Basic V Guide

Indice generale

Introduction....2
History....2
Features....2
Notation....2
Constants....2
Variables....2
Keywords....2
Line Numbers....3
Operators....3
Assignment Operators....3
Indirection Operators....3
Array Operations....4
- Assignment....4
- Addition and Subtraction....4
- Multiplication and Division....4
- Matrix Multiplication....4

Built-in Functions....4
Pseudo Variables....6
Procedures and Functions....8
Error Handling....8
Issuing Commands to the Underlying Operating System....9
Statement Types....9
Statements....9
- Local Arrays....11
- Multiple Prompts....13
- The Format Variable @%....18
- Sending Trace Output to a File....21

Commands....22
The Program Environment....24
Screen Output....24
- VDU Driver....24
- Colours....25
- 2, 4 and 16 Colour Modes....25
- 256 Colour Modes....25
- Text and Graphics Windows....25
- Text-only Modes....25

VDU Commands....25
PLOT Codes....27
Mode Variables....27
Basic Keywords, Commands and Functions....28
- Basic Keywords....28
- Basic Commands....28
- Functions and Pseudo Variables....28

Introduction

The following notes give a brief introduction to Basic V and to the environment that the interpreter emulates. They describe the entire language but not in any great detail; more attention is given to features specific to this version of Basic. Useful information can be found on the web site 'The BBC Lives!' where scanned version of manuals such as the 'BBC Microcomputer User Guide' can be found. The information in these manuals is not 100% relevant to Basic V but they are good for background information and many details of Basic II, the predecessor of (and to all intents and purposes, a strict subset of) Basic V.

These notes describe the Basic language. The file 'use' contains information on how to use the interpreter and on the features and limitations of the different versions of the program.

History

At the start of the 1980s the British Broadcasting Corporation was looking for a microcomputer to be used for their series 'The Computer Programme'. The machine chosen became known as the 'BBC Micro' and it was made by Acorn Computers. It was an extremely potent and flexible little computer that some people still use to this day. The dialect of Basic on it was called 'BBC Basic'. This was an extended Basic that added such features as procedures and multi-line functions. It was also one of the fastest Basic interpreters available on an eight-bit computer. The interpreter was well integrated with the rest of the machine: it was possible to directly call operating system functions from Basic programs and there was also a built-in assembler. If something could not be done in Basic or was too slow it was possible to write the code in assembler. Many programs were written that used a combination of Basic and assembler. Compilers and interpreters for languages such as BCPL, C and Pascal were written for the BBC Micro, but by far the most popular language was Basic.

In 1987 Acorn brought out the Archimedes. This included a new version of the BBC Basic interpreter that had many additional features such as 'CASE' statements and multi-line 'IF' statements. It kept its reputation for speed. This version of Basic was called 'Basic V' and it is the dialect of the language implemented by this interpreter.

The operating system that ran on the Archimedes and the machines that have succeeded it over the years is called 'RISC OS'. This was designed and written by Acorn Computers.

Features

The main features of Basic V are:
1) It is a structured Basic with a full range of statement types such as WHILE and REPEAT loops, a block IF statement and a CASE statement.
2) It has procedures and multi-line functions which can have local variables and arrays.
3) It has 'indirection operators' which allow data structures to be constructed and manipulated. Memory can be allocated from the Basic heap that can be referenced using these operators. (This is not to say that the dialect includes data structures per se but they can be set up and used in a way that appears to be reminiscent of BCPL.)
4) The Acorn-written interpreters include an assembler. Programs can be written using a mix of Basic and assembler. All the features of the Basic interpreter are available to the assembler, so that, for example, functions written in Basic are used as macros.
5) Speed: the interpreter is very fast.

Notation

A few words on the notation used in these notes might be in order.
In describing the syntax of statements, parts of the statement are often put in angle brackets <like this>. The purpose of this is to say what goes at that part of the statement, for example:

```
        GOSUB <line number>
```

This says that the keyword GOSUB is to be followed by a line number if a GOSUB statement is used in the program. Another example:

```
        ON ERROR <statements>
```

This one says that in an 'ON ERROR' statement, the keywords 'ON ERROR' are followed by one or more Basic statements.

In some cases, parts of a statement are in square brackets, for example:

```
        IF <expression> THEN <line number>
                                    [ ELSE <line number> ]
```

This means that that part of the statement is optional. In the example, the '[ELSE <line number>]' part of the statement can be omitted.

Constants

The interpreter supports five types of variable:
- Decimal integer
- Hexadecimal integer
- Binary integer
- Floating point
- String

Hexadecimal constants begin with a '&' and binary ones with a '%'. Strings are enclosed in '"'. It is possible to embed a '"' is a string by preceding it with another '"'. Examples:

```
        &FFFF
        &123456
        %1001101
        %1
        "abcdefghij"
        "klmnop""qrst"
```

Variables

The interpreter supports three main types:

```
        Integer                 e.g. abc%
        Floating point          e.g. def
        String                  e.g. ghi$
```

Integer variables are 32 bits wide. A variable is denoted as being an integer by having a '%' at the end of the name.

Floating point variables are 64 bits wide. If a variable does not have either a '%' or a '$' suffix then it is a floating point variable.

String variables have a '$' suffix at the end of their name. They can refer to strings that have a maximum length of 65,536 characters.

Note that it is possible for variables of different types to have the same name, for example, 'abc%', 'abc' and 'abc$' can all exist at the same time. The '%' and '$' are considered to be part of the name. Similarly, it is possible to have arrays and simple variables of the same name, for example:

```
        abcd$           <-- String variable
        abcd$()         <-- String array
```

What happens is that the '(' is considered to be part of the name of the array.
Variable names are case sensitive, so that 'abcd' and 'AbCd' are different variables.
Basic V has two classes of variables, the normal dynamic variables that are created when the program runs and a set of what are called 'static' variables which comprises of the integer variables A% to Z%. These variables are independent of any program in that their valaues are not reset or changed when a program is loaded or modified. They can, for example, be used to pass values from one program to another.

Keywords

Keywords are Basic's reserved words. They are split into two types in this interpreter, Basic keywords and Basic commands. Examples of the former are 'IF', 'ENDPROC' and 'WHILE'. Examples of commands are 'LOAD', 'LIST' and 'NEW'. A complete list of keywords is given at the end of these notes.
Basic keywords have to be in upper case. On the other hand,

commands can be given in either lower or upper case to make it more convenient to type them in.

The interpreter tries to be clever with keywords. In general keywords cannot be used as variable names but there are cases where if a keyword is followed by a letter it is not identified as a keyword, for example, 'COUNT' on its own is a keyword but 'COUNTER' can be used as a variable without any problems. The keywords taht can are treated in this way are marked with a '*' in the keyword list at the end of the notes.

Line Numbers

Each line in a Basic program has a line number. This is used when editing a program at the command line and in the program by such statements as 'GOTO', 'GOSUB', 'ON GOTO', and 'RESTORE'. Basic V is a structured Basic and line numbers are largely superfluous if statement types such as these are not used. They are still needed to identify the line on which an error was detected when a program runs. Programs and libraries can be written without line numbers using a text editor such as 'vi'. Line numbers will automatically be added when the program is loaded. Similarly, programs can be saved without line numbers, although the default is to include them.
Line numbers are in the range 0 to 65279.

Operators

Basic V supports the usual range of operators. The following list details what is available. The operators are given in priority order, with operators of the same priority grouped together. From highest to lowest priority they are:

```
        ^               Exponentiation
-----------------
        *               Multiplication
        /               Division
        DIV             Integer division
        MOD             Integer modulus
-----------------
        +               Addition and string concatenation
        -               Subtraction
-----------------
        =               Equals
        <>              Not equals
        >               Greater than
        <               Less than
        >=              Greater than or equal to
        <=              Less than or equal to
        <<              Left shift
        >>              Arithmetic right shift
        >>>             Logical right shift
-----------------
        AND             Logical AND
-----------------
        OR              Logical OR
        EOR             Exclusive OR
```

There is a point to watch out for when using the comparison and shift operators. It is not possible to chain these together in an expression, for example:

```
        abc% >> 2 <= 10
```

will give an error. The solution is to put brackets around the first part of the expression thus:

```
        (abc% >> 2) <= 10
```

This is a feature of Basic V.

Assignment Operators

As well as using '=' for assignments in the normal way, Basic V has two other assignment operators:

```
        <variable> += <expression>
        <variable> -= <expression>
```

'+=' adds <expression> to the variable <variable> and '-=' subtracts it.
Examples:

```
        abc% += 1
        ghi(N%) -= count
        xyz$ += "abcd"
        table%!offset% -= X%
        $(table%+name%) += "xyz"
```

Indirection Operators

These are equivalent to 'peek' and 'poke' in other versions of Basic except that they are far more flexible and powerful. Strictly speaking these are not proper operators but language constructs.

There are two types of operator, unary and dyadic. The unary ones are:

```
        ?       Reference to a one byte integer
        !       Reference to a four byte integer
        |       Reference to a floating point value
        $       Reference to a string
```

The dyadic operators are:

```
        ?       Reference to a one byte integer
        !       Reference to a four byte integer
```

Note that there are not dyadic versions of '$' and '|'.

Unary operators can be followed by a variable, array reference or an expression in parentheses. For dyadic operators, the item before the operator must be a variable or array reference. A variable, array reference or expression in parentheses follows the operator. Examples of unary operators:

```
        $pointer%
        !abc%
        ?(abc%+10)
        $text%
        $text% = "abc"
```

Examples of dyadic operators:

```
        pointer%!offset%
        abc%?next%
        array%(N%)!field%
```

The operators all work in the same way. The value of the expression after the operator (unary version) or of the variable before the operator (dyadic version) is interpreted as an address. The value after the operator in the dyadic version is a byte offset from that address.

Indirection operators cannot be chained together, that is, they cannot be used in an expression such as:

```
        pointer%!offset%!field%
```

In general, indirection operators can be used in the same way and places as normal variables and the two forms of reference can be freely mixed. Examples:

```
        IF $text%="quit" THEN STOP
        table%!offset% = table%!offset2
        !(table%+offset%) = !(table%+offset2)
        abc = |address+1.0
        PROCabcd(!table%, table%!4)
        FOR table%!8=1 TO 10: NEXT
```

The interpreter limits the range of addresses that can be read from or written to using indirection operators to the Basic workspace. It is not possible to access any location outside this block of memory.

Indirection operators can be used to built up and manipulate data structures. However they are not true data structures in the sense of structs in a C program and there are no checks on the legality of references (beyond ensuring that the addresses are in range). It is possible for a program to allocate memory from the Basic heap to be used used for data structures and accessed via the indirection operators. A special form of the DIM statement is used for this:

```
        DIM table% 100
```

Strictly speaking this allocates a byte array with indexes 0 to 100. From a more practical point of view, it allocates a 101 byte block of memory and puts its address in table%. This block can then be manipulated using the indirection operators as desired, for example:

```
        $table%="an error message"
        table%!0 = 0: table%!4 = 99
```

In fact, blocks of memory allocated this way can only be referenced via indirection operators.

Array Operations

The interpreter supports some arithmetic operations on entire arrays. There are some restrictions: the arrays have to be the same size (number of dimensions and size of each dimension) and of exactly the same type. Also, general expressions involving arrays are not allowed, nor is it possible to return an array as the result from a function. What is allowed is as follows:

Assignment

```
<array 1> = <array 2>
The contents of <array 2> are copied to <array 1>

<array> = <expression>
All elements of array <array> are set to <expression>

<array> = <expression 1> , <expression 2> , ... , <expression n>
Each expression <expression x> is evaluated and then assigned
to the x'th element of the array <array>. There can be fewer
expressions than there are elements in the array, in which case
the remaining array elements are left unchanged.

<array 1> += <array 2>, <array 1> -= <array2>
Each element of <array 2> is added to <subtracted from) the
corresponding element in <array 1>.

<array> += <expression>, <array> -= <expression>
The expression <expression> is evaluated and the result added
to <subtracted from) each element of <array>.
Examples:
         abc%() = def%()
         ghi$() = "test"
         jkl() = 0.0, 1.1, 2.2, 3.3, FNxyz(4.4)
         abc%() += def%()
         jhl() -= PI
```

Addition and Subtraction

```
<array 1> = <array 2> + <array 3>
Add the corresponding elements of <array 2> and <array 3> and
store the result in the same element in <array 1>.

<array 1> = <array 2> - <array 3>
Subtract the elements in <array 3> from the corresponding element
in array <2> and store the result in <array 1>.

<array 1> = <array 2> + <expression>
<array 1> = <expression> + <array 2>
Add <expression> to each element of <array 2>, storing the result
of each addition in the corresponding element of <array 1>.

<array 1> = <array 2> - <expression>
<array 1> = <expression> - <array 2>
Subtract <expression> from each element of <array 2>, storing the
result of each subtraction in the corresponding element of
<array 1>.

Examples:
         abc%() = def%() + ghi%()
         jkl() = mno() - pqr()
         aaa$() = bbb$() + "ccc" + FNddd(eee$)
         abc%() = 1 - def%()
```

Multiplication and Division

```
<array 1> = <array 2> * <array 3>
Multiply each element of <array 2> by the corresponding element
in <array 3> and store the result in <array 1>.

<array 1> = <array 2> / <array 3>
Divide each element of <array 2> by the corresponding element
in <array 3> and store the result in <array 1>.

<array 1> = <array 2> DIV <array 3>
Carry out an integer division of each element of <array 2> by the
corresponding element in <array 3> and store the result in
<array 1>.

<array 1> = <array 2> MOD <array 3>
Carry out an integer division of each element of <array 2> by the
corresponding element in <array 3> and store the remainder in
<array 1>.

<array 1> = <array 2> * <expression>
<array 1> = <expression> * <array 2>
Multiply each element of <array 2> by the value <expression> and
store the result in the corresponding element in <array 1>.

<array 1> = <array 2> / <expression>
Divide each element of <array 2> by the value <expression> and
store the result in the corresponding element in <array 1>.

<array 1> = <expression> / <array 2>
Divide <expression> by each element of <array 2> and store the
result in the corresponding element in <array 1>.

<array 1> = <array 2> DIV <expression>
Carry out an integer division of each element of <array 2> by the
value <expression> and store the result in the corresponding
element in <array 1>.

<array 1> = <expression> DIV <array 2>
Carry out an integer division of <expression> by each element of
<array 2> and store the result in the corresponding element in
<array 1>.

<array 1> = <array 2> MOD <expression>
Carry out an integer division of each element of <array 2> by the
value <expression> and store the remainder in the corresponding
element in <array 1>.

<array 1> = <expression> MOD <array 2>
Carry out an integer division of <expression> by each element of
<array 2> and store the remainder in the corresponding element in
<array 1>.

Examples:
         abc() = def() * ghi()
         abc() = 10.0 * ghi()
         jkl%() = mno%() MOD 100
         abc() = 1 / abc()
```

Matrix Multiplication

```
<array 1> = <array 2> . <array 3>
Perform a matrix multiplication of <array 2> and <array 3> and
store the result in <array 1>.
Note that '.' is used as the matrix multiplication operator.
```

Built-in Functions

The interpreter has a fairly standard set of functions. One feature of this dialect of Basic is that many of the functions look like monadic operators, for example, a call to the 'LEN' function can be written as 'LEN abc$' as well as 'LEN(abc$)'.

Function names can often be abbreviated when typing them at the command line. The abbreviated version of the name is the first few characters of the name followed by a dot, for example, the 'LE' is the abbreviated form of 'LEFT$('. The names are given in full when the program is listed.

Following is a list of functions implemented and a summary of their actions. More detailed information on the vast majority of them can be found in the manuals on the 'The BBC Lives!' web site.

Entries marked with a '*' after the name are functions added in this interpreter.

<factor> represents a simple expression that consists of just a variable name, array reference or constant or a complete expression in parentheses.

<expression> is a full expression. Sometimes this is written as <string expression> or <numeric expression> to qualify the type of expression, or abbreviated to <expr> to reduce clutter.
<array> is a reference to a whole array.

```
ABS
       Use: ABS <factor>
       Returns the absolute value of the numeric value
       <factor>

ACS
       Use: ACS <factor>
       Returns the arccosine of the numeric value <factor>

ADVAL
       Use: ADVAL <factor>
       This is an unsupported function. Either use of it is
       flagged as an error or it returns zero, depending on the
```

options used to start the interpreter.

ARGC *
Use: ARGC
Returns the number of parameters on the command line. This will be zero if there are no parameters.

ARGV$ *
Use: ARGV$ <factor>
Returns parameter number <factor> on the command line as a string. ARGV$ 0 returns the name of the program. ARGV$ 1 is the first parameter, ARGV$ 2 the second and so forth. ARGV$ ARGC is the last parameter.

ASN
Use: ASN <factor>
Returns the arcsine of the numeric value <factor>

ATN
Use: ATN <factor>
Returns the arctan of the numeric value <factor>

BEAT
Use: BEAT
Returns information from the RISC OS sound system. This function returns zero.

BGET
Use: BGET# <factor>
Returns the next byte from the file with handle <factor>

CHR$
Use: CHR$ <factor>
Returns a string consisting of a single character with ASCII code <factor>

COLOUR
Use: COLOUR(<red expression>, <green expression>, <blue expression>)
This takes the colour with the specified colour components and returns a number that represents the closest match to that colour in the current screen mode. This value is for use with the 'COLOUR OF' and 'GCOL OF' statements. It has no meaning otherwise.

Example:
red = COLOUR(255, 0, 0): COLOUR OF red

COS
Use: COS <factor>
Returns the cosine of the numeric value <factor>

COUNT
Use: COUNT
Returns the number of characters printed on the current line by PRINT.

DEG
Use: DEG <factor>
Converts the angle <factor> from radians to degrees.

DIM
Use: a) DIM(<array>)
b) DIM(<array>, <expression>)
a) returns the number of dimensions in array <array>.
b) returns the highest index of dimemsion <expression> of array <array>.

END
Use: END
Returns the address of the top of the Basic heap.

EOF
Use: EOF# <factor>
Returns TRUE if the file with handle <factor> is at end of file.

ERL
Use: ERL
Returns the number of the line that contained the last error encountered by the interpreter or zero if no error has been seen.

ERR
Use: ERR
Returns the error number of the last error encountered by the interpreter or zero.

EVAL
Use: EVAL <factor>
Evaluates the string <factor> as if it were an expression in a statement in the program and returns the result.

EXP
Use: EXP <factor>
Returns the exponentional of the numeric value <factor>.

FALSE
Use: FALSE
The function returns the value corresponding to 'false' in the interpreter (zero).

GET
Use: GET
Returns the next character pressed on the keyboard as a number, waiting if there is not one available.

GET$
Use: a) GET$
b) GET$# <factor>
a) Returns the next character pressed on the keyboard as a one character string, waiting if there is not one available.
b) Returns the next line from the open file with handle <factor> as a character string.

INKEY
Use: INKEY <factor>
If numeric value <factor> is greater than or equal to zero, return the next character pressed on the keyboard as a num but only wait for <factor> centiseconds.
Return -1 if no key pressed in that time.
If numeric value <factor> is -256, return a number that identifies the operating system under which the program is running. (See the 'use' guide for the values returned.)
If numeric factor <factor> is less than zero and greater than -256, return TRUE if the key with RISC OS internal key number <factor> is being pressed otherwise ret. FALSE.

INKEY$
Use: INKEY$ <factor>
This is the same as INKEY but returns its result as single character string. In the case of a keyboard read with timeout, an empty string is returned if the time limit expires instead of -1.

INSTR(
Use: INSTR(<expr1> , <expr2> [, <expr3>])
Search string <expr1> for the string <expr2> returning the index (starting from 1) of the start of <expr2> in <expr1> if the string is found otherwise return zero. <expr3> is an option expression that gives a starting point in <expr1> at which to start looking for <expr2>.

INT
Use: INT <factor>
Returns the integer part of number <factor>, rounding down (towards minus infinity).

LEN
Use: LEN <factor>
Returns the length of string <factor>.

LISTO *
Use: LISTO
Returns the current LISTO setting.

LN
Use: LN <factor>
Return the natural log of number <factor>.

LOG
Use: LOG <factor>
Returns the base 10 log of number <factor>.

MOD Use: MOD <array>
Returns the modulus (square root of the sum of the squares) of numeric array <array>.

MODE Use: MODE
Returns the number of the current RISC OS screen mode.

NOT
Use: NOT <factor>
Returns the logical negation (ones complement) of numeric value <factor>.

OPENIN
Use: OPENIN <factor>
Opens the file named by the string <factor> for input and

```
       returns its numeric handle or zero if it cannot be opened.

OPENOUT
       Use: OPENOUT <factor>
       Opens the file named by the string <factor> for output and
       returns its numeric handle. If the file exists already its
       length is reset to zero.

OPENUP
       Use: OPENUP <factor>
       Opens the file named by the string <factor> for both input
       and output and returns its numeric handle.

PI
       Use: PI
       Returns the value PI.

POINT(
       Use: POINT(<x expr>,<y expr>)
       Returns the colour number of the point on the graphics
       screen with graphics coordinates (<x expr>, <y expr>).

POS
       Use: POS
       Returns the offset (from 0) of the text cursor from
       the left-hand side of the screen.

QUIT
       Use: QUIT
       Returns TRUE if the interpreter was started with the
       option '-quit', that is, the interpreter will be exited
       from when the Basic program finishes running.

RAD
       Use: RAD <factor>
       Convert the angle <factor> given in degrees to radians.

REPORT$
       Use: REPORT$
       Returns the message for the last error encountered.

RND
       Use: a) RND
            b) RND(<negative expr>)
            c) RND(0)
            d) RND(1)
            e) RND(<expression>)
       a) Return a pseudo-random number in the range -2147483648
          to 2147483647
       b) Initialises the random number generator with seed value
          <negative expr>.
       c) Returns the last number generated by RND(1).
       d) Returns a floating point number in the range 0 to 1.
       e) Returns an integer number in the range 1 to <expression>.

SGN
       Use: SGN <factor>
       Returns -1 if the numeric value <factor> is less than
       zero, zero if it is zero or 1 if it is greater than zero.

SIN
       Use: SIN <factor>
       Returns the sine of numeric value <factor>.

SQR
       Use: SQR <factor>
       Returns the square root of numeric value <factor>.

STR
       Use: a) STR <factor>
            b) STR~ <factor>
       a) Converts the numeric value <factor> to a decimal
       string.
       b) Converts the numeric value <factor> to a hexadecimal
       string.

STRING$(
       Use: STRING$( <expression>, <string expr> )
       Returns a string made from the string <string expr> repeated
       <expression> times.

SUM
       Use: SUM <array>
       If <array> is a numeric array it returns the sum of all of
       the elements of the array. If <array> is a string array it
       returns a string made from all of the elements of <array>
       concatenated.

SUM LEN
       Use: SUM LEN <array>
       Returns the total length of all of the strings in string
       array <array>.

TAN
       Use: TAN <factor>
       Returns the tangent of numeric value <factor>

TEMPO
       Use: TEMPO
       Unsupported RISC OS feature. The function returns zero or
       generates an error depending on intepreter command line
       options.

TINT
       Use: TINT(<x expr>,<y expr>)
       Returns the TINT value of the position with x and y
       graphics coordinates <x expr> and <y expr> on the screen
       in 256 colour modes.

TOP
       Use: TOP
       Returns the address of the first byte after the Basic
       program.

TRACE
       Use: TRACE
       Returns the handle of the file to which TRACE output is
       being written or zero if a file is not being used.

TRUE
       Use: TRUE
       Returns the value used by the interpreter for 'true' (-1).

USR
       Use: USR <factor>
       Calls machine code at address <factor>. Not implemented
       this interpreter except in one special case. See the
       'use' guide.

VAL
       Use: VAL <factor>
       Converts the string <factor> to a number.

VERIFY( *
       Use: VERIFY(<expr 1>, <expr 2> [, <expr 3>] )
       Returns the offset of the first character in string
       expression <expr 1> that is not in string <expr 2>
       or zero if all characters in <expr 1> are in <expr 2>.
       <expr 3> is the optional position at which to start
       the search.

VDU
       Use: VDU <factor>
       Returns the value of the RISC OS mode variable given by
       <factor>. This can be used to determine such things as
       the screen width, the number of colours and so forth.
       Refer to the section 'Mode Variables' below for more
       details.

VPOS
       Use: VPOS
       Returns the offset (from zero) from the top of the screen of
       the text cursor.

WIDTH
       Use: WIDTH
       The function returns the current value of 'WIDTH' (the
       width of the current line as set by the program) or zero
       if a line width has not been defined via the WIDTH
       statement.

XLATE$( *
       Use: a) XLATE$(<expr 1>)
            b) XLATE$(<expr 1>, <expr 2>)
       a) Returns the string expression <expr 1> with all
       upper case characters converted to lower case.
       b) Returns the string expression <expr 1> with the
       characters translated using string expression or
       string array <expr 2>. Characters in <expr 1> are
       replaced with the character in the position
       corresponding to the ASCII code of the original
       character.
```

Pseudo Variables

Pseudo variables are a half way house between a function and a variable. When they appear in the right hand side of an expression they are functions but they can also appear on the left hand side of an expression too.

Pseudo variables cannot follow the keyword 'LET', that is, using them in statements such as:

```
        LET FILEPATH$="."
```

is not permitted. Similarly, they can only be followed by '=', that is, assignments of the form '<something>+=<value>' are not allowed.

EXT
As a function:
Use: EXT# <factor>
Returns the size of the open file with the handle <factor>. Example:

```
oldsize% = EXT#file%
```

On left-hand side:
Use: EXT# <factor> = <expression>
Change the size of the open file with handle <factor> to <expression> bytes.
Example:

```
IF action$="delete" THEN EXT#file% = 0
```

FILEPATH$
As a function:
Use: FILEPATH$
Returns the list of directories to search when trying to find a library or a program.
Example:

```
PRINT"Current search path: ";FILEPATH$
```

On left-hand side:
Use: FILEPATH$ = <expression>
Sets the directory list to the string expression <expression>. Note that there are no checks to make sure that the directory names are valid. Setting FILEPATH$ to an empty string is allowed.
Example:

```
FILEPATH$ = "/home/mine,/usr/local/basic"
```

HIMEM
As a function:
Use: HIMEM
Returns the address of the end of the Basic workspace.
Example:

```
PRINT "Top of workspace is at ";~HIMEM
```

On left-hand side:
Use: HIMEM = <expression>
This sets the address of the top of the Basic workspace to the value of the numeric expression <expression>. If the new value of HIMEM puts it in the Basic program or outside the Basic workspace, the statement is ignored.
Example:

```
HIMEM = HIMEM-1000
```

The places where HIMEM can be changed are limited. It cannot be altered in a procedure, function or subroutine, nor in the body of a loop. The only safe place to change it is at the start of a program.

LEFT$
As a function:
Use: LEFT$(<string expression> [, <expression>)
Returns the left-hand <expression> characters from the string <string expression>. <expression> can be omitted, in which case <string expression> with the last character removed is returned.
Examples:

```
LEFT$(abc$, 4)
LEFT$($table%, 10)
LEFT$(xyz$(X%))
```

On left-hand side:
Use: LEFT$(<string variable> [, <expression>])
= <string expression>
This replaces the left-hand characters of string <string variable> with the string expression <string expression>. <expression> says how many characters to replace. If it is omitted then the length of <string expression> is used. If this value exceeds the original length of <string variable> then the length of <string variable> is used instead.

<string variable> can be a normal string variable, an array reference or a string referenced by the string indirection operator.
Examples:

```
LEFT$(abc$)="1234"
LEFT$(abc$, 2)="abcdefgh"
LEFT$(xyz$(X%), 5)="123"
LEFT$($table%, 3)="pqrst"
```

LOMEM
As a function:
Use: LOMEM
Returns the address of the start of the Basic heap
Example:

```
PRINT"Variables start at ";~LOMEM
```

On left-hand side:
Use: LOMEM = <expression>
This changes the address of the start of the Basic heap to the numeric value <expression>. All of the variables created so far are discarded when LOMEM is changed. If the value is outside the range TOP to HIMEM it is ignored. The assignment is also ignored if LOMEM is changed in a function or procedure.
Examples:

```
LOMEM = LOMEM+1000
```

MID$
As a function:
Use: MID$(<string expression>, <start expression>
[, <length expression>])

Returns a substring from the string <string expression> starting at character position <start expression>. The substring is of length <length expression> characters. <length expression> can be omitted, in which case the the string from the <start expression>'th character to the end of the string is returned.

If <start expression> is negative or exceeds the length of the string the empty string is returned. If <length expression> is negative or exceeds the number of characters in the string, the original string is returned.
Examples:

```
A$ = MID$(B$, 10)
A$ = MID$(B$, 10, 20)
A$ = MID$($table%, 5, 10)
```

On left-hand side:
Use: MID$(<string variable>, <start expression>
[, <length expression>]) = <string expression>
The characters of the string <string variable> starting at character position <start expression> are overwritten by characters from string <string expression>. <length expression> is optional and says how many characters are to be taken from <string expression>. If it is omitted then all of <string expression> is used.

Only the existing characters of <string variable> are overwritten. The length of the string is never changed.

If <start expression> is negative then <string variable> is overwritten from the start of the string. If it exceeds the length of <string variable> then nothing is changed. If <length expression> is negative then the length of <string expression> is used instead.
Examples:

```
MID$(A$, 5) = "ABCD"
MID$(A$, 10, 10) = X$ + "abcdefghijklmnop"
MID$($table%, 10) = "12345"
```

PAGE
As a function:
Use: PAGE
Returns the address of the start of the Basic program.
Example:

```
PRINT"The program starts at ";~PAGE
```

On left-hand side:
Use: PAGE = <expression>
This sets the address of the start of the Basic program in memory to <expression>. Any program currently loaded is discarded. The change is ignored if the value of <expression> is outside the range of the value of PAGE when the interpreter was started to HIMEM.

Note: one trick possible with the Acorn interpreter is to hold several programs in memory at the same time and to switch between them by altering PAGE. This does not work with the current version of this interpreter.
Example:

```
PAGE = PAGE + 1000
```

PTR
As a function:

```
        Use: PTR# <factor>
        Returns the value of the offset in bytes of the file
        pointer in the open file with handle <factor>.
        Example:
                place = PTR# thefile%

On left-hand side:
        Use: PTR# <factor> = <expression>
        Sets the file pointer of the open file with handle <factor>
        to <expression>. This is offset into the file in bytes.
        Example:
                PTR# thefile% = 0

RIGHT$
As a function:
        Use: RIGHT$(<string variable> [, <expression>] )
        Returns a string containing the right-most <expression>
        characters from the string <string variable>. It is
        possible to omit <expression>, when just the last
        character is returned. If <expression> is zero or
        negative, the empty string is returned. If it exceeds
        the length of <string variable>, the original string is
        returned.
        Examples:
                RIGHT$(A$, 5)
                RIGHT$(X%) + RIGHT$(Y$)

Om left-hand side:
        Use: RIGHT$(<string variable> [, <expression>]) =
                                            <string expression>
        The right-most <expression> characters of the string
        <string variable> are overwritten with character from
        <string expression>. If <expression> is omitted, the
        length of the string <string expression> is used instead.
        If it is zero or negative, <string variable> is left
        unchanged. The length of the string <string variable>
        is never changed.
        Examples:
                RIGHT$(A$) = "1234"
                RIGHT$(A$, 5) = "abcdefgh"
                RIGHT$($table%, 1) = A$

TIME
As a function:
        Use: TIME
        Returns the current value of the centisecond counter.
        This is a 32-bit timer updated one hundred times per
        second, although the real accuracy depends on the
        underlying operating system.
        Example:
                newtime = TIME+100

On left-hand side:
        Use: TIME = <expression>
        Sets the centisecond counter to <expression>.
        Example:
                TIME = 0

TIME$
As a function:
        Use: TIME$
        Returns the current date and time as a string in the
        form: "www,dd mmm yyyy.hh:mm:ss"
        Example:
                now$ = TIME$

On left-hand side:
        Use: TIME$ = <expression>
        Sets the date and time to the string expression
        <expression>. This feature is not implemented.
```

Procedures and Functions

Basic V has both procedures and multi-line functions. They can both take parameters of any sort, including arrays, and it is possible to return values via parameters. They can also have local variables.

Procedures and functions are declared in the same way:

```
Procedures:       DEF PROC<name> [ ( <parameter list> ) ]
Functions:        DEF FN<name> [ ( <parameter list> ) ]
```

<name> is the name of the procedure or function. The keywords 'PROC' and 'FN' are considered to be part of the name.

<parameter list> is the list of variables to be used as formal parameters. There can be any number of these. Names are separated by commas. Parameters where values are to be returned are preceded by the keyword RETURN.

Examples:

```
        DEF PROCabc
        DEF PROCxyz(aa$)
        DEF FNpqr(X%, abc%, def$)
        DEF PROCijk(RETURN X%, RETURN Y%)
        DEF FNmno(array())
        DEF PROCpqr(array1$(), RETURN array2$())
```

When a procedure or function is called, the current values of the variables that are to be used as parameters are saved before they are set to the values they will take for the call. When the call ends their old values are restored.

All of the parameters are evaluated before the values are assigned to the parameter variables.

Procedures and functions are called in the normal way, for example:

```
        PROCxyz("abcd")
        value = FNpqr(A%+1, B%+2, C$+"3")
```

Calls to procedures are ended with ENDPROC. Calls to functions end with an '=' followed by the value the function is to return, for example:

```
        DEF PROCabc(X%)
        IF X%=0 THEN ENDPROC
        Y% = Y% DIV X%
        ENDPROC

        DEF FNxyz(X%)
        IF X%=0 THEN = 0
        = Y% DIV X%
```

Note that the name of a function does not include the type of the result it will return. It is possible for the same function to return both string and numeric values, but this is of very limited use. (The only place it will not give an error is in a PRINT statement.)

Recursive calls to procedures and functions are allowed. The only limit on the depth of the recursion is the amount of memory available.

Procedures and functions can have local variables. These are defined by means of the LOCAL statement. The format of this is:

```
        LOCAL <list of variables>
```

for example:

```
        LOCAL abc%, def, ghi$, jkl$
```

Any number of local variables can be declared. It is also possible to have local arrays. These are slightly more complicated in that the array is declared to be local and its dimensions defined separately, thus:

```
        LOCAL array()
        DIM array(100)
```

The scope of local variables is dynamic, that is, they are not restricted to the procedure or function in which they were defined. It is perhaps easier to understand this by considering what happens: when a variable is declared to be local, its value (if the variable exists already) is saved and the value reset to zero (or the empty string). When the procedure or function is returned from, the old value is restored. Declaring a variable to be local does not create a new version of that variable. The same variable is still used but its old value is saved first.

Error Handling

Basic V provides two statements for dealing with errors, ON ERROR and ON ERROR LOCAL. ON ERROR is the less sophisticated of the two. If an error is detected, the interpreter ends all loops and returns from all procedures, functions and subroutines before continuing with the statements after ON ERROR. It is not possible to recover from the error and restart the program at the point where it occured. Most of the time all that can be done is to tidy up and abort the program. ON ERROR LOCAL gives more control in that when an error occurs, the statements after the ON ERROR LOCAL are executed but everything is left as it was at the time of the

error. This means it is possible to trap errors and recover from them.

The statement 'ON ERROR OFF' turns off the trapping of errors in the program. This should be used at the start of an error handler to prevent errors in the error handler causing an infinite loop.

To allow for finer control over errors, Basic V also has two statements that allow different error handlers to be used at different points in the program, LOCAL ERROR and RESTORE error. LOCAL ERROR stores details of any existing error handler and allows a new one to be set up. RESTORE ERROR restores the error handler to the saved one. If LOCAL ERROR is used in a function or procedure, the old error handler is restored when the function or procedured calls ends.

Care should be taken if using ON ERROR LOCAL within the body of a loop. If an error is detected once the program has exited from the loop, it will branch back into it when the interpreter goes to the statements after ON ERROR LOCAL. The interpreter is unaware of the context of the error handler (that is, it has back into the body of a loop) and unpredictable results might ensue.

The function REPORT$ returns the last error message. ERR returns the number of that error and ERL the number of the line in which it occured. Note that this does not say whether the error occured in the program or a library. The REPORT statement displays the last error message. The ERROR statement can be used to report user-generated errors. The interpreter allows the use of the LIST command in programs, so it is possible for error handlers to list the line in which the error occured by the statement

```
LIST ERL
```

Note that the line listed will always be in the Basic program so if the error occured in a library the wrong line will be shown.

Some of the errors that the interpreter flags are classed as 'fatal', for example, running out of memory. The Basic program is always abandoned if a fatal error occurs. It is not possible to trap them with ON ERROR or ON ERROR LOCAL.

Issuing Commands to the Underlying Operating System

There are two ways in which commands can be send to the operating system on which the interpreter is running. The most flexible way is the OSCLI statement. The second way is to put the command on a line preceded by a '*', for example:

```
10 *date
20 IF flag THEN *help
```

Whatever follows the '*' up to the end of the line is passed as the command.

There is no restriction on the commands that can be issued this way.

Statement Types

Statements are broken into two types, executable statements that can appear in a program and commands that, in general, can only be used on the command line.

Many of the keywords and commands can be abbreviated when they are typed in. They will be shown in their complete form when the program is listed. The rule is to type in the minimum number of characters of the keyword and then follow it with a dot, for example:

```
P.
```

can be type instead of 'PRINT'. The minimum abbreviation for each keyword and command is given in the section 'Basic Keywords, Commands and Functions' at the end of these notes.

All of the Basic V statement types are described below. However, not all versions of the Basic V interpreter support all of them, in particular, the graphics statements might not be available.

Statements

In the following:

<factor> represents a simple expression that consists of just a variable name, array reference or constant or a complete expression in parentheses.

<expression> is a full expression. Sometimes this is written as <string expression> or <numeric expression> to qualify the type of expression, or abbreviated to <expr> to reduce clutter.

<array> is a reference to a whole array.

<statements> is one or more Basic statements.

Items in square brackets are optional.

BEATS
Unsupported statement for controlling the RISC OS sound system.

BPUT
Syntax: a) BPUT#<factor>, <expression> [;]
b) BPUT#<factor>, <expr 1>, <expr 2>, ... ,<expr n> [;]

BPUT is used to write data to a file. The handle of the file is given by <factor>. <expression> is the value to be written. If <expression> is numeric, the result of the expression modulo 256 (that is, the least significant byte) is written to the file. If <expression> is a string, the complete string is written to the file. If the string expression is followed by a ';' that is all that happens. If the ';' is absent, a 'newline' character (ASCII code 10) is also written to the file after the string.

The second form of the BPUT statement is the same as the first except that a list of items to be written separated by commas can be supplied. If the last item is a string expression then a new line character is also written to the file unless the expression is followed by a ';'.
Examples:

```
BPUT#outfile, X%
BPUT#outfile, A$
IF TRACE THEN BPUT#TRACE, "Result so far is "+STR$X%
BPUT#outfile, 1, 2, 3, 4, 5
BPUT#outfile, STR$A%, " ", STR$B%
```

CALL
This is an unsupported statement that allows machine code subroutines to be called.

CASE
Syntax: CASE <expression> OF

This marks the start of a CASE statement. This statement must be the last one on a line.

The complete syntax of a CASE statement is:

```
CASE <expression> OF
WHEN <expression 1>: <statements>
WHEN <expression 2>: <statements>
OTHERWISE: <statements>
ENDCASE
```

The expression <expression> is evaluated. The interpreter then searches for 'WHEN' statements and evaluates each expression after the 'WHEN' in turn and compares it to the result of <expression>. If they are equal it starts executing the statements <statements> after the 'WHEN' from that point to the next 'WHEN', 'OTHERWISE' or 'ENDCASE'. At this point it jumps to the statements after the ENDCASE.

Any number of expressions (limited by the length of the line) can follow the WHEN keyword. They are separated by commas.

The expressions after the WHEN keyword do not have to be constants. Any type of expression is allowed and they can be of numeric or string types.

The WHEN keyword must be the first item on a line after the line number (except for any intervening blanks). The same goes for the OTHERWISE and ENDCASE keywords.

CASE statements can be nested to any depth.

Examples:

```
CASE day$ OF
WHEN "Monday": PRINT"It is Monday"
```

```
        WHEN "Tuesday", "Thursday":
          PRINT"It is Tuesday or Thursday"
        WHEN "Friday":
          PRINT"It is Friday"
       WHEN "Wednesday": PRINT"It it the middle of the week"
        OTHERWISE
          PRINT"It is the weekend"
       ENDCASE
```

As general expressions can be used after the WHEN, CASE statements are very flexible. Here one is being used in place of a series of IF statements:

```
        CASE TRUE OF
        WHEN abc%<10: state%=1
       WHEN abc%=10: state%=2
       WHEN abc%>20 AND abc%<20: state%=3
       OTHERWISE: state%=99
       ENDCASE
```

CHAIN
Syntax: CHAIN <string expression>

CHAIN loads and runs the program named by the string <string expression>. The programs currently in memory is replaced by this one and the values of all variables lost, with the exception of the static integer variables.

Extension: The interpreter searches for the program to load in the directories given by the pseudo-variable FILEPATH$.

CIRCLE
Syntax: a) CIRCLE <x expression>,<y expression>,<expression>
        b) CIRCLE FILL <x expression>,<y expression>,<expression>

a) This draws a circle outline centred at (<x expression>, <y expression>) and with a radius <expression> in the current graphics foreground colour.

b) This version plots a filled circle centred at (<x expression>, <y expression>) and with a radius <expression> using the current graphics foreground colour.

CLG
Syntax: CLG

This statement clears the graphics window (normally the whole screen) to the current graphics background colour.

CLEAR
Syntax: CLEAR

CLEAR discards all of the variables and arrays created so far in the program. It also clears the chain of called procedures and functions so it should not be used in a procedure or function.

Example:

```
        IF silly% THEN CLEAR
```

CLOSE
Syntax: CLOSE# <factor>

The CLOSE statement closes one or more open files. <factor> is a numeric value that gives the handle of the file to close. If <factor> is zero then *all* files that have been opened by the Basic program are closed.

Example:

```
        CLOSE#outfile%
```

CLS
Syntax: CLS

CLS clears the screen (or the current text window if one is being used). It also sends the cursor to the top left-hand corner of the screen.

Example:

```
        IF full% THEN CLS
```

COLOUR
Syntax: a) COLOUR <expression>
        b) COLOUR <colour expression> TINT <tint expression>
        c) COLOUR <red expression> , <green expression> ,
                                    <blue expression>
        d) COLOUR <logical expression> , <physical expression>
        e) COLOUR <colour expression> , <red expression> ,
                                  <green expression> , <blue expression>
        f) COLOUR OF <expression> ON <expression>
        g) COLOUR OF <red expression>, <green expression>,
                <blue expression> ON <red expression>,
                <green expression>, <blue expression>

The COLOUR statement is used to change the colour being used when writing text on the screen. It is also used to change the physical colour corresponding to the given logical colour.

a) This sets the colour to be used when writing text on the screen. <expression> is a numeric value. It is reduced modulo the number of colours available in the current screen mode. The colour changed is the logical colour.

If the original value of <expression> is less than 128, the colour changed is the text foreground colour; otherwise the background colour is altered. 128 is subtracted from the vale of <expression> to get the colour number.

b) This version of the statement is used in 256 colour modes to set the colour used for writing text on the screen. <colour expression> sets the logical colour and <tint expression> sets the 'tint' value. If the value of <colour expression> is less than 128 then the foreground colour is changed otherwise the background one is altered. Whether or not <colour expression> is less than 128 affects whether the foreground tint value or background tint value is the one changed by <tint expression>.

The colour value is reduced modulo 64 to obtain the colour. The tint value has set values: 0, 64, 128 and 192. The increasing tint value has an effect on the brightness of the colour. See the section '256 Colour Modes' below for an explanation of how the colour and tint work together.

c) This sets the current text foreground colour to the colour with components <red expression>, <green expression>, <blue expression>. The values of the colour components are reduced modulo 256. The colour is mapped to the nearest equivalent colour in the current screen mode.

d) This version of the COLOUR statement changes the mapping between the logical colour number and the colour displayed on the screen in screen modes which have less than 256 colours. <logical expression> is the logical colour number whose mapping is to be changed. The value is reduced modulo the number of colours in the current screen mode. <physical colour> is the colour number of the physical colour to be used. The colour numbers are given in the section '2, 4 and 16 Colour Modes' below.

e) This version of the statement is related to c) above in that it alters the colour displayed for a given logical colour number in screen modes with more than sixteen colours available. <colour expression> is the number of the colour to change and <red expression>, <green expression> and <blue expression> are the red, green and blue components of the new colour. These are reduced modulo 256. The colour number is also reduced modulo 256.

f) This version of the statement has two parts, the 'OF' part and the 'ON' part. The 'OF' part gives the foreground colour and the 'ON' part the background colour. Either of these parts can be omitted but not both of them. The value <expression> is the colour number to use.

g) This version has two parts, the 'OF' part and the 'ON' part where the 'OF' part is used to give the foreground colour and the 'ON' part the background colour. The three expressions after each keyword, <red expression>, <green expression> and <blue expression>, give the red, green and blue components of the colour to use. The colour that will actually be used is the closest match to this colour in the current screen mode. Either the 'OF' or the 'ON' part can be left out but not both.

Examples:

```
        COLOUR 5
        COLOUR 23 TINT &C0
        COLOUR 2, 128, 128, 128
        COLOUR OF 0, 0, 255 ON 0, 0, 0
        COLOUR ON 191, 191, 191
        COLOUR ON COLOUR(191, 191, 191)
```

The 'COLOUR OF' statement (along with 'GCOL OF') represent a new way of selecting colours to use on screen. Along with the 'COLOUR()' function they provide a means of specifying colours independently of the screen mode, for example:

```
        pink = COLOUR(255, 127, 255)
        blue = COLOUR(0, 0, 255)
        COLOUR OF pink ON blue
```

This avoids use of the 'TINT' keyword.

```
DATA
Syntax: DATA <items of data>
The DATA statement provides the data to be read by a READ
statement. It must be the first keyword on a line after the
line number.

<items of data> is one or more items of data separated by
commas.

DEF
'DEF' marks the start of a procedure or function definition. It
must be the first non-blank item on a line after the line number.
Refer to the section 'Procedures and Functions' above for more
information.

Examples:
        DEF PROCfirst
        DEF FNsecond(X%)

DIM
Syntax: DIM <list of arrays>

The DIM statement is used to declare arrays. <list of arrays>
contains one or more arrays to be declared separated by
commas. There are two types of array declation. The
format of a normal array declaration is:

        <name>( <expression 1> , ... , <expression n> )

where <name> is the name of the array and <expression 1> to
<expression n> are the array's dimensions. Arrays can have up to
ten dimensions and their size is limited only by the available
memory.

Array indexes start at zero with the size given in the array
declaration as the highest index of each dimension.

Note that the '(' is considered part of the array name.

Examples:
        DIM abc%(100), def(10,10), ghi$(size%+10)
The second form is used to allocated blocks of memory. There
are two versions of this:

        <variable> <expression>
        <variable> LOCAL <expression>

where <variable> is the name of a variable and <expression> is the
size of the block to allocate in bytes. The address of the block
is stored in <variable>.

The first form can be used anywhere but the second can only be
used in a procedure or function. In the second case the block
is allocated on the Basic stack and is automatically returned
when the procedure or function containing the DIM statement ends.
In the first case memory is obtained from the Basic heap and
it cannot be returned to the heap for reuse later.

To be more accurate, the second form allocates a byte array with a
low index of zero and a high index of <expression>. For this
reason the size of the block allocated is actually <expression>+1
bytes.

Examples:
        DIM heap% 100000, block% 100
        DIM abc%(10), block% 1000
        DIM xyz% LOCAL 1000
        LOCAL ptr%: DIM ptr% LOCAL 100

One trick is to declare an array in this way with a size of -1
bytes. This stores the address of the current top of the Basic
heap in the variable, for example:

        DIM heaptop% -1

The same effect can be obtained using the function 'END'.
```

Local Arrays

```
The interpreter allows local arrays to be created in procedures
and functions. The memory for these is reclaimed when the procedure
or function call ends.

The way in which local arrays are defined is as follows:
        LOCAL <array>
        DIM <array>

In other words, the array is declared local first and then its
dimensions are defined, for example:

        LOCAL abc$()
        DIM abc$(10,10)

DRAW and DRAW BY
Syntax: a) DRAW <x expression> , <y expression>
        b) DRAW BY <x expression> , <y expression>

The DRAW statement draws a line in the current graphics foreground
colour from the current graphics cursor position to the one given
by <x expression>, <y expression>.

a) <x expression> and <y expression> are absolute coordinates.

b) <x expression> and <y expression> are the offsets from the
current graphics cursor position of the end point of the line.

Examples:
        DRAW 500,100: DRAW 500,500: DRAW 100,100
        DRAW BY 400,0: DRAW BY 0,400: DRAW BY -400,-400

ELLIPSE
Syntax: a) ELLIPSE <x expression> , <y expression> ,
                        <semi-major> , <semi-minor> , <angle>
        b) ELLIPSE FILL <x expression> , <y expression> ,
                        <semi-major> , <semi-minor> , <angle>

The ELLIPSE statement draws an ellipse. The coordinates of the
centre are (<x expression>, <y expression>). <semi-major> and
<semi-minor> are the lengths of the semi-major and semi-minor axes
respectively. <angle> is the angle between the semi-major axis and
the x axis in radians.

Note that values other than zero for the angle only work in the
RISC OS version of the program in the current version of the
interpreter.

a) This draws an ellipse outline.

b) This draws a filled ellipse.

Example:
        ELLIPSE 500, 500, 400, 200, 0.5

ELSE
Syntax: ELSE

The ELSE statement is part of an IF statement. Refer to the
section on the IF statement for more information.

END
Syntax: END

As a statement, END stops the program.

Example:
        IF alldone THEN END

ENDCASE
Syntax: ENDCASE

Part of a 'CASE' statement. It marks the end of the statement.
Refer to the section on the 'CASE' statement above for more
details.

ENDIF
Syntax: ENDIF

Part of a block 'IF' statement. It marks the end of the statement.
Refer to the section on the 'IF' statement below for more
information.

ENDPROC
Syntax: ENDPROC

The ENDPROC statement is used to return from a procedure to the
calling routine. Variables corresponding to RETURN parameters are
set to the their returned values, all local variables declared in
the procedure are restored to their original values and local
arrays destroyed. The effects of any LOCAL DATA or LOCAL ERROR
statements are undone. Control then passes back to the statement
after the procedure call.

Example:
        IF count%=0 THEN ENDPROC

ENDWHILE
Syntax: ENDWHILE

ENDWHILE marks the end of a WHILE loop. Refer to the section
below on the WHILE statement for more information.
```

```
Example:
        ENDWHILE

ENVELOPE
Syntax: ENVELOPE <expression 1> , ... , <expression 14>

This is an unsupported statement. It was used as part of the sound
system on the BBC Micro but it is completely redundant in Basic V.

ERROR
Syntax: ERROR <error expression> , <string expression>

The ERROR statement is used to generate a user-defined error.
<error expression> is the number of the error and <string
expression> is the error message. Errors raised this way can be
trapped just like any other using ON ERROR and ON ERROR LOCAL.

Example:
        ERROR 25, "Bad error"

FALSE
Syntax: FALSE

FALSE returns the value corresponding to 'false' in the
interpreter, zero.

Example:

        flag = FALSE

FILL and FILL BY
Syntax: a) FILL <x expression>, <y expression>
        b) FILL BY <x expression> , <y expression>

The FILL statement is used to flood-fill areas with the current
graphics foreground colour.

a) <x expression> and <y expression> give the coordinates of the
point at which to start the flood fill.

b) <x expression> and <y expression> give the offsets from the
current graphics cursor position at which to start the flood
fill.

Example:
        FILL 500,100

FOR
Syntax: FOR <variable> = <start expression> TO <end expression>
                          [ STEP <step expression> ]

The FOR statement marks the start of a FOR loop. <start
expression> and <end expression> are numeric values that give the
start and end values for the loop index, <variable>. <step
expression> is optional and gives the amount by which the loop
index is incremented or decremented on each iteration. If omitted,
it defaults to one.

Execution continues from the statement after the FOR statement to
the first NEXT statement encountered. At this point the loop index
is incremented (or decremented if the step value is negative) and
compared against the end expression. If it exceeds that value (or
is less than it if the step is negative) the loop is terminated,
otherwise program execution continues back at the statement after
the FOR.

The body of the loop will be executed at least once.

<end expression> and <step expression> are evaluated only once, at
the start of the loop.

As with all the loop constructs, exiting the loop will
automatically undo the effect of any LOCAL DATA or LOCAL ERROR
statements that were used in the loop. Loops badly nested within
the FOR loop body will also be silently ignored.

Examples:
        FOR N% = 1 TO 10: NEXT
        FOR N% = 10 TO 1 STEP -1: NEXT
        FOR abc = FNstart TO FNfinish STEP 10: NEXT
        FOR array%(5) = 1 TO 10: NEXT array%(5)

The last example shows that the loop index does not have to be
a simple variable.

GCOL
Syntax: a) GCOL <colour expression>
        b) GCOL <colour expression> TINT <tint expression>
        c) GCOL <action expression> , <colour expression>
        d) GCOL <action expression> , <colour expression>
                                          TINT <tint expression>
        e) GCOL <red expression> , <green expression> ,
                                              <blue expression>
        f) GCOL OF <expression> ON <expression>
        g) GCOL OF <action expression>, <expression>
                  ON <action expression>, <expression>
        h) GCOL OF <red expression>, <green expression>,
                 <blue expression> ON <red expression>,
                 <green expression>, <blue expression>
        i) GCOL OF <action expression>, <red expression>,
                 <green expression>, <blue expression>
                 ON <action expression>, <red expression>,
                 <green expression>, <blue expression>


GCOL is used to set the graphics foreground or background colour.
The various combinations are as follows:

a) Set the graphics colour to logical colour number <colour
expression>. This value is reduced modulo the number of colours
available in the current screen mode in 2, 4 and 16 colour modes
and modulo 64 in 256 colour modes. If the value is less than 128
then the foreground colour is altered. If it is 128 or more, 128
is subtracted from the colour number and the background colour
changed.

b) Set the graphics colour to logical colour number <colour
expression> and the 'tint' value to <tint expression>. This
version is used in 256 colour modes. The colour number is reduced
modulo 64. The tint value should be set to 0, 64, 128 or 192.

c) This form changes the graphics colour to logical colour number
<colour expression> and sets the graphics plot action to <action
expression>. The RISC OS version of the interpreter supports the
full range of plot actions but others are restricted to just plot
action zero, where each point plotted overwrites the one already
there.

d) This is a combination of cases b) and c) and is used in 256
colour modes. The graphics colour is set to colour number <colour
expression> with tint value <tint expression>. The graphics plot
action is set to <action expression>.

e) The current graphics foreground colour is set to the colour
with colour components <red expression>, <green expression> and
<blue expression>. The colour used will be the closest match to
the specified colour in the current screen mode.

f) and g) are very similar. There are two parts to the statement,
the 'OF' part and the 'ON' part. The 'OF' part gives the colour
to use for the graphics foreground colour and the 'ON' part
the background colour. In case f), only the colour is given but
in g) both the colour and the graphics plotting action are
supplied. It is possible to leave out either of the 'OF' or
the 'ON' parts but not both.

h) and i) are similar.  There are two parts to the statement,
the 'OF' part and the 'ON' part. The 'OF' part gives the
components of the colour to use for the graphics foreground
colour and the 'ON' part the components of the background
colour. The actual colour used will be the closest match to
the specified colour in the current screen mode. In case h),
only the colour is given but in i) both the colour and the
graphics plotting action are supplied. It is possible to
leave out either of the 'OF' or the 'ON' parts but not both.

Examples:
        GCOL OF 192,192,192
        GCOL OF 0, 0, 0 ON 255,255,255
        GCOL OF 1, 15
        GCOL OF COLOUR(0, 0, 255)

The 'CCOL OF' statement (along with 'COLOUR OF') represent
a new way of selecting colours to use on screen. Along with
the 'COLOUR()' function they provide a means of specifying
colours independently of the screen mode, for example:

        pink = GCOL(255, 127, 255)
        blue = GCOL(0, 0, 255)
        GCOL OF pink ON blue

This avoids the use of the 'TINT' keyword, which is really
only of use in old-style RISC OS 256 colour modes.

GOSUB
Syntax: a) GOSUB <line number>
        b) GOSUB ( <expression> )

The GOSUB statement calls a subroutine. Control passes back to the
statement after the GOSUB by means of a RETURN statement.

a) The subroutine to be called starts at line <line number>.
```

b) The line number of the subroutine to be called is found given by the numeric expression <expression>.

Note that the RENUMBER command will update the line numbers in the first form of GOSUB above but it cannot handle the second form.

Example:
```
        GOSUB 1000
```

GOTO
Syntax: a) GOTO <line number>
        b) GOTO (<expression>)

Program execution continues at the line specified on the GOTO statement.

a) The number of the line is given explicitly and so the interpreter can go to that line immediately.

b) The line number is given by the numeric expression <expression>.

Note that the RENUMBER command will update the line numbers in the first form of GOTO above but it cannot handle the second form.

IF
```
Syntax: a) IF <expression> [ [ THEN ] <statements> ]
                                  [ ELSE <statements> ]
        b) IF <expression> THEN <line number>
                                    [ ELSE <line number> ]
        c) IF <expression> THEN
               <statements>
           [ ELSE
               <statements> ]
           ENDIF
```

a) The first form of IF statement is the single line IF.

The numeric expression <expression> is evaluated. If the result is not zero ('true') the statements after the THEN keyword are executed otherwise the ones after the ELSE keywords are executed.

There can be any number of statements after the THEN or ELSE separated by colons. The only limit is the length of the line. Those statements can include further IF statements and some confusion can arise as to which IF statement an ELSE clause applies. If the interpreter has to look for an ELSE as the result of the IF expression is zero, it stops at the first one it finds after the current IF.

It is possible to omit the THEN clause or the ELSE clause. It is also possible to leave out the THEN. If the item after the IF expression is neither a THEN nor a ELSE keyword, the THEN keyword is assumed.

Examples:
```
        IF abc$="xyz" THEN PROCabc: PROCdef
        IF def<2.0 ELSE PRINT"def is too large"
        IF X% PROCpqr
        IF abc$="abc" THEN PRINT"abc": abc$="def" ELSE abc$="abc"
```

b) This is a variation on the single line IF. Either or both of the THEN and ELSE keywords can be followed by a line number instead of one or more statements. Program execution continues at that line iIf the interpreter finds a line number after a THEN or ELSE keyword.

Examples:
```
        IF abc%=1 THEN 100
        IF def<2.0 THEN PROCabc ELSE 200
        IF ghi$="abc" ELSE 300
```

c) The second form of the IF statement is the block IF.

If there is nothing else on the line after a THEN keyword or just a REM statement then the IF statement is a block IF.

The ELSE part is optional but if it is included the ELSE keyword must be the first item on the line after the line number and any leading blanks. The statements for the ELSE part can start on the same line as the ELSE.

The ENDIF must be the first item on the line after the line number and any preceding blanks.

Block IF statements can be nested to any depth.

Examples:
```
        IF X%=1 THEN
          PROCabc
        ENDIF

        IF X%=2 THEN
          PROCabc
        ELSE
          PROCdef
        ENDIF

        IF X%=2 THEN
          IF Y%=1 THEN STOP
        ELSE
          PROCabc
          PROCdef
        ENDIF

        IF X=2 THEN REM X is set to 2
          PROCabc
        ELSE
          PROCdef
        ENDIF
```

Note in this last example that there is a REM statement after the THEN but the statement is still seen as a block IF.

INPUT
```
Syntax: a) INPUT <list of variables>
        b) INPUT# <factor>, <list of variables>
        c) INPUT LINE <list of variables>
        d) LINE INPUT <list of variables>
```

The INPUT statement is used to read data from the terminal of from a file.

a) This form of INPUT statement reads from the terminal. The format is more accurately specified as:

```
        INPUT [<prompt> [,]] <list of variables>
                 [<prompt> [,]] <list of variables>] ...
```

<prompt> is a prompt displayed before the input is read. It can be made up of any number of the following: a string, the function TAB() and "'". TAB() is used to display the prompt at a specific position on the screen. "'" causes a skip to the next line. Examples:

```
        INPUT TAB(10, 10) "Value: "
        INPUT "Size"
        INPUT TAB(0, 10) "Enter coordinates:" ' "X: "
```

TAB() takes two parameters, the column number and the row number to which to move the text cursor, that is, the point at which the next character will be written.

If the prompt is followed by a comma, a '?' is printed after the prompt.

If the prompt is left out, a '?' will be displayed as the prompt.

<list of variables> is the list of variables to receive the values read. There can be any number of these. The variable names can be separated with blanks, commas or semicolons.

Examples:

```
        INPUT X%,Y%
        INPUT abc(1), abc(2), abc(3)
        INPUT xyz$, ghi%
```

Values typed in are separated by commas. Leading blanks are ignored when they are read. When reading a string, trailing blanks as far as the next comma or the end of the are considered to be part of the string.

Strings can be enclosed in double quotes.

Numeric values can be entered in hexadecimal or binary as well as decimal. Hexadecimal values are preceded with a '&' and binary ones with a '%'. (Note that this is an extension in this interpreter.)

If insufficient values are supplied or an error found, the interpreter prompts again for a value to be entered.

Multiple Prompts

It is possible to repeat as many prompt and variable list sequences as desired. When 'return' is pressed the next prompt is displayed and the next set of values read. Any values not used

from the previous read are discarded.

Example:

```
        INPUT "Name: " name$ "Address: " address$
```

b) The second form of INPUT statement is used to read data from a file. <factor> is the handle of the file from which input is to be taken and <list of variables> gives the variables to received those values.

Note that the data is assumed to be formatted binary data produced using PRINT#. INPUT# cannot be used to read text from a file.

Example:

```
        INPUT# file% , abc(1), abc(2), xyz$
```

c) and d) These are a variation on format a). The difference is that each value read is taken from a new line.

Example:

```
        LINE INPUT "Coordinates: " X% Y%
```

LET
Syntax: LET <variable> = <expression>

<expression> is evaluated and the result assigned to the variable <variable>.

Only the '=' assignment operator is allowed here.

Pseudo-variables, for example, PTR#, cannot be used after LET.

LIBRARY
Syntax: a) LIBRARY <expression 1>, <expression 2>, ... , <expression n>
        b) LIBRARY LOCAL <list of variables>

This statement has two purposes. In case a), it is used to read libraries of Basic procedures and functions into memory. <expression 1> to <expression n> are strings that give the names of the libraries. There can be any number of these. The libraries are held in memory until the Basic program is run again or edited or the statements NEW or CLEAR used.

When a procedure or function is called, the interpreter checks to see if it is one already encountered. If not it searches the Basic program for it. If it cannot be found the interpreter then searches the loaded libraries and it if still cannot locate it, the libraries loaded via the INSTALL command. Libraries are searched in the reverse order to which they were loaded, for example, given:

```
        LIBRARY "aaaaa"
        LIBRARY "bbbbb"
        LIBRARY "ccccc"
```

'ccccc' will be searched first, then 'bbbbb' and lastly 'aaaaa'.

Libraries can be seen as an extension of the program in memory rather than separate entities. They can have their own private variables (declared using LIBRARY LOCAL) but any other variables created in procedures and functions in the library will be added to those the program creates.

LIBRARY LOCAL is used to define variables and arrays that will be private to a library. Only the library will be able to reference these variables and arrays.

The syntax of the statement is as follows:

```
        LIBRARY LOCAL <list of variables and arrays>
```

where <list of variable and arrays> is a list of variable and array names separated by commas, for example:

```
        LIBRARY LOCAL abc%, def%, ghi$, jkl(), mno(), pqr$()
```

In the case of arrays, this merely defines that the array is local to the library. The dimensions of the array have to be declared using a DIM statement in the normal way, for example:

```
        LIBRARY LOCAL jkl(), pqr$()
        DIM jkl(100), pqr$(table_size%)
```

There can be as many LIBRARY LOCAL and DIM statements as necessary but they have to be before the first DEF PROC or DEF FN statement in the library. They also have to be the first item on the line.

The variables can only be referenced in the library. They are not visible outside it. This is different to the way in which local variables are dealt with in a procedure or function, where local varables can be accessed by any procedure of function called by the procedure in which they were declared. They can duplicate the names of variables in the Basic program or other libraries.

When looking for a variable in a library, the interpreter first searches for it amongst the library's private variables and then in the Basic program's.

The variables and arrays are created the first time the library is referenced. In practice this means that they are set up when the interpreter has to search the library for a procedure or function.

Private variables in a library can further be used as local variables in a procedure or function. Note that they can only be accessed in the library, for example:

```
        LIBRARY LOCAL abc%, def, ghi$, jkl(), mno$()
        DIM jkl(100)

        DEF PROCaaa(abc%)
        LOCAL def, ghi$
        ENDPROC

        DEF PROCbbb(def, jkl())
        LOCAL mno$()
        DIM mno$(100)
        ENDPROC

        DEF PROCccc(xyz, abc%)
        ENDPROC
```

Here, abc%, def, ghi$, jkl() and mno$() are all declared to be private to the library. The dimensions of jkl() are also defined.

In PROCaaa, abc% is used as a formal parameter (effectively a local variable) and def and ghi$ declared to be local to the procedure. Any procedure or function *in the library* that PROCaaa calls that use def and ghi$ will use PROCaaa's local versions. Any procedure or function that PROCaaa calls that are *outside* the library *will not* see these variables.

In PROCbbb, def and jkl() are used as formal parameters and mno$() is defined as a local array and its dimensions given. Note that this is the first place where the dimensions have been defined.

In PROCccc, two variables are used as formal parameters, xyz and abc%. This case is more complex in that abc% is one of the library's private variables whereas xyz is not. xyz is one of the Basic program's variables. abc% can only be referenced in the library but xyz is visible anywhere.

The rules for the scope of private library variables may sound complex but they are quite simple. The point to remember is that a private variable in *only* accessible in the library in which it was declared. If a variable is not declared on a LIBRARY LOCAL statement then it is visible anywhere.

LINE
Syntax: a) LINE <x expression 1> , <y expression 1> ,
                        <x expression 2> , <y expression 2>
        b) LINE INPUT <list of variables>

a) This form of the statement draws a line on the screen in the current graphics foreground colour from coordinates
(<x expression 1> , <y expression 1>) to
(<x expression 2> , <y expression 2>).

Example:

```
        LINE 100, 100, 400, 500
```

b) This is a version of the INPUT statement. It is described in the section on INPUT above.

LOCAL
Syntax: a) LOCAL <list of variables>
        b) LOCAL DATA
        c) LOCAL ERROR

a) This version of the LOCAL statement is used in a procedure or function to declare local variables. The keyword LOCAL is followed by any number of variable names separated by commas, for example:

```
        LOCAL abc%, def, ghi$, jkl
```

Arrays can also be declared to be local, but the definition of

a local array is slightly more complicated in that the new array dimensions have to be given on a DIM statement, for example:

```
        LOCAL xyz()
        DIM xyz(100,10)
```

LOCAL statements can only appear in procedure or function.

The term 'local variables' is somewhat misleading in this context in that the variables are accessible not just in the procedure or function in which they are declared but anywhere in the program. What actually happens is that if the variable already exists, its old value is saved and then it is reset to zero (or the empty string in the case of string variables). When the procedure or function in which the variable was declared local is exited from, the old value is restored. Variables that do not exist are created by the LOCAL statement but they are not destroyed when the procedure or function is left.

Anything that can appear on the left-hand side of an assignment can in fact be declared as a local variable, so that, for example, individual elements can be declared local if need be! Example:

```
        LOCAL abc(25), abc(30)
```

This would declare two elements of array abc(), elements 25 and 30, to be local. The array abc() has to exist already for this to work. Another example:

```
        LOCAL block%!4, block%?20, $text%
```

This would preserve the integer value at address block%!4, the byte at block%?20 and the string at $text%.

b) 'LOCAL DATA' saves the current value of the DATA statement pointer. It can be reset to its original value by 'RESTORE DATA'. Example:

```
        LOCAL DATA
        RESTORE 100
        READ abc%
        RESTORE DATA
```

Uses of LOCAL DATA can be nested without any problems.

Note that there are times when the DATA pointer will be set back to its old value automatically:

1) If LOCAL DATA is used in a procedure or function the old value will be restored when the function or procedure is exited from.

2) If it is used inside a WHILE, REPEAT or FOR loop, the value will again be restored when the loop ends.

It is possible to leave out the RESTORE DATA but it is probably best to include it.

c) The 'LOCAL ERROR' statement is used to save the details of the current 'ON ERROR' error handler so that it can be changed and restored later. 'RESTORE ERROR' is used to reset it. Example:

```
        LOCAL ERROR
        ON ERROR LOCAL PROClocal_error: ENDPROC
        X=X/0
        RESTORE ERROR
```

There is no limit on the number of times LOCAL ERROR can be used (other than the memory available). It is possible to nest LOCAL ERROR statements, that is, LOCAL ERROR can be used in a procedure, say, and then it can be used again in the procedures that that procedure calls without any problems.

As with LOCAL DATA, there are times when the old error handler details will be restored automatically. They are:

1) If LOCAL ERROR is used in a procedure or function the old handler will be restored when the function or procedure is exited from.

2) If it is used inside a WHILE, REPEAT or FOR loop, the handler will again be restored when the loop ends.

So RESTORE ERROR can be omitted but it is probably best to include it.

MODE
Syntax: a) MODE <expression>
        b) MODE <x expression>, <y expression>, <depth>, <rate>

a) The MODE statement is used to change the screen mode. <expression> is the new screen mode. It can be either a number or a string.

If the mode is numeric, it has to be in the range 0 to 127. This gives the RISC OS screen mode number. The range of modes defined corresponds to those available under RISC OS 3.1 (modes 0 to 46) but whether or not all of these are available depends on the machine on which the program is being run. Mode numbers greater than 46 are undefined and mode 0 is used instead. There is one special mode, mode 127. This corresponds to the screen or window size of the environment in which the interpreter is being run, for example, if the program is being run in an xterm under NetBSD with a window size of 96 characters by 50 lines, mode 0 (80 by 32) will use only part of this but mode 127 will switch to the entire window.

The details of the screen mode can also be given as a string. This can be the RISC OS screen mode number as a string or a more precise specification of the resolution and the number of colours to be used.

A mode string has the format:

```
        X<x resolution> Y<y resolution> [ C<colours> ] [ G<levels> ]
```

where <x resolution> and <y resolution> give the screen size in pixels, <colours> gives the number of colours for a colour screen mode and <levels> the number of levels for grey-scale. Example:

```
        MODE "X800 Y600 C256"
```

The number of colours or grey scale levels is optional. The parts of the mode string can be separated by any number of commas or blanks.

As with numeric screen modes, whether or not the screen mode is available depends on the machine on which the interpreter is being used.

b) In this version of the statement, <x expression> and <y expression> give the desired size of the screen in graphics units. <depth> is a value that gives the number of colours and <rate> the frame rate. The last parameter, <rate> can be left out, in which case the highest frame rate available will be used.

The program will attempt to match the details of the requested mode with those available and will only switch to that mode if it finds a match.

The possible values for <depth> are as follows:

```
        1       2 colours
        2       4 colours
        4       16 colours
        6       256 colours, old-style RISC OS mode
        8       256 colours, new-style RISC OS mode
        16      32K colours
        32      16M colours
```

The values possible depend on the version of the program being used. In general, depths of 16 and 32 are only available in the version of the program that runs under RISC OS.

Under RISC OS, a depth of 6 specifies an old-style Archimedes type 256 colour screen mode in which the extent to which colours can be changed is limited. A depth of 8 indicates that a newer type (RISC OS 3.5 and later) screen mode is to be used where all of the colours can be changed.

Examples:

```
        MODE 1024, 768, 8
        MODE 640, 512, 2, 75
```

MOUSE
Syntax: a) MOUSE ON
        b) MOUSE OFF
        c) MOUSE <x variable> , <y variable> , <button variable>
                                  [, <timestamp variable>]
        d) MOUSE STEP <expression> [, <expression>]
        e) MOUSE COLOUR <expression> , <red expression>,
                        <green expression> , <blue expression>
        f) MOUSE RECTANGLE <left expression> , <bottom expression>,
                        <right expression> , <top expression>
        g) MOUSE TO <x expression> , <y expression>

The MOUSE statement is used to control various aspects of the mouse.

a) MOUSE ON turns on the mouse pointer if it is not being

displayed.

b) MOUSE OFF turns off the mouse pointer.

c) This version of the statement reads the current position of the mouse and the button state. The values are stored in <x variable>, <y variable> and <button variable>. There is one optional parameter, <timestamp variable>, which is set to the time at which the mouse position was recorded derived from the centisecond timer.

Example:
```
MOUSE xpos%, ypos%, buttons%
```

d) MOUSE STEP is used to the mouse multiplier, that is the number of graphics units on the screen that each step of the mouse makes. One or two values can be given. <expression> is a number greater than or equal to zero. If one value is supplied then both the X and Y steps are set to this value. If there are two values, the X multiplier is set to the first value and the Y multiplier to the second. It is possible to set the multiplier in either direction to zero, in which case the mouse will not move in that direction.
Examples:

```
MOUSE STEP 4,3
MOUSE STEP 2,0
```

e) MOUSE COLOUR sets the colour of the mouse pointer on screen. <expression> is in the range one to three and says which of the three mouse colours to change. <red expression>, <green expression> and <blue expression> are the colour components of the new colour.

Example:
```
MOUSE COLOUR 1, 255, 0, 0
```

f) MOUSE RECTANGLE defines a box on the screen outside of which the mouse pointer cannot be moved. It requires four parameters, <left expression>, <bottom expression>, <right expression> and <top expression> which give the coordinates of the bottom left-hand and top right-hand corners of the box in graphics units.

Example:
```
MOUSE RECTANGLE 100, 100, 900, 600
```

g) The MOUSE TO statement moves the mouse pointer to the coordinates (<x expression>, <y expression>) on the screen. The position is expressed in graphics units.

Example:
```
MOUSE TO 600, 800
```

MOVE and MOVE BY
Syntax: a) MOVE <x expression> , <y expression>
```
b) MOVE BY <x expression> , <y expression>
```

The MOVE statements move the graphics cursor to the position specified without drawing a line.

a) MOVE moves the cursor to position <x expression>, <y expression>.

b) MOVE BY moves the cursor by an amount <x expression> and <y expression> in the X and Y directions relative to the old graphics cursor position.

Examples:
```
MOVE 500, 500
MOVE BY 100, 60
```

NEXT
Syntax: NEXT [<variable name>]

The NEXT statement is part of the FOR loop. It marks the end of the loop. <variable name> is optional, and is the name of the FOR loop index variable. Refer to the section above on the FOR statement for more details.

Example:
```
NEXT N%
```

It is possible to end a number of loops on a single NEXT statement. The syntax for this type of NEXT is:

```
NEXT [ <variable name 1> ] , [ <variable name 2> ] , ...
                    [ <variable name n> ]
```

In other words, NEXT is followed by a list of variable names separated by commas. However, the variable names can be omitted, in which case NEXT is followed by a series of commas, one less in total than the number of FOR loops being ended at that point.

Examples:
```
FOR X% = 1 TO 10
  FOR Y% = 1 TO 10
NEXT Y%,X%

FOR abc = 1 TO 100
  FOR def = 1 TO 100
NEXT ,
```

OF
'OF' is part of a CASE statement. Refer to the section on CASE statements above for more details.

Example:
```
CASE abc% OF
WHEN 1: PRINT"1"
ENDCASE
```

OFF
Syntax: OFF

This statement turns off the text cursor so that it is not displayed.

Example:
```
IF hide% THEN OFF
```

ON
Syntax: a) ON
```
b) ON <expression> GOTO <line number 1> , ... ,
                    <line number n> [ ELSE <statement> ]
c) ON <expression> GOSUB <line number 1> , ... ,
                    <line number n> [ ELSE <statement> ]
d) ON <expression> PROC<name 1>, ... ,
                    PROC<name n> [ ELSE <statement> ]
e) ON ERROR <statements>
f) ON ERROR OFF
g) ON ERROR LOCAL <statements>
```

The ON keyword is used in a variety of statements as follows:

a) ON on its own turns on the text cursor if it is turned off so that it is being displayed on the screen.

b) In this form of ON statement, the expression <expression> is used to control which line to branch to next. <expression> is a number greater than or equal to one. If it is one, the first line listed after the GOTO keyword is the one to branch to, if it is two, the second one list is the destination and so forth. Two things can happen if <expression> is less than one or greater than the number of line numbers given. If the ELSE part is supplied, the interpreter jumps to the statement after this. Note that only a single statement is allowed here. If there is no ELSE part, an error is raised.

The line numbers after the GOTO can be given as expressions if desired, but the RENUMBER command will probably fail if this is done because it cannot deal with line numbers given in this way.

Examples:
```
ON abc% GOTO 100, 500, 900
ON def GOTO 1000, 2000, 3000, 4000 ELSE STOP
ON ghi% GOTO FNline1, FNline2, 1000, abc%*10+1
```

c) ON ... GOSUB is similar to ON ... GOTO. The expression is used to control which subroutine to call. When the subroutine call ends, the program continues to run at the statement after the ON ... GOSUB. <expression> is used as an index to choose which line number after the GOSUB to call. If the value of <expression> is n, the n'th line number is the one selected. There are two possibilities if <expression> is less than one or the greater than the number of line numbers supplied. If there is an ELSE part, the interpreter branches to that. If there is no ELSE, an error is flagged.

The line numbers can be given as expressions if required.

Example:

```
ON abc% GOSUB 1000, 2000, 3000
ON def GOSUB 1000, 2000 ELSE PROCerror: PRINT"Done"
```

In second example, if def is set to one, the subroutine at line 1000 will be called. When it returns, execution resumes at the statement 'PRINT"Done"' as this is the statement after the ON ... GOSUB. Similarly, when PROCerror returns, execution also continues with the PRINT statement.

d) ON ... PROC is again similar to ON ... GOTO. The result of numeric expression <expression> is used as an index to locate the

procedure to call. If the value of <expression> is n, the n'th procedure listed is the one called. When the procedure call has ended, execution continue at the statement after the ON ... PROC. There are two possibilities if <expression> is out of range, that is, is less than one or greater than the number of PROC statements after the expression. If there is an ELSE part, the interpreter continues at that statement, otherwise an error is reported.

Example:
```
        ON abc% PROCaaa, PROCbbb(X%), PROCccc("abc")
        ON def PROCaaa, PROCaaa, PROCbbb(1) ELSE STOP
```

e) ON ERROR
The ON ERROR statement is used to deal with errors that the interpreter detects in the Basic program. When an error occurs the interpreter continues with the statements after the 'ON ERROR'. However, before it does so it automatically returns from all procedures, functions and subroutines as well as ending any loops. It is therefore not possible to recover from an error and resume execution of the program at the point at which it was detected.

Example:
```
        ON ERROR PROCcomplain: END
```

The section 'Error Handling' above discusses trapping errors in programs in more detail.

f) ON ERROR OFF
This statement turns off the trapping of errors by the Basic program.

g) The ON ERROR LOCAL statement is used to trap errors in the Basic program. When one of these statements is executed by the interpreter it notes the address of the statements after the keywords ON ERROR LOCAL. When an error is detected it continues at those statements.

ON ERROR LOCAL statement gives more control than ON ERROR in that everything is left as it was at the time of the error. This means that it is possible to trap errors and recover from them, resuming at the point of the error.

Example:
```
        ON ERROR LOCAL PRINT"Error - ";REPORT$
        INPUT X,Y
        PRINT X/Y
```

Here, if an error occured when inputting the values X and Y or when X is divided by Y, the interpreter would branch to the PRINT statement after the ON ERROR LOCAL, display the error message and ask for input again.

The section 'Error Handling' above discusses trapping errors in programs in more detail.

ORIGIN
Syntax: ORIGIN <x expression> , <y expression>

This statement changes the coordinates of the graphics origin. <x expression> and <y expression> specify the new coordinates.

Example:
```
        ORIGIN xplace, yplace
```

OSCLI
Syntax: OSCLI <string expression> [TO <string array>
[, <variable>]]

The OSCLI statement is used to issue a command to the operating system on which the interpreter. <string expression> is the command. <string array> is an array that will be used to hold the output from the command. It is optional. If it is not present then the command output goes to the normal place. <variable> is set to the number of lines stored in <string array>. Again, it is optional.

The existing contents of <string array> are discarded before the output from the command is stored in it. Elements of the array that are not used are set to the empty string. The first element of the array used is 1, so the output is found in elements 1 to <variable>. If there is more output than will fit in the array the excess is discarded. There is nothing to indicate that this has happened so it is up to the user to ensure that the array is large enough.

Examples:
```
        F% = OPENIN filename$: size% = EXT#F%: CLOSE#F%
        DIM block% size%
        OSCLI "load "+filename$+" "+STR$~block%
        OSCLI <command> [ TO <string array> [ , <variable> ] ]

        OSCLI "ex" TO array$(), lines%
        FOR N%=1 TO lines%
          IF LEFT$(array$(N%), 1)="a" THEN PRINT array$(N%)
        NEXT
```

OTHERWISE
Syntax: OTHERWISE [: <statements>]

Part of a 'CASE' statement. When none of the cases given on WHEN statements match the expression on the CASE statement, the program continues at the statements after OTHERWISE. Refer to the section on the 'CASE' statement above for more details.

OVERLAY
Syntax: OVERLAY

This statement is not supported by the interpreter.

PLOT
Syntax: PLOT <code> , <x expression> , <y expression>

This statement carries out the graphics operation with code <code>. <x expression> and <y expression> are the two parameters that plot codes use.

Plot codes are at the heart of the graphics support. They carry out graphics operations such as drawing lines and shapes. Many of the more common operations have their own statements, for example, MOVE and DRAW. PLOT allows any of the codes to used in a program. The section 'Plot Codes' below gives more details.

Examples:
```
        PLOT 4, 100, 100
        PLOT 5, 500, 100
        PLOT 5, 500, 500
        PLOT 5, 100, 100
```

This plots a triangle. It could also be written as:

```
        MOVE 100, 100
        DRAW 500, 100
        DRAW 500, 500
        DRAW 100, 100
```

On the other hand, a filled triangle would have to be drawn using:

```
        MOVE 100, 100
        MOVE 500, 100
        PLOT 85, 500, 500
```

as there is no Basic statement to draw one.

POINT and POINT BY
Syntax: a) POINT <x expression> , <y expression>
b) POINT BY <x expression> , <y expression>

The POINT statement is used to plot a single point on the screen.

a) This form of the statement plots the point at coordinates (<x expression>, <y expression>) on the screen.

b) This versions of the statement plots the point at the position <x expression> and <y expression> graphics units in the X and Y direction relative to the current graphics cursor.

Examples:
```
        POINT 500, 100
        PONT BY xoffset%, yoffset%
```

POINT TO
Syntax: POINT TO <x expression> , <y expression>

This statement is similar to the MOUSE TO statement. Normally the pointer on screen is tied to the mouse but it does not have to be. POINT TO is used to move the pointer when it is not following the mouse. <x expression> and <y expression> give the x and y coordinates in graphics units to which the pointer is to be moved.

Example:
```
        POINT TO 500, 500
```

PRINT
Syntax: a) PRINT <list of expressions>
b) PRINT# <factor>, <list of expressions>

The first version of the PRINT statement displays data on screen and the second writes it to a file.

a) This version of the statement is used to display information on the screen. <list of expressions> is a list of items to be displayed. These are separated by blanks, ',' ';' or "'". In addition there are two functions specifically for controlling output.

Each item in <list of expression> can be an expression whose value is to be displayed or a print function. The print functions are:

TAB(<expression>)
This moves the text cursor to character position <expression> on the current line. If the text cursor is already beyond that column, move to the next line and skip to the required column.

TAB(<x expression>, <y expression>)
Move the text cursor to column <x expression>, row <y expression> on the screen.

SPC <expression>
Print <expression> blanks.

The rules describing how values are displayed are quite involved:

1) The print format of numbers is controlled by a special variable called '@%'. See the section below for details on the values that this can take.

2) Whilst @% affects the format used for a number, ';' and ',' affect the way in which it will be laid out.

   ',' causes two things: numbers will be printed right-justified occupying the number of characters set by field width in @%. Secondly, it causes the text cursor to be moved to the next column whose number is a multiple of the field width.

   ';' causes numbers to be printed left-justified. They occupy only as many characters as needed to express the number.

   By default, numbers are printed right-justified.

   If the first expression after the PRINT keyword is numeric and the result has to be printed left-justified, the expression has to be preceded with a ';' thus: 'PRINT ; abc%'.

3) If a numeric expression is preceded by a '~' then the value is printed in hexadecimal. In fact the '~' acts as a switch and and the results of all numeric expressions from this point to the next ';' or ',' will be printed in hexadecimal.

4) If an expression is separated from the previous one by a space then the second expression is formatted in same way as the previous one.

5) If an expression is separated from the previous one by a "'" the second expression is displayed on the next line. It will be formatted in the same way as the previous expression.

6) The functions TAB() and SPC do not affect how numbers will be formatted.

7) Strings are always printed left-justified and do not use the field width set by @%.

Examples:

```
PRINT abc def
PRINT ~pqr% xyz%
PRINT ghi$, "abcde", ~abc%
PRINT abc$ TAB(20) def
PRINT TAB(1,1) "a" TAB(2,2) "b" TAB(3,3) "c"
PRINT ; xyz%
PRINT CHR$13 ; SPC20 ; X%
PRINT "Line 1" ' "Line 2" ' "Line 3"
```

The Format Variable @%

This controls how numbers are formatted when they are printed. It can also affect the function STR$.

@% can best be thought of as being made up of four distinct byte-sized fields. They are laid out as follows:

<flags> <format> <places> <width>

with <flags> in the most significant byte of @% and <width> in the least signicant byte.

<flags> contains two flag bits:
&80 Use ',' as decimal point
&01 Format is used by the function STR$

<format> controls how numbers are output. The values it can take are:

0 General format
1 Exponent format
2 Floating point format

<places> gives the number of digits to print. The range is 1 to 255.

<width> is the field width, the number of characters that can be used when displaying a number. The range is zero to 255 characters. This affects how a number will be displayed as well as the number of characters skipped when using a ',' in a PRINT statement.

The default value of @% is &90A, that is, use general output format, display up to nine digits after the decimal point and use a field width of ten characters.

To make it easier to use, @% can be set using by assigning the desired format to it as a string.

@% = "<format string>"

<format string> is composed as follows:

[+] [<format>] [<width>] [. <places>]

Starting the string with a '+' indicates that the format is used to be used by the function STR$.

<format> gives the format to use:
G General format
E Exponent format
F Floating point format

<width> is the field width.

<places> gives the number of digits to print. If the format is F, it is the number of digits after the decimal point.

If ',' is used instead of '.' before the number of places, ',' is used as the decimal point character instead of '.'.

Note that all of these parts are optional. Only the parts of the format supplied in the string will be changed.

Examples:

```
@% = "G15.12"
@% = "E"
@% = "10"
@% = ","
@% = "+.10"
```

b) PRINT# writes data to the file with file handle <factor>. <list of expressions> is the data to be written. Note that the output is in binary, not text. It is designed to be read by the INPUT# statement.

Examples:

```
PRINT#file%, xyz, abc%(X%), "abcdefghij"
```

QUIT
Syntax: QUIT [<expression>]

The QUIT statement is used to exit from the interpreter. It canoptioanlly be followed by a numeric expression. This value is passed back to the operating system under which the program is running as a return or status code. If omitted it defaults to zero.

READ
Syntax: READ <list of variables>

The READ statement is used to read values supplied via DATA statements elsewhere in the program. <list of variables> is the list of variables to which the values will be assigned.

When reading the value of a numeric variable, the value in the data statement is treated as a Basic expression. The text on the DATA statement is read as far as the next comma or the end of the line and then evaluated. The result of the expression is the value assigned to the variable.

If the variable being read is a string variable, the text on the DATA statement is read as far as the next comma or the end of the line. Blanks preceding any text are ignored but trailing blanks are considered to be part of the string. The string can be enclosed in double quotes if need be.

Examples:

```
        abc% = 100
        READ xyz%, pqr%
        DATA abc%+10, 99
```

Here the expression 'abc%+10' is evaluated when xyz% is read and xyz% will be set to 110.

```
        READ abc$, def$
        DATA     aaa   , "   bbb   "
```

abc$ will be set to the string 'aaa   ' and def$ to '   bbb   '.

The RESTORE statement can be used to change the DATA statement from which data will be read. LOCAL DATA and RESTORE DATA can be used to save the current DATA statement pointer and to retrieve its value later.

RECTANGLE
Syntax: a) RECTANGLE <left expression> , <bottom expression>, <width expression> [, <height expression>]
        b) RECTANGLE FILL <left expression> , <bottom expression>, <width expression> [, <height expression>]
        c) RECTANGLE <left expression> , <bottom expression>, <width expression> [, <height expression>]
                TO <x expression> , <y expression>
        d) RECTANGLE FILL <left expression> , <bottom expression>, <width expression> [, <height expression>]
                TO <x expression> , <y expression>

The RECTANGLE statement has two uses, to draw a rectangle and to move or copy a rectangular portion of the screen to another part of the screen.

a) This form draws a box on the screen. The botton left-hand corner of the box is at the coordinate (<left expression> , <bottom expression>) and the width and height are given by <width expression> and <height expression>. <height expression> can be omitted, in which case the height is taken to be the same as the width.

b) This version draws a filled rectangle. The botton left-hand corner of the rectangle is at the coordinate (<left expression> , <bottom expression>) and the width and height are given by <width expression> and <height expression>. <height expression> can be omitted, in which case the height is taken to be the same as the width.

c) This statement copies an area of the screen. The botton left-hand corner of the area is given by the coordinate (<left expression> , <bottom expression>) and the width and height are given by <width expression> and <height expression>. <height expression> can be omitted, in which case the height is taken to be the same as the width. <x expression> and <y expression> are the coordinates of the bottom left-hand corner of the copied area.

d) This form moves an area of the screen. The botton left-hand corner of the area is given by the coordinate (<left expression> , <bottom expression>) and the width and height are given by <width expression> and <height expression>. <height expression> can be omitted, in which case the height is taken to be the same as the width. <x expression> and <y expression> are the coordinates of the bottom left-hand corner of the location to which the area will be moved. The old area is set to the background graphics colour.

Examples:

```
        RECTANGLE 400, 400, 500, 200
        RECTANGLE FILL 200, 200, width%, height%
        RECTANGLE FILL 200, 200, size%
        RECTANGLE left%, bottom%, with% TO 1000, 1000
        RECTANGLE FILL left%, bottom%, with% TO 1000, 1000
```

REM
Syntax: REM <comments>

The REM statement is used to include comments in a program. When a REM statement is encountered, the interpreter skips to the next line in the program.

Example:

```
        REM Here be dragons
```

REPEAT
Syntax: REPEAT

REPEAT marks the start of a REPEAT loop. The format of one of these is:

```
        REPEAT
          <statements>
        UNTIL <expression>
```

The block of statements <statements> is executed. The numeric expression <expression> is evaluated when the UNTIL is reached. If the result is zero (the value Basic used to represent false) the program jumps back to the start of <statements> and continues from that point again. If the result is not zero, execution continues with the statement after the UNTIL. In other words, <statements> will be executed repeatedly until the value of <expression> is not zero.

REPEAT loops can be nested to any depth.

It is possible to start the loop on the same line REPEAT, missing out the ':' statement separator.

The interpreter is not fussy about where the UNTIL is located, for example, the following sort of code is allowed:

```
        IF flag THEN UNTIL X%>10 ELSE UNTIL Y%>10
```

The interpreter silently ignores incorrectly nested loops of different types so that, for example, the following will not give an error:

```
        REPEAT
          FOR X% = 1 TO 10
        UNTIL Y% = 0
```

Note that the effect of any LOCAL DATA or LOCAL ERROR statements in the body of the loop will be reversed when the loop ends, that is, the data pointer and error handler details will be restored to the values they had when the LOCAL DATA or LOCAL ERROR was encountered.

REPORT
Syntax: REPORT

The REPORT statement displays the error message for the last error encountered in a program.

Example:

```
        DEF PROCerror
        PRINT"Program failed with error ";
        REPORT
        PRINT" at line ";ERL
        END
```

REPORT is used to print the error message. There is a function REPORT$ that returns it as a string then would be more convenient in code such as the example above. Other useful functions in this context are ERR and ERL which return the number of the last error and the line in which it occured.

RESTORE
Syntax: a) RESTORE [<line number>]
        b) RESTORE + <expression>
        c) RESTORE DATA
        d) RESTORE ERROR

The RESTORE statement has two uses: firstly, it is used to control which DATA statement is to be used to provide data for a READ statement and secondly it returns various internal pointers to values previous saved.

a) This version of the statement sets the data pointer to point at the DATA statement at or after line <line number>. The line number can be an expression. It can also be omitted, when the data pointer is reset to the first DATA statement in the program.

Example:

```
        RESTORE
        RESTORE 100
        IF flag THEN RESTORE start% ELSE RESTORE finish%
```

b) This form of the statement sets the data pointer to a line relative to the one in which the statement is found. <expression>

gives the number of lines to skip. It has to be a positive value greater than or equal to zero. The data pointer is moved to the DATA statement at or after the line indicated by the RESTORE statement.

```
        RESTORE +0
        RESTORE +10
        RESTORE +offset%
```

c) RESTORE DATA restores the data pointer to the value it had the last time a LOCAL DATA was executed. RESTORE DATA has to match up with a LOCAL DATA statement.

Note that RESTORE DATA is not always necessary as the data pointer will be automatically restored under some conditions, for example, if LOCAL DATA is used in a procedure, there will be an implicit RESTORE DATA when the procedure call ends. The data pointer is also restored automatically if LOCAL DATA is used inside a FOR, WHILE or REPEAT loop when the program leaves the loop.

Example:
```
        LOCAL DATA
        RESTORE 100
        READ X%
        RESTORE DATA
```

d) RESTORE ERROR restores the Basic error handler set up by ON ERROR or ON ERROR LOCAL to its condition when the last LOCAL ERROR statement was executed. RESTORE error has to match up with a LOCAL ERROR statement.

Note that RESTORE ERROR is not always necessary as the details of the error handler will be automatically restored under some conditions, for example, if LOCAL ERROR is used in a procedure, there will be an implicit RESTORE ERROR when the procedure call ends. The error handler details are also restored automatically if LOCAL ERROR is used inside a FOR, WHILE or REPEAT loop when the program leaves the loop.

Example:
```
        LOCAL ERROR
        ON ERROR LOCAL REPORT
        INPUT X%
        RESTORE ERROR
```

RETURN
Syntax: RETURN

The RETURN statement is used to return from a subroutine invoked via GOSUB to the statement after the GOSUB.

Example:
```
        IF X%=0 RETURN
```

RUN
Syntax: a) RUN [<line number>]
        b) RUN <string expression>

a) The first form of the RUN statement starts a program running. The line number <line number> is optional. If supplied, execution starts at that line in the program. If omitted, execution starts at the beginning of the program.

b) The second version of the RUN statement is identical to the CHAIN statement. The expression <string expression> provides the name of the program to run. If the program can be found it is loaded into memory and run. Any variables that existed at the time the RUN command (with the exception of the static integer variables, A% to Z%) are destroyed.

SOUND
Syntax: a) SOUND OFF
        b) SOUND ON
        c) SOUND <channel> , <amplitude> , <pitch> ,
                                <duration> , <delay>

This statement is used to make a sound. It is only supported under the RISC OS version of the interpreter.

a) SOUND OFF turns off the sound system.

b) SOUND ON turns on the sound system.

c) This version of the statement is used to make a sound.

STEP
Syntax: STEP <expression>

The STEP statement is part of a FOR statement. The expression <expression> gives the amount by which the FOR loop index variable is incremented (or decremented if negative) on each iteration of the FOR loop. Refer to the section on the FOR statement for more details.

Example:
```
        FOR N% = 10 TO 1 STEP -1: NEXT
        FOR X% = 1 TO 5 STEP 2: NEXT
```

STEREO
Unsupported statement for controlling the RISC OS sound system.

STOP
Syntax: STOP

The STOP statement ends the run of a program. It differs from END in that STOP is trapped as an error.

Example:
```
        IF bad THEN STOP
```

SWAP
Syntax: SWAP <variable 1> , <variable 2>

The SWAP statement exchanges the values of the two variables <variable 1> and <variable 2>. The variables have to be of the same type, that is, numeric variables can switch values and string variables can but types cannot be mixed.

Arrays can also be swapped but they have to be of exactly the same type, although the array dimensions can be different.

Examples:
```
        SWAP X%, Y%
        SWAP abc(X%), abc(X%+1)
        SWAP array1(), array2()
        DIM aaa(25), bbb(10,10,10): SWAP aaa(), bbb()
```

The last example demonstrates that arrays of different sizes and even with differing numbers of dimensions can be swapped.

SYS
Syntax: SYS <swi expression>, <expression 1>, ... ,
                <expression n> [TO <variable 1> , ... ,
                <variable n> [; <flag variable>]]

The SYS statement is only supported by the RISC OS version of the interpreter. It is used to issue a SWI (RISC OS operating system call) from a Basic program.

<swi expression> identified the SWI to call. This can be a number or a string giving the name of the SWI. <expression 1> to <expression n> are the parameters for the call. They can be numbers or strings. Basic strings are converted to null-terminated strings for the call. It is not possible to check that the types of the parameters supplied are correct for the SWI call so it is up to the programmer to ensure they are right.

Values can be returned from the SWI call. They are stored in the variables <variable 1> to <variable n>. The type of the variable is used to decide on the type of the value returned. Strings returned by the SWI are converted to Basic strings. The processor flags can also be returned by the call if <flag variable> is supplied. It is possible to discard values returned by leaving the position for that parameter after the 'TO' empty. The third example below illustrates this (the first two values returned by the SWI are not wanted).

Examples:
```
        SYS "OS_Write0", "abcdefgh"
        SYS 3
        SYS "OS_Byte", 134 TO , , row
```

TEMPO
Unsupported statement for controlling the RISC OS sound system.

THEN
Syntax: THEN

The THEN keyword is used in an IF statement. Refer to the section above on the IF statement for more information.

Example:
```
        IF X%=1 THEN X%=2
```

TINT
Syntax: TINT <expression> , <tint expression>

The TINT statement only has an effect in 256 colour screen modes. It sets the tint number <expression> to <tint expression>. <tint expression> can take the values 0, 64, 128 or 192.

There are four colours settings used by the interpreter as follows:

```
        Text foreground colour
        Text background colour
        Graphics foreground colour
        Graphics background colour
```

Each of these has a tint value associated with it that is used in 256 colour screen modes. This statement allows the tint values to be changed without having to change the colours as well. The values to use to identify the tint to change are as follows:

```
        0       Text foreground
        1       Text background
        2       Graphics foreground
        3       Graphics background
```

Example:
```
        TINT 0, 64
```

TO
Syntax: TO <expression>

The TO keyword is used in FOR loops to give the final value for the loop variable. Refer to the section on the FOR statement above for more information.

Example:
```
        FOR N% = 1 TO 10: NEXT
```

TRACE
Syntax: a) TRACE ON
```
        b) TRACE OFF
        c) TRACE TO <expression>
        d) TRACE CLOSE
        e) TRACE PROC
        f) TRACE GOTO
```

The TRACE statement is used to help debug Basic programs. It controls the various trace options possible.

a) This form turns on the line number trace. This lists the number of each line executed.

b) This turns off all the traces.

c) This sends the trace output to the file named by the string expression <expression>. Normally output goes to the screen but it is directed to a file if this statement is used.

d) This closes the trace file. Trace output continues on the screen.

e) TRACE PROC displays the names of each procedure or function as it is entered and left.

```
===>PROC<name>    indicates that the procedure has been entered.
PROC<name>--->    indicates that the procedure has been left.
```

The trace can be turned off with the statement:

```
        TRACE PROC OFF
```

f) TRACE GOTO lists the origin and destination line numbers every time a branch occurs in a program, for example, whenever a procedure or function is called or there is a jump to the top of a loop. It can be used to see the path through the program in a more concise form than just displaying the number of every line executed. The output from the trace is of the form:
```
        [<line number 1>-><line number 2>]
```

This says that a branch has occured from <line number 1> to <line number 2>.

This trace can be turned off with the statement:

```
        TRACE GOTO OFF
```

Sending Trace Output to a File

'TRACE TO' is used to send trace output to a file. The handle of the file can be found by using TRACE as a function. If it is not zero, output is going to the file with this handle. It is possible to put extra data into the trace file, for example:

```
        IF TRACE THEN BPUT#TRACE,"initialisation finished"
```

TRUE
Syntax: TRUE

The TRUE keyword returns the value Basic uses to represent 'true' (-1).

Example:
```
        flag = TRUE
```

UNTIL
Syntax: UNTIL <expression>

The UNTIL keyword forms part of a 'REPEAT UNTIL' loop. Refer to the section on the REPEAT statement for more information.

Example:
```
        REPEAT
          X%+=1
        UNTIL X%=10
```

VDU
Syntax: VDU <expression 1> , ... , <expression n>

The VDU command sends output to the screen. The numeric expressions <expression 1> to <expression n> are evaluated one by one and the result sent to the screen driver. The main purpose of this command is to send screen control codes (the so-called 'VDU commands').

The amount of data sent for each expression depends on what follows the expression. If there is a comma after it or it is the last expression in the statement, one byte of data (the least significant byte of the result) is sent. If there is a semicolon after it, two bytes of data are sent, the two least significant bytes of the result. The least significant byte is sent first, for example:

```
        VDU 31, 10, 10
```

This VDU command sends three bytes of data to the screen driver.

```
        VDU 29, 640; 512;
```

This example sends five bytes, 29 as one byte, 640 as two bytes as it is followed by a ';' and 512 as two bytes.

If a '|' is used instead of a comma or semicolon, nine zeroes are sent. This is useful when using the VDU 23 commands as nine bytes of data has to be provided for these when the actual command might need only one or two bytes. The extra zeroes are ignore or treated as no-ops.

Examples:
```
        VDU 31, 1, 1, ASC"a", 31, 2, 2, ASC"b"
```

The VDU statement allows any VDU command to be issued but the most common ones are more conveniently handled by Basic statements, for example, VDU 17 can be used to change the colour used for displaying text but the COLOUR statement is easier.

VOICE
Unsupported statement for controlling the RISC OS sound system.

VOICES
Unsupported statement for controlling the RISC OS sound system.

WAIT
Syntax: a) WAIT
```
        b) WAIT <expression>
```

a) This version of the statement causes the program to be halted until the next vsync interrupt. It is used to synchronise drawing output on the screen with the hardware to make the output look neater.

Example:
```
        WAIT
```

b) This version is used to make the program wait for a period of time. <expression> is a numeric expression that gives the time for which the program will pause in centiseconds. Pressing the 'escape' key (either 'Esc' or control C, depending on the environment) will cause the program to resume execution.

Example:
```
        WAIT 500
```

This will make the program pause for five seconds.

WHEN
Syntax: WHEN <expression> [,<expression>] :

Part of a 'CASE' statement. Refer to the section on the 'CASE' statement above for more details.

WHILE
Syntax: WHILE <expression>

The WHILE statement marks the start of a WHILE loop. The format of one of these is:

```
WHILE <expression>
<statements>
ENDWHILE
```

The numeric expression <expression> is evaluated and if it is not zero the loop entered. The program then continues until it comes across an ENDWHILE. This is the end of the loop. The expression <expression> is evaluated again and as long as the result is not equal to zero the program branches back to the first statement after the WHILE statement.

Note that the first ENDWHILE found is considered to be the end of the loop. It is possible for different ENDWHILEs to be used in the same loop, for example, it is possible to write code such as:

```
WHILE X%<10
  X%+=1
IF X%<5 THEN ENDWHILE ELSE ENDWHILE
```

If the expression <expression> is zero the first time the WHILE statement is executed, the loop is skipped. Note that the interpreter searches for the first ENDWHILE it can find and assumes that that is the end of the loop. This could give a problem if code like the example above is used.

The interpreter allows loops to be nested incorrectly and silently ignores this sort of error, for example:

```
WHILE X%<10
  X%+=1
  REPEAT
    Y%+=1
ENDWHILE
```

The interpreter will not complain about the missing 'UNTIL' part of the nested REPEAT loop.

Note also that the effects of any LOCAL DATA or LOCAL ERROR statements encountered in the loop will be reversed when the program leaves the WHILE loop.

WIDTH
Syntax: WIDTH <expression>

This statement sets the number of characters printed on each line by the PRINT statement to <expression> characters. It automatically wraps round to the next line when that number is reached. The default is the width of the screen or text window.

This version of the interpreter ignores WIDTH.

Example:

```
WIDTH 40
```

Commands

Commands can only be entered on the command line with the exceptions of LIST and LVAR which can be used in programs as well. As with keywords, commands can often be abbreviated by typing the first few characters of the command name and following that with a dot, for example, 'l.' is the abbreviated form of the 'list' command. The section 'Basic Keywords, Commands and Functions' at the end of these notes gives the abbreviated versions of each command.

On the command line, commands can be entered in mixed case (as opposed to keywords, which have to be in upper case). If in a program they are treated as normal keywords and have to be in upper case. This can give rise to unexpected results, for example the program:

```
10 list = 1.23
20 PRINT list
```

will print the number 1.23, but if:

```
list = 1.23
```

is entered on the command line it gives a syntax error as 'list' is seen as a command. 'list' is in lower case in the program and therefore it is not identified as the 'list' command. It is a variable called 'list' in the program. It is best to avoid the use of commands as variable names in programs where possible.

The commands available are:

APPEND
Not implemented.

AUTO
Not implemented.

CRUNCH
Ignored.

DELETE
Syntax: DELETE <line number 1> [, <line number 2>]

The DELETE command is used to delete a single line or a range of lines from a Basic program. If a single line number is given then only that line is removed. If two line numbers are provided then all lines in that range are erased. It is not possible to retrieve lines that have been deleted in error.

Example:

```
DELETE 100
```

EDIT
Syntax: EDIT [<line number>]

If a line number is supplied after the EDIT command, that line is transfered to the command line where it can be edited.

If no line number is given, the program in memory is transferred to a text editor where it can be more easily edited. When the program is saved, the interpreter loads it back into memory.

The program is transferred to the editor using the current LISTO settings to format it. This is not always the best way to do it so there is a variation on the EDIT command, EDITO that can be used instead.

Examples:

```
EDIT 100
EDIT
```

EDITO
Syntax: EDITO <expression>

EDITO is a variation of the EDIT command. It allows the format of the program to be controlled when it is transferred to the editor. <expression> is a numeric expression that gives the 'LISTO' value to be used to format the program. As with EDIT, the program is loaded back into memory when the program is saved in the editor and the editor exited from.

The expression <expression> controls the format. The LISTO values useful in this context are:

```
1  Insert a space after the line number.
2  Indent program structures such as block IF statements.
8  Omit line numbers.
```

The LISTO value is obtained by adding together these numbers. If <expression> is 10, the program is transferred to the editor without line numbers and with structures indented.

Example:

```
EDITO 8
```

HELP
Syntax: HELP

HELP displays some information on the program in memory and LISTO and TRACE debug options.

INSTALL
Syntax: INSTALL <expression 1>, <expression 2> , ... , <expression n>

This command loads the Basic libraries named by the string expressions <expression 1> to <expression n> into memory. If only the names of the libraries are supplied, that is, the directory in which they live are not given explicitly, the interpreter searches for them in the directories given by the pseudo-variable FILEPATH$.

The libraries are permanently loaded into memory, unlike libraries that are loaded via the LIBRARY statement which are discarded under some circumstances. It is not possible to unload these libraries. On the other hand the same library can be loaded via LIBRARY and it will be searched before the version loaded using INSTALL.

The library search order is:

1) Search libraries loaded via LIBRARY in the reverse order to that in which they were loaded.
2) Search libraries loaded via INSTALL in the reverse order to that in which they were loaded.

Note that libraries are seen as an extension to the program in memory rather than separate entities. They can have their own private variables (declared using LIBRARY LOCAL) but any other variables created in procedures and functions in the library will be added to those the program creates.

Example:
```
        INSTALL "proclib"
```

LIST
Syntax: LIST [<line number 1> [, <line number 2>]]

The LIST command lists lines in the Basic program. If no line number is given then the whole program is listed. If one line number is supplied then just that line is displayed. If two line numbers are given then all lines in that range are shown.

The lines are formatted according to the current setting of LISTO. One useful option when listing a large number of lines is LISTO 32, which causes the interpreter to pause when twenty lines have been listed until either the space bar or return is pressed.

The line numbers can be given by expressions and, unlike other commands, LIST can be used in a running program.

Examples:
```
        LIST
        LIST 100,200

        DEF PROCerror
        PRINT REPORT$;" at line ";ERL
        LIST ERL
        ENDPROC
```

LISTIF
Syntax: LISTIF <search string>

The LISTIF command provides a simple string search facility. <search string> is the string to look for. Every line where the string is found is listed.

There are no wild card facilities.

Example:
```
        LISTIF PROCerror
        LISTIF abc%
```

LISTO
Syntax: LISTO <expression>

LISTO is primarily used to control how programs are formatted when they are listed. It also provides the default format used when transferring a program to a text editor or when saving a program.

<expression> gives the new LISTO value. This is formed by adding together one or more of the following values:

```
        1  Insert a space after the line number.
        2  Indent program structures such as block IF statements.
        8  Omit the line number.
        16 List keywords in lower case.
        32 Pause after displaying twenty lines when listing program.
```

The LISTO value is obtained by combining these, for example, indent structures, omit line numbers and list keywords in lower case would be 2+8+16 = 26.

As noted above, the LISTO value is used by default when transfering programs to an editor or saving them. This might not always be the best format so the commands affected, EDIT and SAVE, have variations where the LISTO value to be used for that command invocation is given.

Really, the LISTO values are bit settings and the LISTO value is obtained by OR'ing them together.

Examples:
```
        LISTO 26
        LISTO (2+8+16)
        LISTO (LISTO+1)
```

LOAD
Syntax: LOAD <string expression>

The LOAD command is used to load a program into memory. Any existing program and any variables created are discarded before the new program is read.

If the name is just the name of the file, that is, it does not contain any directory names, the interpreter looks first in the current directory and the searches through the directories listed by the pseudo-variable FILEPATH$ to find the program.

The name of the file is recorded and will be used as the name to use by default by the SAVE command when the program is saved. Note that the name (as shown by 'HELP') will normally include the name of the directory in which the program was found.

Example:
```
        LOAD "abc"
        LOAD FNget_name
```

LVAR
Syntax: LVAR [<letter>]

The LVAR command is used to list the variables, procedures and functions that have been created or called in the program. If a letter is supplied then only items whose name starts with that letter are listed. Note that only a single letter can be given: it is not possible to give a pattern for the names to be listed.

Variables are listed with their current values. The dimensions of arrays are given but not the values of the elements. The types of the parameter of procedures and functions are given. Some entries for procedures and functions give the line where they were found and no parameters. These are procedures and functions that the interpreter has found in the program but that have not been called yet.

Examples:
```
        LVAR
        LVAR x
```

NEW
Syntax: NEW [<expression>]

The NEW command has two purposes. Its main use is to discard the program in memory and any variables that have been created. If the keyword NEW is followed by an expression, it not only does this but changes the size of the Basic workspace to <expression> bytes.

The size of the Basic workspace is fixed. It can be specified when the interpreter starts via the command line option '-size'. The only way to change it when the interpreter is running is via the NEW command.

Libraries loaded via INSTALL are retained over the change of workspace size but the program in memory is lost and cannot be recovered.

Example:
```
        NEW
        NEW 1000000
```

OLD
Syntax: OLD

The OLD command is used to attempt to recover a program in memory. It is of limited use. It only stands a chance of working after a plain NEW command (that is, one that does not change the workspace size). If it appears that there is a program in memory it carries out a number of checks to see if the program is intact. If so it sets up the program again so that it can be run. If any errors are detected the program is marked as a 'bad program'. It cannot be run or edited. The checks carried out are not foolproof and it is possible that a corrupted program will pass the tests.

Example:
```
        OLD
```

RENUMBER
Syntax: RENUMBER [<start expression> [, <step expression>]]

The renumber command renumbers the lines of a program in memory. <start expression> and <step expression> give the line number to use for the first line and the increment from line to line. One or both can be omitted, in which case they both default to ten.

<start expression> is in the range zero to 65279.
<step expression> is in the range one to 65279.

The renumber command will renumber the lines of the program and any explicit references to line numbers on GOTO, GOSUB, ON GOTO, ON GOSUB and RESTORE statements. Line numbers which are given by expressions cannot be dealt with. Any line number references that refer to lines in the program that are missing are left unchanged and a warning message put out. If the highest line number allowed (65279) is exceeded the command renumbers the program again with a start value of one and a step of one.

Examples:
```
        RENUMBER
        RENUMBER 1000
        RENUMBER 100,1
```

SAVE
Syntax: SAVE [<string expression>]

The SAVE command saves the program in memory in the file named by <string expression>. The file is saved as plain text so that it can be edited with any text editor.

<string expression> can be omitted. If this happens, the command looks for a file name in two places:

1) It checks the first line of the program. If there is a '>' that is followed by some text, that text is used as the name of the file. (This follows an Acorn convention where the first line of a file gives its name).

2) If the first check fails, the name saved from the last 'LOAD' or 'SAVE' command is used.

An error is reported if a name cannot be found.

The program is saved in text form using the current LISTO settings to format it. If this is not what is desired, the SAVEO command can be used.

The name <string expression> is saved for use by later SAVE commands if it is supplied.

Examples:
```
        SAVE "abcdefgh"
        SAVE A$+"/abc"
        SAVE
```

SAVEO
Syntax: SAVEO <expression> [, <string expression>]

The SAVEO command is just like the SAVE command. It saves the program in memory in the file named <string expression>. The difference between it and SAVE is that the LISTO value to be used to format the program when writing it to the file is specified by <expression>. The LISTO values useful in this context are:

```
        1  Insert a space after the line number.
        2  Indent program structures such as block IF statements.
        8  Omit line numbers.
```

The LISTO value is obtained by adding together these numbers. If <expression> is 10, the program is saved without line numbers and with structures indented.

As with SAVE, <string expression> can be omitted. If this happens, the command looks for a file name in two places:

1) It checks the first line of the program. If there is a '>' that is followed by some text, that text is used as the name of the file. (This follows an Acorn convention where the first line of a file gives its name).

2) If the first check fails, the name saved from the last 'LOAD' or 'SAVE' command is used.

An error is reported if a name cannot be found.

The name <string expression> is saved for use by later SAVE commands if it is supplied.

Examples:
```
        SAVEO 10 , "abcdefgh"
        SAVEO 3
```

TEXTLOAD
Syntax: TEXTLOAD <string expression>

This command is a synonym for the LOAD command in this interpreter. Refer to the section on the LOAD command above for more details.

TEXTSAVE
Syntax: TEXTSAVE [<string expression>]

This command is a synonym for the SAVE command in this interpreter. Refer to the section on the SAVE command above for more details.

TEXTSAVEO
Syntax: TEXTSAVEO <expression> [, <string expression>]

This command is a synonym for the SAVEO command in this interpreter. Refer to the section on the SAVEO command above for more details.

TWIN
Syntax: TWIN

This command is a synonym for the EDIT command in this interpreter. Refer to the section on the EDIT command above for more details.

TWINO
Syntax: TWINO <expression>

This command is a synonym for the EDITO command in this interpreter. Refer to the section on the EDITO command above for more details.

The Program Environment

The interpreter provides a partial emulation of the operating system under which the Acorn interpreter runs, RISC OS. The emulation is limited to supporting the control codes used for screen output and the file handling used by the program. It is not an attempt to write a RISC OS emulation layer.

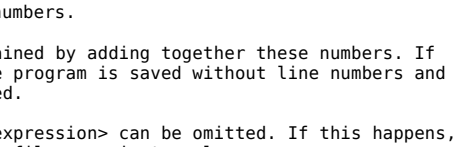
Screen Output

The interpreter emulates the RISC OS VDU drivers for screen output so some details of how these work and the facilities available follow.

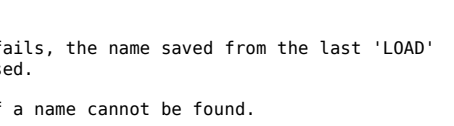
VDU Driver

The VDU driver is the portion of RISC OS that handles screen output. RISC OS mixes text and graphics on the same screen. The screen size can be measured in text coordinates or graphics coordinates.

Text coordinates have their origin at the top left-hand corner of the screen and extend to the left and downwards:



Graphics coordinates have their origin at the bottom left-hand corner of the screen and extend to the left and upwards:



The values in the diagrams are just examples.

Text coordinates are just character positions on screen. The range of the coordinates varies according the screen resolution.

Graphics coordinates are more complex. Going back to the days of the BBC Micro, the range of the coordinates was fixed at 0 to 1279 in the X direction and 0 to 1023 in the Y direction. This was independent of the screen resolution; the operating system mapped the graphics on to it. Later the mapping was changed when the range of screen resolutions available was increased and larger

sizes became available. Up to a resolution of 640 by 512 pixels, the graphics coordinates are in the range given earlier (with a couple of exceptions). For sizes larger than that, the general rule is that the graphics coordinate range is twice that of the pixels, for example, the graphics coordinate range for a screen resolution of 800 by 600 is 1600 by 1200 (0 to 1599 and 0 to 1199).

Graphics coordinates in the X and Y direction are in the range -32768 to 32767.

The screen resolution and number of colours can be set using the 'MODE' command. This can be either a number that identifies the mode to use or a string that gives its details.

Colours

RISC OS supports colour depths of 1, 2, 4, 8, 15 and 24 bits per pixel. The interpreter support 1, 2, 4 and 8 bit. The way colours are dealt with in 2 colour (1 bit), 4 colour (2 bit) and 16 colour (4 bit) modes is straight forwards but manipulating colours in 256 colour modes is slightly awkward due to the way the colour system was originally designed back in the days of the BBC Micro. (The old system was probably kept for compatibility.)

2, 4 and 16 Colour Modes

There are logical colours and physical colours. The colour corresponding to each logical colour can be changed by means of VDU 19. The number corresponding to each physical colour is as follows:

```
0 (8)    Black
1 (9)    Red
2 (10)   Green
3 (11)   Yellow
4 (12)   Blue
5 (13)   Magenta
6 (14)   Cyan
7 (15)   White
```

(Note: in this program colour numbers 8 to 15 simply repeat colours 0 to 7. Under RISC OS they give flashing colours.)

256 Colour Modes

In 256 colour modes the colour is broken into two components, the colour and a 'tint' value used to modify it. The colour portion is six bits and the tint two bits. The colours in these modes are constructed from sixty three colours, each of which has four brightness levels. Unlike the other colour depths, the 256 colour palette is essentially fixed and the colour number made up of the colour and tint determines which of these fixed colours to used.

The tint value affects the brightness of the colours. It changes the level of all three (red, green and blue) colours components at the same time. It has four values: 0, 1, 2 and 3 but these are written as 0, 64, 128 and 192.

The VDU drivers works in terms of the following colours when writing text or plotting graphics:

```
Text foreground colour
Text background colour
Graphics foreground colour
Graphics background colour
```

Text and Graphics Windows

It is possible to define text and graphics windows so that output goes to a restricted portion of the screen. Effectively the windows by default occupy the entire screen but they can be changed at any time.

Text-only Modes

Three of the screen modes, modes 3, 6 and 7, do not support graphics. There are a hangover from the days of the BBC Micro.

VDU Commands

The VDU commands are the escape sequences that the interpreter uses to control output, for example, to move the cursor to a specific location on the screen. It is customary to refer to these as 'VDU commands'.

The ASCII character codes in the range 0 to 31 are used. The format of a VDU code is a single command byte followed by up to nine bytes of data. The complete list follows:

```
0        Does nothing
1        Send next character to the printer
2        Enable sending of characters to the printer
3        Disable sending sending of character to the printer
4        Display text at the text cursor position
5        Display text at the graphics cursor position
6        Enable the VDU drivers
7        Sound the console bell
8        Move the cursor back one character
9        Move the cursor forwards one character
10       Move the cursor down one line
11       Move the cursor up one line
12       Clear the text screen
13       Move the cursor to the start of the line
14       Enable page mode
15       Disable page mode
16       Clear the graphics window
17       Change the current text colou
18       Change the current graphics colour
19       Map logical colour to physical colour
20       Reset logical colours to default values
21       Disable the VDU driver
22       Change the screen mode
23       Various commands
24       Define the graphics window
25       Issue a graphics command
26       Restore default text and graphic windows
27       Do nothing
28       Define the text window
29       Define the graphics origin
30       Send cursor to top left-hand corner of window
31       Send cursor to given position in window
```

The interpreter does not support all of these. The following table says what works where. The column marked 'Text' says whether the code is supported (yes), flagged as an error (no) or ignored (ignore) in text-only versions of the program. The column 'Graphics' gives the same information for versions that support graphics.

Code	Text	Graphics
0	yes	yes
1	yes	yes
2	ignored	ignored
3	ignored	ignored
4	ignored	ignored
5	no	yes
6	ignored	ignored
7	yes	yes
8	yes	yes
9	yes	yes
10	yes	yes
11	yes	yes
12	yes	yes
13	yes	yes
14	ignored	ignored
15	ignored	ignored
16	no	yes
17	yes	yes
18	no	yes
19	yes	yes
20	yes	yes
21	ignored	ignored
22	yes	yes
23	yes	yes
24	no	yes
25	no	yes
26	yes	yes
27	yes	yes
28	yes	yes
29	no	yes
30	yes	yes
31	yes	yes

Details of the supported commands follow.

VDU 0

Does nothing.

VDU 1
Under RISC OS this sends the next character to the printer stream. Under other operating systems it sends the next character uninterpreted to the output stream, that is, it can be used to output values that would normally be treated as VDU commands to the output stream.

VDU 4
Display text at the text cursor position.

VDU 5
Display text at the graphics cursor position. Text is sent to the current graphics cursor position with the graphic cursor defining the position of the top left-hand corner of the character. The graphics cursor position in the X direction is updated by the width of a character expressed in graphics units. If the edge of the window is reached output continues on the next line, that is, the graphics Y coordinate is decreased by the height of a character expressed in graphics units and the X coordinate is reset to the edge of the graphics window.

Note that text written in this mode overwrites what is currently on the screen without clearing the background of each character.

VDU 7
Sound the console bell.

VDU 8
Move the cursor back one character. The text cursor is moved back one character, moving up a line and wrapping around to the right-hand edge of the text window if it is at the left-hand edge of the window. If the cursor is at the top left-hand corner of the window it wraps around to the end of the line and scrolls the window down by one line.

If text output is being sent to the graphics cursor the actions are basically the same but they affect the graphics cursor instead and leave the text cursor unchanged. The only difference is that the cursor wraps around to the bottom right-hand corner of the window if it was originally positioned in the top left-hand corner instead of scrolling the screen. Changes in the X and Y coordinates are by the width and height of a character expressed in graphics units respectively.

VDU 9
Move the cursor forwards one character. The actions performed follow the same pattern as VDU 8. The text cursor is moved forwards by one character, moving down a line and wrapping around to the left-hand edge of the text window if it was originally positioned at the right-hand edge of the window. If it was originally positioned at the bottom right-hand corner of the window it is move to the left-hand edge and the window is scrolled up one line.

If text output is being sent to the graphics cursor the actions afffect the graphics cursor instead of the text cursor. As in the case of VDU 8, the only difference is when the graphics cursor is at the bottom right-hand corner of the graphics window. It wraps around to the top left-hand corner.

VDU 10
Move the cursor down one line. The text cursor is moved down the window by one line. If it was originally on the bottom line of the text window it remains on that line and the window is scrolled upwards by one line. If text is being sent to the graphics cursor, the Y coordinate of the graphic cursor position is decreased by the height of a character expressed in graphics units. If the graphics cursor was originally at the bottom of the window it wraps around to the top.

VDU 11
Move the cursor up one line. Again this follows the same pattern as VDU 10. The text cursor is moved up by one line unless it was originally positioned on the top line of the window, in which case it is left there and the window is scrolled down by one line.

If output is going to the graphics cursor, the Y coordinate of the cursor is increased by the height of a character expressed in graphics units unless the cursor was located at the top of the window, in which case it is moved to the bottom.

VDU 12
Clear the text window. The text window is set to the text background colour.

VDU 13
Move the cursor to the start of the line. The cursor is moved to the left-hand edge of the window on the current line. If output is being sent to the graphics cursor it is the graphics cursor that is moved otherwise it is the text one.

VDU 16
Clear the graphics window. The entire graphics window is set to the current graphics background colour.

VDU 17
Change the current text colour. This is followed by one byte of data, the new colour number. If this is less than 128 the foreground colour is changed, otherwise the background colour is the one altered, the number of the colour to use being given by the colour number-128.

The colour number is reduced modulo the number of colours available in the current screen mode in 2, 4 and 16 colour modes. In 256 colour modes it is reduced modulo 64. This alters the colour value but not the tint (see the section '256 Colour Modes' above for an explanation of this).

VDU 18
Change the current graphics colour. This is followed by two bytes of data:

Byte	1	Plot action
	2	Colour number

The 'plot action' controls how graphics are plotted on the screen. Currently the only value supported here is zero, causing each point to overwrite what was previously at that place on the screen.

If the colour number is less than 128 the foreground graphics colour is changed, otherwise the background colour is updated. 128 is subtracted from the colour number to get the number of the colour to use. The colour number is then reduced modulo the number of colours available in the current screen mode in 2, 4 and 16 colour modes. In 256 colour modes it is reduced modulo 64 to change the colour value. It does not affect the tint (see the section 256 Colour Modes' above for more information on this).

VDU 19
Map logical colour to physical colour.

This requires five bytes of data after the command as follows:

Byte	1	Logical colour number
	2	Physical colour number
	3	Red component
	4	Green component
	5	Blue component

The logical colour number is reduced modulo the number of colours available in the current screen mode in 2, 4 and 16 colour modes.

If the physical colour number is in the range zero to fifteen the physical colour corresponding to the given logical colour is changed to the value supplied. The red green and blue colour components are ignored. This version of the VDU command only has an effect in 2, 4 and 16 colour modes.

If the physical colour number is sixteen, the colour for the given logical colour number is changed to the one with the red, green and blue components supplied in bytes 3, 4 and 5.

Physical colour numbers greater than sixteen are ignored in versions of the interpreter that do not run under RISC OS.

VDU 20
Reset logical colours to default values.

VDU 22
Change the screen mode. This takes one byte of data, the new screen mode. It causes the text and graphics windows to be set to cover the whole screen, resets the colour palette to its default for that mode and moves the text and graphics cursors to their respective origins.

VDU 23
VDU 23 has two purposes. Firstly, it allows any of the characters with codes in the range 32 to 255 to be redefined and, secondly, it is the code used for a range of commands that affect the operation of the VDU drivers.

VDU 23 requires nine bytes of data. The format is:

Byte	1	Command or character code
	2 to 9	Command or character data

If the first byte, the command byte, is in the range 32 to 255 VDU 23 is being used to define the layout of the character with that code. The eight bytes of data that follow contain the image for

that character. If the command byte is in the range 0 to 31 then it is a command for the VDU driver. Eight bytes of data must be supplied but the command might ignore them.

Only two of the VDU 23 commands are supported:

```
        23,1     Hide or show the text cursor.
        23,17    Set the 'tint' value in 256 colour screen modes.
```

23,1

The two values defined are:

```
        23,1,0   This removes the text cursor
        23,1,1   This displays the text cursor
```

23,17

The format of this command is:

```
        23,17,<option>,<tint>
```

where:

```
<option> says what to set:
        0        Set foreground text tint value
        1        Set background text tint value
        2        Set foreground graphics tint value
        3        Set background graphics tint value
        5        Swap foreground and background text colours
```

<tint> sets the new tint value. Four values are allowed: 0, 64, 128 and 192.

VDU 24
Define the graphics window.

This requires eight bytes of data after the command as follows:

```
Bytes 1 and 2     x coordinate of left-hand side of window.
Bytes 3 and 4     y coordinate of bottom of window.
Bytes 5 and 6     x coordinate of right-hand side of window.
Bytes 7 and 8     y coordinate of top of window.
```

Each pair of bytes is treated as a two byte signed integer with the least significant byte sent first. The coordinates are given relative to the current screen origin. The command is ignored if any of the coordinates lie outside the screen.

VDU 25
Issue a graphics command. Any graphics operation such as drawing a line or plotting a circle goes can be carried out using VDU 25. The code is followed by five parameter bytes as follows:

```
Byte     1        Graphics PLOT code
         2        Low byte of first parameter
         3        High byte of first parameter
         4        Low byte of second parameter
         5        High byte of second parameter
```

See the section 'PLOT Codes' below for details of the graphics commands allowed.

VDU 26
Restore default text and graphic windows. The text and graphics windows are set to full screen. The text cursor is moved to the top left-hand corner of the screen and the current graphics position set to (0, 0), which is at the bottom left-hand corner of the screen. The graphics origin is also reset to this position.

VDU 27
Does nothing.

VDU 28
Define the text window.

This command requires four bytes of data as follows:

```
Byte     1        column number of left-hand side of window.
         2        Row number of bottom of window.
         3        Column number of right-hand side of window.
         4        Row number of top of window.
```

If any of the coordinates are off the screen the command is ignored.

VDU 29
Define the graphics origin.

This requires four bytes of data after the command as follows:

```
Bytes 1 and 2     x coordinate of graphics origin.
Bytes 3 and 4     y coordinate of graphics origin.
```

The pair of bytes is treated as a two byte signed integer with the low-order byte send first. The coordinates are absolute, that is, they are not specified relative to the current graphics origin.

VDU 30
Send cursor to top left-hand corner of window. The text cursor (or the graphics cursor if text is being plotted at the graphics cursor) is sent to the top left-hand corner of the window.

VDU 31
Send cursor to given position in window.

This requires two bytes of data after the command:

```
Byte     1        Column number
         2        Row number
```

The text cursor (or the graphics cursor if text is being plotted at the graphics cursor) is sent to the position corresponding to text coordinate (column, row) on the screen.

PLOT Codes

Plot codes form the heart of the graphics support. Normally use of these is hidden behind Basic statements such as 'MOVE', 'DRAW' and 'CIRCLE' but they can be used directly in programs by means of the 'PLOT' statement and VDU 25.

The codes are in the range 0 to 255. Each code can really be thought of as containing three fields which specify the operation, how the graphics coordinates are to be interpreted and how to treat the graphics colours. The graphic operation is given by (plot code DIV 8). How to treat the coordinates and colours is given by (plot code MOD 8). It is easier to understand plot codes if they are written in binary. They can be expressed in this way as:

```
        xxxxx y zz
```

where 'xxxxx' is the graphics operation, 'y' says how to treat the coordinates and 'zz' how to handle the colours. 'y' and 'zz' are defined as follows:

```
'y' values:
        0        Treat coordinate as relative
        1        Treat coordinate as absolute

'zz' values:
        0        Only move the graphics cursor.
        1        Plot graphics using graphics foreground colour.
        2        Plot graphics using logical inverse of colour at
                 each point.
        3        Plot graphics using graphics background colour.
```

The interpreter does not support the full range of PLOT codes. The ones implemented are:

```
          0-  7  Draw a solid line
         64- 71  Plot a single point
         80- 87  Draw a filled triangle
         96-103 Draw a filled rectangle
        112-119  Draw a filled parallelogram
        128-135  Flood fill background
        144-151 Plot a circle outline
        152-159 Draw a filled circle
        184-191 Move or copy a rectangle
        192-199  Draw an ellipse outline
        200-207 Plot a filled ellipse
```

Typical plot codes used are 4 (move graphics cursor), 5 (draw a line) and 69 (plot a point).

Mode Variables

The VDU function can be used to find out information about the current screen mode being used, for example, the width of the screen in characters or in pixels. It is used as in the following example:

```
        width% = VDU 1
```

```
        PRINT"Screen width in characters is "; width%

The allowed values and what they return are as follows:

  0  ModeFlags             0 if mode is capable of graphics
                           1 if mode is text only.
  1  ScrRCol               Width of screen in characters - 1
  2  ScrBRow               Height of screen in characters - 1
  3  NColour               Number of colours allowed - 1
 11  XWindLimit   Width of screen in pixels - 1
 12  YWindLimit   Height of screen in pixels - 1
128  GWLCol                Graphics window left limit in pixels
129  GWBRow                Graphics window bottom limit in pixels
130  GWRCol                Graphics window right limit in pixels
131  GWTRow                Graphics window top limit in pixels
132  TWLCol                Text window left limit in characters
133  TWBRow                Text window bottom limit in characters
134  TWRCol                Text window right limit in characters
135  TWTRow                Text window top limit in characters
136  OrgX                  Graphics X origin in RISC OS graphics
units
137  OrgY                  Graphics Y origin in RISC OS graphics
units
153  GFCOL                 Graphics foreground colour
154  GBCOL                 Graphics background colour
155  TForeCol              Text foreground colour
156  TBackCol              Text background colour
157  GFTint                Graphics foreground tint value
158  GBTint                Graphics background tint value
159  TFTint                Text foreground tint value
160  TBTint                Text background tint value
161  MaxMode               Highest screen mode number supported

The names in the second columns are what the mode variables are
called in RISC OS documentation.

If the value passed to the function is not one of the above,
the function returns 0.

The mode variables that return graphics information will return
zero in versions of the program that do not support graphics.

The RISC OS version of the program supports the full range of
mode variables. The ones given above represent only a selection
of them. The complete list can be found on pages 703 to 705 and
710 and 711 of volume 1 of the RISC OS Programmer's Reference
Manual.
```

Basic Keywords, Commands and Functions

Basic Keywords

This is a list of all of the Basic keywords and their minimum abbreviations. The '.' after each abbreviated name is considered to be part of the name.

Keyword	Abbreviation	Keyword	Abbreviation
AND	A.	BEATS	BE.
BPUT	BP.	CALL	CA.
CASE	CAS.	CHAIN	CH.
CIRCLE	CI.	CLG	CLG
CLEAR	CL.	CLOSE	CLO.
CLS	CLS	COLOUR	C.
DATA	D.	DEF	DEF
DIM	DIM	DIV	DI.
DRAW	DR.	ELLIPSE	ELL.
ELSE	EL.	END	END
ENDCASE	ENDC.	ENDIF	ENDI.
ENDPROC	E.	ENDWHILE	ENDW.
ENVELOPE	ENV.	EOR	EOR
ERROR	ERR.	FALSE	FA.
FILL	FI.	FN	FN
FOR	F.	GCOL	GC.
GOSUB	GOS.	GOTO	G.
HIMEM	H.	IF	IF
INPUT	I.	LET	LET
LIBRARY	LIB.	LINE	LIN.
LOCAL	LOC.	MOD	MOD
MODE	MO.	MOUSE	MOU.
MOVE	MOV.	NEXT	N.
NOT	NOT	OF	OF
OFF	OFF	ON	ON
OR	OR	ORIGIN	OR.
OSCLI	OS.	OTHERWISE	OT.
OVERLAY	OV.	PLOT	PL.
POINT	POINT	PRINT	P.
PROC	PROC	QUIT	Q.
READ	REA.	RECTANGLE	REC.
REM	REM	REPEAT	REP.
REPORT	REPO.	RESTORE	RES.
RETURN	R.	RUN	RU.
SOUND	SO.	STEP	S.
STEREO	STER.	STOP	STO.
SWAP	SW.	SYS	SY.
TEMPO	TE.	THEN	TH.
TINT	TIN.	TO	TO
TRACE	TR.	TRUE	TRU.
UNTIL	U.	VDU	V.
VOICE	VOICE	VOICES	VO.
WAIT	WA.	WHEN	WHE.
WHILE	W.	WIDTH	WI.

Basic Commands

Following is a list of all of the Basic commands and their minimum abbreviations.

APPEND	AP.	AUTO	AU.
CRUNCH	CR.	DELETE	DEL.
EDIT	ED.	EDITO	EDITO
HELP	HE.	INSTALL	INSTAL.
LIST	L.	LISTIF	LISTIF
LISTO	LISTO	LOAD	LO.
LVAR	LVA.	NEW	NEW
OLD	O.	RENUMBER	REN.
SAVE	SA.	SAVEO	SAVEO
TEXTLOAD	TEX.	TEXTSAVE	TEXTS.
TEXTSAVEO	TEXTSAVEO	TWIN	TWIN
TWINO	TWINO		

In addition, although 'RUN' and 'QUIT' are classed as keywords they are really commands and be entered in mixed case like any of the commands listed above.

Functions and Pseudo Variables

Following is a list of all of the Basic functions and pseudo variables and their minimum abbreviations.

ABS	AB.	ACS	AC.
ADVAL	AD.	ARGC	ARGC
ARGV$	ARGV$	ASC	AS.
ASN	ASN	ATN	AT.
BEAT	BEAT	BGET	B.
CHR$	CHR$	COS	COS
COUNT	COU.	DEG	DE.
EOF	EOF	ERL	ERL
ERR	ERR	EVAL	EV.
EXP	EXP	EXT	EXT
FILEPATH$	FILE.	GET	GET
GET$	GE.	INKEY	INKEY
INKEY$	INK.	INSTR(	INS.
INT	INT	LEFT$(	LE.
LEN	LEN	LN	LN
LOG	LOG	LOMEM	LOM.
MID$(	M.	OPENIN	OP.
OPENOUT	OPENO.	OPENUP	OPENU.
PAGE	PA.	PI	PI
POS	POS	PTR	PTR
RAD	RA.	RIGHT$(	RI.
RND	RN.	SGN	SG.
SIN	SI.	SQR	SQR
STR$	STR.	STRING$(	STRI.
SUM	SU.	SUMLEN	SUMLEN
TAN	T.	TIME	TI.
TIME$	TIME$	TOP	TOP
USR	US.	VAL	VA.
VERIFY(	VE.	VPOS	VP.
XLATE$	XL.		