLOBAL REPLACE and the appropriate string.

To replace all occurrences of the target string with the replacement string, use:

GLOBAL REPLACE <target string>/<replacement string> RETURN

To delete all occurrences of the target string, use:

GLOBAL REPLACE <target string>/ RETURN

To count the number of occurrences of the target string:

GLOBAL REPLACE <target string> [RETURN]

To find the next occurrence of the target string:

FIND STRING <target string> [RETURN]

To selectively delete occurrences of the target string use:

FIND STRING <target string> / [RETURN]

To selectively replace occurrences of the target string with a single replacement string, use:

FIND STRING <target string> / <replacement string> [RETURN]

To selectively replace occurrences of the target string with different replacement strings use:

FIND STRING <target string> [RETURN]

To use the previous target and replacement strings:

FIND STRING [RETURN]
GLOBAL REPLACE [RETURN]

To cancel an incomplete string or abort a command while it is operating, press [ESCAPE].

The following characters may be used freely in target and replacement strings as they have no special meaning to the search and replace commands (ie they are interpreted as themselves):

capital and lowercase letters
numbers (eg 12 will match 12 only)
spaces

the symbols: `<,>? ;:}{_£!"'()=` may be used in target and replace strings
the symbols: `%&` may only be used in target strings
the symbols: `.#@^*~-/` may only be used in replace strings

Other characters do not stand for themselves but have special meanings, as described later (see chapter R.15). Capital and lower case letters are equivalent in the target string, since they will both match the same letters of either case (eg cat will match `CAT` or `Cat` or `caT` etc).

If you need to match characters other than those above, refer to chapter R.15.

The FIND STRING and GLOBAL REPLACE commands operate differently, as follows:

GLOBAL REPLACE takes the target and optional replacement string typed in response to the prompt and acts on all matching strings found in the text

buffer, giving a count of the number found. If a single mark is set, it will only search between that mark and the cursor.

FIND STRING takes the target and optional replacement string and searches from the present cursor position towards the end of the text buffer. When a match is found the cursor is moved to the found string and the screen updated if necessary to show the found string, then you are prompted with `R(eplace), C(ontinue)` or `ESCAPE`. This allows you to check each target string before changing it:

To replace the target string, press `R`

to move to the next matching string without changing it press `C`

to exit FIND STRING press [ESCAPE]

If you type `R` for replace but did not specify a reeplacement string in the answer to the `Find and replace` prompt, you will be prompted for it now, with `Replace string:` – this string may be [RETURN] (to delete the found string) or <replacement string> as defined above.

R.12 Printing text

To print the text in the buffer, use [f8] (PRINT TEXT). This command takes the text in the text buffer, formats it and then prints the result. This will not affect the text in the buffer, but see the note below.

This command should not be used to list programs as the formatting it performs is quite inappropriate. (Return the program to its language and list it.)

This command uses two prompts, as follows:

`S(creen), P(rinter) or H(elp) ?` asks you whether you want to print the formatted text on the screen, print it via the currently selected printer driver, or print information on the print formatter commands on the screen.

If you select `S` or `P`, `C(ontinuous) or P(aged)?` asks you whether you want the printing to be continuous, or wait at the bottom of each page (or screen, if printing is to the screen) for you to press [SHIFT]. This allows you time to check the printing or insert paper in a printer using single sheets

Note: if you have typed control codes in your text, when you print to the screen these will be obeyed by the VDU driver as VDU commands. If the editor behaves strangely after printing your text to the screen, use [BREAK] and OLD TEXT to restore normal operation with your text intact. If behaviour is still incorrect, try entering and leaving command line mode (COMMAND LINE). To avoid this potential problem, print control codes by using the `.oc` or `.on` formatter commands in your text (see the next Section for details).

The text formatter does not print the text exactly as you typed it, but formats it first, as follows. It considers each sequence of text between return characters as a paragraph, assumes a line length of 76 characters, and then divides the paragraph up into a number of lines that will fit this length without splitting words across lines. It then expands all the lines except for the last in the paragraph to 76 characters by inserting spaces between the words, and prints the formatted paragraph. This formatting process is called 'justification'.

The text formatter also keeps a count of the number of lines that have been printed on each page and when this reaches 58, prints a page number and a form feed (ASCII 12) which should move the printer to the start of the next page. This should produce a neat page on printers which use the standard character spacing of 10 characters per inch and line spacing of 6 lines per inch, when using 8½" × 11" size paper.

You can alter the way your text is formatted in a variety of ways (including turning off justification), by inserting text formatting commands in the text. These commands are described in detail in the next Section.

R.13 Saving and loading text

Saving text files

The text in the text buffer may be saved to a named file in the filing system at any time, using [f3](SAVE FILE) and typing the filename followed by [RETURN].

SAVE FILE produces a bleep (or other tone programmed for [CTRL] + G) to warn you that you may be about to overwrite a file, in case you selected SAVE FILE by mistake.

To save just part of the buffer in a file, mark one end of the part required (using MARK PLACE), move the cursor to the other end and then use SAVE FILE.

The file name specified should be valid for the current filing system. The text is saved as a file which may be reloaded using LOAD FILE or INSERT FILE, *TYPEd or *DUMPed etc.

To save the file under the same name as last typed for save, load or insert you may use [COPY] then [RETURN].

To include the filename within the file, put `>filename` on the first line in the buffer and simply press [RETURN] when prompted for the filename. This line must be less than 129 characters long. `>filename` need not be at the start of the line eg to prevent this line being printed when the text is formatted, you can start it with `.co`.

Saving programs

If the text in the editor is a program, saving it as a text file (using SAVE FILE) will not be enough to create program that you can run. To do this, you must either return the buffer contents to the language using [SHIFT]+[f4](RETURN TO LANGUAGE), or save the program as a text file then select the language and *EXEC the saved text file. Using either method, you will now be in the language with a (tokenised) program in memory which can be run or saved using the language's appropriate filing system command.

If the buffer contains a BASIC program written without line numbers (as described in 'Typing text' above), you cannot create a runnable program by simply returning it to BASIC using RETURN TO LANGUAGE. Instead, you must put an AUTO command at the start of the buffer, save the program as a text file, select BASIC, then *EXEC the text file. This will load the program into memory, inserting line numbers automatically, after which you can RUN or SAVE it as a normal BASIC program.

Note: if the file was originally created using *SPOOL, it may contain CTRL J at the start of each line. These *must* be deleted before returning the program to

the language or *EXECing it, as lines starting with CTRL J will be ignored by the language interpreter. Lines starting CTRL @ will also be ignored.

Loading text and program files

The text in the buffer may be replaced at any time using f2 (LOAD FILE) and typing the filename followed by RETURN.

The file named should be an existing file on the current filing system. Any type of file currently supported by an approved filing system or language will load correctly (eg editor files, files created with *SPOOL or *BUILD, assembler object programs etc).

Some types of file (such as BASIC programs, files created using PRINT# or an assembler) although they will load correctly will not be very readable, as they do not consist of ASCII coded text. In particular, programming languages convert keywords in their programs into ASCII codes above 127. Programming languages may provide commands (such as the BASIC EDIT command) to pass their tokenised programs to the editor, expanding them to readable plain text in the process – refer to the language's documentation for guidance. If the language does not provide such a command, you should load the program into the language then use *SPOOL <filename>, list the program and use *SPOOL again to produce a plain text file, which you can then load into the editor.

The loaded file completely replaces the existing text in the buffer. (If you load a new file by mistake, you may be able to recover some of the old text – see Section T).

RETURN and COPY RETURN may also be used as described under 'Saving text files'.

Combining files

Files can be added to the text in the buffer rather than completely replacing it by using SHIFT+f2 (INSERT FILE). The file is inserted at the cursor position.

R.14 Using filing and operating system commands

Any of the filing and operating system commands used by typing *<text> (as described in other sections of this manual) may also be used from the editor, by first pressing f1 (COMMAND LINE):

You may continue typing these commands (which don't need the initial *) until either you type RETURN on a line by its own or press ESCAPE, when you will be returned to normal text editing. While in the command line mode, f0-f9 will function as soft keys ie will generate any text assigned to them previously using *KEY0-9.

While in command line mode, the only 'editing' functions available are cursor copying (selected by using the cursor keys) and deleting characters typed on the command line with DELETE.

Note however that you should not use certain operating system commands because they are either overridden by the editor or may interfere with its functions. You are unlikely to want to use these commands, but see chapter R.17 for further details.

R.15 Finding and replacing text (advanced)

The information in this chapter builds on that given in chapter R.11. You must read that chapter first to make sense of the additional details presented here.

Movement of the search pointer

After FIND STRING or GLOBAL REPLACE finds a matching string in the buffer, searching resumes at the next character after the last one in the found string, eg if the target string is aa and the text in the buffer aaabaababababaaa, matches will be found at the points indicated by the ↑:

```
aaabaababababaaa
↑   ↑         ↑
```

and not:

```
aaabaababababaaa
↑↑  ↑         ↑↑
```

If a found string is replaced, searching resumes after the last character of the replacement string.

Specifying the target string

The target string consists of one or more typed characters. These may simply spell out the word(s) you want to match, or may include certain other symbols or combinations of symbols which have special meanings as listed below. (These symbols are called 'special characters'.) In this way the target string contains a 'character specification' for each character which must be matched in a found string.

You may use any combination of the following specifiers to define the target string. Really lengthy or complex target strings may be processed very slowly or be rejected by the editor as too complex.

Specifying single characters

To match:	Use:
a letter of either case (a or A):	a or A
a letter of the specified case (D):	\D
a specified number (4):	4
a return character:	$
a control character ([CTRL] + A):	¦A
characters above ASCII 128 (eg 129):	¦!¦A, ¦!4 etc
<,>?+;:}{_£!"():	themselves
the special characters % or &:	themselves
other special characters (* ^\/@#.¦$-~):	\special character

Ambiguous single character specifiers – one of a predefined group

To match:	Use:
any letter, digit or _ (a-z, A-Z, Ø-9 or _):	@
any digit (Ø to 9):	#
any character (ASCII 0-255):	.

Ambiguous single character specifiers – one of a group defined by the user

To match:	Use:
any one of a set of characters (1, 3 or c):	13c
any one of a range of characters (e to g):	e-g
any character other than that specified:	~character

Notes:

- the case of letters used in specifying groups is not interpreted ambiguously as it is with explicit specifiers, (eg a will match a or A, but a-b will match a or b, not A or B)
- control characters can also be typed as [CTRL]+key, with the exception of CTRL M, CTRL U and CTRL[(these are **RETURN**, delete line and **ESCAPE**)
- characters in the range 128–137 can also be typed using [f0]-[f9], in the range 144–159 by [SHIFT]+[f0]-[f9], 160–169 by [CTRL]+[f0]-[f9]. [SHIFT]+[CTRL]+[f0]-[f9] produce nothing by default but can be programmed to produce ASCII codes from 2 upwards using *FX228,<n>.

Ambiguous single character specifiers – using character specifiers in combination

The character specifications that define groups of characters may use other single character specifiers to define that group, so that the following are all valid specifications of single characters:

@\$	matches any alphanumeric character or the dollar sign
\@a-z	matches any lower case letter or @

~#	matches any non-numeric non-space character
~3-6	matches any character other than 3, 4, 5 and 6
a-ce-z	matches any lower case letter other than d

The main exceptions are in specifying the end of a range, where the only special characters that retain their functions are ¦ and $ (eg !-# would match ! ' or #).

Multiple character target strings

A multiple character target string simply consists of a combination of single character specifiers as described above. Except as noted for , - and ~, each character specifier in the string stands for a single character within the target.

Specifying variable length strings

Using the symbols described already, you can match any strings in your text that are of a fixed known length. Two extra symbols are provided to allow matching of variable length strings, provided that the variable length portions can be defined as repetitions of single character specifications. The extra symbols are:

as many of a character as possible:	^<character specification>
as few of a character as possible:	*<character specification>

These symbols are used in the target string by placing then immediately before a single character specification defined as above; the resulting object is a string specification.

The first symbol, ^, creates a string specification that will match as many repetitions (one or more) of the following character specification as it can. Thus:

^a	matches a or AAA or aAAAaAAAaAAAaaaaaa etc
^@	matches any word (a, restitution, PL6)
^#	matches any number (6, 66, 3454545)
^~$$	matches a non-blank line

The second symbol, *, by contrast creates a string specification that will match as few (down to zero) repetitions of the following character specification as it can! Thus, *. will match nothing. String specifications using * are not really complete until you add another character specification after it which cannot be interpreted as null, eg:

*~$$	matches any line

The * symbol is most useful for specifying 'unimportant' filler characters ie characters that need not be there at all to match the target string.

The ^ symbol is generally easier to use, since use of * often requires specification of one more character than you actually want to match, causing occasional skipping of the next found string in the buffer. Searching using it is however much slower than using *.

Examples

Target	Matches
cat	Cat or CAT or cat etc
c*at	Cat or CT or caaaaT etc
d^og	Dog or dooog or DOG etc
ca*t	CA, ca etc
do^g	dog, DOGGG, etc
$	a return character
^@	a word

To count words of 10 or more characters: GLOBAL REPLACE @@@@@@@@@@ ^@ [RETURN]

To find BASIC comments that are at the start of program lines:

FIND STRING $^#*: REM [RETURN]

(carriage return, one or more numbers, zero or more spaces and colons, then REM)

To find integer variables in a BASIC program:

FIND STRING A-Za-z\@*@\% [RETURN]

(a letter or @, zero or more numbers or letters, then %)

To count the number of hexadecimal constants in a BASIC program:

GLOBAL REPLACE \&^#A-F [RETURN]

(&, then as many as possible numbers or characters A-F)

To find assembler labels:

FIND STRING \.A-Za-z*@ :$ [RETURN]

(full stop, a letter, zero or more alphanumerics, then either a space or colon or carriage return)

To find uses of multiple spaces:

FIND STRING ^ [RETURN] (space, then one or more spaces)

Note that while variable length strings are defined using single character specifiers, they cannot be treated as single character specifiers eg although ^@ matches words, ~^@ does not match non-words, but any character other than^ followed by an alphanumeric. Use ^~@ to match non-words.

Specifying the replacement string

The replacement string consists of zero or more typed characters; if you simply type <target string>/ [RETURN], the replacement string is null so found strings will be deleted from the buffer. Like the target string, the replacement string may consist of normal text or may also include special characters, as follows:

Specifying characters explicitly

a letter of the specified case (D):	D
a specified number (4):	4
RETURN:	$
a control character ([CTRL] + U):	¦U
characters above ASCII 128 (eg 176):	¦!Ø
the special characters .#@^*~-\:	themselves
other special characters (%, &, ¦ or $):	\special character

Notes:

- control characters can also be typed as [CTRL]+letter, with the exception of CTRL M, CTRL U and CTRL [(these are **RETURN**, delete line and **ESCAPE**)
- characters in the range 128–137 can also be typed using [f0]-[f9], in the range 144–159 by [SHIFT] [f0]-[f9], 160–169 by [CTRL] [f0]-[f9]. [SHIFT] + [CTRL] [f0]-[f9] produces 160–169 by default but can be programmed to produce ASCII codes from 2 upwards using *FX228,<n>.

Specifying the use of text from the found string

the whole found string:	&
part of the found string:	%number

Using %, the parts of the found string which you can use in the replacement string are those that were specified by ambiguous character or string specifications in the target string. To identify which parts of the found string to use, the ambiguous elements are numbered according to their position in the target string, from left to right, starting at 0.

Examples

To bracket words (one or more alphabetic/numeric characters, preceeded and followed by anything else): ^@/(&) or ^@/(%Ø)

To remove trailing spaces from lines: ^ $/$

To put a '+' in front of all single digit numbers (a single digit preceded by a space and followed by a space or common punctuation mark): <space># ,:;./ <space>+%Ø%1 (or <space># ~ #E./<space>+%Ø%1)

To put words that are in capitals (two or more capitals, preceded by a space or **RETURN** and followed by anything not a number or lower case letter), onto lines separate from the surrounding text: `$ A-Z^A-Z~a-z#/$%1%2$%3`

R.16 Automatic editing: editor command files (advanced)

Although the editor is normally used by pressing keys on the keyboard, it can also be used by storing the required keystrokes (commands and text) in a text file and then taking that file as input by using f1 (COMMAND LINE) and typing *EXEC <file name> (or * <file name> if using the ADFS). Text files used in this way are called 'command files'.

Command files are best used when you wish to repeat the same editing operations on several files or when the editing is such a complex sequence that you will want to refine it in steps.

While in text entry modes, the f0 - f9 function keys, TAB, COPY and the cursor keys generate ASCII codes which when read by the editor from the input buffer, are translated into the appropriate editor functions. The codes generated are:

	alone	+SHIFT	+CTRL
f0 - f9	128-137	144-153	160-169
TAB	138	154	170
COPY	139	155	171
←	140	156	172
→	141	157	173
↓	142	158	174
↑	143	159	175

To create a command file, you need to store ASCII codes 128–175 in the text buffer rather than have them intercepted and interpreted as commands by the editor. The best way to do this is to type the required functions as normal text, then use GLOBAL REPLACE to replace each function name with the correct ASCII code eg use fn0-fn9 for f0 - f9, sf0-sf9 for SHIFT + f0 - f9, etc then:

Global replace:

fn0/GOTO LINE RETURN
fn1/COMMAND LINE RETURN
sf0/DISPLAY RETURNS RETURN
tab/¦!¦J RETURN
esc/¦ RETURN

etc

When you press GOTO LINE, COMMAND LINE etc, unusual characters such as accented letters will be displayed. This is simply how ASCII codes above 127 are displayed (they can be used to make it easier to display languages that contain these characters).

Alternatively, you can first program the [f0]-[f9] keys to generate ASCII codes 128–137 by selecting command line mode with COMMAND LINE then using *KEY0¦!¦@ for GOTO LINE, *KEY1¦!¦A for COMMAND LINE etc. Then, whenever you want to type in a command normally accessed by function key, select cursor editing mode ([SHIFT]+[COPY]) and press that function key. The [SHIFT]+fn and [CTRL]+fn function keys will generate the correct codes without any programming. This method will however not allow you to place [TAB], [COPY] or cursor keys in the text buffer directly.

The only keystrokes that you cannot put into the edit buffer using any of these methods are [ESCAPE], [SHIFT] and [CTRL].

The first line of a command file should usually be blank, to take the editor out of command line mode.

Since command files execute virtually without interaction from you (all you can do is watch what happens and press [ESCAPE] if you see anything go wrong), be very careful to check that a command file does exactly what you want before using it for an important task.

The following example command files illustrate some common uses. Remember that the keystrokes shown (such as GLOBAL REPLACE) will have to be typed indirectly, using one of the two techniques described above.

To compress text to standard abbreviations:

```
GLOBAL REPLACE Cc\a\t\a\l\o\g\u\e/%0at!!
GLOBAL REPLACE Dd\i\r\e\c\t\o\r\y/%0ir!!
```

...etc

SAVE FILE [COPY]

This command uses global replacing to substitute shorter tokens for specified long words. Note in particular the use of explicit case specification (`\a\t\a` etc) so that words which were all in capitals will not be condensed, avoiding the possibility of changing the case of words by compressing and then expanding them. Note also the use of ambiguous specifications (`Cc`, `Dd`) to allow you to preserve the initial letter of words in the condensed version (they are copied across as %0 into the replacement string)

To expand standard abbreviations:

```
GLOBAL REPLACE Cc\a\t!!/%0atalogue
GLOBAL REPLACE Dd\i\r!!/%0irectory
```

...etc

SAVE FILE [COPY]

This command file uses global replacing to expand the compressed text file generated by the previous example. Note that this will only work if the tokens (`at!!`, `ir!!` etc) do not occur in the uncompressed text and relate to the same strings in both command files.

To print a file after inserting a standard file (`head`) at its start:

```
GOTO LINE 1
INSERT FILE head
PRINT TEXT PC
```

This sequence of commands moves the cursor to the start of the text buffer, inserts the file `head`, then prints the new buffer contents to the printer in continuous mode.

To print a fixed sequence of files (`file1`, `file2`, `file3`):

```
LOAD FILE file1
PRINT TEXT PC
LOAD FILE file2
PRINT TEXT PC
LOAD FILE file3
PRINT TEXT PC
```

This sequence of commands loads each file in turn, printing it to the printer in continuous mode.

(This will usually be done better by chaining the files together using the formatter comand `.ch`, as this command produces consecutive page numbering for consecutive files – see section S)

By including `*EDIT` as the first line of a command file, it can be used to select the editor from another language, and then perform editor functions. This type of command file is often used as a filing system '!boot' file, to select the editor and set options (such as display returns, release scroll margins, select text colour) at the start of editing.

R.17 The editor environment

Facilities available

When selected, the editor pages out the current language ROM (ie the editor becomes the current language) so that that language's facilities are no longer available.

Machine operating system, filing system and VDU commands may all be used by first using [f1] (COMMAND LINE). The effects of certain operating system and many VDU commands will however only last until you leave command line mode, as they would disrupt normal functioning of the editor. In particular, if the display mode has been changed while in command line mode, it is changed back again.

Soft key expansion for keys 0–9 is not available in the INSERT and OVER text entry modes of the editor, as the function keys are used to provide access to editor facilities. Soft key strings are however preserved and available in the command line and cursor editing modes. (See below for further details.)

Soft key expansion for [TAB], [COPY] and the cursor keys is not available during any of the text entry modes, but any strings defined for soft keys 10–15 are preserved and may be while used in command line mode only, after explicitly enabling their expansion using *FX219 ([TAB]) or *FX4,2 (others).

Memory usage

EDIT always uses shadow screen memory for its display to allow the maximum text buffer size. The text buffer starts at the Operating System High Water Mark (OSHWM) +1 which is usually &0E01, and ends at &7FFE, allowing a maximum of 29180 characters (including return characters).

Print formatting does not require any extra storage, so that if a file will fit in the buffer, there is room to format and print it. The editor uses &0400–&07FF (the language workspace) to store variables such as line length and number register settings, and a single byte of the CMOS RAM for the default screen mode, display carriage returns and insert/over status.

The header, footer and macro definitions used by the print formatter (see the next Section) are not stored but picked up from the text in the buffer as needed.

Text is stored in the edit buffer as a continuous sequence of ASCII codes in the order displayed on the screen, with return characters stored as ASCII 13. Text is actually stored in two sections, split at the position of the text cursor. Text before the cursor is stored at the start of the buffer, text at and beyond the cursor is stored at the end of the text buffer, with free space in the middle. This

sectioning of the text is used to make it quicker for EDIT to insert text at the cursor. EDIT stores return characters at &0E00 and &7FFF to simplify text formatting.

The marker displayed at the end of the text (a black asterisk on a white block, or a white block in mode 7) is not stored in the buffer, but generated by the editor.

Locations &70–&8F of zero page are left free by the editor, for utility programs to use. Other locations in zero page may be available depending on the current filing system – see Section O for further details.

Utility programs may be loaded into the free space between the start and end sections of the text, ideally by assembling the program to load at &0E01, and always moving the cursor to the start of the text (so that all the text is moved out of the way to the end of the buffer) before loading and running it. Programs may also be loaded and run in the transient program area if they are no longer than 512 bytes (see Section O for further details).

The editor and the MOS

On entry, the editor initialises as a language ROM, then selects the display, insert/over and carriage return display/conceal modes last used with the editor (which were stored in CMOS RAM). It also does the following:

programs [TAB] as soft key 10 (*FX219,138)
programs [COPY] and cursor keys as soft keys 11–15 (*FX4,2)
defines the soft keys to be based (*FX225–227) as follows:

Soft key codes in text entry mode

	alone	+[SHIFT]	+[CTRL]
[f0] - [f9]	128–137	144–153	160–169
[TAB]	138	154	170
[COPY]	139	155	171
←	140	156	172
→	141	157	173
↓	142	158	174
↑	143	159	175

(The [SHIFT]+[CTRL] level of soft keys is not changed – the system default is for these to be ignored.)

The editor also alters the MOS break vector (BRKV) so that it takes over error handling, as is usual for the current language.

The function keys are reset to their system default behaviour whenever you select command line or cursor editing modes or leave the editor, as follows:

	alone	+[SHIFT]	+[CTRL]
[f0] - [f9]	expansion	128-137	144-153
[TAB]	9	9	9
[COPY]			
← → ↓ ↑	cursor editing		

S The EDIT text formatter

S.1 Introduction

What the formatter does

The text formatter is an integral part of the editor, used by the PRINT TEXT command to arrange the text in the buffer into a neat format for printing to screen or printer.

The formatter actually recognises two types of information in its buffer: text, which is for printing and formatter commands, which it obeys.

If the buffer contains no formatter commands, the text is formatted as follows:

- right justified lines, with 76 characters per line
- pages of 58 lines of your text each, with no standard text at the top but standard text at the bottom of one blank line and a line with Page <page number> in its centre
- page numbers start at 1, increasing by 1 each page
- pages are separated by a form feed (ASCII 12)

If this is sufficient for your purposes, you need not use any of the formatter commands described in this section.

By including text formatter commands in the buffer, which are all of the format .<2 character name> <parameters>, you can make use of the many sophisticated formatting facilities provided by the formatter, such as:

- catering for different paper widths and lengths
- printing in bold and underlining
- changing the standard text at the top and bottom of each page
- controlling pagination
- centering, right justifying or indenting text
- storing numbers and performing arithmetic with them
- storing standard text and commands for reuse
- generating indexes and contents lists

Text formatter commands have no effect on the format of the text in the buffer, only on the way it is formatted when printed using PRINT TEXT.

Terms used in this section

The following words and phrases are key terms used throughout this section:

text – text is a continuous sequence of one or more characters that will be printed rather than obeyed by the formatter

paragraph – a paragraph is a continuous piece of text or commands which starts after a return character and ends with a return character

right justification – right justification produces text paragraphs whose lines all end at the same column (the current right margin); the formatter produces right justified text by inserting spaces between words before printing the line

formatter commands – are 'words' of the form .<aa>, which are not printed but obeyed by the formatter

parameters – a character, words or numbers after a formatter command that are used by that command

command paragraph – a paragraph that consists only of formatter commands

A full summary of the EDIT formatter commands is given in section S.4.

S.2 Printing to the screen

Before printing your text onto paper, you should always print it to the screen to check that the layout is as intended:

f8 (PRINT TEXT)

```
S(creen), P(rinter) or H(elp) ? S RETURN

C(ontinuous) or P(aged) ?
```

P RETURN (to view the formatting a screenful at a time)

or

C RETURN (to view the formatting in a continuous stream)

or

H RETURN (to view a summary of the formatter commands' syntax)

If you select the paged option, you must press SHIFT after each screenful has been printed to see the next. If you select the continuous option, you can freeze the display by holding down CTRL and SHIFT together.

The formatter displays text as a very good approximation to the way it will appear on the printer, but you should be aware of the following minor limitations:

- underlining and bold will not be displayed correctly in `MODE 7`.
- if the formatter line length (`.ll`) plus the page offset (`.po`) is longer than can be displayed on a single line in the current screen mode, formatted lines may spread over more than one screen line.
- if you have used the `.oc` or `.on` commands to send codes to the printer only, these codes will not be seen on the screen, to avoid potentially confusing results (such as causing screen mode changes).
- characters will be displayed using the system's character set, not the printer's (printers often print # as £, £ as # etc and are unlikely to print characters above ASCII 127 as displayed).
- by default, each page ends with a form feed (ASCII 12), which clears the screen – this may make it difficult to check text near the ends of pages (redefine the footer using `.fo` if it does).

S.3 Printing on a printer

When you are satisfied that the text formats correctly, use PRINT TEXT to print it on the printer:

f8 (PRINT TEXT)

```
S(creen), P(rinter) or H(elp) ? P RETURN

C(ontinuous) or P(aged) ?
```

P RETURN (to print a page at a time eg on single sheet paper)

or

C RETURN (to print the text in one go – on continuous stationary)

or

H RETURN (to view a summary of the formatter commands' syntax)

If you select the paged option, you must press SHIFT after each sheet has been printed to print the next.

The print formatter is designed to get the best out of the wide variety of printers it might be used with. Most of the formatter commands will work with most printers, but for all of the commands to work, your printer should:

– not do an automatic line feed when sent carriage return (ASCII 13).

– use fixed character spacing (ie not proportional spacing).

– feed the paper to the top of the next sheet when sent form feed (ASCII 12).

– use the standard ASCII character set, including control codes.

If your printer obeys all these constraints, you should set the printer ignore character to null before printing (using *CONFIGURE IGNORE RETURN followed by CTRL+BREAK to store it 'permanently' or *IGNORE RETURN). If your printer does not behave as required, you may be able to 'reprogram' it so that it will, by setting switches on the printer or by sending it special software codes – refer to the printer manual for guidance.

If your printer performs an automatic line feed when sent carriage return, you should set the printer ignore character to line feed (using *CONFIGURE IGNORE 10 RETURN followed by CTRL+BREAK to store it 'permanently' or *IGNORE 10 or *FX6,10) and should not use underlining (.bu .eu .ul), bold text (.bb .eb .bl) or 'no line feed' (.nn) commands.

If your printer does not obey form feed correctly, you will need to redefine the text footer in each file, using `.lv <n>` in the footer to make the printer feed to the next page, and avoid using the form feed command (`.ff`).

If your printer cannot print monospaced text (which is most unlikely), facilities such as right justification (`.rj`), centring (`.ce`) and tabs will work inaccurately.

If your printer does not use the standard ASCII character set, you can correct for use of different codes (but not omissions) by using the translate (`.tr`) command.

S.4 The formatter commands

Command syntax

Formatter commands are of the general structure .<name><parameter>. Parameters are described in the next chapter.

The <name> is always lower case and two characters long. The dot placed at the start of the command name identifies it as a command or macro (see 'Macros', below) and is termed the control character. You may change the control character if '.' is not convenient, using .cc<new control character>, eg .cc@

The commands are:

.af	assign register format
.an	assign register number
.bb	begin bold
.bl	bold line
.bp	begin page
.bu	begin underline
.cc	set control character
.ce	centre line
.ch	chain file
.co	comment line
.dm	define macro
.ds	set double spacing
.eb	end bold
.ef	define even page footer
.eh	define even page header
.en	end of .dm, .ef, .eh etc
.ep	begin even page
.eu	end underline
.ff	form feed
.fo	define footer text
.he	define header text
.ic	close index file
.ig	ignore
.in	set left margin indent
.io	open index file
.ix	index text
.ju	right justify
.ll	set line length
.ls	set line spacing

.lv	leave blank lines
.ne	needs lines
.nj	do not right justify
.nn	no new line
.oc	output character and count it
.of	define odd page footer
.oh	define odd page header
.on	output character and not count it
.op	begin odd page
.os	operating system command
.pl	set page length
.po	set page offset
.rØ-9	print register contents
.rf	right justification
.sp	insert spacing lines
.ss	set single spacing
.ta	set tabs
.tc	set tab character
.ti	temporary indent
.tr	set character translation
.ul	underline line

These commands are described below, grouped by function rather than alphabetically.

Command positioning

Commands only affect the text that comes after them, so that their positioning is vital. It is particularly important to put commands that define text layout (page length, header text, etc) right at the start of the text, before any text which will be printed. Header, footer and macro definitions only work if they are still in the text buffer at the time of formatting, so they must be repeated in each file to be printed.

The following commands can be used anywhere in printed text, but should not be mixed with other commands:

.bb, .bu, .eb, .eu, .oc, .on, .rØ-r9

Thus

```
and here is .buthe.eu news
```

is valid, printing as

and here is the news

The other commands (and macros) must be used in paragraphs consisting only of commands (with their parameters) and macros – such a paragraph will be referred to as a 'command paragraph' in this section. Commands in command paragraphs may be separated by any number of spaces to aid legibility. Thus the following are all valid:

```
.bp.bl.tr#£ RETURN
```

```
.bp .bl .tr#£ RETURN        (identical)
```

```
.bp RETURN
```

```
.bl RETURN
```

```
.tr#£ RETURN
```

```
.ta 10,20,30 .ioindex1 RETURN
```

The .os and .io commands must always be used at the end of command paragraphs.

The commands .he, .fo, .eh, .oh, .ef, .of, .ix, .dm are used to store the text and/or commands that comes after them, up to a .en, for later use. Formatter commands or text to be used immediately should not be placed between these commands and the .en that terminates their text:

eg

```
.he.bl.ce
Provisional draft
.en
```

This defines standard text for the top of pages – none of the commands or text will be used at this point in the formatting.

Command positioning errors

If you use an unrecognised command or use a valid command in the wrong place (eg .bp in the middle of a printed line), it will not be obeyed but it and the rest of the line will be printed as text.

If you follow a command (other than .bb, .bu, .eb, .eu, .oc, .on, .r0-r9) with anything other than the parameter(s) it requires, another command or spaces then the rest of the line will not be obeyed nor printed.

Thus the following are all invalid:

`.bp !.bl.tr#£ RETURN`	non-space character
`.bpbl.tr#£ RETURN`	missing control character
`that was .blnasty too`	a command in text paragraph

Finally, the maximum number of characters that you should include in a command paragraph is 255. This limitation is most unlikely to cause any problems.

S.5 Command parameters

Parameter types

Formatter commands other than `.ta` always have the same number (0 or 1) and type of parameters. The main types are:

numeric – either a constant in a defined range, or an increment/decrement operation (`+3` or `−14`)

register expression – complex; see 'Storing and manipulating text and numbers – registers and macros', below

character – a single character (`A`, `b`, <space>)

block – any number of commands and text, terminated by `.en`

The `.io` and `.ch` commands have a file name parameter, 1 or more characters in length (length and legal characters are filing system dependent). The `.ta` command takes a variable number of numeric constants as parameters, separated by spaces or commas. The `.os` command takes a string of one or more characters, that should be a legal `*` command without the asterisk (eg `.os adfs`)

Parameters other than the single character parameters may be separated from the command by any number of spaces to aid legibility; single character parameters must follow the command immediately ie without any intervening spaces, as the space character is itself a legal character parameter.

The `.dm` command has a 2-character macro name followed by a block.

Parameter errors

If a numeric parameter is out of range or non-numeric, it is taken as 0. If an increment/decrement operation would take a command parameter outside its defined range, the parameter is set to 0. The legal range for most numeric parameters is 0-255.

If a register expression is non-numeric, it sets the register to 0. If a register expression is assigned a number greater than 255, the number used is <n> MOD 256. If an increment operation takes a register value over the allowed maximum (4095) it will become this value MOD 4096. If a decrement operation takes the register value below 0, when that register is printed it will be displayed in a strange format or produce other unpredictable results.

S.6 Defining the style of the page

Pages will normally be printed with a maximum of 58 lines of your text, without any standard text (header) at the top, but with standard text of a blank line followed by 'Page n' centred in another blank line at the bottom (footer text). The number of blank lines after the page number to the text on the next page depends on the form length used by your printer, since after printing the Page number at the bottom of the page, the formatter prints a form feed character (ASCII 12). The text is printed at the left margin.

The following commands allow you to redefine this style extensively. They should only be used at the start of the text buffer (ie before any printed text) to avoid problems with changes in formatting half way through a page.

To change the number of lines of text per page, redefine the *total* number of lines, using .pl <lines> or change the number of lines by <n> using .pl +<n> or .pl −<n>. The number of lines specified should not include those used at the top and bottom of the page for the header and footer.

To print the page away from the left margin on the printer (eg to allow room for stapling or punching), use .po <spaces>. Note that this offset does not change the line length ie it is entirely separate from text indentation. To change the page offset by <n>, use .po +<n> or .po−<n>.

To change the page number printed on the pages, use .an 0<page number>. (This is an example of using a numeric register – this subject is dealt with fully later.)

To change the standard text printed at the head of each page, use .he; the following lines, up to .en, will be formatted and printed at the top of each subsequent page.

If you wish to define different header text for use on odd-numbered and even-numbered pages, define the even page header using .eh instead of .he and the odd page header using .oh.

To change the standard text printed at the foot of each page, use .fo; the following lines, up to .en, will be formatted and printed at the bottom of each subsequent page.

If you wish to define different footer text for use on odd-numbered and even-numbered pages, define the even page footer using .ef instead of .fo and the odd page footer using .of.

You may use most of the other formatter commands within the header and footer definitions. The current page number is stored in a numeric register – r0.

To print this in header or footer text, use `.rØ` where you want the page number to appear.

Note that in the header and footer definitions, all the text following the command (`.he`, `.ef` etc) is significant, including the return character after the command. Thus 'footer 1' would be printed preceded by a blank line; 'footer 2' would not:

```
.fo
footer 1
.en
.fofooter 2
.en
```

S.7 Controlling pagination

The formatter keeps a record of the number of lines printed on a page (in register r1). When this count reaches the page length (which defaults to 58 but can be changed using .pl<n>), the formatter produces a new page automatically by printing the footer text and resets the line counter to 0. The following commands are provided to allow you to override this mechanism where you wish.

To force a new page to be started regardless of the number of lines printed, use .bp, .ep or .op. The .bp command ends the page by printing blank lines until the footer position is reached, then produces a footer as automatic pagination would. The .ep and .op commands end the current page as .bp does, but will also print another blank page if necessary to ensure that the following text will be printed on an even or odd-numbered page, respectively. The .ep and .op commands are useful for ensuring that parts of a two-sided document fall on the appropriate side (eg so that all new chapters start on a right-hand page).

To force a new page if you are within a specified number of lines of the bottom of the page, use .ne <number of lines>. This is a useful way of keeping text (such as tables or text figures) from being split across pages. (Also see .lv, in 'Defining the style of paragraphs', above.)

To force a new page without producing headers and footers, incrementing the page number or resetting the line counter, use .ff. This command should be used at the end of footer text definitions (.fo) to advance the printer to the start of the next page.

S.8 Defining the style of paragraphs

A paragraph is one or more lines of text terminating in a return character. By default, paragraphs are right justified to a line length of 76 characters, single spaced, with no offset from the left margin. The following commands can be used to change the formatting of subsequent paragraphs from this default.

Right justification inserts spaces into all the lines of a paragraph except the last so that these lines all finish at the right margin. To turn right justification off, use `.nj`. To turn right justification on again, use `.ju`.

To set the line length, use `.ll` <number of characters>. (This effectively sets the right margin.) To change the line length by <n>, use `.ll +`<n> or `.ll -`<n>.

To indent paragraphs from the left margin, use `.in` <number of spaces>. To change the indentation by <n>, use `.in +`<n> or `.in -`<n>. As this command does not change the position of the right margin, indented paragraphs are narrower than those at the left margin.

To alter the line spacing, use `.ss`, `.ds` or `.ls` <line spacing>. These force single spacing, double spacing and spacing of 1 to 10 lines, respectively. To change the line spacing by <n> use `.ls +`<n> or `.ls -`<n>.

To put blank lines in your text, you may either type a number of return characters, or use `.lv` <lines> or `.sp` <lines>. The former ensures that these lines are not split across a page break, the latter allows them to be split. Use `.lv` to leave space in the printed text for insertion of drawings etc.

S.9 Defining the style of single lines

The following commands are provided so that you can print a single line with a different format to that currently set, without disturbing these settings. Note that the length of this line is determined by the current settings of `.ll` and `.in`; if the line of text formats to more than one line, only the first one will be printed in the different format.

To centre a line, use `.ce` on the preceding line.

To right flush a line (equivalent to turning justification off and moving the text across the page so that it ends at the right margin), use `.rf` on the preceding line.

To change the indentation for a single line, use .ti <spaces> on the preceding line; to change it by <n>, use `.ti +`<n> or `.ti -`<n>.

To prevent the return character at the end of a line being printed, use `.nn` on the preceding line. This allows you to combine formatted pieces of text defined on separate lines into a single line, as in the following 2 examples.

To specify left flush, centred and right flush components of a single line, use:

```
.nn
Left bit
.nn
.ce
middle portion
.nn
.rf
right bit
```

Which produces:

```
Left bit                    middle portion                    right bit
```

Examples

To number indented paragraphs outside the paragraph left margin, you can use:

```
.in20
.nn
.ti15
1
```

text of the first paragraph, which is formatted neatly into a narrow column with the paragraph number outside its left margin

```
.nn
.ti15
2
```

second paragraph ... etc

```
.in0
```

Which produces:

```
1    text for first paragraph which is formatted neatly into
     a narrow column with the  paragragh  number outside its
     left margin


2    second paragraph etc
```

(For another method, see 'Using tabs', below.)

S.10 Underlined and bold text

To underline one or more characters anywhere in the text, precede the characters with .bu and follow them with .eu.

To underline a whole line of text, put .ul on the previous line. Note that if this 'line' of text formats to more than one line, only the first one will be underlined.

To make one or more characters bold anywhere in the text, precede them with .bb and follow them with .eb.

To make a whole line of text bold, put .bl on the previous line. Note that if this 'line' of text formats to more than one line, only the first one will be emboldened.

You can use these commands in combination ie have bold underlined text.

S.11 Using tabs

Tabs provide a convenient way of lining up columns of text or numbers that also take up less room in your documents than using spaces. To use tabs, type the tab character immediately before the text you want tabbed; when that text is formatted, it will be printed starting at the next tab stop to the right or if there are no more tab stops to the right, the tab is ignored. Formatter commands are provided to redefine the tab character and the position of the tab stops.

Note that this tabbing facility is quite different from that provided by the editor: in the editor TAB moves the cursor, while the formatter tab facility affects the format only when text is printed. Tables constructed using the tab facility will take up much less space than tables constructed using the TAB key.

The tab character defaults to [CTRL]+I. To change the tab character use `.tc`<character> eg `.tc>` to set it to >.

To redefine the tab stops, use `.ta` <n>,<n>,<n> etc. By default they are set at intervals of 8 ie columns 8, 16, 24, 32, 40 etc up to the line length set by `.ll`.

Tabs are frequently used in tables to align the columns, but also provide a simple way of implementing indented paragraphs with a number or title at the left margin, eg to indent the paragraph 20 spaces:

```
.ta 20
.in20
.ti0
Example.<[CTRL]+I>    and now the paragraph text, which is moved to
start at the indent by the embedded tab character
```

Which is printed as:

```
Example.        and now the   paragraph text,  which is moved   to start at
                the indent by the embedded tab character
```

(A similar result can be acheived without using tabs – see 'Defining the style of single lines', above.)

S.12 Ignoring text and commands

Although you will usually want text in your files to either be printed or executed as formatter commands, this is not always the case. The formatter provides commands which mark text so that it will be ignored by the computer. This is useful for leaving notes in a file for you to read when editing it; when you print the file, the notes will not be printed. This facility can also be used for making trial 'deletions' to see whether the formatted text is better without certain commands or text.

To stop a single paragraph being executed/printed, put it after `.co` on the same line eg `.co` this is a comment line

To stop a group of lines being executed/printed, precede the group with `.ig` and follow it with `.en`.

Example

```
.ig and all the text and commands from here
.bp.in10.ll-10
to there will be ignored
.en
```

S.13 Linking files for printing

If you are writing lengthy documents, you will have to split the text across several files. The formatter provides a method of linking these files together so that to print the whole document, you need only load and print the first file.

This facility is used by placing `.ch` <next file name> on the last line of each file, then printing the first file. Make sure that filing system monitoring has not been turned on (using `*OPT1`), as any messages displayed by the filing system will be printed too.

In the formatted output, the transition between chained files is only shown as a carriage return.

Formatter commands placed in the first file will be effective for all subsequent chained files, with the exception of header, footer and macro definitions, which *must be repeated at the start of each file.*

It is possible to print multi-file documents without using the chain command, but you will then need to load them by hand and repeat header, footer and macro definitions at the start of each file and set page numbers too.

S.14 Printing special codes and translating characters

To use some of the facilities on your printer such as alternative print styles, vertical tabbing or setting character spacing, you may need to 'print' control sequences that include characters which it is difficult or impossible to type into your text with the editor. To solve this problem, two commands are provided so that you can specify characters for printing by their ASCII codes:

- to output a character and count it as a character for line formatting, use `.oc` <ascii code>.

- to output a character and not count it for line formatting, use `.on` <ascii code>.

When the text is formatted, the specified code will be sent to the printer only (by preceding it with ASCII 1), to avoid problems caused by the VDU driver executing VDU commands. If your printer can obey backspace (ASCII 8), these commands can be used to construct new characters from combinations of characters on the printer eg `u.on8.on34` would be printed as u, backspace, double quotes – ü, the german character 'u umlaut'; `=.on8.on47` would be printed as =, backspace, oblique – ≠, 'not equal to'.

The formatter also allows you to change how any ASCII character will be printed, using `.tr`<typed character><printed character>. This specifies that wherever the <typed character> occurs in the text, it will be printed as the <printed character>.

You may translate as many characters as you wish. Initially, the only character that is translated is [CTRL] + J which prints as a space, so that by typing [CTRL] + J instead of a space between words in a phrase you can stop the formatter splitting the phrase across lines. You may translate any characters (ie ASCII 0–255) to any others.

Translation is useful for printers with incompatible character sets (eg which print # as £, £ as #: `.tr£# .tr#£`) or incomplete character sets (eg missing , so translate them to normal brackets: `.tr( .tr)`). (The same effect could be achieved by editing the problem characters in the file before you print it, but the file would then seem to contain incorrect characters and so be more awkward to edit.)

S.15 Storing and manipulating text and numbers

Most of the output from the formatter is text that you typed explicitly, like constants in a programming language. To provide more flexibility, the formatter provides numeric registers, which are designed to hold variable numbers, like variables in a programming language. The contents of these registers may be changed, used to change other registers or printed.

The formatter also provides a 'shorthand' method of specifying frequently used text or commands: by first defining the text or commands as a macro, you can use them subsequently by just typing the macro name.

S.16 Numeric registers

There are ten numeric registers, named r0 to r9. Each register may hold a positive integer in the range 0 to 4095. Attempting to produce positive contents outside this range results in <number modulo 4096>; attempting to produce negative contents changes the display format and produces a result of <number modulo 4096>. Two of these registers are used by the formatter and should not be used for other purposes: r0 holds the page number and r1 the line number (you can change these directly eg to start page numbering from a number other than one).

The numeric registers may be used to hold values either assigned to them or produced using simple arithmetic and may be printed in a range of formats.

All the numeric registers initially contain 0. To change the value of a register, use .an<register number><register expression>, as follows. A space between .an and the register number is optional, but you must not leave a space between the register number and expression, unless specified below.

To assign a value to a register, use .an <register number> <value>. The value must be in the range 0 to 255 (values outside this range are taken as <value> MOD 256). Note that you must type a comma or one or more of spaces between <register number> and <value>.

To assign the current value of one register to another, use .an <number of register to change>=<number of register to use>.

To change the current value of a register by a fixed amount, use .an <register number><+ or -><change>

To print the value of a register, use .r<register number> where you want the text to appear. The contents will be output in the format specified for that register (see below) and formatted as if that text had been typed in the normal manner.

The intial output format of each numeric register is 0. To define the output format, use `.af` <register number> <format number>. The formats available are as follows:

0	gives 0,1,2...10,11,12...100,101,102
1	gives 00,01,02...10,11,12...100,101,102
2	gives 000,001,002...010,011,012...100,101,102
3-7	gives print fields of 4–8 minimum width, similar to formats 0–2
8	gives capital roman numerals ie 0,I,II,III,IV...
9	gives lowercase roman numerals ie 0,i,ii,iii,iv...
10	gives 0,A,B...AA,BB,CC...AAA...
11	gives 0,a,b...aa,bb,cc...aaa...

Note that register contents can only be used as text or in the `.an` command – they cannot be used in other commands eg `.ll.r3` will not set the line length to the current contents of r3.

S.17 Macros

Macros are named blocks of text and/or formatter commands in the text buffer, which can be 'typed' over and over again by typing just the name of the macro in the text buffer (much like a procedure or function call in BASIC).

Macros provide a quicker and easier way of typing text that contains repeated elements. The resulting text is more compact, so you can fit more into the buffer or onto a disc. By using macros for repeated elements, you also make it far easier to check and alter them (you just check or alter the one definition of the macro).

Defining macros

Before you can use a macro, you must define it. To define a macro, put `.dm` <name> before the lines to include in the macro, and `.en` after them. The macro name can be any two character string except defined formatter commands. The case used for the name is significant (eg `.dm BZ` and `.dm Bz` define different macros). Any text on the same line after `.dm` <name> will be ignored.

The macro text can include any text or formatter commands, and can use other macros as long as they will be defined by the time the macro is used. The definition must not use itself.

The macro definition is not printed by the formatter.

If a definition includes any of the commands which are terminated by a `.en` command (`.dm`, `.ig`, `.ix` and the header/footer definition commands), this must be included within the macro definition ie the definition will contain two `.en` commands – one to end the included command, one to end the macro definition.

Using macros

To use a macro after defining it, put `.`<name> at the start of a line or in a command paragraph (as described in 'The formatter commands', above). Since macros cannot be used in text paragraphs, the shortest text element they will normally be used for is a complete paragraph.

A macro can only be used if its definition is still in the text buffer and that definition has already been read by the formatter.

Examples

For example, to start each section on a new page, preceded by the centred bold text 'Section nn', where nn is an automatically incrementing number, you could use the following formatter commands at the start of each Section:

```
.bp.ce.bl.an3+1
Section .r3
```

Alternatively, you could define a macro to do this:

```
.dm hd
.bp.ce.bl.an3+1.nn
Section .r3
.en
```

And then simply put the following at the start of each section:

```
.hd
```

S.18 Generating contents lists and indices

Formatted text is normally printed on the screen or screen and printer, but there are times when it would be convenient to have it stored in a file. While it is possible to store output in a file using *SP00L, the formatter provides a more elegant method, that allows you to store all or part of the formatted text in a specified file called the 'index file'.

To specify the index file, use .io <index file name> before the text you want indexed; this opens a file of that name. If you do not specify an index file, indexing will be ignored. You may only have one index file open at a time (see .ic below).

To indicate the text you want placed in the index file, precede the block with .ix and follow it with .en. The text can follow .ix on the same line; .en must be at the start of a line.

You may index almost any text or commands you like; this text will be formatted as if it were text to print, except that headers and footers will not be provided and the line counter and page number will not be affected. Indexed text will *not* be printed, only sent to the index file, although the index file can be printed automatically, using .ic and .ch (see below).

Note: Formatter commands within this text will be obeyed normally, so you would not normally include commands that change the page or paragraph style (such as .pl, .in, .he).

A typical index entry would be:

```
.io rev2ind
<normal text for printing>
<normal text for printing>
<normal text for printing>
.ix
.nn
text formatting
.rf
Page .rØ
.en
<normal text for printing>
<normal text for printing>
```

If this was used on page 22 of the document, the following would be written to rev2ind:

```
Text formatting                                    Page 22
```

To reduce the amount of typing required to use indexing in this way, you can use global editing. Mark words to be indexed uniquely, on a line of their own eg

`!!text!!` RETURN

then use global editing:

GLOBAL REPLACE: `$!!*.!!/$.ix$.nn$%Ø$.rf$Page .rØ$.en$`

To format the whole text buffer into a file, bracket the whole text with `.ix` and `.en`. The text will be formatted correctly with the exception that header and footer text will not be produced and the page number and line number registers will not be incremented. This method is useful for converting text containing formatter commands to 'plain text', eg for editing by editors that format as text is entered, rather than as text is printed (such as VIEW).

When the editor finishes formatting, the index file will be closed automatically, but you may close it before this using `.ic`. This allows you to route index information from different parts of a document to different index files by closing one index file with `.ic` then opening another with `.io`, or to print the index file at the end of the current file by chaining it with `.ch` <index filename>.

T System editor/formatter error messages

T.1 Introduction

This section describes error handling, diagnosis and correction for the system editor and text formatter. The individual topics are:

– meaning and resolution of editor error messages

– detection, analysis and correction of formatting errors

– dealing with other problems

Each of these subjects is described in turn below. MOS and filing system errors are dealt with separately, in the appropriate sections.

When errors are detected, they are handled by the editor and print formatter in quite different ways. The editor reports the error and aborts the command; the formatter does not report any errors and continues formatting as best it can.

T.2 Editor errors

The editor always reports detected errors by printing error messages at the bottom of the screen, so that detecting editor errors is not a problem.

The command in error is not obeyed. The text in the buffer will be left unchanged but the editor may clear marks or move the text cursor to the start of the buffer or screen line.

If the editor stops functioning, refer to 'Dealing with other problems', below.

General purpose error messages

Press ESCAPE to continue

To make sure that you notice the error message, this message is displayed with the majority of error-specific messages, in a distinct block at the bottom of the screen. You cannot proceed any further until you press [ESCAPE] to confirm that you have seen the error message.

For help type: shf-f5 D RETURN

To remind you how to obtain more information on the editor commands, this message is displayed with the majority of error-specific messages if you are not already in descriptive mode. After pressing [ESCAPE] to acknowledge the error message, you may select tutorial mode by pressing (SET MODE) and then typing D (or d) [RETURN].

Error-specific messages:

Bad marking

You have tried to set more than 2 marks or have not set the right number of marks (with MARK PLACE) for use with this command eg you are trying to delete a block with 2 or 0 marks set rather than 1. The marks will be cleared.

Bad number

You have typed a number outside the allowed range or letters rather than digits, in response to an editor prompt for a number. Note that numbers are always decimal – you cannot use & to specify a hexadecimal number.

Bad replace field number

In specifying use of parts of a target field in a replacement field, you have use a field number (n in %n) which is either non-numeric or too large

Bad use of stored name

Using INSERT FILE you have either pressed [RETURN] or [COPY] [RETURN] in response to the prompt for the file to insert. You must specify the file to load by typing its name with this command.

Error with \

\ is used to turn off the special meaning of the following single character in a search or replace string. You have used it incorrectly by not following it with a character. To specify the character \, use \\.

Error with ¦

¦ is used in search and replace strings to specify a control character (eg ¦A specifies **CTRL** A). You have used it incorrectly by not following it with a character. To specify the character ¦, use \¦.

Error with ~

~ is used in target strings to specify 'any character other than the following'. You have used it incorrectly by not following it with a character. To specify the character ~, use \~.

File too big

There is not sufficient room in the text buffer for the file that you have tried to load or insert.

Line not found

You have specified a destination line with GOTO LINE which is beyond the end of the file. (To count the number of lines in the buffer, use GLOBAL REPLACE $ [RETURN].)

Mark(s) set

You have tried to use a command that cannot be used with marks set (such as INSERT FILE or FIND STRING with a replace string): use CLEAR MARKS first.

No name found

Using the load, save or insert file commands you have either replied [COPY] [RETURN] to the prompt for the filename and have not yet typed a filename, or have replied [RETURN] and the >filename string at the start of the file is either missing or invalid (eg no > character or the line is more than 127 characters long).

No previous string

You have replied [RETURN] to the FIND STRING or GLOBAL REPLACE prompt, but have not previously specified a find/replace string for that command. (Note that the string previously used with FIND STRING is *not* available for use with GLOBAL REPLACE or vice versa.)

No room

The text buffer is full so there is no room left for you to type more text. If you are preparing a document, split it into separate files and chain them together for printing. If you are preparing a program, split it into separate files, then *EXEC them after selecting the language to produce the complete program.

Not enough space to return to language

When you use RETURN TO LANGUAGE, text in the buffer is effectively *EXEC'ed into the selected language. This error message indicates that there is not enough memory space for the program text in the text buffer and the tokenised program text in the language workspace. Save the program as a text file, select the language, then *EXEC the text file from the language, to convert it into a program.

Only 0, 1, 3, 4, 6, 7, D and K

The only valid display modes are 0, 1, 3, 4, 6, 7, D and K. The editor is always in shadow mode and MODEs 2 and 5 provide too few characters per line for editing work.

Syntax incorrect

The target or replacement string you have specified with FIND STRING or GLOBAL REPLACE, uses incorrect syntax (eg a-z/[RETURN]). Refer to Section R (The System Editor) for details of correct usage.

Too many find multiples

You have used * and ^ more than four times in total in a target string with FIND STRING or GLOBAL REPLACE. Use a simpler expression.

T.3 Text formatting errors

Detecting formatting errors

The text formatter does not report *any* formatter command errors, so detecting them is entirely up to you. Check the source text (usually by viewing it in the editor) against the printed results, being particularly careful to check for missing text.

Analysing formatting errors

If the text has not formatted precisely as you wanted it to, then it contains formatting errors. Formatting errors may be caused by any of the following:

- incorrect formatter commands (spelling, syntax, positioning, parameter range)
- correct use of the wrong formatter commands (ie a 'design' error)
- incorrect setting up of the editor/printer interface (printer switches, ignore character, character translation, avoiding unsupported commands)
- it is not possible to achieve what you want (ie a 'design' error)

In general, the position of a formatting error in the printed document shows where the error is in the source document. The error may actually be earlier in the document for held settings such as line length (`.ll`) or character translation (`.tr`), but will *never* be later in the document.

The cause of most formatting errors will be obvious after inspecting the source text in the buffer, but the following summary of symptoms and causes may be helpful:

- failure to obey a command: check the command spelling, parameter range and position, and the current control character.
- failure to obey a macro: check the macro spelling and position, the current control character and that a valid macro definition is still in the text buffer, before the macro invocation.
- text all in bold or all underlined: you have used `.bb` or `.bu` without including `.eb` or `.eu` respectively.
- bold text prints as normal intensity, repeated on the next line: the printer is performing a line feed when it receives carriage return. Reprogram the printer or avoid use of bold and underlining.
- underlined text is printed with the underline characters on the following line: the printer is performing a line feed when it receives carriage return. Reprogram the printer so that it does not perform line feed when it receives carriage return, or avoid use of bold and underlining.

- text prints all on one line: you have not changed the printer ignore character to allow linefeeds to be sent to the printer. Use *IGNORE [RETURN], or *CONFIGURE IGNORE [RETURN] followed by [CTRL] + [BREAK].
- parts of text missing: check for previous use of .ig (ignore), .co (comment), define header, footer, macro or index entry commands. Text that follows commands other than .bb, .bu, .eb, .eu, .oc, .on or .rØ–9 in the same paragraph will also be ignored. Check that the missing text is not being read as one of the preceding seven commands (eg an assembler label declaration such as .buffer would actually be read as <begin underline>ffer).
- formatter freezes or repeats the same text: check for command paragraphs of more than 255 characters, or text paragraphs containing a sequence of more than 254 spaces.

T.4 Dealing with other problems

Editor locks up

If the editor seems to be ignoring your commands and typing entirely, or if you want to abort a current operation, press [ESCAPE]. You should now be able to continue using the editor with any text in the buffer intact.

If [ESCAPE] fails, press [BREAK] and then [f9](OLD TEXT) to restore the text buffer.

If the editor still does not respond after [BREAK], press [CTRL]+[BREAK] – you will no longer be able to recover the buffer contents using OLD TEXT, but can do so indirectly, using the method described under the next heading.

Recovering lost text

If you have lost the text in the buffer by pressing [BREAK] (or [CTRL]+[BREAK] with EDIT configured as the default language), you can recover it by pressing OLD TEXT *before you type anything else.*

If you have deleted a large block of your text by mistake (using MARKED DELETE), overwritten it by loading another file, or lost it by pressing [CTRL]+[BREAK] (with another language configured as the default), you may be able to recover most or all of the text by saving the entire text buffer contents then reloading it, as follows.

The editor text buffer stretches from the operating system high water mark (OSHWM – usually &0E00 in the I/O processor) to &7FFF, with the text split into two parts (to increase the speed of inserting and deleting single characters). The text from the start of the file to the cursor is stored from &0E01–<n1>, and text from the cursor to the end of the file is stored from <n2>–&7FFE. ASCII 13 (carriage return) is always stored at &0E00 and &7FFF.

Although you will usually use the editor SAVE FILE command (**f3**), you can also save the entire buffer contents by using `*SAVE` <filename> `801 7FFE`. This is only necessary if are trying to recover text that the editor has lost track of (eg after [CTRL]+[BREAK], a block delete or loading a new file).

U The TERMINAL Emulator

U.1 Introduction

The computer has been designed to offer an impressive range of facilities and processing power in the standard configuration, which can easily be extended by adding a variety of 'add-ons'. The TERMINAL terminal emulator provides you with a route to possibly the ultimate 'add-on', by giving you access to remote computers and computerised systems. This can provide access to facilities such as:

- Use of very fast computers with large memory and disc capacity, to develop and use programs that would be difficult or impossible to work with on a microcomputer.
- Sending or receiving messages, data, telex etc worldwide, through Electronic Mail facilities such as TELECOM GOLD (Dialcom).
- Rapid, automated access to up-to-date well-organised information sources, such as airline schedules, stock exchange prices & research paper indexes by using any of the thousands of information databases available worldwide.
- Informal interchange of general information, programs, advertisements etc through computer club 'bulletin boards'.

To use the facilities of such 'remote systems', you need three main things:

- a terminal (a device containing a display, keyboard and interface circuitry)
- some means of connecting the terminal to the system you want to use
- permission to use the system!

A terminal usually has a serial interface such as RS232 or RS423 which is used to connect to the remote system, and may provide a range of facilities independently of the remote system ('local intelligence') such as graphics or editing lines of text before they are sent to the remote system.

A terminal is usually connected either by wiring it directly to the (local) system through its serial interface, or by wiring the serial interface to a modem or acoustic coupler which then is connected to the (remote) system via telephone lines or a dedicated data network.

To spare you the expense of buying a terminal, the computer has terminal emulation software and a serial interface built in, so that it can be used as a terminal. The terminal emulation software is a program in ROM that runs as a 'language', making the computer behave as a dedicated terminal while still letting the user take advantage of BBC microcomputer features such as high resolution colour graphics, built-in filing systems and use of a local printer.

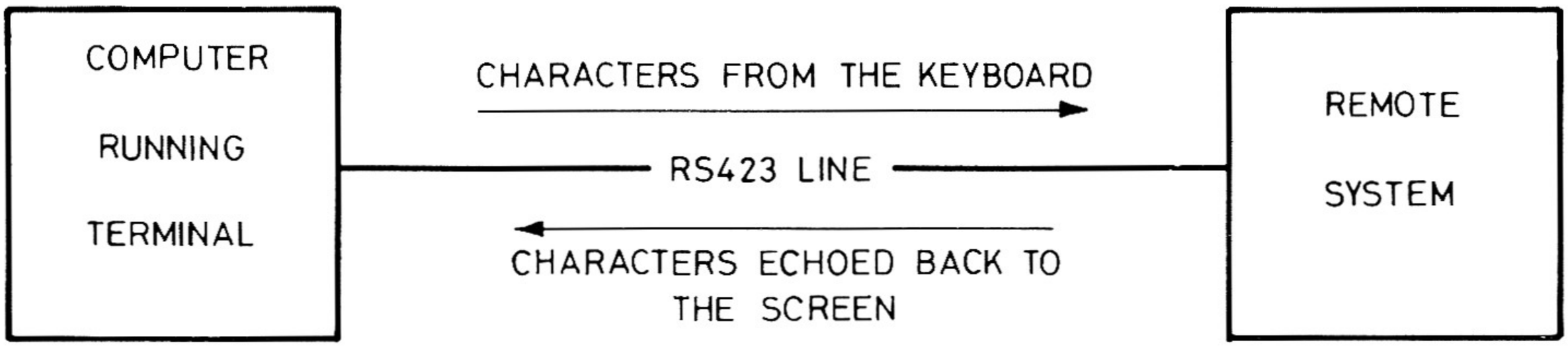
The terminal emulator is highly intelligent. It gives you:

- full screen working, access to all of the computer's operating system facilities and use of powerful control sequences. All screen MODES can be used

- full access to the computer's VDU drivers, including the option of translating certain received ASCII sequences as control characters, to allow communication over most types of networks without the problem of transmitting control characters

- the possibility of switching between emulation modes, both from the remote system and locally, using the keyboard

When the terminal program is active, characters typed at the keyboard are transmitted to a remote computer through the computer's RS423 port. It is always assumed that the remote system will 'echo' the characters back to the user, ie the terminal operates in full duplex mode. The diagram below illustrates this:



The format of the data sent (word length, transmission speed and parity) must be set to that required by the remote system using operating system `*FX` or `*CONFIGURE` commands in order for the terminal to work correctly. If set using `*CONFIGURE`, the information is stored in the battery-backed RAM and so will only need to be specified once. Thereafter it will be 'remembered' each time the computer is turned on. The two commands are:

```
*CONFIGURE BAUD <Ø-8>
*CONFIGURE DATA <Ø-7>
```

The first one sets the default receive and transmit rates for the RS423 system. The interpretation of the number is:

0	9600	baud
1	75	baud
2	150	baud
3	300	baud
4	1200	baud
5	2400	baud
6	4800	baud
7	9600	baud
8	19200	baud

If you require different transmit and receive rates, you must use a *FX7 or *FX8 command to change one of them explicitly.

The parameter of the DATA configuration command determines the format of the transmitted and received data, ie the length, parity and number of stop bits:

length bits	**word length**	**parity**	**stop bits**
0	7	even	2
1	7	odd	2
2	7	even	1
3	7	odd	1
4	8	none	2
5	8	none	1
6	8	even	1
7	8	odd	1

Once configured the computer should be reset using [CTRL]+[BREAK] to make the new parameters come into effect.

One of the emulator's operating modes is as an ANSI (American National Standards Institution) terminal. This is a device which understands a set of standard escape sequences sent from the remote computer; for example to move the cursor around the screen, clear parts of the screen, provide functions such as autowrap at the ends of lines etc. If a program on the remote system knows that it is dealing with a terminal accepting ANSI control information, it can use these facilities to make functions such as screen editing quicker and easier.

It is possible to enter escape sequences on the keyboard ('locally'), ie without having to send them to the remote computer for echoing, by entering local mode (see below). As all of the escape sequences consist of printable ASCII characters, they are quite easy to type on the keyboard. In addition, operating system commands and TERMINAL control commands are available directly from the keyboard.

The cursor keys usually move the COPY cursor around, for cursor editing, so that the user can perform powerful cursor edit functions, even when the machine is acting as a terminal. You can also make the cursor keys send the codes required to move the text cursor to the remote system instead.

Function keys used with [SHIFT] or [CTRL] issue terminal commands (see below). By default, function keys used on their own, or in conjunction with [SHIFT]+[CTRL], behave as normal within the terminal emulator:

Function keys on their own generate the strings programmed using *KEY. [SHIFT]+[CTRL] function keys are ignored.

U.2 Using the TERMINAL emulator

The terminal is called up with the operating system command:

```
*TERMINAL
```

The minimum abbreviation is '*TE.' The terminal is a language and can be called from any other language such as BASIC or EDIT and will be copied into either a 65C102 Co-processor or a 6502 Second Processor if one is in use.

Upon entry, the screen clears and the start-up message is printed at the top of the screen. At this stage the user may communicate with the remote system (assuming the RS423 connections and software parameters have been set-up correctly), or issue commands directly to terminal or operating system using the function keys. The facilities available from the keyboard are described below.

On entry, the terminal emulator prints

```
TERMINAL
 =
```

indicating that it is in terminal command state (see below). To put it into communicating mode, press [RETURN] or [ESCAPE].

If a suitable connection has been made with a remote computer, typing at the keyboard will result in the characters being echoed onto the screen. It is assumed that the user knows the correct logging-on sequence for the system being used. If no response can be obtained from the remote system, check that the transmit and receive baud rates are correct, and that the data format has the correct parity and number of stop bits. If these are correct and it is still not possible to get a response from the remote machine, it is possible that there is a hardware fault and you should consult your system expert or dealer.

The default terminal mode is ANSI compatible, as this provides a large number of terminal facilities and access to the computer's operating system. The other modes are really restricted versions of ANSI mode.

The terminal does not use any RAM from OSHWM up, so BASIC programs should remain intact during a terminal session, and should be recoverable by typing `OLD` after returning to BASIC. It is a wise precaution, however, to save any program or data before calling TERMINAL from another language, as * commands called from TERMINAL which load files may corrupt the program area.

U.3 Using the function keys with TERMINAL

When using the terminal emulator, you can type commands by pressing one of the red function keys (marked [f0] to [f9]) at the same time as [CTRL] or [SHIFT]. To make it easier to use the function keys, a function key card is supplied for TERMINAL which gives the name of each command. These command names are used throughout this guide. Insert the function key card under the transparent function key strip (behind the function keys) when you are using TERMINAL.

For reference purposes throughout this section, the functions are:

	+[SHIFT]	+[CTRL]
f0	Clear screen	Clear screen
f1	Command mode	Command mode
f2	–	–
f3	Printer ON	Printer ON
f4	Printer OFF	Printer OFF
f5	Printer toggle	Printer toggle
f6	Spool toggle	Spool toggle
f7	–	–
f8	Local/line	Local line
f9	Short break	Long break

Note: these keys will not work correctly if you change the function key bases (with *FX225 - 228) after entering TERMINAL.

Apart from [f9], the [SHIFT] function keys produce the same effects as [CTRL] function keys, but you should use the [CTRL] function keys to maintain compatibility with other versions of TERMINAL. TERMINAL sets the base of the [CTRL] function keys to &A0, so if you are including terminal commands in an *EXEC file, use &A0 for [f0] (CLEAR SCREEN), &A1 for [f1] (COMMAND MODE) and so on.

All of these commands act locally, ie do not rely on the remote machine responding (or even existing) to work.

Three of the commands act as toggle switches: PRINTER TOGGLE, SPOOL TOGGLE and LOCAL/LINE. Pressing one of these keys once turns the facility on (indicated by a high-pitched bleep) and pressing it again turns the facility off (indicated by a lower-pitched bleep). However, the status of these toggles varies according to the default settings on power-up. By default:

SPOOL TOGGLE is ON
PRINTING TOGGLE is ON (but see below)
LOCAL/LINE toggle is in LINE mode.

CLEAR SCREEN - [CTRL]+[f0]

This command clears the screen and moves the cursor to the top lefthand corner of the screen. If a text window has been defined using [CTRL] + [f1] (`DCS VDU`), only this area of the screen will be cleared. It is not recommended that you define windows in ANSI terminal mode as TERMINAL assumes the screen to have fixed dimensions. If these are altered by the presence of a window, many of the emulator's facilities will not work correctly.

COMMAND MODE - [CTRL]+[f1]

This prompts with a '=' and waits for the user to type a command line, terminated by [RETURN]. To return to normal communication mode, press [ESCAPE] or type a blank line. If the first character typed is a "`*`" the string is treated as an operating system command and is sent to the operating system command line interpreter (OSCLI) for execution; if the first character is a "`!`" it is treated as an Application Program Command (see below), otherwise it is treated as one of TERMINAL's inbuilt commands (as a Device Control String), listed below. Typical uses are obtaining filing system information (eg `*CAT`), setting configuration options (`*FX`, `*CONFIGURE`), finding out the time (`*TIME`), altering TERMINAL's options such as turning word wrap mode on (`WWM ON`), or allowing operating system commands from the serial line (`PROT OFF`).

TERMINAL recognises the following commands:

`MODE` <number>			eg `MODE 7`
`VDU` <number> [`,`<number>] etc			eg `VDU 17,2` or `VDU19,0,2,0,0,0`
`TERMINAL`			select ANSI mode
`BBC`			select BBC transparent mode
`GS`			select BBC GSREAD mode
`TTY`			select "dumb terminal" mode
`AWM`	`ON/OFF`	(Auto-wrap mode)	eg `AWM ON`
`CKL`	`ON/OFF`	(Cursor keys local)	eg `CKL OFF`
`MCL`	`ON/OFF`	(Modem control)	eg `mcl on`
`PROT`	`ON/OFF`	(Protection)	eg `p. of.`
`RFC`	`ON/OFF`	(Receive flow control)	eg `rfc ON`
`TFC`	`ON/OFF`	(Transmit flow control)	eg `TFC off`
`WWM`	`ON/OFF`	(Word wrap mode)	eg `WWM on`

The first two commands are very similar to the BASIC commands of the same name. `MODE` changes the machine's display mode and `VDU` sends specified ASCII

codes (characters) to the VDU driver. The numbers are in decimal and in the case of VDU should be separated by commas. Note that the syntax used by VDU is different from that used by the BASIC VDU keyword: the TERMINAL VDU does not allow you to use semicolons for separators or | for a terminator.

The second set of commands is used to switch between the four emulation modes. 'TERMINAL' sets the default mode. 'BBC' mode provides full access to the BBC VDU drivers, so you can draw circles, for example, from the remote machine. 'GS' mode is similar to 'BBC' but enables you to encode control codes using only printable characters; this is a 'safer' alternative to use than 'BBC' mode on systems that take exception to control codes. TTY is the 'dumb terminal' mode. See chapter U6 for more details.

The third group of commands controls the various mode flags. These are described in detail below. As shown in the examples, TERMINAL's commands may be given in upper or lower case and abbreviated using a full stop.

CTRL+f2

This command is currently unused and is reserved for use in later versions of the emulator.

PRINTER ON - CTRL+f3

This command enables output to the printer by sending a control code (ASCII 2) to the BBC VDU driver. Subsequent screen output will be sent to the selected printer as well. The printer type is as selected by previous use of *CONFIGURE PRINT or *FX5.

PRINTER OFF - CTRL+f4

This command turns off output to the printer by sending a control code (ASCII 3) to the BBC VDU driver.

PRINTER TOGGLE - CTRL+f5

This command enables and disables the printer, once it has been turned on using PRINTER ON. It controls printing by changing the output stream (using *FX3). You should use it in preference to a series of PRINTER ON and PRINTER OFF commands if you are using an Econet printer server. This prevents the printer server from printing banner and end text every time you turn printing off and on.

PRINTER TOGGLE is 'on' by default; when you first select PRINTER ON, printing is turned on immediately. Printing is disabled the first time that you press PRINTING TOGGLE and re-enabled when you press PRINTING TOGGLE again. If you have disabled the printer by using PRINTER TOGGLE and subsequently turned the printer off using PRINTER OFF, remember to set PRINTER TOGGLE to 'on' again. If you have not done this, the next time that you wish to use the printer and

press PRINTER ON, nothing will happen - the PRINTER TOGGLE will be set to 'off' and the printer disabled.

SPOOL TOGGLE - CTRL + f6

This command controls whether output to the screen will also be sent to the *SPOOL file. It assumes the *SPOOL command to open the file has been issued already (using COMMAND MODE for example). The sequence of commands required to send a section of screen output to a file is:

SPOOL TOGGLE	(Turn spooling off as it is on by default)
COMMAND MODE*SPOOL myfile RETURN	(Open the *SPOOL file with a *SPOOL command)
SPOOL TOGGLE	(Turn *SPOOL file on at the desired point; displayed output is now sent to myfile)
SPOOL TOGGLE	(Turn spooling off again)
COMMAND MODE*SPOOL RETURN	(Close the *SPOOL file)

Note that *SPOOL and *EXEC files do not work with TERMINAL under the cassette filing system, because the RS423 system used by TERMINAL and the cassette filing system compete for system resources.

CTRL + f7

This command is currently unused and is reserved for use in later versions of the emulator.

LOCAL/LINE - CTRL + f8

By default, TERMINAL is in 'line' or 'remote' mode, meaning that all characters typed by the user on the keyboard are sent to the remote machine so that they can be echoed back and interpreted by the terminal software. Pressing LOCAL/LINE once will put the user into 'local' mode. In this mode, characters typed at the keyboard are sent straight to the terminal software and do not appear on the output line to the remote machine. This enables you to type escape sequences directly without fear of 'disturbing' the remote machine. Remember that many of the facilities available through the escape sequences can instead be used by typing commands after COMMAND MODE.

Another press of LOCAL/LINE will return the TERMINAL to line mode.

LONG BREAK - CTRL + f9

Pressing LONG BREAK causes TERMINAL to send a 'long break' of duration 3.5 seconds to the remote machine. This is required by some modems to sever a connection. Consult your system expert about whether this command is necessary.

SHORT BREAK - [SHIFT]+[f9]

This has a similar effect to LONG BREAK, but the break lasts for about 233ms. Again, you are advised to consult your system expert where and when you need to use SHORT BREAK.

U.4 The ANSI compatible mode

This 'central' mode is described first. ANSI terminal mode ignores most of the usual control characters that the BBC VDU drivers use (ie those with codes between 0 and 31, and 127). Cursor movement, clearing the screen, changing mode etc. are controlled by sending escape sequences to the terminal. An escape sequence is the ASCII character ESCAPE (code 27) followed by one or more printable characters. As mentioned above, escape sequences may be typed locally by using LOCAL/LINE. Alternatively, of course, TERMINAL may obtain escape sequences from the remote machine.

Below is a list of the control codes that are understood by the emulator, irrespective of the current terminal mode:

BEL	(ASCII 7)	Produces a 'bleep'
BS	(ASCII 8)	Moves the cursor back by one position (see CUB).
TAB	(ASCII 9)	Moves the cursor forward by one position (see CUF).
LF	(ASCII 10)	Moves the cursor down a line, scrolling if necessary.
CR	(ASCII 13)	Moves the cursor to the start of the current line.
ESC	(ASCII 27)	Introduces an escape sequence, (ignored in 'dumb' mode).

In LOCAL mode, pressing [ESCAPE] followed by the key or code indicated will produce the specified result. In LINE mode, ESCAPE refers to ASCII 27 as transmitted from the remote computer to the terminal.

U.5 ANSI Mode escape sequences

The escape sequences recognised by TERMINAL in ANSI mode may be divided into three categories: control strings, control sequences, and mode parameters.

Control strings

consist of text strings which are passed to various parts of the system. For example, the OSC control string is passed to the machine's `OSCLI` routine.

Control sequences

are like control strings but shorter. They are used to perform such tasks as positioning the cursor, clearing the screen etc.

Mode parameters

are 'flags' which may be set or reset. An example is CKL, which controls the action of the BBC microcomputer's cursor keys.

The table below summarises the escape sequences that the terminal understands in ANSI mode. Each sequence has a mnemonic, which is easier to refer to than the complete escape sequence.

Mnemonic	Type	Description
APC	S	Application program command
AWM	P	Auto wrap mode. Controls action of cursor at line ends
CKL	P	Cursor keys local. Controls action of cursor keys
CPR	C	Cursor position report
CUB	C	Cursor backward
CUD	C	Cursor down
CUF	C	Cursor forward
CUP	C	Cursor position
CUU	C	Cursor up
DA	C	Device attributes. Identify terminal type
DCS	S	Device control string. (See COMMAND MODE in chapter U.3)
DSR	C	Device status report
ECH	C	Erase character(s)
ED	C	Erase whole or part of display
EL	C	Erase whole or part of line
HVP	C	Horizontal and vertical position. See CUP
IND	C	Index. Cursor down with scrolling

MCL	P	Modem control lines. Used for flow control
NEL	C	Next line. As carriage return, line feed.
OSC	S	Operating system command: send a command to `OSCLI`
PM	S	Private message. As OSC but the command isn't printed
PROT	P	Protect against remote interference
RI	C	Reverse index. Cursor up with scrolling
RIS	C	Reset to initial state (as after `*TERMINAL`)
RM	C	Reset a mode parameter
RFC	P	Receive flow control. Controls transmission of XOFF/XON
SCI	C	Special code interpretation. Sets other terminal modes
SCS	C	Select character set
SM	C	Set a mode parameter
ST	S	String terminator. Used to end a control string
SU	C	Scroll up without changing cursor position
TFC	P	Transmit flow control. Controls significance of XOFF/XON
WWM	P	Word wrap mode. Controls whether words 'wrap' at line ends

Type S means 'control string'
Type C means 'control sequence'
Type P means 'mode parameter'

The main use of these sequences is in controlling the terminal from a program (eg a screen editor) running on the remote machine. Some of them are available using the [CTRL] function keys, as described in chapter U.3. The others may be obtained from the keyboard by entering local mode and typing the sequence ie LOCAL/LINE [ESCAPE] followed by the appropriate sequence of characters.

In the examples of escape sequences below, spaces may be put in to aid readability. These should not appear in the actual characters sent to the terminal, unless they form a part of a message. So ESCAPE % 5, for example, would actually be sent as ASCII 27, followed by a '%' character, followed by a '5' character.

The headings below have four parts: the mnemonic and the long form name, followed by the escape sequence in ASCII, and escape sequence in hexadecimal.

Control string sequences

APC (Application Program Command)

ESCAPE _ (&1B &5F)

This sequence marks the start of a command string to be passed to the terminal program. The string, like all control strings, is terminated by the string terminator, ST (ESCAPE \). The string is displayed on the screen. If this is not desired, the VDU drivers should be disabled first using `DCS VDU 21`. They can be re-enabled using `DCS VDU 6`.

Usually the terminal will respond to an APC string by bleeping. However, it is possible for the user of TERMINAL to set-up a machine code routine which can interpret his own APC commands. Note that APCs may be entered from the keyboard in response to a = prompt (eg after COMMAND MODE) by starting the command with !.

To access the string, a program must be executed which makes the indirection vector `IND2V` (at locations &232-&233) in the language processor point to the user's code. The language processor is the BBC microcomputer if no 65C102 Co-processor or 6502 Second Processor is active, otherwise it is the Co-processor or Second Processor. This code will then be called by TERMINAL whenever an APC sequence is received. The string (without the APC and ST codes) is put at location &500. It is terminated by a carriage return (supplied by TERMINAL) and may be up to 254 characters long.

The user's routine is called with interrupts enabled but, while it is executing, no other input processing (ie keyboard and RS423) is carried out. This means that the routine should return as quickly as possible (eg in under 5mS). Locations &80-&8F are available to the routine as workspace. The program itself may be located anywhere between `OSHWM` and `HIMEM` when using the current version of TERMINAL, though this may become more restricted in later versions. You are recommended to put your routines as high up in memory as possible. The simplest way of returning from the routine is with an '`RTS`' instruction. An alternative method is decribed after the example program.

As an example of how to intercept `IND2V`, the program below allows the remote machine to set the computer's real-time clock. To do this it sends the string:

`APC !TIME` <time string> `ST`

where the format of the <time string> is:

`Tue, 1 Jan 1972`	(set date only)
`21:12:00`	(set time only)
`Tue, 1 Jan 1972.21:12:00`	(set time and date)

The program is listed below:

```
1000 REM  Source program to interpret APC string
1010 DIM code 100
1020 org=&2F00
1030
1040 osword=&FFF1
1050 com$=EMIT!" : REM !TIME" backwards
1060 strPtr=&80  : REM Pointer into APC string
1070 start=&81   : REM Pointer to start of TIME$ string
1080 ind2v=&232  : REM Address of OS vector used by APC
1090 apcBuff=&500: REM Address of APC string buffer
1100 setTime=15  : REM OSWORD number for TIME$=
1110 cr=13       : REM End of line marker
1120
1130 FOR pass=4 TO 6 STEP 2
1140 P%=org : O%=code
1150 [ opt pass
1160 .setup
1170   lda #apc MOD &100/Point IND2V at our code
1180   sta ind2v
1190   lda #apc DIV &100
1200   sta ind2v+1
1210   rts
1220
1230 .apc
1240   clr strPtr/Point to first char of APC string
1250   jsr skipSpaces/Skip spaces between APC and first char
1260   ldx #LENcom$/Check for the command string
1270 .comLoop
1280   cmp command-1,x/Does it match the command?
1290   bne ret/No, return with no action
1300   jsr getChar/Get the next APC character
1310   dex/Next command char
1320   bne comLoop
1330   jsr skipSpaces/Matched the command, so skip to the argument
1340   sty start/Save pos. of first byte of arg.
1350 .findEnd
1360   jsr getChar/Find the carriage return
1370   cmp #cr
1380   bne findEnd
1390   tya/Get length of command line
1400   sbc start/(Carry set from the cmp)
1410   ldx start/Store it just before the string
1420   dex/and set X for the OSWORD
1430   sta apcBuff,x
```

```
1440   ldy #apcBuff DIV &100 /XY points to the time string
1450   lda #setTime/Do the OSWORD
1460   jsr osword
1470 .ret
1480   rts
1490
1500 .getChar
1510   ldy strPtr/Get a character from the APC buffer
1520   inc strPtr/and point to the next one
1530   lda apcBuff,y
1540   cmp #ASCa"/See if it's in range a-z
1550   bcc notLower
1560   cmp #ASCz"+1
1570   bcs notLower
1580   and #&DF/Convert to upper case
1590 .notLower
1600   rts
1610
1620 .skipSpaces
1630   jsr getChar/Get characters until not space
1640   cmp #ASC "
1650   beq skipSpaces
1660   rts
1670
1680 .command
1690   EQUS com$/String containing APC command
1700
1710 ]
1720 NEXT
1730 OSCLISAVE APCOBJ +STR$code+" +STR$0%+" +STR$org+" +STR$org
```

When executed the program saves the object code under the name 'APCOBJ'. To test it, enter TERMINAL as usual, then type

COMMAND MODE*RUN APCOBJ [RETURN]

This causes IND2V to be set-up so that it uses the program listed from the label 'apc' above. Then to set the time, enter local mode, then press [ESCAPE] _ to start the APC sequence. Follow this by a command sequence of the form mentioned above, eg '!TIME 10:32:00' and then type [ESCAPE] \ to terminate the APC string. The real-time clock will be set to the time given.

Of course, the APC sequence would usually be been generated from the remote computer - in order to synchronise the machines' clocks, for example.

The user's APC program may use an alternative method to return to TERMINAL. This is to jump to the address held in `IND2V` before it was overwritten by the user routine's address. If this method is used, the carry flag should be cleared ('`clc`') if the user routine was successful, and set ('`sec`') otherwise. In the former case, TERMINAL will carry on as usual. If an error is implied by a set carry flag, TERMINAL will bleep and the next request for an error status (DSR) will indicate that an error has occured.

DCS (Device Control String)

ESC `P` (&1B &50)

This sequence marks the start of a string which will perform some change in the state of the terminal. The DCS sequence is followed by a command string which is terminated by the ST sequence. For a description of the possible commands see chapter U.3. For a description of the mode flags see 'Mode parameters', below.

OSC (Operating System Command)

ESC `]` (&1B &5D)

This should be followed by an operating system command string. As usual, ST terminates the command. The carriage return that the MOS requires at the end of the string is supplied automatically by TERMINAL. Control characters are ignored and will not be passed to the operating system or acted upon.

The string sent to the OS is displayed on the current line. An example is:

```
OSC *CAT ST
```

PM (Privacy Message)

ESC `^` (&1B &5E)

This marks the start of a string which is treated in exactly the same way as an `OSC` message, with the exception that it is not echoed on the screen, or the `*SPOOL` file or printer if these are enabled. The main use of this sequence is to execute * commands (for example programming the function keys) from the remote machine without disturbing the screen.

ST (String Terminator)

ESC `\` (&1B &5C)

This is not a control string, but marks the end of APC, DCS, OSC and PM strings.

Control Sequences

Many of the control sequences use the common sequence **ESC** `[` at the start. This has the name *CSI* (control sequence introducer), which is used below to

stand for ESC [. In addition, <n> stands for an optional number given as a decimal string (eg "23": &32 &31). This usually defaults to 1 if omitted.

Cursor position sequences

It is possible to move the cursor in each of the four main directions. It is also possible to position the cursor on a given line and column. Note that when the cursor is moved by CUB, CUF, CUU and CUD, it is not possible to make the screen scroll. Thus an attempt to do a CUU on the top line, a CUD on the bottom line, CUB on the very first screen position, or a CUF on the very last screen position, will be ignored. When the CKL flag is reset, the arrow keys can be made to produce the ESCAPE sequences expected by your remote system for cursor movement (by programming function keys 11 to 15 with *KEY).

CUU (Cursor Up)

CSI <n> A (&1B &5B <n> &41)
This moves the cursor up by <n> lines, without altering its horizontal position. If the cursor is already on the top line, it will not move. The cursor movement sequences CUU, CUD, CUF and CUB will never cause the screen to scroll - it is always in a 'fixed' state.

CUD (Cursor Down)

CSI <n> B (&1B &5B <n> &42)
This moves the cursor down by <n> lines, without altering the horizontal position.

CUF (Cursor Forward)

CSI <n> C (&1B &5B <n> &43)
This sequence moves the cursor forward by <n> character spaces. The action of CUF when the cursor is at the last position on the line depends on the state of the auto wrap mode flag. If this is set, a CUF when the cursor is in column 80 (or 40 in MODE 7) will cause the cursor to move to the start of the next line. If AWM is off (reset), a CUF will not change the cursor position at column 80.

A CUF when the cursor is at the bottom right-hand corner will cause no movement of the screen or cursor, regardless of the state of AWM.

CUB (Cursor Back)

CSI <n> D (&1B &5B <n> &44)
This behaves in the opposite way to CUF. The cursor is moved back by <n> character positions, without overwriting the character already there. If AWM is on, a CUB when the cursor is in column 1 causes the cursor to move to the end of the previous line. If AWM is off, a CUB in column 1 has no effect.

CUP (Cursor Position)

CSI <n> ; <n> H (&1B &5B <n> &3B <n> &48)
This sequence may be used to position the cursor at any column and line. The first <n> is the line number (between 1 and 25) and the second is the column number. As both of these default to 1, a 'home cursor' may be obtained simply by *CSI* H. To move to the 'centre' of the screen in MODE 3 use:

ESC [12 ; 40 H

HVP (Horizontal/Vertical Position)

CSI <n> ; <n> f (&1B &5B <n> ; <n> &66)
This sequence acts in the same way as CUP and the parameters have the same meanings. CUP is the preferred method of positioning the cursor.

IND (Index)

ESC D (&1B &44)
This moves the cursor down by one line (as CUD) and scrolls the screen up if the cursor is on the bottom line (unlike CUD). The state of the cursor (pending or otherwise) is preserved.

NEL (Next Line)

ESC E (&1B &45)
This moves the cursor to the start of the next line down, performing an action similar to the sequence carriage return, line feed.

RI (Reverse Index)

ESC M (&1B &4D)
This moves the cursor up by one line (as CUU) and scrolls the screen down if the cursor is on the top line (unlike CUU). The state of the cursor (pending or otherwise) is preserved.

SU (Scroll Up)

CSI <n> S (&1B &5B <n> &53)
This scrolls the screen up by <n> lines without altering the position of the cursor on the screen.

DSR (Device Status Report)

CSI <parms> n (&1B &5B <parms> &6E)
When received, this sequence causes the terminal program to report its status. The <parms> understood by the program are '5' and '6'.

A received parameter '5' means 'error status?'. The terminal will respond to it by sending a DSR with a parameter '0' to the remote station if no errors have

occured, or a parameter '3' (meaning 're-try') if an error has occurred since the last DSR with a '0' parameter was sent. Once an error condition has been sent by DSR, it will be cleared.

A received parameter '6' means 'cursor position?'. The terminal will reply by sending a CPR sequence as described below.

CPR (Cursor Position Report)

CSI <n> ; <n> R (&1B &5B <n> ; <n> &52)
This sequence is sent by the terminal in response to a DSR 6 (see above). The two numbers are the row and column of the cursor respectively. The row number is in the range 1 to 25 and the column is in the range 1 to 81. A column number of 81 (or 41 in a 40 column display mode) implies that the cursor is in pending state.

DA (Device Attributes)

CSI <n> c (&1B &5B <n> &63)
This sequence causes the terminal to identify its type. If the parameter is 0 or absent, TERMINAL will respond with the sequence

```
CSI > 5 c
```

That is, a DA with parameter 0 of type '>'. The '>' means 'Acorn' and the 5 is the version number. This may be different in other versions of TERMINAL.

ECH (Erase Character)

CSI <n> X (&1B &5B <n> &58)
This overwrites <n> characters starting from the cursor without changing the cursor position.

ED (Erase part of Display)

CSI <n> J (&1B &5B <n> &4A)
This sequence has three possible effects, depending on the value of <n>:

<n>='0' (or is absent) Erase to the end of the display from the cursor
<n>='1' Erase from the start of the display to the cursor
<n>='2' Erase the whole display

In all cases, the cursor position is unchanged

EL (Erase part of Line)

CSI <n> K (ASC &5B <n> &4B)
This sequence has three possible effects, depending on the value of <n>:

<n>='0' (or is absent) Erase to the end of the line from the cursor
<n>='1' Erase from the start of the line to the cursor
<n>='2' Erase the whole line

In all cases, the cursor position is unchanged

RM (Reset a Mode flag)

CSI <parms> l (&1B &5B <parms> &6C)

SM (Set a Mode flag)

CSI <parms> h (&1B &5B <parms> &68).eb
These two sequences have identical formats, but opposite effects. The <parms> part is a sequence of characters indicating the parameter to be reset (or set). These are described in the next section. An example is 'reset receive flow control':

ESC [> 3 l

RIS (Reset to Initial State)

ESC c (&1B &63)
This sequence resets all of the mode parameters to their initial states, viz:

AWM	is on	Characters wrap around at line ends
CKL	is on	Cursor keys act as 'cursor edit' keys
MCL	is off	Modem control lines are inactive
PROT	is on	Remote OSCs and PMs are not executed
RFC	is on	XOFF is sent when the receive buffer becomes full
TFC	is on	TERMINAL responds to XOFF/XON when sending data
WWM	is off	Words do not wrap automatically at line ends

Other characteritics of the terminal emulator, eg the display mode etc. are unaltered by RIS.

SCI (Special Code Interpretation)

ESC % <n> (&1B &25 <n>)
This sequence controls the mode in which the terminal operates, ie ANSI, BBC transparent etc. Some of these modes may be called up from the keyboard using COMMAND MODE (see chapter U.3) The value <n> has the following interpretations:

ESC % 0This is the default ANSI mode, treating incoming characters as 7-bit bytes and obeying all of the ESCAPE code sequences.

ESC % 1This mode also uses 7-bit characters, but ignores all ESCAPE sequences and unknown control codes. Because of the lack of ESCAPE

facilities, it is impossible to exit from this mode 'remotely'. This is an extremely 'dumb' mode.

ESC % 2This is the same as ANSI mode (**ESC** % Ø), except that all eight bits of the character codes are used, and so it is possible to receive codes greater than 128 to display characters from BBC's extended character set. Note however that the *CONFIGURE DATA setting may only permit 7 bit codes to be received.

ESC % 3This is the same as **ESC** % 1 mode, but 8-bit characters are used.

ESC % 4See BBC VDU driver mode below.

ESC % 5See BBC VDU driver mode below.

ESC % 6Incoming characters are interpreted as pairs of hexadecimal digits which are converted in pairs into bytes that are sent directly to the BBC VDU drivers. If the hexadecimal pair &1B is received, and no multi-byte VDU command is active (eg PLOT or VDU 23) then this is taken to mark the start of an ESCAPE sequence and subsequently decoded bytes are sent to the ESCAPE interpreter.

ESC % 7This mode causes control characters to be ignored until the next ESCAPE sequence or printable character is received. Flow control using XON/XOFF continues if enabled. After the ESCAPE or printable character is received, the terminal mode reverts to that in use before the ESC % 7.

SCS (Select Character Set)

ESC (<n> (ESC (<n>)
In the soft display modes (all but MODE 7), this sequence causes a particular character set to be activated. Possible values for <n> are:

'A' The UK ASCII character set is selected. This differs from the usual BBC character set, in that ASCII 35 is printed as the pound-sign, and the ASCII 96 is displayed as grave accent.

'B' The US ASCII character set is selected. This is similar to the UK ASCII character set, but ASCII 35 is printed as hash. This character set is selected by default.

'6' The characters with ASCII codes &40-&7F are replaced by those with codes &80-&BF. On the BBC microcomputer these default to accented letters, the Greek alphabet, line drawing characters, and some mathematical symbols. They may be redefined using VDU 23.

'7' The characters with ASCII codes &40-&7F are replaced by those with codes &C0-&FF. On the BBC microcomputer, these default to upper and lower case Greek letters and mathematical symbols. They may be redefined using VDU 23.

Unknown control sequences

If the terminal receives a control sequence introduced by *CSI* which it does not recognise, it passes control to a user's program which can try to interpret the sequence instead. The user's program must lie at the address held in `IND1V` (at addresses &230-&231) in the language processor.

The format of a *CSI* sequence is:

CSI <parms> <intermediate chars> <final char>

<parms> is zero or more parameters separated by semicolons '`;`'. A parameter consists of a decimal number in ASCII optionally preceded by a type character, which may be '<', '=', '>' or '?'. Currently '>' is used by Acorn and '?' denotes a DEC parameter.

<intermediate chars> is a sequence of zero or more characters in the range ASCII &20 to ASCII &2F.

<final char> is any single character in the range ASCII &40 to ASCII &7F. This character determines the meaning of the whole sequence.

TERMINAL calls the user routine with the final character in the accumulator and a block of up to eight parameters starting from location &0000. Location &0000 holds a count of the number of parameters (0-8) and subsequent locations hold the parameters in the format: type byte, two-byte parameter value (low byte first). The intermediate bytes are discarded by the terminal. If there are more than 8 parameters, &0000 is set to 9, but only the first eight are stored.

The example below uses the unknown *CSI* facility to provide access to the BBC's PLOT command. The control sequence

CSI `>p1 ; p2 ; p3 P`

will perform the same function as `PLOT p1,p2,p3` in BASIC, ie `p1` is the plot code, `p2` is the X co-ordinate and `p3` is the Y co-ordinate. For example to draw a line to co-ordinates (1000,1000) this sequence might be used:

CSI `>5;1000;1000P`

The first parameter must be preceded by the Acorn type character '>' for the command to be recognised. The page zero parameter block for this sequence would look like this:

&0009	18	High byte of third parameter
&0008	2	Low byte of third parameter
&0007	Don't care	Type of three (Y co-ord) parameter

&0006	3	High byte of second parameter
&0005	232	Low byte of second parameter
&0004	Don't care	Type of second (X co-ord) parameter
&0003	0	High byte of first parameter
&0002	5	Low byte of first parameter
&0001	'>'	Type of first (plot mode) parameter
&0000	3	Number of parameters

```
1000 REM  Source program to interpret CSI sequence
1010 DIM code 70
1020 org=&2F00
1030
1040 oswrch=&FFEE  :REM  Operating system write character
1050 ind1v=&230    :REM  Holds the address of the user routine
1060 acorn$=>"     :REM  The Acorn type byte
1070 plot$=P"      :REM  The final character for CSI plot
1080 plot=25       :REM  The VDU code for PLOT
1090 parBlk=0      :REM  The address of the CSI parm block
1100 :
1110 FOR pass=4 TO 6 STEP 2
1120 P%=org : O%=code
1130 [ opt pass
1140 .setup
1150   lda #csi MOD &100/Point ind1v at our code
1160   sta ind1v
1170   lda #csi DIV &100
1180   sta ind1v+1
1190   rts
1200\
1210 .csi
1220   cmp #ASCplot$/The right final character?
1230   bne ret/No action
1240   lda parBlk/Correct number of parameters?
1250   cmp #3
1260   bne ret
1270   lda parBlk+1/First type byte '>'?
1280   cmp #ASCacorn$
1290   bne ret
1300   lda #plot/PLOT = VDU 25
1310   jsr oswrch
1320   ldx #2/Send the plot code and skip high byte
```

```
1330   jsr send1
1340   inx/Skip next type byte
1350   jsr send2/Send X co-ordinate and skip next type
1360 .send2
1370   lda parBlk,X/Send the Y co-ordinate
1380   inx
1390   jsr oswrch/Send it
1400 .send1
1410   lda parBlk,X/Get high byte
1420   inx
1430   jsr oswrch
1440   inx/Always skip a byte
1450 .ret
1460   rts
1470 ]
1480 NEXT
1490 OSCLISAVE CSIOBJ +STR$code+" +STR$O%+" +STR$setup+" +STR$org
```

Once TERMINAL has been called, the routine is initialised by

COMMAND MODE*RUN csiobj [RETURN]

After this, all CSI sequences of the type described above will be sent to the user routine. Although the example given is quite useful, it is mainly illustrative as the same operation can be performed in other ways (eg by sending bytes directly to the VDU drivers). However, the user *CSI* facility should prove useful in other situations. For example, external hardware (graph plotters, turtles etc.) could be controlled using special *CSI* sequences.

Note that example above uses '`rts`' to return to TERMINAL. The alternative method described in the section on unknown APCs may also be used.

Mode parameters

These are set with the SM control sequence and reset with the RM control sequence. RM and SM are described above. For the default values of the mode flags, see RIS above.

AWM (Auto Wrap Mode)

CSI ? 7 h/l (&1B &5B &3F &37 &68/&6C)

Auto wrap mode controls what happens when the cursor reaches the end of the line. When it is on (the default), printing a character in column 80 will put the cursor into pending state, which is indicated by the cursor remaining in column 80, under the character just printed. The next character will be printed on the first column on the next line down, and the screen will scroll up if necessary. The cursor will no longer be in the pending state.

When AWM is off, the pending state is again entered by printing a character in column 80. However, subsequent characters will not be printed and the cursor stays at column 80, with the effect that long lines will appear to be truncated on the screen. This continues until pending state is released. In pending state, the cursor is said to be in column 81.

The effect of AWM on CUB and CUF is described in section 3.1.2.

CKL (Cursor Keys Local)

CSI > 0 h/l (&1B &5B &3E &30 &68/&6C)
This flag controls the action of the BBC microcomputer's cursor keys. When it is set (the default state), the arrow keys and [COPY] produce the normal editing effect, ie characters may be copied from other lines. Cursor editing is terminated when [RETURN] is pressed.

If CKL is reset, the arrow keys behave as function keys 11 to 15 ([COPY], cursor left, right, down and up, respectively). To make the cursor keys produce the codes for cursor movement, use the following definitions:

```
*KEY12|[[D [RETURN]
*KEY13|[[C [RETURN]
*KEY14|[[B [RETURN]
*KEY15|[[A [RETURN]
```

The arrow keys will then produce the control sequences ESC [A etc. In addition, [COPY] can be programmed to produce a 'home cursor' sequence:

```
*KEY11|[[H [RETURN]
```

MCL (Modem Control Lines)

CSI > 1 h/l (&1B &5B &3E &31 &68/&6C)
This flag, which defaults to off, controls whether the terminal will use one of the signals on the RS-423 interface to stop the remote station from sending data. If MCL is set, the CTS line on the interface will be taken low when the RS-423 input buffer becomes full.

PROT (Protect Against Interference)

CSI > 2 h/l (&1B &5B &3E &32 &68/&6C)
When this flag is set (the default state), the terminal will not act on OSC or PM control strings from the remote machine. This stops the computer being taken over by, say, another user connected to the remote computer. When an OSC or PM sequence is sent to a protected terminal it bleeps.

RFC (Receive Flow Control)

CSI > 3 h/l (&1B &5B &3E &33 &68/&6C)

If this mode is set (the default), the terminal will send XON and XOFF control codes to the remote machine to control the flow of incoming bytes. When the receive buffer becomes full, XOFF is sent to stop the incoming data. When the receive buffer begins to empty, XON will be sent to re-enable transmission from the remote machine. It is necessary to use RFC when TERMINAL is operated at high baud rates.

TFC (Transmit Flow Control)

CSI > 4 h/l (&1B &5B &3E &34 &68/&6C)

This is the counterpart of RFC; it enables a remote machine to tell the terminal to stop sending, by transmitting an XOFF control code. TFC is set by default. The terminal will start to send data again when an XON is received. The terminal may send XOFF/XON itself, even if currently XOFFed by the remote machine. It is necessary to use RFC when TERMINAL is operated at high baud rates.

WWM (Word Wrap Mode)

CSI > 5 h/l (&1B &5B &3E &35 &68/&6C)

When WWM and AWM are both set, an attempt to print a word across two lines will result in the whole of the word being printed on the lower line. This makes text which is formatted for, say, 80 columns much more readable when displayed in a 40 column mode (say MODE 7). WWM is off by default.

U.6 The BBC VDU driver modes

These terminal modes are entered using SCI sequences or DCS strings. The BBC modes are similar to each other, but transparent mode passes every character received to the VDU drivers, and GSREAD mode applies decoding of received characters. In transparent mode ANSI ESCAPE sequences can be accessed normally by sending ASCII 27 to the terminal. GSREAD mode requires ESCAPE to be sent encoded as ¦[(see 'BBC GSREAD mode', below).

BBC Transparent mode

In this mode, all eight (or seven; see `*CONFIGURE` in chapter U1) bits of received characters are available, and characters are sent directly to the BBC VDU drivers, although ESCAPE sequences are acted upon. This means that BBC graphics etc may be accessed from the remote machine simply by sending the appropriate codes.

Note that although the sequences to enable/disable word-wrap mode are recognised, they will not come into effect until ANSI mode is selected, as the BBC VDU modes do not perform word-wrapping.

In BBC transparent mode with flow control enabled, the characters for `XON` and `XOFF` (ASCII 17 and ASCII 19 respectively) are intercepted before they reach the VDU drivers. This means that setting the text colours (ASCII 17) and logical colours (ASCII 19) is not possible using simple control codes when flow control is enabled. Some alternative (eg `DCS VDU`) must be used).

BBC GSREAD mode

This is substantially the same as transparent mode, but incoming characters are decoded assuming GSREAD format before being displayed to the screen. GSREAD format is described below. This mode provides a very 'safe' method of using the BBC microcomputer's VDU drivers from the remote machine. For example, to change logical colours using ASCII 19, a pure text sequence such as '¦S¦A¦C¦@¦@¦@' might be used, instead of control codes which might be intercepted between the remote machine and the BBC microcomputer (for example by the terminal driver software in the remote computer).

A danger using GSREAD mode is if the text originating from the remote computer contains vertical bars (¦) which will cause the next character to be converted to a control character before being passed to the VDU drivers. This may have undesirable effects (eg ¦L will clear the screen).

When a vertical bar is received in GSREAD mode followed by a sequence of control characters, all characters are disregarded until the next printable

character (including the initial ¦). Thus simply by ensuring that ¦ is the last character printed on a line, a stream of GSREAD encoded characters may be sent without being disturbed by the 'new line' sequence of the remote machine.

Another minor difference between GSREAD and BBC modes is that receiving DEL in the former has no effect, while in the latter it deletes the preceding character on the screen.

GSREAD encoding

This is a method by which control characters and characters greater than 127 are rendered printable. It will be familiar to those who have used the BBC microcomputer's softkey facility. To embed a control character in the string, say **[CTRL]** A, the character should be preceded by a vertical bar: ¦A. All characters may be obtained in this way:

Code	GSREAD encoding
0	¦@
1	¦A
2	¦B
....	
....	
25	¦Y
26	¦Z
27	¦[
28	¦\
29	¦]
30	¦^
31	¦_
32 to 126 (The printable characters always stand for themselves)	
127	¦?

To obtain the vertical bar symbol, use ¦¦.

To add 128 to the code of a character, precede it with ¦!. For example, code 128 is ¦!¦@, code 156 is ¦!¦Z, code 173 is ¦!A and code 255 is ¦!¦?. Using this method, any ASCII code from 0 to 255 can be encoded and decoded without needing to transmit control characters.

U.7 Summary of ESC sequences in ASCII order

ESC % Ø	SCI	7 bit codes, ANSI mode
ESC % 1		7 bit codes, dumb mode
ESC % 2		8 bit codes, ANSI mode
ESC % 3		8 bit codes, dumb mode
ESC % 4		8 bit transparent BBC codes
ESC % 5		8 bit GSREAD BBC codes
ESC % 6		Hexadecimal digits
ESC % 7		Ignore ctrls until ESC or ASCII 32-126
ESC (Ø	SCS	CHR$&40-CHR$&7F=CHR$&80-CHR$&BF
ESC (1		CHR$&40-CHR$&7F=CHR$&C0-CHR$&FF
ESC (A		UK character set
ESC (B		US character set
ESC D	IND	Index (Cursor down)
ESC E	NEL	Next line (like CR LF)
ESC M	RI	Reverse index (cursor up)
ESC P <string> **ESC **	DCS	Device control string
ESC c	RIS	Reset to initial state
ESC [	CSI	Control sequence introducer
ESC [<n> A	CUP	Cursor up
ESC [<n> B	CUD	Cursor down
ESC [<n> C	CUF	Cursor forward
ESC [<n> D	CUB	Cursor backward
ESC [<row> ; <column> H	CUP	Cursor position
ESC [Ø J (**ESC [J**)	ED	Erase to end of screen
ESC [1 J		Erase to start of screen
ESC [2 J		Erase whole display
ESC [Ø K (**ESC [K**)	EL	Erase to end of line
ESC [1 K		Erase to start of line
ESC [2 K		Erase whole line
ESC [<row> ; <column> **R**	CPR	Cursor position report
ESC [<n> **S**	SU	Scroll up
ESC [<n> **X**	ECH	Erase character
ESC [<parm> **c**	DA	Device attributes
ESC [<row> ; <column> **f**	HVP	Horizontal and vertical position
ESC [<parm> **n**	DSR	Device status report
ESC [<options> **h**	SM	Set mode
ESC [<options> **l**	RM	Reset mode
ESC [> Ø h	CKL	Cursor keys local (set)
ESC [> Ø l		Cursor keys local (reset)
ESC [> 1 h	MCL	Modem control lines (set)
ESC [> 1 l		Modem control lines (reset)
ESC [> 2 h	PROT	Protect (set)
ESC [> 2 l		Protect (reset)
ESC [> 3 h	RFC	Receive flow control (set)
ESC [> 3 l		Receive flow control (reset)

ESC [> 4 h	TCF	Transmit flow control (set)
ESC [> 4 l		Transmit flow control (reset)
ESC [> 5 h	WWM	Word wrap mode (set)
ESC [> 5 l		Word wrap mode (reset)
ESC [? 7 h	AWM	Autowrap mode (set)
ESC [? 7 l		Autowrap mode (reset)
ESC \	ST	String terminator
ESC] <string> **ESC** \	OSC	Operating system command
ESC ^ <string> **ESC** \	PM	Private message
ESC _ <string> **ESC** \	APC	Application Program Command

Index

ABS L.2-1
absolute
– address P.3-2
– addressing P.3-3
– indexed addressing P.3-4
– indirect addressing P.3-3
accumulator P.4-4
– addressing P.3-2
acoustic coupler U.1-1
ACS L.2-1
actual parameter L.2-26
ADC P.4-1
address
– evaluation O.7-3
– field P.3-1,Q.3-3
addressing modes P.3-1,P.4-1
ADVAL L.2-2,N.5-1
American National Standards Institution U.1-3
amplitude L.2-20,L.2-62
– envelope L.2-21
AND L.2-3,P.4-2
ANSI U.1-3
ANSI compatible mode U.4-1
APC U.5-3
application program command U.5-3
arithmetic shift P.4-2
arrays K.2-2,L.2-15
– with CALL L.2-8
ASC L.2-4
ASL P.4-2
ASN L.2-4
assembler O.1-1,O.5-1
– errors Q.1-1
– extensions K.1-2
– keywords P.1-1,P.4-1
– macros O.7-5
– options P.4-16
– switches O.7-1
assembler syntax O.7-2
assembly
– errors Q.2-2
– language O.1-1,O.7-1
ATN L.2-4
attack phase L.2-20
AUTO L.2-5
auto wrap mode U.5-14
automatic editing R.16-1
AWM U.5-14

background colour L.2-11
bad program K.3-2
BASIC K.1-6,N.5-1
– commands K.1-6
– error messages M.1-1
– expressions K.2-7
– IV K.1-1
– keywords L.1-1,L.2-1,N.2-1,N.3-1
– line L.1-1
– stack L.2-30,N.4-2
– syntax L.1-1,L.2-1
– syntax notation L.1-2
– technical information N.1-1
– tokens N.3-1,N.2-1
BASIC_ZP_START N.4-3
baud rate U.1-3
BBC
– BASIC K.1-1,N.1-1
– BASIC assembler O.1-1,O.2-1
– GSREAD mode U.6-1
– transparent mode U.6-1
BCC P.4-3
BCS P.4-3
BEQ P.4-3
BGET# K.3-7,L.2-5,N.5-1
BIT P.4-4
BMI P.4-4
BNE P.4-5

bold text S.10-1
BPL P.4-5
BPUT# K.3-7,L.2-6,N.5-1
BRA P.4-5
bracket K.2-8
BRK P.4-6
buffer K.3-6
BVC P.4-6
BVS P.4-6

CALL K.1-5,L.2-6,O.6-3,O.9-1
– parameter block L.2-7
carry P.4-3
CHAIN K.3-2,L.2-9,N.5-1
channel number L.2-61
character specifier R.11-2,R.15-2, R.15-5
CHR$ L.2-9
CLC P.4-7
CLD P.4-7
CLEAR L.2-10
clearing text R.10-1
CLG L.2-10,N.5-1
CLI P.4-8
CLOSE N.5-1
CLOSE# K.3-8,L.2-10
CLR P.4-8
CLS L.2-11,N.5-2
CLV P.4-8
CMP P.4-8
coding errors Q.3-1
COLOR K.1-2,L.2-11
COLOUR K.1-2,L.2-11,N.5-2
combining files R.13-2
command
– files R.16-1
– parameters S.5-1
– positioning S.4-2
concatenation L.2-65
conditional assembly O.7-4
control
– codes R.3-2
– sequence U.5-1,U.5-6,U.5-12
– sequence introducer U.5-6
control string U.5-1,U.5-3
copying text R.9-1
COS L.2-12
COUNT L.2-13
CPR U.5-9
CPX P.4-9
CPY P.4-9
CRL U.5-15
CSI U.5-6,U.5-12
CUB U.5-7
CUD U.5-7
CUF U.5-7
CUP U.5-8
cursor R.6-1
cursor
– back U.5-7
– down U.5-7
– editing R.9-1
– forward U.5-7
– keys local U.5-15
– position U.5-8
– position report U.5-9
– position sequence U.5-7
– up U.5-7
CUU U.5-7

DA U.5-9
DATA L.2-13,L.2-55
– pointer L.2-57
DCS U.5-6
DEA P.4-10
debugging Q.3-2
DEC P.4-10
decay phase L.2-20
DEF L.2-14
DEF PROC L.2-53
DEG L.2-14
DEI P.4-24
DELETE L.2-15
deleting text R.9-1
descriptive mode R.3-1
device
– attributes U.5-9
– control string U.5-6

device status report U.5-8
DEX P.4-10
DEY P.4-11
DIM K.2-6,L.2-15,O.6-1
direct access file K.3-11
display mode L.2-41,R.3-1
DIV L.2-16
DRAW L.2-16,N.5-2
DSR U.5-8
dynamic variable K.2-3

ECF pattern L.2-28
ECH U.5-9
ED U.5-9
EDIT K.1-2,L.2-17,R.1-1,R.2-1
 – buffer R.1-2
 – commands R.4-1
editing R.1-2
editor error T.2-1
EL U.5-9
electronic mail U.1-1
ELSE L.2-18,L.2-31,L.2-45
END L.2-18
ENDPROC L.2-18,L.2-53
ENVELOPE L.2-19,N.5-2
EOF K.3-7,L.2-22,N.5-2
EOR L.2-22,P.4-11
EQUB P.4-12
EQUD P.4-12
EQUS P.4-12
erase
 – character U.5-9
 – part of display U.5-9
 – part of line U.5-9
ERL L.2-23,M.1-1,Q.2-1
ERR L.2-23,M.1-1,N.5-2,Q.2-1
ERROR L.2-24,M.1-1
error
 – handling O.1-3,O.7-1,Q.1-1
 – line L.2-2
 – message T.1-1
 – number L.2-23
 – trapping K.1-5,L.2-44,M.1-1
ESC sequence U.5-1,U.7-1
EVAL L.2-24
exclusive OR L.2-22,L.2-23,P.4-11
EXP L.2-25
exponent format printing L.2-52
expression L.1-2
EXT# K.1-2,L.2-25,N.5-2
extended colour fill pattern L.2-28
external variable O.6-1

factor L.1-2
FALSE L.2-26
fatal error L.2-24,M.2-1
filing system
 – commands R.14-1
 – use from BASIC K.3-1
finding text R.11-1,R.15-1
fixed format printing L.2-52
floating point
 – accumulator O.8-3
 – routines O.8-3
 – variable O.8-3
FN L.2-26
FOR L.2-27,L.2-43,L.2-63,L.2-68
foreground colour L.2-11
formal parameter L.2-26
formatter commands S.4-1
function L.2-39
 – keys R.4-1,U.3-1

GCOL L.2-27,N.5-2
general format printing L.2-52
generating contents lists S.18-1
GET L.2-28,N.5-2
GET$ L.2-29,N.5-3
GOSUB L.2-29,L.2-57
GOTO L.2-30
graphics cursor L.2-42
graphics pixel L.2-50
GSREAD encoding U.6-2

high level language O.1-1
HIMEM L.2-30,N.4-2,N.4-5,N.5-3
horizontal/vertical position U.5-8
HVP U.5-8

identifier K.2-1,L.1-4
IF L.2-18,L.2-31,L.2-66
ignoring
– commands S.12-1
– text S.12-1
illegal address P.3-1
immediate
– addressing P.3-3
– mode L.2-23
implied addressing P.3-2
IND U.5-8
indentation options L.2-38
index U.5-8
indirect
– integer K.2-3
– string K.2-6
indirection operator K.1-3,K.1-5, K.2-3
INKEY L.2-31,N.5-3
INKEY$ L.2-32,N.5-3
INPUT L.2-33,N.5-3
INPUT LINE L.2-34
INPUT# K.3-9,L.2-34
insert mode R.5-1
INSTR L.2-34
instruction P.4-1
– mnemonic P.3-1
INT L.2-35
integer K.3-10,L.1-3
– constant K.2-2
– division L.2-41
– indirect K.2-3
– variable K.2-2
interpreter O.1-2
interrupt mask P.4-24
INX P.4-14
INY P.4-14

JMP P.4-14
JSR P.4-14
justification R.12-1,S.1-2

keyboard R.1-2
keyword mode R.3-1

label O.7-3,Q.3-3
LDA P.4-14
LDX P.4-15
LDY P.4-15
LEFT$ L.2-35
LEN L.2-36
LET L.2-36
LINE L.2-37
line
– number L.1-1,L.2-55
– range L.1-3
– width L.2-71
linking files S.13-1
LIST K.1-1,K.1-3,L.2-37
LIST IF K.1-1,L.2-37
LISTO K.1-3,L.2-38
LN L.2-38
LOAD K.3-2,L.2-39,N.5-3
loading
– BASIC programs K.3-1
– text R.13-1
LOCAL L.2-39
local variable L.2-26,L.2-39
LOG L.2-39
logarithm L.2-39
logical
– AND L.2-3,P.4-2
– OR L.2-47,P.4-18
– shift P.4-15
LOMEM L.2-30,L.2-40,N.4-2,N.4-3
loop L.2-56
– nested L.2-43
– variable L.2-43
low level language O.1-1
LSR P.4-15

machine code K.1-4,O.1-1,O.1-3, O.9-1
– environment O.8-1
– file K.3-5
machine operating system K.1-4, N.5-1
macro O.7-5,S.17-1
MCL U.5-15

memory
– map under BASIC N.4-1
– pointer O.6-2
– usage in assembler O.7-1
merging BASIC programs K.3-3
Microsoft BASIC K.1-4
MID$ L.2-40
minimum abbreviations N.2-1
mnemonics P.1-1
MOD L.2-41
MODE L.2-41,N.5-3
mode parameter U.5-1,U.5-14
modem U.1-1
– control lines U.5-15
MOS K.1-4,N.5-1,O.8-2
MOS type L.2-32
MOVE L.2-42,N.5-3
moving text R.9-1
multi-file document S.13-1
multiple declaration Q.3-2

natural logarithm L.2-38
NEL U.5-8
nested loop L.2-43
NEW L.2-42
NEXT L.2-27,L.2-43
next line U.5-8
noise channel L.2-62
non-fatal error M.2-1
NOP P.4-16
NOT L.2-43
note channel 3 L.2-61
numeric array L.2-16

object program O.6-2
OFF L.2-44
OLD L.2-43,L.2-44,U.2-1
ON K.1-1,L.2-44
ON ERROR K.1-5,L.2-24,L.2-45,
M.1-1,Q.2-1
ON... L.2-45
ON...GOTO L.2-45
ON...PROC L.2-45
OPENIN K.3-6,K.3-14,L.2-46,N.5-3
OPENOUT K.3-6,K.3-14,L.2-47,N.5-3
OPENUP K.3-6,K.3-14,L.2-47,N.5-3
operand L.1-3
operating system K.1-4
– commands R.14-1,U.5-6
operation codes P.2-1
operation
– single byte file K.3-7
– whole file K.3-1
operator K.2-1
– in BASIC K.2-6
– precedence K.2-7
OPT O.6-3,O.7-1,P.4-16
OR L.2-47
ORA P.4-18
OSC U.5-6
OSCLI L.2-48,N.5-4
OSHWM N.4-3
OVER mode R.5-1

PAGE L.2-48,N.4-2,N.4-3,N.5-4
PAGE_1 N.4-3
pagination S.7-1
parity U.1-3
PHA P.4-18
PHP P.4-19
PHX P.4-19
PHY P.4-19
PI L.2-49
pitch L.2-19,L.2-62
– envelope L.2-21
– increment L.2-19
PLA P.4-20
PLOT L.2-28,L.2-49,N.5-4
– action L.2-28
– number L.2-49
PLP P.4-20
PLX P.4-20
PLY P.4-21
PM U.5-6
POINT L.2-50,N.5-4
POS L.2-50,N.5-4
post-indexed indirect
addressing P.3-4

pre-indexed
– absolute indirect addressing P.3-5
– indirect addressing P.3-4
PRINT L.2-50
print formatting K.1-5
PRINT# K.3-9,L.2-53
printer ignore character S.3-1
printing S.2-1
– special codes S.14-1
– text R.12-1
privacy message U.5-6
PROC L.2-53
PROC part L.1-4
procedure L.2-14,L.2-39,L.2-53
PROT U.5-15
protect against interference U.5-15
PTR K.3-12,N.5-4
PTR# L.2-54

RAD L.2-54
radian L.2-54
random
– access file K.3-11
– number L.2-58
– number generator L.2-58
READ L.2-13,L.2-55
real K.3-10
– constant K.2-2
– variables K.2-1
real-time clock L.2-67
receive flow control U.5-16
recovering lost text T.4-1
registers L.2-70,N.5-1,S.16-1
relational operator L.1-3
relative addressing P.3-5
release phase L.2-20
REM L.2-55
remark L.2-55
RENUMBER L.2-55
REPEAT L.2-56,L.2-69
replacement string R.11-1
replacing text R.11-1,R.15-1
REPORT L.2-56,M.1-1,N.5-4
reserving memory O.8-1
– for machine code O.6-1
reset
– mode flag U.5-10
– initial state U.5-10
RESTORE L.2-57
restoring text R.10-1
RETURN L.2-57
reverse index U.5-8
RFC U.5-16
TI U.5-8
RIGHT$ L.2-57
RIS U.5-10
RM U.5-10
RND L.2-58
ROL P.4-21
roman numeral S.16-2
ROR P.4-21
RS423
– data format U.1-3
– parity U.1-3
– stop bit U.1-3
– transmission U.1-2
– word length U.1-3
RTI P.4-22
RTS P.4-22
RUN L.2-59
run-time errors Q.4-1

SAVE K.1-3,K.3-1,L.2-59,N.5-5
saving
– BASIC programs K.3-1
– machine code O.6-3
– text R.13-1
SBC P.4-23
SCI U.5-10
scroll
– margins R.7-1
– up U.5-8
SCS U.5-11
search pointer R.15-1
SEC P.4-23
SED P.4-24
select character set U.5-11

sequential
– file K.3-6
– pointer K.3-11
set a mode flag U.5-10
SGN L.2-59
SIB K.2-4
SIN L.2-60
sine L.2-60
single-byte file operation K.3-7
SM U.5-10
software interrupt Q.3-2
SOUND L.2-19,L.2-60,N.5-5
sound channel L.2-61
source program O.6-2
space L.1-4
SPC L.2-33,L.2-51
special code interpretation U.5-10
SQR L.2-63
square-root L.2-63
ST U.5-6
STA P.4-24
stack O.8-2,P.4-18,P.4-19
– pointer P.4-28
STACK_TOP N.4-5
statements L.1-4
status register P.4-1
STEP L.1-3,L.2-27,L.2-63
STOP L.2-64
STR$ L.2-52,L.2-64
string K.2-4,K.3-10,L.1-4
– array L.2-16
– indirect K.2-6
– information block K.2-4,L.2-8
– terminator U.5-6
– variable K.2-4
STRING$ L.2-65
structured programming K.1-4
STX P.4-25
STY P.4-25
STZ P.4-26
SU U.5-8
subroutine L.2-29,L.2-57,P.4-14
substring L.2-34,L.2-40,L.2-41
sustain phase L.2-20
system
– clock L.2-67
– editor K.1-2,O.1-4,R.1-1,T.1-1
– integer variable K.2-3,L.2-70

TAB L.2-33,L.2-51,L.2-65,N.5-5
tabulation L.2-52,S.11-1
TAN L.2-66
tangent L.2-66
target string R.11-1,R.15-1
TAX P.4-26
TAY P.4-26
Telecom Gold U.1-1
terminal U.3-1
– emulator U.1-1
text
– blocks R.9-1
– buffer R.7-1
– cursor L.2-50,L.2-71
– files R.13-1
– formatter R.12-1,S.1-1
– formatting error T.3-1
– window L.2-71
TFC U.5-16
THEN L.2-31,L.2-66
TIME K.1-2,L.2-67,N.5-5
TIME$ K.1-2,L.2-67,N.5-5
TO L.2-27,L.2-68
TOP K.3-3,L.2-68,N.4-2,N.4-3
TRACE L.2-68
translating characters S.14-1
transmit flow control U.5-16
TRB P.4-27
TRUE L.2-69
TSB P.4-27
TSX P.4-28
TXA P.4-28
TXS P.4-28
TYA P.4-29
type
– byte K.3-10
– number L.2-7

underlined text S.10-1

UNTIL L.2-56,L.2-69
user-defined function L.2-14,L.2-26
USER_ZP_END N.4-3
USR K.1-5,L.2-70

VAL L.2-70
VAR_TOP N.4-4,N.4-5
variable K.2-1,L.1-4
 – format in files K.3-10
 – type K.2-1
VDU L.2-70,N.5-6
 – driver mode U.6-1
VPOS L.2-71,N.5-6

white noise L.2-62
whole file operation K.3-1
WIDTH L.2-71
window R.7-1
word wrap mode U.5-16
WORK_END N.4-3
WORK_START N.4-3
WWM U.5-16

zero page
 – addressing P.3-2,P.3-3
 – indexed addressing P.3-4
 – indirect addressing P.3-3

! K.1-5,K.2-3
$ K.1-5,K.2-6
' L.2-33,L.2-51
*CODE O.9-1
*EDIT R.2-1
*EXEC K.3-3,K.3-4
*GO O.9-1
*GOIO O.9-1
*LINE O.9-1
*RUN O.9-1
*SAVE O.6-3
*SPOOL K.3-3
*TERMINAL U.2-1
, L.2-51
– L.2-51
. Q.3-3

6502 assembler O.3-1
65C12 K.1-2
 – assembler O.1-1
 – microprocessor O.4-1
; L.2-51
= U.2-1
> K.1-6
? K.1-5,L.2-33
@% L.2-51
[O.7-1,O.7-2
] O.7-1,O.7-2
¦ L.2-71