BAS12K

PROGRAMMING MANUAL.

Copyright HISOFT 1980

CONTENTS

SECTION 1. PRELIMINARIES.

1.1 Introduction.

This manual is not intended for those who have never encountered the BASIC language before; it is assumed that, if you see the need for a 12k BASIC then you have previously used some other version of the language on your machine and found it inadequate. Thus, you will not find any detailed advice on programming practice within this manual - it is intended as a reference guide to enable you to use BAS12K efficiently and without error.

It is recommended that you read through the whole manual before attempting anything more than a superficial program in BAS12K, with particular regard to Section 3 which gives full details of the commands etc. available in this version of the language.

1.2 Conventions.

To clarify definitions and ease understanding, several conventions will be used throughout this manual. No claim is made that these conventions are industry standard although they should not be unfamiliar in appearance.

1.2.1 Syntactical conventions.

Throughout this document, although primarily in Section 3, any reference to BAS12K statements and commands will obey the following rules.

The statement or command being described, and any other keyword essential to the definition, will be printed in capital letters. Angle brackets (< >) will be used to denote elements that are an essential part of the syntax of the statement or command while square brackets ([ ]) will denote those components which are optional.

< essential element > [optional element]

1.2 Keyboard usage.

Certain routines within BAS12K are reached by holding the 'control' key (or @ key on NASCOM 1's) down and then pressing another key; throughout this manual this feature is represented by the word CTRL followed by the character that is to be keyed with the 'control' key.

e.g. CTRLA represents holding the 'control' key down and then keying 'A'.

Similarly, the 'shift' key has no effect when used alone; only when another key is pressed with the 'shift' key will a value be returned by the keyboard. Whenever it is necessary to use the 'shift' key this will be represented by the word SHIFT followed by the key that is to be pressed with 'shift'.

e.g. SHIFTA holding the 'shift' key down and then keying 'A'.

Certain CTRL characters are recognised by the NASMON routine which blinks the cursor and reads a character from the keyboard. These special characters are:

CTRLK — use of this inverts the function of the SHIFT key on alphabetics only.

CTRLB — decreases the rate at which a character repeats if its key is held down.

CTRLD — increases the rate at which a character repeats if its key is held down.

CTRLC — enters the extended screen editor of NASMON with the display set to the start of the current text.

CTRLZ — enters NASMON's Front Panel mode.

The use of CTRLC does not convert BAS12K's compacted text to NASMON text format; see Section 2.3.3 and the command SCREEN for details of text conversion. Also CTRLC does not reset any of NASMON's vectored jumps.

Note: BAS12K accepts all its commands and statements in either upper or lower case so that the use of CTRLK will not affect the response of the interpreter.

1.3 Constants.

1.3.1 Numeric constants.

Numeric values are either held in integer form or as a floating point number with signed mantissa and exponent. The conversion from one to the other is done automatically and the user has no control over this internal representation.

Externally (on the video screen or printer) the representation of numerics depends on the PRECISION set i.e. what value that you

have set (using the PRECISION statement) for the number of significant figures to which numerics are to be displayed. If the precision has been set to X then all numbers below 10↑X will be displayed normally; numbers >= 10↑X will be displayed in scientific notation i.e. with a signed mantissa, followed by 'E' and a signed exponent. Also, if the precision is less than the maximum (11) then any number less than 0·1 will be displayed in scientific format, whereas if the precision is 11 then numbers less than 0.01 will be converted to scientific notation.

Examples:

PRECISION = 5	0.2345	printed as	0.2345
	0.0143	printed as	1.4300E-02
	1E4	printed as	10000
	3.2E7	printed as	3·2000E+07
PRECISION = 11	34563.2	printed as	34563·2
	2E10	printed as	20000000000
	0.0143	printed as	0.0143
	0.0045	printed as	4·5000000000E-03
	55.4E17	printed as	5·5400000000E+18

The largest number that can be held withoutan 'ARITHMETIC OVERFLOW' error being displayed is 1·7014118346 E+38.

1.3.2 Character constants.

Another type of constant is the character constant or string. It is different from other constants both in the way that it is defined and in the way that it is used. Character string constants will most frequently be seen in PRINT and INPUT statements although they may be assigned to variables in the same way as any other constant.

String constants are defined by placing any ASCII character or group of characters between double quotes '"'. A string may be of a length between 0 and 255 characters inclusive. A string of zero length is called a null string.

Examples:

"How are you?"	
"I'm very well - thank you"	(includes single quote)
"*"	
""	(null string)

Note that double quotes are not allowed within string expressions - they are only used to define strings.

1.4 Variables.

A variable is an item of data that may take on different values. For example, it can be assigned a value by the programmer and later be changed by the program during execution. Within BAS12K two types of variable exist; 'numeric' and 'character string' variables.

Variables are identified by their unique names which may be defined as follows:

Variable names:

Numeric variables - names are defined by any alphabetic character followed by any number of alphanumeric characters. Note that only the first two characters of this name are taken as the unique identifier of the variable.

String variables - names are defined as for numeric variables except that the terminating character of the name must be a dollar sign ($). Again only up to the first two characters of the name (not including the dollar sign) are taken.

It should be noted that some combinations of letters intended as variable names or as parts of variable names are not valid. These are the BAS12K keywords.

Either of the variable types may be subscripted and this will be described below. The same variable name may be used for a numeric or a character variable within the same program e.g. the names 'DU' and 'DU$' may be used in the same program since they are considered different names.

1.5 Dimensioning.

Both numeric and string variables may be subscripted using the DIM statement. A variable is subscripted (dimensioned) in a DIM statement by following the variable name with an integer enclosed in parentheses - this would create a one-dimensional array. It is possible to create n-dimensional arrays simply by using more than one integer between the parentheses with the integers separated by commas. Note that BAS12K arrays have a 0th element so that, in a statement such as DIM A(5), an array of six elements is defined.

In the absence of a DIM statement it is possible to refer to and to assign an array as long as none of the subscripts used exceed 10; one can think of a default size of 10 for each dimension.

If an attempt is made to re-dimension an array in the same program (i.e. to DIM the same variable twice) then an error will be issued.

Note that each element of a character string array is a full character string variable - you are not limited to single characters in array elements.

Examples:

```
DIM  A(12), BR$(3,2), AC(2,2,4)
DIM  PQ$(20)
```

with reference to the above DIM statements consider:

```
A(11) = 5.6              ok.
A(14) = 12               illegal - subscript too large.
AC(1,1) = 56.4           illegal - missing subscript.
BR$(0,0) = "ONE"         ok.
PQ$(15) = BR$(1,2)       ok.
BR$(1,1) = A(3)          illegal - type mismatch.
C = A(0) + AC(1,2,3)     ok.
D(11) = 78.956           illegal - D() not DIMensioned and subscript
                         greater than 10.
```

Finally note that, if the dimensioning of a variable uses many subscripts or if the subscripts are large, then it is possible that there will not be enough room within memory to hold the array. When this happens a 'SUBSCRIPT OUT OF RANGE' error will be generated on reading the DIM statement.

1.6 Operators.

There are three different classes of operator available in BAS12K. The class of operators which is most familiar is that of the Mathematical Operators. These comprise addition, subtraction, division, multiplication and exponentiation: +, -, /, * and ↑ .

The second class is the Logical Operators. They are used to perform bit by bit operations on integer quantities and are used extensively in conditional tests and for masking. BAS12K automatically converts floating point quantities to integers when these numbers appear as arguments of these operators.

The third class of operators are called Relational Operators. They are also used in conditional tests but in a different fashion from Logical Operators. They are commonly used in IF statements to determine whether one expression is greater than another. When integer and floating point quantities are compared the integer is first converted, by BAS12K, to floating point.

The above operators are now discussed in more detail.

1.6.1 Mathematical Operators.

Symbol	Example	Meaning
+	X + Y	Add X and Y
-	X - Y	Subtract Y from X
/	X / Y	Divide X by Y
*	X * Y	Multiply X and Y
↑	X ↑ Y	Raise X to the power Y

When an arithmetic expression containing several of the above symbols is to be evaluated, it is processed by BAS12K according to the following priority scheme:

↑ , unary minus, * and / , + and - .

This means that when BAS12K is evaluating an expression containing a mixture of mathematical operators, it will first do the exponentiation, then take into account any unary minus signs (such as -0.456 or -X). Next it will do multiplications and divisions followed by additions and subtractions. When signs of equal priority are encountered then BAS12K does the left one first since it evaluates expressions from left to right. The above order can be altered by the use of parentheses. BAS12K evaluates quantities in parentheses first and, in the case of nested parentheses, it starts with the innermost set and works its way out.

The above should be read carefully if there is ever any doubt as to how a particular arithemetic expression will be evaluated by BAS12K.

1.6.2 Logical Operators.

When Logical Operators are used on one or two they perform the desired operation on the corresponding bits of the number or numbers. If the arguments of these operations are floating point numbers then BAS12K converts them to integer and the arguments must be greater than -65537 and less than 65536.

Example:

```
          A = (0011110001010111)
          B = (1001100000111001)

A  AND B = (0001100000010001)    output=1 if both inputs=1
A  OR  B = (1011110001111111)    output=1 if either input=1
NOT  B   = (0110011111000110)    output=1 if input=0 and vice
                                                      versa
```

These operators used like this, operating on one or two numerical expressions, yield a single numerical result. The above example is rather complicated but shows the bitwise effect of the operators. As a simpler example consider:

```
63 (111111) AND 16 (10000)  =  (010000)  =  16
-1 (1111111111111111)  OR  4 (100)  =  (1111111111111111)  =  -1
                NOT 0 (0)  =  (1111111111111111)  =  -1
```

Also note that in general: NOT X = -(X+1).

The logical operators have a very different effect when they are used in an expression that is the test condition of an IF ... THEN ... ELSE statement. In this case the expression is being logically evaluated to see if it is TRUE or FALSE. An expression that is evaluated to TRUE has a value of -1 whilst an expression which is FALSE has a value of 0. The expression(s) are evaluated to either -1 or 0 and then the logical operators operate on the result(s) to produce either -1 (TRUE) or 0 (FALSE). If the final result is -1 then the statement(s) after the THEN are executed whereas if the final result is 0 then the statements after the ELSE are obeyed or if the ELSE clause is not invoked then control passes to the statement after the IF statement.

Example:

```
10 A = 56.03 : B = -200
20 IF A > 100  AND  (A-B)  0  THEN  TQ = TQ + 1
```

The conditions A > 100 and (A-B) 0 are evaluated to 0 and -1 respectively. Now 0 AND -1 equals 0 (FALSE) so that TQ is <u>not</u> increased and control passes to line 30.

We see that, although the effect is different, the function of the logical operators is the same in the above two cases.

1.6.3 Relational Operators.

As the name implies, this group of operators tests the relation of variables to other variables or constants. The six relational symbols and their meaning within BAS12K are given over the page.

Relational Operators:

Symbol	Example	Meaning
=	X = Y	is X = Y ?
< >	X < > Y	is X not equal to Y ?
<	X < Y	is X less than Y ?
>	X > Y	is X greater than Y ?
< =	X < = Y	is X less than or equal to Y ?
> =	X > = Y	is X greater than or equal to Y ?

The results of all these tests will either be TRUE or FALSE and these logical values are assigned the numerical values -1 and 0 respectively. These relational operators are most often used in IF ... THEN ... ELSE statements but they may also be used in the following manner:

A = B = 9	would assign the value -1 (TRUE) to A if B = 9, otherwise 0 (FALSE) would be assigned to A.
C = D < 100	D < 100 is evaluated and the result assigned to the variable C.

1.6.4 String Operators.

The string operators consist of the concatenation operator ('+') and the relational operators. The '+' operator is used to join two strings together (e.g. "ABCD" + "XYZ" = "ABCDXYZ"). The relational operators, when applied to string operands, indicate alphabetic sequence. If one string is 'less than' another string this implies that the first string would appear before the second if they were sorted into alphabetic order. If two strings of unequal length are compared then the shorter string is padded with trailing spaces to make it the same length as the other string. Otherwise trailing spaces are ignored. A null string is considered to be completely blank and is less than any other string.

Examples:

"ABCD" < "XYZ" evaluates to TRUE(-1)

"FRED" >= "FREDA" evaluates to FALSE(0)

"" = " " evaluates to FALSE(0)

1.6.5 Operator Precedence.

The overall operator precedence is shown below. The operator at the top of the list has the highest precedence. Operators of equal precedence are evaluated from left to right.

Operator Precedence:

1. ( ) Expressions enclosed in parentheses.
2. ↑ Exponentiation.
3. - Unary minus.
4. * / Multiplication and division.
5. + - Addition and subtraction.
6. Relational operators.
7. NOT The negation operator.
8. AND The disjunction operator.
9. OR The conjunction operator.

1.7 Running a Program.

Once a program has been created it may be executed using the RUN command. This clears all variables and enters the program either at the beginning or at a specified line number (see RUN in Section 3). A program may also be executed from a particular line number by the use of the GOTO statement - this does not clear variables and may be useful for restarting the program.

If loading a program from tape then the program may be run automatically as soon as it is loaded by using the LOADGO command (see Section 3).

When a program is running, execution may be halted at any time by pressing any key on the keyboard - the program will go into a wait state until you either press ESCape (SHIFTENTER) which will return you to BAS12K command level or any other key upon which execution will be resumed normally. This is useful if you feel that the program has gone into an endless loop or you wish to inspect its progress - break out, do a list of variables (LVAR), amend if necessary and then restart execution using CONT (see Section 3 for full details).

SECTION 2. PROGRAM ENTRY AND EDITING.

2.1 Entry Modes.

Text may be entered either within BAS12K or by using NASMON's extended screen editor.

Entering text within BAS12K is done either via the AUTO command or by typing in lines directly; line number followed by text.

Text is entered within the screen editor in the normal way (see the NASMON Programmer's Manual) but you must remember to give each text line a line number and line numbers must be in numerical order.

Alternatively, text may be created within BAS12K and then converted to NASMON editor format by using the SCREEN command. Note that there must be enough memory in the machine to hold both the compacted text of BAS12K and the ASCII text of NASMON at the same time. Thus occasionally, for very large programs, you will not be able to use the screen editing facility because of lack of memory.

Text may be converted from NASMON format to BAS12K compacted format by use of the NASMON editor 'J' command, assuming that BAS12K has been initialised beforehand by executing it at 4300H/4303H. BAS12K need only be initialised once in a session as long as a cold start to the monitor is not made. If a cold start (e.g. RESET) is made then either re-initialise BAS12K or set up locations CVECT+1 and CVECT+2 to 01H and 40H respectively.

2.2 Program Entry.

The format of a BAS12K program line is:

nnnn <statement>[<:statement>][.....]

where 'nnnn' is the line number of the line and this must be in the range 0-65529. Within BAS12K lines need not be typed in following numerical order; the interpreter automatically orders the lines as they are entered.

Lines may be entered by typing the line number followed by a BAS12K statement or statements separated by colons ':'.

Alternatively the AUTO command may be used to automatically generate line numbers for you - you simply type the statement(s).

BAS12K accepts lines up to 255 characters in length; any entry from the keyboard that causes a line to exceed this length will be echoed as a carriage return/line feed and ignored within the line.

Lines are terminated by ENTER/NEWLINE.

As discussed above, lines may also be entered with NASMON's extended screen editor but, before running the program, the text must be converted to BAS12K's compacted text format. This is achieved through the use of NASMON's 'J' editor command - see above.

Programs may also be entered from cassette within BAS12K using the LOAD, ALOAD and AMERGE commands (see Section 3).
LOAD loads compacted text from tape straight into memory whereas ALOAD and AMERGE convert the ASCII form stored on tape into the compacted form first.

Note: ALOAD and AMERGE cannot be used to load a NASMON textfile and vice versa. Text should have been ASAVEd if ALOAD or AMERGE is to be used.

2.3 Program Editing.

Errors in your program are almost inevitable especially in a language like BASIC which does not encourage the use of flowcharts - it is too easy to write in BASIC.
Therefore a good BASIC must provide adequate editing facilities for correcting the errors. The following details the facilities available within BAS12K.

2.3.1 Deleting lines.

Within BAS12K single lines may be deleted simply by typing the relevant line number followed by ENTER/NEWLINE.

Blocks of lines may be erased from the program using the DELETE command (Section 3) from within BAS12K or the 'X' command from within NASMON's editor. Note that, when using DELETE, the line numbers given as arguments to the command must exist in the program, otherwise an error message will be generated - this is a safety feature to prevent accidental erasure.

If, in the process of typing a line, you decide that you do not want to enter this line then type ESCape (SHIFTENTER). This will cause a carriage return/line feed to be issued and the message '*BREAK' to be displayed.

2.3.2 Line Editor.

The line editor within BAS12K may be entered from BAS12K by using the command EDIT. Full details of this command are given in Section under EDIT. Note that if you attempt to edit a non-existent line with this command then an 'ILLEGAL FUNCTION' error will result.

2.3.3 Screen Editor.

Refer to the NASMON Programmer's Manual for full details of how to use the extended screen editor.

2.4 Direct Mode.

Most BAS12K statements and all the commands may be executed from within the interpreter without a line number i.e. directly. This enables BAS12K to act as a calculator or a test-bed for certain simple constructs; variables may be assigned and PRINTed,

FOR...NEXT loops obeyed (if on one line) etc. Of course some statements may lead to rather odd results if used in this way. Two statements will result in the error 'ILLEGAL DIRECT' if used without a line number - they are INPUT and DEF FNV.

2.5 Use of the BACKspace Key.

Use of the BACKspace key from within BAS12K will produce a different result from its use in screen editing. While still deleting the previous character from the text buffer within BAS12K, the character is not deleted on the screen - instead the character that has actually been 'backspaced' over is displayed again between slashes '/'. This is so that BAS12K can be used with a hard terminal.

Example:

Say you have just typed in 'PRUMT' instead of 'PRINT'; hit BACKspace three times and then type 'INT'. The output that will appear is:

PRUMT/TMU/INT

it is clear exactly which characters you have backspaced over although this may take a little while to get used to for those familiar with screen editing.

SECTION 3. COMMANDS, STATEMENTS AND FUNCTIONS.

The following is a comprehensive list of the various commands, statements and functions available under BAS12K together with detailed explanations.

Throughout this section the following definitions hold:

Commands:	those instructions to BAS12K that are normally issued in the 'direct' mode, that is - not imbedded in the program. Although all commands may be written into a program, most of them will give errors when used in this way and usually result in control being passed back to BAS12K 'direct' mode with 'Ready' displayed.
Statements:	those instructions to BAS12K that are normally issued within a program. Most statements may also be used in the 'direct' mode (one exception being INPUT) without line numbers. If an attempt is made to use a statement illegally in the 'direct' mode then the message 'ILLEGAL DIRECT' will be displayed.
Functions:	these are like subroutines; they return a value given certain arguments. BAS12K provides many mathematical, Input/Output and character handling functions - these are intrinsic functions. The programmer may also define his own functions using the DEF statement.

The above distinction between commands and statements is rather loose but perfectly adequate for the purposes of this manual.

For information on the range of variables etc. and other conventions used in the following section see Section 2.

3.1 Commands.

ALOAD

This command is used to load the ASCII version of a program that has been previously dumped out to tape using ASAVE. Normally a program is saved on tape in a condensed form in order to economise on space; however it may be useful to save the program as pure ASCII text if it is, say, required to transfer the program to another system/BASIC - ALOAD and its associated commands allow you to do this.

ALOAD prompts for a filename of up to 10 characters and then searches the tape for the file in the same way that LOAD does. The command issues a NEW before loading the tape.

AMERGE

This is similar to ALOAD in that it loads a program that has been dumped out to tape in ASCII. However this command does not do a NEW before reading the tape and so lines from the tape can be merged or appended to program lines in memory. This command prompts for a filename of up to 10 characters in length.

ASAVE

Saves the current program on tape in ASCII not in condensed form that SAVE does. This enables programs to be merged and appended with each other and, indeed, transferred to other systems. ASAVE prompts for a filename of up to 10 characters under which to save the program on the tape.

AUTO [<line number>] [<,increment>]

This command causes entry to the automatic line numbering mode. Line numbering will begin at the line number specified in the AUTO command and the increment between line numbers will be as given by the second argument of the command. The default for both these numbers is 10.

Examples:

AUTO 5 causes line numbering to start at 5 and increase in steps of 10 (by default).

AUTO 100,5
causes line numbers to be generated starting at 100 and increasing in steps of 5.

CONT

This causes program execution to continue after the execution of a STOP or END statement or after ESCape has been input from the keyboard. This command is particularly useful as a debugging tool since you can put a STOP statement in your program at some relevant point, then list the variables in use (using LVAR), check their correctness, amend if necessary and then re-enter the program using CONT.

Note that CONT will not function if any direct-command error is encountered or if the program is modified at all before typing CONT.

Note also that this command should not be used when the 'break' out of the program using ESCape has been from an INPUT statement. If it is required to re-enter the program in this particular case then a GOTO statement must be used.

DELETE <line number>[<- line number>]

This command will delete blocks of lines from the program beginning at the first line number entered following the command and finishing at the second line number entered i.e. the delete is inclusive. If only one line number is entered then only that line will be erased.

Examples:

DELETE 95 - 150 will delete all the lines with line numbers between 95 and 150 inclusive.

DELETE 45 deletes line 45 only.

If either line number does not exist in the program then an error will be generated as a safety check.

EDIT < line number >

This is a most powerful command giving access to the inbuilt line editor of BAS12K.

The commands available under the editor are as follows:

A - abandon the changes made so far and reload the working buffer with the original line (used when you have made an irrecoverable error while editing but still want to edit this line.

[n]D - delete the next 'n' characters starting at the current position of the pointer. D alone deletes the next character.

[n]F< character>
- this command 'finds' the nth occurrence of 'character' within the line starting from the current position of the pointer and then sets the pointer before the 'found' character. This is useful for skipping quickly to the portion of the line that you wish to edit.

I - insert text before the character currently pointed to. To end the insertion of text use either ESCape (to stay in the edit) or CR (to return to BAS12K command level).

K - 'kill' the rest of the text from the character currently pointed to until the end of the line i.e. delete the rest of the line.

L - list the line currently being worked on to the video screen.

Q - 'quit' the edit and return to BAS12K command level leaving the line as it was i.e. ignoring all changes made in this session.

[n] R - replace the 'n' characters from the current pointer position with 'n' characters which should follow the R. The default on 'n' is 1.

X - set the pointer to the end of the line and enter the 'I'nsert mode - useful for appending to a line.

[n]␣ - move the pointer 'n' characters to the right.

[n] BS - move the pointer 'n' characters to the left.

CR - return to BAS12K command mode and print the final version of the edited line.

In order to clarify the use of this line editor an example is set out below. This example is not meant to show a realistic use of the editor, its purpose is to illustrate the way in which the more common editor commands are used - it is certainly not the most efficient way of using the line editor.

Example of editor use:

Consider the following line of text:

```
90 PRINT "PAYROLL - details.":GOSORB 1999
```

This line is incorrect and should read as follows:

```
90 PRINT "PAYROLL - Personnel details.":GOSUB 1000
```

We can use the line editor to amend the incorrect line as shown below (note that the commands themselves are not echoed as they are typed in; only replaced or inserted characters are echoed):

1. Enter the editor to work on line 90.

```
EDIT 90
90
```

2. List the line (for convenience).

```
L    90 PRINT"PAYROLL - details.":GOSORB 1999
     90
```

3. Set the pointer to just after the '-'.

```
F-␣  90 PRINT"PAYROLL -
```

4. Now insert the word 'Personnel', terminated by ESCape.

```
I    90 PRINT"PAYROLL - PersonnelESCape
```

5. Now set the pointer just after the 'S' of GOSORB.

```
2F0  90 PRINT"PAYROLL - Personnel details.":GOS
```

6. Delete the '0'.

```
D    90 PRINT"PAYROLL - Personnel details.":GOS/0/
```

7. Replace the 'R' with a 'U'.

```
RU   PRINT"PAYROLL - Personnel details.":GOS/0/U
```

8. Set the pointer after the '1' of 1999.

```
␣␣␣  90 PRINT"PAYROLL - Personnel details.":GOS/0/UB 1
```

9. Replace '999' with '000'.

```
3R000 90 PRINT"PAYROLL - Personnel details.":GOS/0/UB 1000
```

10. Return to BAS12K command mode.

```
CR   90 PRINT"PAYROLL - Personnel details.":GOS/0/UB 1000
```

Note that, in the above example, the editor commands and other entries from the keyboard are underlined for clarity. Note also that any deleted character(s) will be enclosed by slashes '/'; this will also be the case when backspace BS is used in BAS12K command mode, if you are not in the screen editing mode.

You are encouraged to work through the above example until you fully understand how it works and then try many more examples of your own before attempting to edit real BASIC programs.

LIST [< line number>][< ->][< line number >]

This command is used to display portions of the current program onto the video screen. A few examples will clarify its use:

Examples:

LIST 5	will print line 5 only on the screen.
LIST 10-30	will print all lines from line 10 to line 30 (inclusive).
LIST -50	will list all lines from the start of the program up to and including line 50.
LIST	simply prints the whole program line by line on the video screen.

If, during an extended listing, you wish to stop the list at any stage in order to inspect it, then you simply hit any key on the keyboard (apart from CTRL, SHIFT or GRAPH) and the listing will be halted. You will then have the option of continuing the listing by hitting any key (apart from CTRL etc) or returning to BAS12K command mode by hitting ESCape (this is SHIFT CR).

LLIST[< line number>][< - >] [< line number >]

Functions in exactly the same way as LIST except that the output is directed through the serial port (normally one would expect this to be connected to a printer) intead of to the video screen.

LOAD

This command causes a BASIC program to be loaded into memory from tape. It first prompts for the filename of the program - this filename may be up to 10 characters in length). After the filename has been entered the tape machine should be switched to 'PLAY' and BAS12K will begin searching the tape for a file with the required filename. If a file with a different filename should be encountered then its filename will be displayed and searching will continue. On finding the correct filename the program will be loaded. Programs are stored on tape in 256 byte blocks and as the found program is loaded into

memory, a block count is displayed alongside the filename.
If a checksum error should be encountered then the message 'RDERROR' will be displayed and you will be returned to BAS12K command mode. You can rewind the tape beyond the defective block and try to load it again by using LOAD and entering the relevant filename.

LOAD?

This command does not actually load a file from tape but simply compares the fileon the tape with the program currently held in memory. If, at any stage during the comparison a mismatch is found then the message 'FILES DIFFERENT' is displayed.
This command is useful for checking that your program has been successfully dumped to tape.

LOADGO

A command that is exactly the same as LOAD except that, once the program has been successfully loaded, a RUN command is executed.
Again, this command prompts for a filename of up to 10 characters.

LVAR

Produces a list of the values of all the variables (apart from arrays) that are currently activated. A variable is activated by occurring on the left hand side of an assignment statement or in an INPUT or READ statement.
This command is most useful as a debugging tool e.g. interrupt the program (using ESCape twice), do an LVAR to inspect the state of the variables, alter the variables if necessary (using a direct assignment statement) and then re-enter the program with CONT.

LLVAR

Exactly the same as LVAR except that the output goes to the serial port and not to the video screen.

NEW

The command that deletes all record of the current program and clears all variables. Used before entering a new program into memory.

RENUMBER[< line number>][<,increment >]

Renumbers the current program so that the renumbered program begins at the line number given and its line numbers increase according to the increment given. The default of both arguments is 10.
The renumber command amends all GOTOs, GOSUBs and ON....GOTOs so that they refer to the new line numbers.

RUN [line number]

Causes execution of the current program starting at the line number given. If no line number is given then execution

begins from the lowest line number in the program.
This command clears all variables.

SAVE

Used to save the current program on tape. The command prompts for a filename under which to save the program and then immediately dumps the program to tape in 256 byte blocks with a checksum at the end of each block.
Note that the tape should be running before you enter the filename to give sufficient blank leader on the tape.
The filename may be up to 10 characters in length.

WIDTH [X]

This sets the width of the lines of text that BAS12K works on i.e. the maximum number of characters per line of text. 'X' may be any integer value between 14 and 255 inclusive.
This command is especially useful when IF..THEN..ELSE are used with multiple statements between the IF and the THEN and the THEN and the ELSE - this technique makes for clear, concise code but does lead to rather lengthy lines, hence the need for WIDTH.
Note that WIDTH sets the length of lines sent to the video screen.

LWIDTH [X]

Exactly the same in function as WIDTH except that this command sets the width of the lines of BAS12K text sent to the serial port. The same restrictions on 'X' apply.

NULL < X >

This sets the number of nulls to be printed on the video screen after the end of each line. X must be in the range 0 - 49 inclusive.

LNULL < X >

This has the same effect as NULL except that it sets the number of nulls sent to the printer at the end of each printed line. As in NULL, X must be in the range 0 - 49 inclusive.

NASMON

This command returns control to NASMON's extended screen editor. However, unlike CTRLC, this command also resets the editor vectored jump 'J' so that it now has the same effect that it had before BAS12K was initialised in this session. This is useful if you want to go back to NASMON permanently or if you wish to use NASGEN, our 3K assembler which also uses the 'J' command within the editor. Re-entering BAS12K at 4300H or 4303H will set the 'J' vector again so that its future use from within NASMON will convert NASMON text to BAS12K compacted text and enter BAS12K.
If you wish to leave BAS12K but not alter the 'J' vector then use CTRLC.

SCREEN

This is used to convert the compacted text of BAS12K to a format suitable for the NASMON extended screen editor.

The command first prompts with 'Text:' and you should respond by entering the hexadecimal address at which you wish the NASMON text to reside. This text should obviously not reside on top of the interpreter or on top of the compacted text (which lies immediately after the interpreter) or, if you wish to keep the NASMON ASCII text as a backup, on top of the BAS12K variables (which lie immediately after the compacted text). Use FRE to find out how much space is available after the compacted text.

While SCREEN is converting the text to NASMON format, which may take some time for a long program, the message 'Converting text ....' will be displayed. When finished you will be left in NASMON's editor mode - see the NASMON Programmer's Manual for details of the screen editing facilities available.

3.2 Statements.

CALL <(X)>

This statement is used to call up a machine code routine whose start address (in decimal) is given by the expression X. Unlike USR there is no need to POKE this address into a location within BAS12K.

Note, however, that no parameters may be passed to or returned from the machine code program - although it is simple to call a machine code routine this way, it is not as flexible as USR. You might use CALL to output titles quickly or to erase part of the screen quickly without having to POKE into locations C00H - C02H (see NASMON manual).

Example:

You might have set up the following machine code at E00H:

```
E00  111000          LD   DE,16
E03  210A09          LD   HL,090AH
E06  0605            LD   B,5
E08  C5       BLOCK  PUSH BC
E09  0630            LD   B,48
E0B  3620     LINE   LD   (HL),' '
E0D  23              INC  HL
E0E  10FB            DJNZ LINE
E10  C1              POP  BC
E11  19              ADD  HL,DE
E12  10F4            DJNZ BLOCK
E14  C9              RET
```

The above code will erase five lines from the video screen starting at line 5. It can be invoked from a BAS12K program by CALL(3584) - E00H being 3584 decimal.

The above will run twice as fast as POKEing locations C00H - C02H and then using CLS and many, many times faster than actually POKEing the screen with spaces.

Note that, to return to your BAS12K program from the machine code routine, you simply execute a RET (C9H) instruction.

CLEAR [<X>]

This statement has two functions; with no argument it simply clears all variables - numerics set to zero, character strings set null - however, if X is a valid numeric expression then this statement allocates space for string variables - the number of bytes that it allocates being equal to the value of X. Initially, on cold start, BAS12K allocates no string space and the message 'NO STRING SPACE' will be issued if any attempt is made to assign a value to a character variable. So it is up to the user to reserve as much space as he thinks he needs by using CLEAR X.

Examples:

CLEAR	clears all variables.
CLEAR 100	reserves 100 bytes of memory for string variables.

CLS

A very straightforward statement, this simply clears the video screen. Normally, this will clear 16 lines starting from 080AH i.e. the entire screen. However parts of the screen may be cleared by POKEing the number of lines to be cleared in location 0C00H and the address from which clearing is to start in locations 0C01H and 0C02H (low order in 0C01H) and then issuing a CLS - see the NASMON manual for full details of this partial screen clearing feature.

Example:

```
POKE 3072,4 : POKE 3073,74 : POKE 3074,9
CLS
```

This will clear four lines starting from line 6.

DATA <list>

Specifies the data that can subsequently be read by a READ statement. Elements in the list may be numeric or string data - the elements must be separated by commas.
DATA statements may occur anywhere in your program.

Example:

```
DATA   5,10,-5.7,"A","HENRY",79.6
```

DEF FNV <(list of parameters)>

Defines a user-defined function V (V can be any of the 26 alphabetic characters). Unlike 8K Basic user-defined functions, those supported by BAS12K are not restricted to occur on single lines or to pass only single parameters.
The list of parameters may be any number (greater than zero) of variables, numeric or character, that will fit on a line. The list separator is the comma.
The function is not defined on the line of the DEF statement as with smaller Basics but on subsequent lines. To indicate the value of which local variable (local to the function) is to be passed back to the code calling the function the FNRETURN X statement is used. The end of the function definition is indicated by the FNEND statement.
Two examples will help to explain the use of these statements. The first example is a function which returns the ARCSIN of a number, giving the result in radians. This function can be approximated by the expression:

$$ARCSIN(X) = ARCTAN(X/\sqrt{1-X^2})$$ result in radians.

Evaluation of this expression may generate an error in BASIC since X may be greater than 1, resulting in an error when the $SQR(1-X^2)$ is evaluated. Also, if $SQR(1-X^2)$ is zero then a 'CAN"T / 0' error will be generated on evaluation of the expression whereas the value PI/2 should actually be returned.
In BAS12K one can get round all these problems in the following manner:

Example 1.

```
  5 REM Function to return ARCSIN of a number X.
 10 DEF FNS(X)
 20 IF X*X > 1 THEN A = 0 : GOTO 50
 30 IF X*X = 1 THEN A = 1.5708 : GOTO 50
 40 A = ATN(X/SQR(1-X*X))
 50 FNRETURN A
 60 FNEND
 70 REM Code to call function.
100 INPUT M
110 N = FNS(M)
120 PRINT "The ARCSIN of ";M;" = ";N
```

Note that the variables X and A are local to the function and these labels may be used outside of the function definition with no ambiguity.

The second example shows the use of recursion in BAS12K function calls. This example is a function to evaluate the factorial of a number.

Example 2.

```
  5 REM Function to calculate the factorial of Y.
 10 DEF FNF(Y)
 20 IF Y < 1 THEN Y=1
 30 IF Y = 1 THEN F=1 ELSE F = Y*FNF(Y-1)
 40 FNRETURN F
 50 FNEND
100 REM Code to call the function.
110 INPUT I
120 J=FNF(I)
130 PRINT " The factorial of ";I;" = ";J
```

Note that line 30 calls FNF again and again until the parameter supplied is 1 at which point the function is exited - in other words the function calls itself - recursion.
Line 20 is needed since the factorials of numbers less than 1 are arbitarily defined as being equal to 1.
Note that type integer is assumed for Y.

DIM < V(<I>[<,J, ...>])>[, ...]

Used to set aside space for arrays (numeric or character). Arrays may be of any dimension and size provided sufficient memory exists to hold them. More than one array may be listed in one DIM statement, the names being separated by commas. The smallest allowable subscript is 0.
If a DIM satement is not issued for a particular array then any reference to the elements of the array must not be outside the range 0 - 10 i.e. the default size of any dimension is the first 11 elements. So, without a previous DIM statement, the following references are valid:

```
A(9,5) = 6     :     F$(10,10,10) = "chip"
```

whereas the following will generate errors:

```
S = X(20)      :     U(3,11) = 45.7
```

Of course, all these references are valid if a suitable DIM statement has been issued for the arrays.

END

Used to signify the end of program execution - it results in a return of control to the BAS12K 'direct' or 'command' mode. A CONT command may be issued after program execution has been terminated by END.

EXCHANGE < V,U >

As its name implies this statement exchanges the values of the variables U and V, assuming that they are of consistent type. The variables exchanged may be numeric or character but note that you cannot use this statement to exchange whole arrays, only their individual elements.

FNEND

The statement used to signify the end of the definition of a user-defined function (see DEF).

FNRETURN [X]

Signifies what value is to be returned from a user-defined function - must be type-consistent with the variable that the function is being assigned to. For more detail see DEF.

FOR <V> = <X1> TO <X2> [< STEP X3 >]

Enables a body of code to be repeated many times without having to use IF and GOTO statements which make the understanding of the flow of the program more difficult.

Initially V is set to the value of X1 and then the statements below the FOR are executed until a NEXT V statement is encountered. X3 is then added to Vand the test:

(X3<0)AND(V>=X2)OR(X3>0)AND(V<=X2)

is evaluated. If the result of this test is TRUE then control is passed back to the statement after the FOR statement. Otherwise program execution continues at the statement immediately following the NEXT.

Note that, if STEP is omitted then the default on X3 is +1.

GOSUB <line number >

Causes an unconditional call of the subroutine at the line number given. The statements at and following this line number will be obeyed until a RETURN statement is found, at which point control will be passed back to the statement immediately after the GOSUB.

GOTO <line number >

Results in an unconditional transfer of control to the statement at the line number given.

IF <condition> THEN <statement>[<:statement>][:.....]
[ELSE <statement>[<:statement>][:.....]]

Allows different paths through the program to be taken depending on the evaluation of 'condition'. If the result of the condition is TRUE then the statement(s) after the THEN and before the ELSE (if it is present) are executed. On the other hand, if the condition evaluates to FALSE, then control is passed immediately to the statement following the IF statement or, should the ELSE clause be invoked, then control is passed to the statement(s) after the ELSE. Assuming that the statement(s) after the THEN or the ELSE do not produce a transfer of control (GOTO) then after executing the statement(s) control will be passed to the statement immediately after the IF.

Note that, if more than one statement is to follow the THEN or the ELSE then these statements must be separated by a colon ':'.

Examples:

```
IF A=1 THEN PRINT"Heads" ELSE PRINT"Tails"

IF FLAG$="ON" AND TEMP>CRITICAL
  THEN GOSUB 1000 : PRINT "***TEMP***" : GOTO 90
  ELSE IF FLAG$="ON" THEN PRINT "***WARNING***"
```

Note that the second example is set out on more than one line - this is simply for clarity - you would, of course, type this in on only one line in your program. Remember that although the names of variables can be as long as you like (starting with an alphabetic) BAS12K only takes the first two characters as the label for the variable so you must be careful so as to avoid duplication.

INPUT [<literal ;>]<V>[<,V1> .....]

This statement is used to assign values to variables V, V1 etc. from the keyboard. You have the option of printing out a literal (say asking for a particular range of values) after which BAS12K will prompt for a value to be fed in from the keyboard by issuing a question mark on the screen. Obviously you must feed in values which are type compatible with the variables to which they are to be assigned or errors will be generated.

You can break out of an INPUT statement by hitting ESCape (SHIFT CR). However you cannot re-enter an INPUT statement using CONT; instead you must use a GOTO statement in the 'direct' mode.

KILL <array name>

A most useful statement - this allows you to initialise an array without having to use FOR.....NEXT loops. All elements of the array addressed by 'array name' are set to zero if the array is of type numeric or to 'null' if the type of the array is character.

Note that this statement only works on arrays - to try and 'KILL' a variable which does not represent an array will have no effect whatsoever.

LET < assignment >

This statement allows you to assign variables, either to other variables or to constants (numeric or character).

Firstly note that the 'LET' is optional and, in fact, rarely used except when portability is important. So the statement LET A(5,4) = 4.24 has exactly the same effect as A(5,4) = 4.24.

Secondly note that using a 'multiple' assignment statement has the following effect:

A = B = 0	in some Basics this would cause both A and B to be set to zero. However in BAS12K this statement has a quite different effect - only the first (from the left) '=' is taken as an assignment operator, any further '=' being assumed to be logical comparison operators. So, in the above statement the condition 'B=0' would be evaluated and the result -1 (TRUE) or 0 (FALSE) returned. This result would be assigned to A.
D = B = F = 6	Here 'B=F' would first be evaluated and either 0 or -1 returned; the returned value would then be compared with the constant 6. Since neither 0 or 1 is equal to 6 the result of the comparison will be FALSE (0). So in this case D will be assigned the value 0.
D = B = (F = 6)	This changes the order of the comparisons; first F is compared with the constant 6; 0 or -1 is returned. Then B is compared with either 0 or -1 depending on the previous result and the final result is assigned to D.

LPRINT [< X>][separator] [<literal>][separator] [.....]

Exactly the same syntax and effect as PRINT except that the output is sent to the serial port (printer) instead of to the VT screen. See PRINT for full details.

LPRINT USING < string>[<, print list >]

Identical to PRINT USING except that the output is sent to the serial port (printer), not the VT screen. See PRINT USING for full details.

LTRACE < X >

Like TRACE but only sends its output to the serial port and not to the video screen (see TRACE).

NEXT[<V>]

Used in conjunction with FOR as a test to see if the loop has terminated (see FOR for details).

ON <X> GOTO <line number>[<, line number>][.....]

This is a type of conditional GOTO statement. The expression X is evaluated and if the result is 1 then a branch is made to the first line number in the list, if the result is 2 then the branch is to the second line number in the list etc.

X may evaluate to a number less than 256 and greater than or equal to 0, the integral part always being taken. If X equals 0 or is greater than the number of line numbers in the list then execution is continued at the statement after the ON ... GOTO statement.

ON <X> GOSUB <line number>[<, line number>] [.....]

Similar to ON ... GOTO, this statement transfers control to the subroutine at the first line number given or at the second line number given etc. depending on the value of X. On exit from the called subroutine control is returned to the statement after the ON ... GOSUB.

OUT <X1,X2>

This takes the byte represented by X2 and outputs it to the port represented by X1. Obviously expressions X1 and X2 must be numeric and both must be in the range 0 to 255 inclusive.

POKE <X1,X2>

Enters the byte X2 into the location X1. This is necessary when interfacing machine code programs to BAS12K programs.

0=<X2<=255 0=<X1<=65,535

PRECISION [<X>]

Sets the number of significant figures that numeric variables will be displayed to although BAS12K always calculates using the maximum - 11. This statement is useful if you want all your results displayed to the same precision without having to use PRINT USING.

0=<X<=11 0 has same effect as 11.

The default value on X is 11.

Example:

PRECISION (5) means that, until another PRECISION statement is read, all numerics will be displayed to 5 significant figures e.g. 123458.9 will be printed as 1.2346 E+05.

PRINT [<X>][<separator>][<literal>][<separator>] .....

Directs output to the video screen. This output may be a blank line (simply PRINT), a literal, a numeric expression or a character expression or a mixture of the last three.

If more than one expression or literal is to be output then they must be separated in some way; the separators allowed and their effects are:

',' (comma)	causes the 'print-head' to be moved to the next 14 character tab position before printing the next character. Allows neat formatting of results although greater flexibility may be gained by using TAB and PRINT USING.
';' (semi colon)	has the effect that the next character printed will be printed 'hard-up' to the previous character i.e. no spaces between them.

Some examples will clarify the use of PRINT:

Ex. 1:
```
10 A = 89.3
20 PRINT "The value of 'A' is: ";A;"."
```

RUNning this program will cause the following to be output to the video screen:

The value of 'A' is: 89.3 .

Note the use of single quotation marks since double quotes cannot be used within a literal. Also note the space generated between the 89.3 and the terminating full-stop; BAS12K inserts a space after printing all numerics.

Ex. 2:
```
10 FOR I = 0 TO 1 STEP  0.2
20 PRINT I,SIN(I),COS(I)
30 NEXT I
```

This will produce a tabulated set of numbers but on a 48 character width screen you will need to set the PRECISION to 10 beforehand because, with full PRECISION, the width of a floating-point numeric goes just over the boundary of the next tab position of ','. Assuming that PRECISION has indeed been set to 10, the above program would produce the following:

0	0	1
.2	.1986693308	.9800665778
.4	.3894183423	.921060994
.6	.5646424734	.8253356149
.8	.7173560909	.6967067093
1	.8414709848	.5403023059

Both ',' and ';' can be used alone in a PRINT statement although this would not normally be advantageous.

Remember that PRINT will output lines of a length previously set by WIDTH (default is 72) so that if a short line length has been selected at any stage then the output produced by PRINT may not be quite as you expected e.g. if a line width of 20 had been selected before running the above program then all the numbers would be printed on separate lines.

The use of LWIDTH will be advantageous when using LPRINT.

PRINT USING < string > < , print list >

This statement is a highly flexible form of the PRINT statement designed to give the user total control of the printing of numerics. The string is an image of the output line except for special characters that are used as formatting instructions. The print list must contain only numerics but otherwise is the same as in the normal PRINT statement where expressions are separated by either a comma or a semi-colon. PRINT USING normally ignores the meaning of these separators unless they occur at the end of the statement where they have their normal meaning. string may either refer to a string variable previously assigned or it may be a literal containing an allowed mixture of the following formatting characters:

POUND SIGN ('£')

The pound sign is used to denote a numeric field.

```
PRINT USING "££££.££", 124.555
124.56
```

If the number to be printed will not fit in the field defined then a percent sign will precede the number and it will be printed as though no PRINT USING statement had been used. Any character other than a comma, period or up arrow will terminate the numeric field.

```
PRINT USING "£.£", 10.3
%10.3

PRINT USING"£.££  £.£", 1,5
1.00  5.0
```

If the fra tional digits of a number do not fit into the field defined then the number will be rounded and then printed. If a number being rounded becomes too large to fit in the field then a percent sign will be printed before the number.

```
PRINT USING"£.£ divided by £.£ = £.£", 9.99;1.99,5.02
%10.0 divided by 2.0 = 5.0
```

DOLLAR SIGN ('$')

The dollar sign is used when printing amounts of money - unfortunately because of the significance of the pound sign it is not possible to print a '£' hard up against a number. The field is defined with the pound sign except that the first two characters of the field must be dollar signs.

```
PRINT USING"$$£££.£",  34.56
  $34.6

PRINT USING "$£££.£",   4.56
$  4.6
```

Note that the use of this option adds an extra character to the size of the numeric field besides specifying the leading dollar. If allowing for this, the number is still too large to fit in the field then a percent sign will be printed before the dollar. Negative numbers will have their sign printed before the dollar sign.

```
PRINT USING"$$££.££",5678.-12.355
%$5678.00-$12.36
```

ASTERISK ('*')

The asterisk is used to fill leading blanks of any numeric field with asterisks. This is especially useful when printing a numeric field that should not be easily altered (e.g. writing cheques). The format is specified exactly as for the leading dollar and is used in much the same way except that the asterisks specify two extra printable characters. If room for at least one asterisk is not available then a percent sign will be printed.
The leading dollar and the asterisk fields may not be defined together in the same field; instead a literal dollar may be used.

```
PRINT USING"**££.££", 1.456
***1.46

PRINT USING"$**£££.££", -12.67
$**-12.67
```

COMMA (',')

The comma is used to insert commas in a numeric field every three places to the left of the decimal point. If at least one comma is embedded in a numeric field and before the decimal point then commas will be inserted appropriately. A comma before the numeric field definition or after the decimal point is considered to be a literal and will simply be printed. If while filling the numeric field, BAS12K runs out of field room then a percent sign will be printed before the number although the number itself will be printed correctly.

```
PRINT USING"££££,.££", 1234.56
1,234.56

PRINT USING"£,,,,,", 1E4
 10,000

PRINT USING"£££,££", 1E6
%1,000,000

PRINT USING"$$£££,£.££", 3456.36
 $3,456.36
```

UP ARROW ('↑')

The up arrow is used to denote scientific format for numeric fields. Four and only four up arrows are allowed and they must trail the numeric field. They are used to denote the 'E±XX' notation used in the scientific format. The final format of the output will depend on the exact nature of the other special characters in the numeric field.

```
PRINT USING"££.£££↑↑↑↑", 12.3456
 1.235E+01

PRINT USING"£££££.££↑↑↑↑", 12.3456
 1234.56E-02

PRINT USING"**£££.£££££↑↑↑↑", SIN(0.3)
*2955.202067E-04
```

LPRINT USING has exactly the same syntax as that outlined above - it simply outputs to the serial port instead of to the video screen.

RANDOMIZE

This is used in conjunction with the RND function. Basically it provides a way of starting at a different point in a particular series of pseudo-random numbers each time that the program is run. Consult the function RND for full details.

READ <V>[<,V1>][<,V2>][.....]

Used to read data from a DATA statement; it assigns to variable V, V1 etc. the next, next but one etc. available entries in the DATA list. Remember that once an element of the DATA list has been read it can only be read again by using the RESTORE statement which resets the DATA list pointer to the first element of data defined in the program. Obviously the type of the entry in the DATA list must be compatible with the type of the variable in the READ statement. Also an attempt to READ an element beyond the end of the DATA list will generate an error.

REM [message]

Used to document your program, REM produces no code and is, in fact, ignored by the interpreter except that the REM statement can be branched to. The message can be any combination of characters and can be left blank. REM statements can appear on lines by themselves or can be compounded with other statements on one line using ':'. The word REM can also be replaced by the single quote ' .

Examples:

```
10 REM  *** TRIGONOMETRIC PACKAGE ***
20 PI=3.1416          ' fundamental definitions.
30 R=PI/180
40 INPUT X            : REM number in degrees.
```

RESTORE [< line number >]

When a READ statement is first encountered within a program then the first element of data from the first DATA statement is read; the next READ statement reads the next element of data and so on (assuming there is only one argument per READ statement). This arrangement is rather inflexible since once an element of data has been read there is no way (using READ alone) that it can be re-read. RESTORE enables you to skip back and read elements of data previously read.

Using RESTORE without <line number> resets the data pointer to the beginning of the program so that, on a subsequent READ, data will be read from the first element of the first DATA statement in the program, then the second etc.

If a line number is included after RESTORE then the data pointer will be reset to the first DATA statement after the line number given - this allows for greater flexibility.

When using RESTORE with the line number option remember that an error will be returned if the data pointer is beyond the end of the last DATA statement in the program.

RETURN

This is used to return to the calling routine from a subroutine. It should be the last statement in the subroutine and will cause control to be passed back to the statement after the GOSUB or ON ... GOSUB that called the subroutine.

Obviously a RETURN should not be encountered without a previous subroutine call - an error will be generated if this happens.

STOP

When a STOP statement is executed, program execution is terminated and a message is printed, informing the user where the break in execution occurred. The program can then be restarted using CONT and execution would resume at the statement following the STOP. In short, STOP works just like END except that a message is printed.

Example:

```
30 STOP
40 GOSUB 2000
```

If the above is executed then the program would be halted and '*BREAK @ LINE 30' would be printed. A CONT command would cause execution to continue from line 40.

TRACE < X >

This allows you to trace the path that your program is taking by enabling you to print out the line numbers of the statements that are being executed. If the expression X is grater than or equal to 1 then the TRACE function will be turned ON until another TRACE statement with X less than 1 is encountered, at which point TRACE will be turned OFF.

When TRACE is ON then as statements are executed their line numbers will be printed out between < >.

LTRACE sends the line numbers to the serial port but otherwise works identically to TRACE.

WAIT < X1 , X2>[<, X3 >]

This causes the status of port X1 to be XOR'd with X3 and AND'd with X2. If the result is non-zero then execution continues, otherwise the program halts until the result becomes non-zero.

The default value on X3 is zero.

Example:

WAIT 4,5	Execution stops until either bit 0 or bit 2 of port 4 are equal to 1.
WAIT 6,255,7	Execution will halt until any of the most significant bits of port 6 are 1 or any of the least significant three bits are zero.

CLR TOP

Causes the top line of the video screen to be filled with spaces. Useful since the standard NASMON scrolling is 15 lines and not 16.

COPY <line number [,line increment] = line number-line number >

This copies the range of statements specified by the line range ('line number-line number') to the destination specified by the first argument and renumbering the block of lines according to the given line increment. If no line increment is specified then an increment of 10 is assumed.

The line range should be in the same format as that specified under LIST.

If an attempt is made to overwrite statements then the error message 'ILLEGAL FUNCTION' will be displayed.

Example:

COPY100,2=30-50	copies all the statements between lines 30 to 50 inclusive to lines 100, 102, 104 etc. assuming no statements exist there already.

LINE INPUT [literal ;]< A$ [,A1$ .....] >

Like INPUT this statement prompts, at execution time, with '? ' but, unlike INPUT, it then reads the whole of the line input up to, but not including, the end-of-line character.

Example:

Execution of '10 LINE INPUT X$' will prompt with '? '; if you now enter the string 'SIN(1.54)' followed by ENTER/NEWLINE then X$ will now be set to that string and may be processed by the string functions detailed in Section 3.3.

3.3 Functions.

ABS(X)

The absolute value of X is returned by this function. If X is positive, then the value returned is X; whereas if X is negative, then the value returned is -X.

ASC(X$)

The argument X$ is a string expression. The value returned by the function is the ASCII numeric value of the first character within the string. If the string is null then an error will be returned.

ATN(X)

Returns the arctangent of the expression X, in radians. The value returned will be between -pi/2 and pi/2 , where pi/2 is approximately equal to 1.5707963268.

CHR$(X)

This returns a single ASCII character (a one character string) whose ASCII value is the argument X.

$$0 <= X <= 255$$

COS(X)

Determines the cosine of the angle X; X is assumed to be in radians.

EXP(X)

The mathematical operation e X (e raised to the Xth power) is performed and the result returned. 'e' is the base of natural logs and is approximately equal to 2.7182818285. The maximun allowable value of X is 88.029691931 and a value greater than this will result in an overflow error being issued.

FRE(X)

X may be either a numeric or character expression. If it is numeric then this function returns the number of free bytes left to the programmer within memory. If X is a character expression then the function returns the number of bytes left available to hold character variables. Remember that this character space may be extended using CLEAR.

INP(X)

This returns a byte read from port X.

$$0 <= X =<255$$

INSTR(X$,Y$,X)

The INSTR function searches for sub-string Y$ in string X$. The last argument, X, specifies the first character of X$ at which to start the search. INSTR returns an integer value specifying at which character the sub-string started. If the sub-string was not found then zero is returned.

INT(X)

The value returned is the largest integer that is not greater than X. A few examples are shown below:

```
INT(5.9) = 5          INT(45.0) = 45
INT(-6.01) = -7       INT(0.54) = 0
```

LEFT$(X$,X)

A character function that returns a string that is the X left-most characters of the string X$. The value of X must be positive and less than 256 to avoid an error.

```
A$ = "NASCOM COMPUTERS"
LEFT$(A$,5) = "NASCO"
```

LEN(X$)

X$ is a string expression and this function returns the number of characters (including spaces and non-printables) within the string X$.

LOG(X)

This returns the natural logarithm (to the base 'e') of the number X. X must be greater than zero.
To translate to logarithms of other bases, where the log to the base A is desired, the following formula may be used:

```
logarithm of X to base A = LOG(X)/LOG(A)
```

The most commom use of this formula will be to convert to base 10 when A will be 10.

LPOS(X)

Returns the current position of the print-head of the printer. X is only a dummy argument and can have any value, including character values. Numbering starts at zero, so that if the printer is ready to start a new line then zero will be returned.

```
LPRINT "This is a test.";
A = LPOS(5)
```

If the above two statements are executed, A will then be assigned the value 15 since that is the position of the print-head after printing the message above (note the presence of the semi-colon).

MID$(X$,X,Y)

This function returns a character string that is a sub-string of the string X$. The returned string starts at position X within the string X$ and has a length Y.

```
0 < X =<255        0 =< Y =<255

MID$("abcdefgh",4,3) = "def"
A$ = "Lunch time"
MID$(A$,7,10) = "time"
```

Note: MIDS may also be used on the left hand side of an assignment statement.

PEEK(X)

Returns the value of the byte at the address specified by X. The value returned will be >= 0 and < 256 and X must be in the range:

$$0 =< X < 65536$$

POS(X)

The value returned will be the current position of the cursor on the video screen. X is a dummy argument and may take any numeric or character value.
The leftmost position on the screen is taken as zero so that, on NASCOM machines, POS will return a value between 0 and 47 inclusive.

RIGHT$(X$,X)

Returns a character string that is the rightmost X elements of the string X$. If the value of X is greater than or equal to the length of the string X$ then the entire string will be returned. An error will be issued if X is less than zero or greater than 255.

RND(X)

This function returns a random number that has a value between zero and one, but not including one.
The argument X has an effect on the number that is generated according to the following rules:

X< 0 A new series of random numbers is started. For different negative values of X, a different sequence is started each time but if the argument remains the same then the function will keep starting the same sequence so that the value returned will be the same each time the function is called.

X = 0 Returns the last random number that was generated.

X > 0 Generates a new random number in the present series. This is the argument that will be used most frequently with RND.

Note that, although different negative values of X will start different series of random numbers, the same negative value will always start the same series. This may be undesirable and, if so, RANDOMIZE can be used; this will 'randomise' the point at which you enter the random series.

```
10 A=RND(-2)
20 FOR I = 1 TO 5
30 A=RND(1)
40 NEXT I
```

```
10 A=RND(-2)
20 RANDOMIZE
30 FOR I = 1 TO 5
40 A=RND(1)
50 NEXT I
```

The program over the page on the left will always return the same six random numbers; 1·17E-10, 0·457, 0·456, 0·208, 0·206 and 0·607 (with PRECISION set to 3).
The program on the right, however - although it will always select the same series (because of the -2) - will enter the series at different points each time the program is RUN because of the presence of RANDOMIZE. For example three runs of this program produced the following results (again with PRECISION set to 3):

Run 1	Run 2	Run 3
1·17E-10	1·17E-10	1.17E-10
0·576	0·385	1.14E-02
0·746	0·314	0.780
8·65E-02	0·683	2·11E-02
0·195	0·620	3·48E-02
0·625	0·574	0·542

SGN(X)

This is the 'sign' function. It returns '1' if the argument is greater than zero, '0' if the argument is zero and '-1' if X is less than zero.

SIN(X)

Returns the sine of the angle X where X is assumed to be in radians.

SPC(X)

This function may only be used in a print statement. It causes 'X' spaces to be printed either on the video screen (if used in a PRINT statement) or on the printer (if using LPRINT).

$$0 =< X < 256$$

SQR(X)

Returns the square root of the argument X. An error will be issued if X is negative.

STR$(X)

This will return a character string that represents the numerical expression X. In other words, the function takes a number and transforms it into a character string. The string is constructed just as it would be output, that is with a leading space or minus sign and no trailing space.

TAB(X)

May only be used in print statements. Its function is to move the cursor to column X on the video screen (if used in a PRINT statement) or to move the print-head to column X on the printer (with LPRINT). If the cursor or print-head is already at this position then no action is taken. The argument must be positive and less than 256.

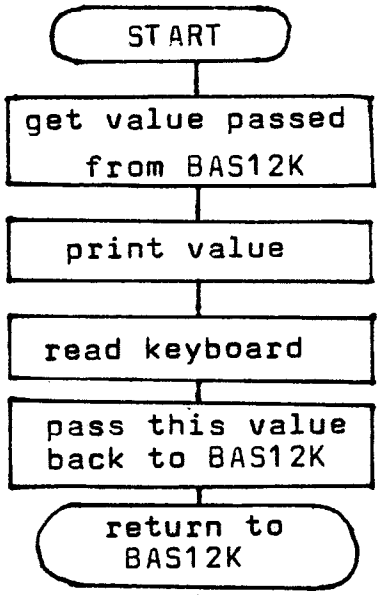
TAN(X)

Returns the tangent of the angle X where X is assumed to be in radians.

USR(X)

This is a function that allows you to interface BAS12K with a machine code routine. It returns the value passed back by the machine code and the value of X is passed to the machine code on entry to it. Details of how to use USR follow:

1. Set up the machine code. Say you want to write a very small routine to output the character passed to the routine and then input a character from the keyboard and pass it back to the BAS12K program. A flowchart for this routine might look like this:



To turn this into machine code we need to know how to pick up parameters passed by BAS12K and how to pass back values. This is achieved as follows:

<u>To pick up a passed value.</u>

Call address 4527H. The twos complement of the passed value will be returned in register pair DE (E low, D high).

<u>To pass a value back.</u>

The number that you wish to pass back should be converted to two's complement, the low order part placed in register B and the high order part in register A. Having done this you should then call address 452AH which will assign the value to the appropriate variable.

<u>To return to BAS12K.</u>

To return to BAS12K from your machine code routine, simply execute a RET (C9) instruction in your routine.

We are now in a position to write the machine code required for this routine. Assume that we are going to place the code at address E00H; then what we need is:

```
CD2745          CALL  GETPAR      (E00H)
7B              LD    A,E
F701            SYS   1
F702      GET   SYS   2
30FC            JR    NC,GET
47              LD    B,A
AF              XOR   A
CD2A45          CALL  PUTPAR
C9              RET               (E0FH)
```

Note that this very simple routine simply outputs the low order half of the number that is passed to it and passes back a single number in the range 0 - 255 i.e. no attempt is made to form the two's complement and cater for negative numbers.

2. Interface the routine to BAS12K. The above machine code may be entered into memory either by using CTRLZ and then entering it directly or by using POKE statements in the BAS12K program which calls the routine. Once it has been entered at E00H onwards then BAS12K must be told where to find this routine so that, when USR is invoked BAS12K knows to which location to transfer control.

   <u>To set up address of routine.</u>

   To tell BAS12K where your machine code resides simply POKE the low order half of the address (in this case 00H) in location 4507H (17671 decimal) and POKE the high order half of the address (here 0EH) in location 4508H (17672 decimal).

   If USR is invoked without first setting up locations 4507H and 4508H then an error will be issued.

The above gives you all the information that you need in order to use USR within your BAS12K program. To further clarify its use, a BAS12K program (simple and rather trivial) that calls the above machine code is listed below:

```
10 REM To illustrate the use of USR.
20 REM Define the machine code as data.
30 DATA 205,39,69,123,247,1,247,2
40 DATA 48,252,71,175,205,42,69,201
50 REM Enter the m/c into memory.,
60 FOR I = 0 TO 15
```

```
 70 READ DUM
 80 POKE 14*256 + I, DUM   ' at E00H etc.
 90 NEXT I
100 REM Set up the address of the m/c
110 REM  within BAS12K.
120 POKE 17671, 0 : POKE 17672, 14
130 REM We are ready to use USR now.
140 REM A simple routine to read a
150 REM  character from the keyboard
160 REM  and display it with its ASCII
170 REM  value, neatly tabulated.
180 REM
190 WIDTH 40                ' for tabulation.
200 CLEAR 50                ' string space.
210 A  = "   £££  "         ' format for
220                         ' PRINT USING.
230 CLS                     ' clear screen.
240 A=8                     ' first argument
250                         ' is a backspace
260                         ' to keep screen
270                         ' clear.
280 Z = USR(A)              ' Z is ASCII no.
290 IF Z=1 THEN GOTO 370    ' exit on CTRLA.
300 A = Z                   ' will print the
310                         ' character whose
320                         ' ASCII value is
330                         ' Z on next USR.
340 PRINT USING A , Z;      ' print the ASCII.
350 GOTO 280                ' back to print
360                         ' the character.
370 WIDTH 48                ' reset width.
380 END
```

With careful study of all the above you should be in a position to use USR with confidence.
You can, of course, interface more than one machine code routine with your program - but you must remember to set locations 4507H and 4508H for each new routine.

VAL(X$)

This function does just the opposite of STR$(X) . VAL(X$) takes a numerical character string and converts it to its numerical value.
Zero is returned if the first non-space character is anything other than a digit or a decimal point, plus sign or minus sign.

Examples:

VAL("- 45.7") = -45·7

VAL(" 540") = 540

VAL("·2345 ") = 0·2345

VAL(" ,10.05") = 0

APPENDIX 1. ERROR MESSAGES.

BAS12K issues full error messages which are usually explicit enough in themselves to enable you to detect exactly what the error was. If the error occurs within a program then the phrase '@ LINE nnnnn' will follow the error message enabling you to identify in which line the error occurred. The following is a list of the possible error messages that could occur.

*EXTRA LOST
ARITHMETIC OVERFLOW
CAN'T CONTINUE
CAN'T /0
FILES DIFFERENT
FNRETURN W/O FUNCTION CALL
ILLEGAL DIRECT
ILLEGAL EOF
ILLEGAL FUNCTION
INVALID INPUT
MISSING STATEMENT NUMBER
NEXT W/O FOR
NO STRING SPACE
OUT OF DATA
OUT OF MEMORY
RD ERROR
RE-DIMENSIONED ARRAY
RETURN W/O GOSUB
STRING TOO LONG
SUBSCRIPT OUT OF RANGE
SYNTAX ERROR
TOO COMPLEX
TYPE MIS-MATCH
UNDEFINED STATEMENT
UNDEFINED USER CALL

ON ISSUING A NEW.

LOCATIONS

7352/3 reset. ~~should~~ to point to next line.
E – H.

4144/B SET START OF TEXT
4150/1 } Set to EOT.
415E/F } END OF THIRD ZERO
4160/1 } AFTER END OF LAST LINE ~~00~~
4162/3 }
4190 ?

RUN

~~4150 7351~~ ✓
~~4154 FFFF~~
~~415E 7362~~

415E/F } 7354 7362
✓ }

4164 7351

APPENDIX 2. IMPORTANT ADDRESSES.

The following is a list of useful addresses within BAS12K:

4000H	start of BAS12K.
7354H	end of BAS12K.
4300H	cold start entry point - clears all variables and program, re-initialises.
4303H	warm start entry point - used to re-enter BAS12K without destroying the program or variables.
4507H	low order half of address of machine code routine called by USR.
4508H	high order half of address of machine code routine called by USR.
4527H	address to call to pick up a value passed to a machine code routine by by USR - the value is returned in register pair DE (E low, D high).
452AH	address to call to pass a value back to the program from a m/c routine called by USR - low order half of value should be in register B, high order in A.