tbas MANUAL v. 1.0.beta

by Antonio Maschio
with the great great help of Tom Lake, Bruce Axtens and Ian Jones
and some suggestions by Marcus Cruz

ABSTRACT

Welcome to the world of **tbas**! **tbas** is a powerful console BASIC interpreter, with many statements and many functions, and with the most advanced features for console programming.

This manual is not a BASIC primer; I assume you have the most common knowledge about programming; it's more like a reference manual that helps in using correctly the various statements, functions and their options.

Read also the **tbas** man page for other more general info and the installing instructions.

Readers are encouraged to report all errors and inconsistencies (and all mistakes in the English sentences) found to

ing dot antonio dot maschio at gmail dot com

The **tbas** international team wants to thank you.

14 September 2019

1. Introduction

tbas is a BASIC language interpreter (with an optional built-in interactive session) that reads textual files written in the BASIC language; files may be in any format - UNIX, DOS, Mac. Statements may be written in lower or upper or mixed case letters, since **tbas** is case insensitive. Line numbers are not necessary, and are required only either as labels for the GOTO/GOSUB jumps, or in the interactive session (option -i) or in case you have to run successively your program in a different numbered-lines BASIC interpreter or compiler. It is completed with the famous MAT statements and a large math functions and operators set.

Three things I want to clarify: First: the classic default console running executes statements as they are found in the file (line numbers, if any, are labels, and they are not ordered before execution); so be careful to sort the lines of a classic numbered BASIC program in order to have the right sequence suitable for **tbas**; in the interactive session, instead, like any classic BASIC environment, lines are reordered according to their numeric labels values, following the old tradition of BASIC; so be careful to put the right line numbers.

Second, **tbas** obeys to a fundamental rule:

ONE STATEMENT PER LINE

This means that the colon delimiter to pack several statements in a single line is not available (apart from assignments and comments). You can use the structured features for building complex but clear programs, but remember that each statement must reside on its own line. This rule is still a milestone in the TRUE Basic environment, the successor of the Dartmouth original first version.

Third, **tbas** was made not to build a hyper-fast-super-number-crunching calculator, but for hobby purposes. So take it for what it is.

1.1. **tbas** compilation

1.1.1. Linux/UNIX

Compilation under Linux/UNIX has no problems, provided you have gcc version >= 3.X, and a terminal window (real or emulated) to run **tbas** on.

As root, you can then install it for everyone. If you want to install for yourself only, edit the makefile, and run 'make install' (this operation needs you know what you are doing...).

Read the man page, which contains the options information, some notes about installation and the bibliography upon which **tbas** is based.

1.1.2. Windows and CygWin

tbas can be easily built and executed under CygWin (and CygWin64), provided that gcc-core, make and libreadline-devel have been installed.

After compilation, to run the program from the Windows CMD Prompt, the PATH should be updated to include C:\cygwin\bin. If this is inconvenient, the following files should be copied from C:\cygwin\bin to a directory on the PATH, or into the same folder as the tbas.exe itself, if you don't want to deal with PATH:

* cygwin1.dll
* cygreadline7.dll
* cygncursesw-10.dll

Remember that the readline features of the interactive session are not perfect, due to differences in the Windows API.

Be also aware that some statements, while correctly interpreted, may not work as expected (for instance PIPE).

1.1.3. Windows and other compilers

tbas can be compiled by under different environments, but at your risk. In particular, MinGW misses some headers and some library files. So compilation may fail. If you want to compile under MinGW (or under another environment which fails) and this fails, write me; I will address you to one of the **tbas** team members that can help you do it.

I suggest to use CygWin to compile and run **tbas** under Windows. CygWin is free and it works. What else?

1.2. **tbas** DESIGN

tbas was born as a further development of decb, my former DEC-20 interpreter. The BASIC of the DEC-20, being one of the most used and one of the most complete, was a good starting point.

Functions and statements available in **tbas**, when not classic or derived from standard behaviours, are based on my imagination, but I was inspired by programming languages created by Institutes that with their compilers were capable of building a sort of a "multilayer standard" for the BASIC language; they are mainly: Dartmouth, DEC (aka Digital), OSU, HP (with the HP-BASIC, an evolution of the DEC BASIC) and the CDC. Even the wonderful Microsoft Quick BASIC was useful, because it became a sort of 'standard' in the Eighties and Nineties. Their manuals were consulted while developing **tbas**; probably some other, more or less important, were left out... my fault.

I don't pretend that **tbas** is not limited, in confront with those giants; I only wanted a typical old-times BASIC programmer feeling acquainted with statements and functions, and that her/his memories could fit the **tbas** syntax without big interventions.

I hope I've made it.

2. **tbas** features

In the following, the whole language is explained, feature by feature, statement by statement, function by function, with examples and comments. If you find some errors, well it's my fault. If you find strange English statements, again it's my fault. I'm no way an English mother-tongue, and I miss some English subtleties. In any case, write me, and I will try to better things up. I'm backed by a great international team, after all!

2.1. Conventions

The following conventions will be respected throughout the whole manual:

```
<c>         is a channel stream identifier (1÷9)
<cond>      is a true/false condition (actually nonzero/zero)
<f>         is a file name (a string identifying a file)
<l>         is a label name (a numeric or an alphanumeric string)
<n>         is any number
<nv>        is a numeric variable
<s>         is any string
<statement> is any valid BASIC statement belonging to the tbas dialect
<sv>        is a string variable
<t>         is any array (matrix or vector)
<true>      is a true condition of type <cond>
<v>         is a one-dimensional array (vector)
<var>       is any variable (numeric or string, simple or array)
```

Each convention element may be followed by an index that has the scope to differentiate two or more items. E.g.

```
<n1>, <n2>, ...
```

Characters enclosed in brackets, as in

```
MID$(<s>,<n1>[,<n2>])
```

are optional; if not given, the function result depends on some explained predetermined values.

Brackets may be used even in statements:

```
[DECLARE] SUB
```

This for instance means that, in the SUB statement, the token DECLARE may or not be used (depending on the context), whereas the token SUB must be specified. Usually, the context is specified. If not, write me.

2.2. Syntax rules

The Alpha version of **tbas** and all previous versions didn't take in account the blanks, in a program line, because all blanks were removed before evaluation. This assured great freedom in writing programs, but prevented the usage of some specific token names of the language as variables or subs names. For instance, the following program line

```
FOR I=TOP TO TOP+10
```

led to an error, because the first TO (in the first TOP) was not the keyword **tbas** was looking for.

In the Alpha testing phase, I received requests from users to enable a more strict syntax evaluation; now **tbas** obeys to stricter syntax rules, explained in the following.

The current version of the BASIC language interpreted by **tbas** has four main types of items:

- statements, that begin a line (remember, one statement per line), like the following examples for GOTO, IF and PRINT:

```
GOTO 100
IF A=0 THEN 500
PRINT "HELLO"
```

- commands, that are statements that don't begin a line, like PRINT and GOTO in the following examples:

```
IF A=0 THEN PRINT X
ON A GOTO doprint, doinput, doerr
```

- keywords, that are constituents of a statement but not statements alone, like THEN, AS, TO or STEP in

```
IF A=0 THEN PRINT X
OPEN AS LIBRARY "LIBS"
FOR I=1 TO 10 STEP 2
```

- labels, that are descriptors of the current action, like LIBRARY in the following example:

```
OPEN AS LIBRARY "LIBS"
```

Apart from the BASIC language syntax that applies to each statement (see the relative entries in this manual), some general syntax rules must be respected while writing programs suitable for **tbas**, where 'blank' is either a space or a tab character:

1) The statements DECLARE, SUB, HANDLER, WHILE, UNTIL, IF must be followed by a blank. Also SELECT and DO, if followed by something. No requirements exist for leading blanks, to let the user adopt her/his favorite indentation rules (all blanks preceding a statement are removed before the string evaluation). All remaining statements don't care of being followed by a blank.

2) The keywords THEN (when followed by something), TO, STEP, BY, AS, OTHERWISE, IN must be written surrounded by blanks. IN, when used in a WHEN ERROR IN statement, is not a keyword but a complement of the statement WHEN ERROR, and so it doesn't need to be surrounded by blanks.

3) The commands FOR, GOTO, GOSUB, ELSE, USE, when not used as statements, must also be surrounded by blanks. GOTO and GOSUB may also be written as GO TO and GO SUB: **tbas** knows what to do.

4) labels don't follow any special rule.

If you should find these rules too difficult to remember, follow this rule of thumb:

ALWAYS SEPARATE STATEMENTS, COMMANDS, KEYWORDS and LABELS WITH BLANKS.

tbas on its own can distinguish which blank is really necessary and which is not, and tell you if something is wrong, so that now, with the current version, statements like

```
FOR I=TOP TO TOP+10 STEP TOP
```

or even the weirder

```
FOR I=STEP TO STEP+10 STEP STEP
```

are correctly interpreted.

2.3. Special structures

The following special structures are available.

2.3.1. Consecutive assignments

Assignments of several variables to one same value can be grouped in one consecutive assignment; a statement like

```
A=B=C=D=0
```

is equivalent to

```
A=0
B=0
C=0
D=0
```

The assignment is evaluated right to left; this implies that an assignment like:

```
D(A)=A=0
```

is not equivalent to

```
A=D(A)=0
```

The first sets A=0 and then D(0)=0 (since A=0). The second sets D(A)=0 (whatever A is) and then sets A=0. Use this feature with care if arrays are involved.

If you ever wanted to assign some truth value (e.g. the result Z=0,

which is true if Z is null and false if Z is not null), by writing

```
A=B=C=Z=0
```

you simply set all variables, included Z, to 0 (i.e. to false); the solution is to enclose the test in parentheses:

```
A=B=C=(Z=0)
```

This feature belonged also to the DEC-20 BASIC.

A note is worthwhile here: parentheses are necessary in general, when the expression in the assignment contains one or more equal sign:

```
LET FNP = A %% I = 1
```

would be interpreted as a (wrong) consecutive assignment; the only safe mode is to write:

```
LET FNP = (A %% I = 1)
```

that will be interpreted as a correct single assignment.

2.3.2. Multiple assignments

Multiple assignments and REM comments are the only cases where the colon separator is legal. For instance:

```
a=3 : H2=a : LET h=24 : A=a*h
```

is correct, as well as

```
CLOSE #3: REM END OF WRITING
```

while

```
PRINT : PRINT
```

or even

```
a=3 : PRINT a
```

are not; everything following a legal statement/assignment which is not a legal assignment or a comment is ignored.

This feature makes programs with multiple assignments of values (typically at the very start of the listing) more elegant and even easier to read. In any case, remember the fundamental rule: I think it's a very important issue.

Note: technically, the ': REM' comment, in the pre-parsing phase, is turned into a tick comment; e.g. the previous example is turned to:

```
CLOSE #3 ' REM END OF WRITING
```

The tick comment is obviously ignored during execution; that's the reason why the colon-REM is not a real colon separation.

2.3.3. Left-assignments

Special left-assignment cases (useful from time to time) are available:

LEFT$(<sv>,<n>)="string"

means: change first <n> characters of <sv> with "string"; if <n> is zero, this is a mere rewriting of <sv>="string"+<sv>; if <n> is greater or equal than LEN(<s>), this is a mere rewriting of <sv>="string"; if <n> is lower than zero, it is automatically set to zero, and if greater than LEN(<sv>)+1 it is automatically set to LEN(<sv>)+1.

E.g.

```
A$="WEATHER"
LEFT$(A$,1)="H"
PRINT A$ ' returns HEATHER
```

RIGHT$(<sv>,<n>)="string"

means: change last <n> characters of <sv> with "string"; if <n> is zero, this is a mere rewriting of <sv>=<sv>+"string"; if <n> is greater or equal than LEN(<sv>), this is a mere rewriting of <sv>="string"; if <n> is lower than zero, it is automatically set to zero, and if greater than LEN(<sv>)+1 it is automatically set to LEN(<sv>)+1.

E.g.

```
A$="WEATHER"
RIGHT$(A$,4)="SE"
PRINT A$ ' returns WEASE
```

MID$(<s>,<n1>,<n2>)="string"

means: change <n2> characters of <s> from position <n1> with "string"; if <n2> is zero, the process is equivalent to the insertion of "string" in <s> at position <n1>; if <n1> is zero, it is a mere rewriting of LEFT$(<s>,<n2>)="string"; if <n1> is equal or greater than LEN(<s>), it is a mere rewriting of RIGHT$(<s>,<n2>)="string"; if <n1> is lower or equal than zero, it is automatically set to 1, and if greater of LEN(<s>)+1 it is automatically set to LEN(<s>)+1; if <n2> is lower than zero, it is automatically set to zero, and if greater than LEN(<s>)+1 it is automatically set to LEN(<s>)+1.

E.g.

```
A$="WEATHER"
MID$(A$,3,3)="ST"
PRINT A$ ' returns WESTER
```

*****

The first argument of a left-assignment must be a string variable and not a literal string; in particular, the variable must be a simple variable and not an element of a string array. If the variable is not instantiated, it will be before the composition.

The insertion string may instead be any string, literal or variable and in particular it can be the void string. There is no limit to the insertion string length, provided the total of the result string is not greater than the maximum length of strings, or an error is raised.

The left-assignments are built with the same exact philosophy of the string functions LEFT$(), RIGHT$() and MID$(), but the difference is crucial: the functions return a value (that may be null if length of extraction is 0), while the left-assignments operate an insertion for the characters that would be returned by the correspondent string function, and then copy back the remaining characters of the source string; in practice they are the negative of the correspondent string functions.

Practice with left-assignments: when you have sufficient experience, they could represent a very fast way for string composition.

2.3.4. Comments and text markers

Comments are a useful way of recording text within the program lines (and it should be noted that it's imperative for programmers to comment code for future reference -- even in case it's the same programmer who takes over the listing some time later).

BASIC had a simple way of storing comments since its beginnings in 1964: the REM statement (from the word "Remark"); its purpose is to discard everything that follows REM. REM extends to the end of line.

tbas of course supports REM, as long as other ways of storing information inside the file; REM may be a statement on its own but can also follow any legal statement if preceded by colon; e.g.

```
PRINT "Result": REM comment
```

is available. You must take in account that such inline comments are discarded during the preprocessing phase, while REM statements (that may the object of a GOTO, for instance) are not.

Another type is the tick comment (introduced by the apostrophe ' sign), which has the property of starting everywhere (even after a statement) and extends to the end of line; e.g.:

```
A=B*3 ' set angle value
```

This kind of comment is discarded by the preprocessing phase as well, but the line number, if present, is retained, because this type of comment (like REM) may be the object of a jump; e.g.

```
    GOTO 35
    ...
35 ' start here
```

There is another comment delimiter: the sharp symbol, that must appear as the first not-blank character of the line; this is a typical UNIX shell comment. This and other non-BASIC comments are introduced in the chapter "Special markers".

There is no multi-line comment in **tbas**. I judge too onerous for the execution time to parse a listing for multi-line comments.

2.3.5. The PRAGMA feature

The PRAGMA features is 'hidden' into a REM statement (to make it harmless if the program is run by another compiler/interpreter. It is used to pass to program some console options, with the following features:

1 it is in the form of a REM comment (that is, it is not executed if found in other dialects); also REMARK can be used. Any or no space or tab can be put between the REM, PRAGMA and option elements.

2 it may be followed by any of options -d -N -r -T -H -D in any order, to pass those options without the need to type them from console. No space between the dash and the option. For the meaning and scopes of these options, type '**tbas** -h' from console or type 'man **tbas**' (if **tbas** is installed).

3 multiple occurrences of the REM PRAGMA may be used. The options are activated when the PRAGMA is met.

So to pass a BASIC program to anyone, populate the line where you want to pass the options (preferably the first before any printing code). E.g.

```
REM PRAGMA -H -T -d -D
```

When this line is met, **tbas** will detect the PRAGMA statement and execute the options. There's no need to specify what options should be used in your documentation: **tbas** will use the PRAGMA line for you where you decided this must be done.[1]

2.3.6. Literal string format

Literal strings are always enclosed in double quotes. Inside strings there is no escaping, unless you use the OPTION RAWPRINT feature.

The quotes have no particular repetition available:

A figure like:

```
PRINT ""HELLO""
```

is seen as the printing of "", the variable HELLO and "", so it is equivalent to PRINT HELLO.

---

[1] Option -H in PRAGMA statement executes immediately. So take care of this and if you want to use it, place it before any printing code.
Observe also that, in the interactive session, PRAGMA has no effect, since all options have a command (e.g. ASPECT) or an OPTION counterpart that can be typed from console before RUN.

A figure like:

```
PRINT """HELLO"""
```

is seen as the printing of "", "HELLO" and "", so it is equivalent to PRINT "HELLO".

The case of characters inside a string is always maintained.

2.3.7. Special markers

Special **tbas** text markers help the programmer to isolate part of code during the development of a BASIC program. They are:

```
  #          - ignore current line (UNIX shell comment)
  @          - suspend/re-enable loading a program
' '          - source blanks
```

Note: characters # and @ can be safely used in BASIC programs (in strings, for instance), since in this case they can never appear as the first not-blank character in the line.

Important note: the special markers are not recognized by the interactive interpreter, so that you may use them only in programs run by the shell console (see also the chapter "The interactive session").

IGNORE LINE

If the character '#' is found as the first non-blank character of a line, that line will not be loaded and run at all, and any information there contained won't be part of the BASIC execution. In practice, it is a special marker for 'Ignore Current Line'. This lets you store specific comments to the source, or build BASIC programs that can act as executables; if you have a file named 'script.bas' whose first line (called the 'she-bang' line) run like this:

```
#!/path/to/tbas
```

you have built a sort of bash scrtipt. For example, the default installation places **tbas** into /usr/local/bin, thus the first line of 'script.bas' should run as:

```
#!/usr/local/bin/tbas <opts>
```

where <opts> are all the **tbas** options you want to execute. In order to run the script directly from the console, turn the script into a sort of executable by running the following (once for all):

```
$ chmod +x script.bas
```

The 'she-bang' line is not loaded in memory.

SUSPEND LOADING

If the character '@' is found as the first non-blank character of a line, the file loading is suspended, and this and all following lines

are read and discarded until another line with '@' as the first non-blank character is found (and discarded too), restarting file loading with the next BASIC line. In practice, it is a special marker for "Suspend Loading a Program". This lets you store any free form text within the source file itself, acting as temporary documentation of the source, or it may be useful for isolating parts of code not to be executed temporarily.

E.g.

```
 PRINT "Ian is a good boy"
 PRINT "Ian is bold"
 PRINT "Ian is a butcher"
```

The previous program, if run, produces:

```
 Ian is a good boy
 Ian is bold
 Ian is a butcher
```

If the suspend character is used:

```
 PRINT "Bruce is a good boy"
 @
 PRINT "Bruce is blond"
 @
 PRINT "Bruce is a shoemaker"
```

The output is:

```
 Bruce is a good boy
 Bruce is a shoemaker
```

If the character @ is used once only, the text after it, until the end of file, is simply ignored.

```
 PRINT "Tom is a good boy"
 @
 PRINT "Tom is brown"
 PRINT "Tom is a teacher"
```

The output is:

```
 Tom is a good boy
```

SOURCE BLANKS

A line that begins with a line number, possibly followed by trailing spaces and nothing else, is stored but not executed. Such lines may safely work as target lines for any GOTO/GOSUB.

2.4. Language operators

tbas has many operators used in algebraic expressions and strings evaluation. In the following they are listed and explained in detail.

2.4.1. Math operators

The following math operators are available:

```
  +              - unary operator for positive numbers (optional)
  +              - addition and string concatenation
  &              - string concatenation (alternating form)
  -              - unary operator for negative numbers
  -              - subtraction
  *              - multiplication
  /              - division
  ^              - power (classic form)
  **             - power (alternating form)
  \              - Classic Math integer division
  %%             - Classic Math integer modulus
  >>             - shift right (divide)
  <<             - shift left (multiply)
```

Note: the division is performed by two operators (/ and \) and two functions (IDIV and DIV), but each has a different behaviour:

- the / infix operator returns the result of the float division of the two operands.

- the function IDIV and the \ infix operator return the integer division (operands are turned to integer and then divided, returning an integer value)

- the function DIV returns the integer division.

Note: the modulo is performed by one operator (%%) and two functions (MOD and REMAINDER), but each has a different behaviour (just like the division):

- MOD and %% return the modulo according to the classic definition of division: A/B ==> A=Bq+r, with |r|<|B|, where r may be positive or negative.

- REMAINDER returns the modulo according to the Number Theory definition of division: A/B ==> A=Bq+r, with 0<=r<|B|, where r is always positive.

2.4.2. Relational operators

The following relational operators are available:

```
  =              - equal
  ==             - absolutely equal (string equality)
  |=             - quasi-equal (numerical quasi-equality)
  <> and ><      - not equal
  <= and =<      - lesser than or equal
  <              - lesser than
  >= and =>      - greater than or equal
  >              - greater than
```

Note: the '=' operator for strings compares the strings after they have been trimmed out of their trailing spaces, so that " string " is equal

to "string" (this is the standard behavior of the DEC-20 BASIC); the '==' operator, instead, considers the strings in their totality, so that the previous test would fail.

Note: the '=' operator for numbers checks if two numbers are equal (two numbers are equal when their bit representation is the same) and this is the classic usage of this operator; the '|=' operator for numbers checks if two numbers can be considered quasi-equal because their difference is lower than 10^-6; this difference limit may be changed by means of the statement OPTION DIFFERENCE <n> where <n> defines 10^-n, the new difference limit. If OFF is used, the default value 6.0 is restored. For instance, in default mode, if a=0.0000005 and b=0.0000006 the test returns false, while the quasi-equal operator returns true (because the difference a-b is 1E-7 and thus lesser that 1E-6). This operator lets the user decide how to discard lesser significant digits in floating point comparisons.

2.4.3. Logical operators

Logical operators have a double nature, as symbols and as alphabetic tokens; there is no real difference between the two (in the pre-processing phase, all alphabetic tokens are translated to symbols), but maybe expression are more easily recognized when using alphabetic tokens.

The following infix logical operators are available (all operators are infix, apart for ~ and NOT which are prefix unary operators):

Symbol	Alpha	
~	NOT	- unary prefix negation [not A]
&&	AND	- conjunction [both A and B]
||	OR	- (inclusive) disjunction [at least A or B]
->	IMP	- implication [A implies B]
##	EQV	- equivalence [A implies B and B implies A]
!!	XOR	- exclusive disjunction [either A or B]
{	NOR	- negation of inclusive disjunction [neither A nor B]
}	NAND	- negation of conjunction [not both A and B]

Here are the logic truth tables of the logic infix operators:

		&&	||	||	##	->	{	}
A	B	AND	OR	XOR	EQV	IMP	NOR	NAND
-1	-1	-1	-1	0	-1	-1	0	0
-1	0	0	-1	-1	0	0	0	-1
0	-1	0	-1	-1	0	-1	0	-1
0	0	0	0	0	-1	-1	-1	-1

In the previous table, A and B symbolize the results of comparison tests (usually returning -1 in case of true, and 0 - zero - in case of false); for instance, A is X>0, and B is Y<24.

Here is the logic truth table of the prefix operators ~ and NOT: where A is a truth value resulting from a logical expression. These operators may be used in mathematical formulae and numerical tests, and also in string tests; the following tests hold:

```
 A    ~ NOT
---------
-1      0
 0     -1

IF NOT(B>0) THEN...

IF NOT A$="TRUE" THEN ...
```

It's interesting (at least for me) noting that the AND, OR, NOR and NAND operators reflect a particular and very ancient point of view in logic, dating back to Chrysippus (3rd century B.C.):

1. the OR operator can be also called the "some" operator;
2. the AND operator can be also called the "all" operator (also "both" in this two-states logic world);
3. the NOR operator can be also called the "none" operator;
4. the NAND operator can be also called the "not all" operator (also "not both" in this two-states logic world).

In particular, the NAND operator, neglected in modern logic, can be used to define all the other operators, included the NOT operator, in the hypothesis of a two-way logic (two truth values); for instance, NOT A may be substituted by A NAND A. The full demonstration is not given here, because out of scope.

Logical operators don't follow any precedence over each other; in particular, AND has not a lower or greater precedence over OR; this means you can use parentheses at will if you are uncertain of the field extension; for instance:

```
PRINT 0 OR 0 IMP NOT 0 AND 0
```

is equivalent to

```
PRINT (0 OR 0) IMP (NOT(0) AND 0)
```

and yields -1 (TRUE), because:

```
step 1: 0 OR 0 = 0 (left part)
step 2: NOT 0 = -1 (NOT of right part)
step 3: -1 AND 0 = 0 (right part)
step 4: 0 IMP 0 = -1 (left and right parts),
```

2.4.4. Bitwise operators

The following infix bitwise operators are available:

```
  &     - bitwise and
  |     - bitwise or
```

The bitwise operators act bit-by-bit on the patterns of the two operands (whose integer part is taken), and the result is returned as integer. They don't operate on the truth value represented by the operands; for instance

```
2 | 4
```

yields 6, because the two patterns are (last byte):

```
 2:  0010
 4:  0100
```

and the bit-by-bit matching yields (OR)

```
 0110
```

which is 6.

2.4.5. Operators Priority

Operators follow a precedence order:

1. the minus sign and the NOT operators are evaluated first, as attributes of the term or of the parenthesis group immediately at the right, changing respectively the sign of the term (the minus) or inverting the truth value of the expression (the NOT).
2. anything enclosed in parentheses is evaluated next, where each item runs this precedence order anew)
3. ^ and ** (exponentiation) and shift operators << and >> are evaluated next
4. *, / and \ (multiplication and division) are evaluated next
5. + and - (summation/subtraction) are evaluated next
6. <, >, =, <=, =<, >=, =>, <>, >< (relational operators) are evaluated next
7. AND, OR, XOR, NOR, NAND, & and | (logical and bitwise operators) are evaluated next
8. EQV and IMP are evaluated as last, having the lower precedence (low-level logical operators).

Keep this precedence order in mind when you write algebraic math expression, and whenever in doubt, use parentheses to isolate those parts of the expression that must be calculated first.

2.5. Numbers picturing

In normal printing mode (OPTION RAWPRINT OFF, which is the default state) numbers are always printed with one extra character before - yielding the sign character, which is a minus symbol in case it's negative or a space in case it's positive - and one extra space after, to separate output ahead and behind.

In raw printing mode (OPTION RAWPRINT ON) numbers are printed as-is, that is no extra space is provided, except for the minus symbol when the number is negative (of course), leaving the output to the total control of the programmer.

The comma tabulating character works as expected in both modes.

2.6. Files types

tbas has several statements that deal with files, and some functions that help in that. But to use them properly, first I must describe the fundamental difference between sequential (or textual) files and random

accessed files.

Sequential files

(freely adapted from the LBMAA-A-D DEC document)
Sequential access files are those files that contain information that must be read or written sequentially, one ASCII character after another, from the beginning of the file. A sequential access file is either in write mode or read mode, but cannot be in both modes at the same time. An important distinction to note about sequential access files is that they can be listed in readable form on the user's terminal or line printer. Sequential access files consist of lines that contain data items. A sequential access file can be either a line-numbered file or a not line-numbered file.

The following conventions must be observed when dealing with a sequential access file at the editing level:

1. A line can contain any number of data items separated from one another by at least one space, a comma, or a tab. However, the line must not be longer than 255 characters (counting the numeric label - if present - and its following delimiter, the line content and the carriage return that terminates the line). It is not necessary to have a space, comma, or tab after the last data item on the line. Note that blank tabs are not ignored in a data file as they are in a program.

2. A data item is any numeric constant or string constant. Numeric constants must not contain blanks or tabs, otherwise more than one numeric constant is read. If a string is to contain a blank, comma, or tab, the user must enclose the string in quotes; otherwise it will be read as more than one data item by the statements that read data.

A distinction between sequential access files is whether the file is a pure data file or a text file. A pure data file is used primarily for the storage of data. A text file contains data that is probably destined for output to the line printer, because it is a report, a financial statement, or the like. The user must follow slightly different procedures in her/his program, depending on the type of file he wishes to handle (for example, a string that contains a blank must be enclosed in quotes when it is written in a pure data file, otherwise it will be seen as more than one string when data is read from the file. However, such a string should not be enclosed in quotes when it is written in a text file because text files are not normally read back in a program, and the superfluous quotes would spoil the appearance of the file when it is printed). See the QUOTE statement for more about this.

A sequential file is made of lines of text, each terminated by the End-Of-Line character. Each character is an ASCII character in the range 1-127, but occasionally, for some environment and characters encoding, you can read and print characters in the 128-255 range.

Such files may be safely read, edited, saved and printed with any text editor.

Random access files

(freely adapted from the LBMAA-A-D DEC document)

Random access files are data files that are not necessarily read or written sequentially. The user can read items from or write items in a random access file without having the items followed one after the other. The items in a random access file are not recorded in a form suitable for listing, and therefore cannot be output to the user's terminal or the line printer. A random access file is any way a regular text file that can be copied to the disk.

Random access files, unlike sequential access files, do not distinguish between read mode and write mode. The user can read or write any item in a random access file at any time by first setting a pointer to that item. A random access file contains either string data or numeric data, but not both. Each data item in a random access file takes up the same amount of storage space, called a record, on the disk. BASIC must know the record size for the random access file in order to correctly move the pointer for that file from one data item to another. The record size for a random access numeric file is set by BASIC because the storage space required for a number in such a file is always the same. The storage space required for a string, however, is dependent upon the number of characters in the string. Thus, for a random access string file, the user must specify the number of characters in the longest string in the file so that BASIC can set the record size accordingly. This specification takes place when the file is assigned to a channel. Refer to the description of the FILES, FILE and OPEN statements in next paragraphs.

When creating a new random access string file, if the user specifies too few characters, an error message is issued when a string too long to fit in a record is written. If too many characters are specified for a record, the strings will always fit, but space will be wasted on the disk (though nowadays this is a secondary problem).

When he is dealing with an existing file, the user does not have to specify a record size. If he does specify a record size for an existing file, the record size must match that with which the file was written.

A random access file can easily be read or written in sequential order.

A random access file is a textual file, at all effects, but since it has fixed length fields and all unused characters are filled with zero, the file cannot be read as a sequential file with a text editor. Moreover, there is no line breaks, so a text editor would read a unique huge line of text, with unpredictable results.

This kind of files start with an 8-characters pattern, which begins with the paragraph § character, followed by the code 364 for random access numeric files, or the code 4XXX, for random access string files, where XXX (padded with zeroes) defines the field length for strings (e.g. 4056). The rest of the 8 characters in the introduction pattern is padded with zeroes. After the 8-characters pattern, all fields follow, consecutively, with no pattern pads between any two fields. Each field is padded with zeroes to fill the entire space.

Since the reading of random files is critical, such files should not be altered with a text editor, unless the user knows very well what he's about to do.

Channels

With "channel", in BASIC lingo, I mean the channel number identifying the input/output file stream. **tbas** has 9 channels always available for input or output, for sequential or random access files, numbered 1 to 9. They can be used simultaneously or not.

tbas always controls if a channel is already open when it tries to open it, or if a file is opened twice on different channels, or if wrong operations are performed on an opened channel (for instance, a write operation in read mode).

2.7. BASIC Statements

Here is the complete list of the BASIC statements and their usage context in detail, with examples you are invited to study, expand and execute.

: (colon)

This statement must be used in conjunction with a numeric label, and its purpose is to define an image line containing characters and markers suitable for the USING feature. The image starts at the character immediately following the colon, and runs until the end of line (see IMAGE).

Tick comments are not allowed in an image line, because the string formatter would use the tick by itself.

See the USING statement for details.

E.g.

```
 10 :The name is 'LLLLLL
    g$="Hector"
    PRINT USING 10,g$

 The name is Hector
```

See also the IMAGE statement.

ABORT

See the END statement.

ACCEPT

See the LINPUT statement.

ACCEPT#

See the LINPUT# statement.

APPEND#

APPEND# sets the file in write mode and brings the file pointer to the end-of-file, so that any further printing is appended. When a sequential file is opened by FILES, FILE or OPEN in OUTPUT mode, and the file exists at that time, it is automatically set in append mode and the file pointer is set to the end-of-file; so APPEND# is not strictly necessary, but it is if your file was opened in INPUT mode (not READONLY), and you want to turn it to OUTPUT mode to add some lines to it. Appending to a random access file has no meaning.

The APPEND# statement has the form:

```
 APPEND <c1>, <c2>, ...
```

where the arguments have the form:

```
 [#]N for sequential files
```

The # character is optional, since there is no ambiguity, because APPEND# must be used only with sequential files and N is a channel stream in the range 1÷9.

At least one argument must be present in an APPEND# statement.

E.g.

```
 100 OPEN "Report.txt" for INPUT as #3
 110 OPEN "Summary.txt" for OUTPUT as #6
 120 APPEND #3,6
```

sets both channels 3 and 6 (which were opened as sequential files), to append-mode; file on channel 3 is converted to OUTPUT mode; from now on, everything printed to channels 3 and 6 will be printed after the existing text. Of course, to avoid deletion of the preexisting file, OPEN must not use the NEW flag and SCRATCH# must not be used before APPEND#.

Note that channel 3 is written with the # symbol, while channel 6 is not.

BREAK

This statement causes an interruption of the execution in the most inner level of a cycle (FOR..NEXT, DO..LOOP, WHILE..END WHILE); it is equivalent to the EXIT statement acting on the most inner cycle.

E.g.

```
 FOR ...
    WHILE ...
       IF ... THEN BREAK
    END WHILE
```

```
    ... ' BREAK resumes here
 NEXT
```

BREAK ignores any given argument.

CALL

The statement CALL is used to call a user or system function as if it were a procedure. Its proper scope is obtaining the side-effects of a function (for instance the printing of some values/states or the setting of variables/constants) without having to build an expression for this. E.g.

```
 SUB CALC(X) ' THIS IS A FUNCTION!
    RETURN  -1      IF  X  <  0  THEN          PRINT  "IRREGULAR  VALUE"
        EXIT SUB     END IF     RETURN X*X
 END SUB
```

Since CALC() is a function, it's not directly executable, and it must be used in expressions only. But if the user's scope is to test some value, the execution of

```
 CALL CALC(T)
```

would print the error message in case T is not a regular accepted value; the return value is in any case ignored.

Of course, if CALL is called upon a user procedure, the total effect is merely to pass control to what follows.

CALL cannot be invoked with system user functions or statements. The reason, beyond the technical matter, is that system functions have no side-effects, and statements can be called directly. E.g.

```
 CALL PRINT "Hello!"
```

may be cute, but it is simply substituted by

```
 PRINT "Hello!"
```

which is more... serious?

CASE

See the SELECT statement.

CASE DEFAULT

See the SELECT statement.

CASE ELSE

See the SELECT statement.

CAUSE ERROR

This statement raises an error condition and calls the error-printing routine, that in turn will print the error message or activate an error-treatment condition (see ON ERROR and WHEN ERROR). Syntax

```
CAUSE ERROR
CAUSE ERROR 0
CAUSE ERROR <n>
```

E.g.

```
CAUSE ERROR 44
```

- If the argument is zero or absent (first two clauses), CAUSE ERROR raises error condition 0, a default error condition not related to BASIC, but with its proper error message. This 0-condition is safe, because it does not appear during the usage of any BASIC statement.

- If the integer part of the argument is given and is not zero (third clause), CAUSE ERROR raises the error condition corresponding to that error number (at all effects, it's like the error condition really occurred), as if that error really appeared.

CAUSE ERROR is a tool for the design of error system treatment routines, because these latter can be tested without raising the effective error (for instance, an error with files, which can be troublesome).

If the argument <n> is not in the range of proper error messages (from 0 to ERRLIM, defined in errors.h), it is resized accordingly, without any error or warning message.

CHAIN

The CHAIN statement transfers the control to the program whose name is given as argument, which is loaded and parsed; the name must be enclosed in quotes, or contained in a string variable, but in any case it must contain the whole path if the CHAINed program is not in the same path of **tbas**. Syntax:

```
CHAIN <f>[,<n>]
```

where <f> is the string identifying the file (with path). The CHAINed file name may be followed by a label <n>, separated by comma; this label must exist in the called program and, in this case, the called program will start from that label (and not from the beginning); I underline here that this number is not the physical line number, but it's a numerical line label. Thus:

```
CHAIN "gross",100
```

will start file "gross.bas" from label 100, but this line is not

necessarily the 100th line of the text.

When CHAIN executes the called file, by default it resets all variables; that is no variables can be passed forward to the called program; this is the default behaviour of the DEC-20 BASIC. But **tbas** has an option that overrides this:

```
OPTION RESET OFF
```

When used, variables (arrays included) are not erased, and the called program inherits all of them. This statement must be put in the first program of the chain that needs to pass variables through; variables are passed through the whole chain, until the end of last program or an OPTION RESET ON is met.

The COMMON statement does a similar task, but selects variables to pass. See COMMON for details.

A final remark, here: the CHAIN statement is a primitive statement, and there is no way to get back to the caller, unless you use another CHAIN. In any case, beware of circular calls, not forbidden by **tbas**, but dangerous because they can lead to infinite cycles and/or memory waste.

CHANGE

The CHANGE statement converts a string to ASCII codes or vice versa. The syntax is:

```
CHANGE <s> TO <v>
```

where <s> is any valid string (literal, string variable, string array element, string expression), and <v> is a numerical one-dimensional array larger enough to host the string length, or

```
CHANGE <v> TO <sv>
```

where <v> is a numerical one-dimensional array properly formatted containing data suitable for CHANGE and <sv> is a string variable (simple or array element).

The CHANGE statement works in two ways:

- to obtain the individual characters in a string and reverse its ASCII values to a one-dimensional vector
- to convert a one-dimensional vector built according to the CHANGE rules to a proper string (in a string variable)

The vector used by CHANGE is formatted as follows:

- in position 0 the length L of the string;
- in position 1 to L the ASCII values of the characters of the string;

whenever the vector is longer than the string, the positions in excess are all set to 0 (they won't be part of the resulting string, anyway).

Note: since a vector is not declared automatically (any subscripted array is set as a two dimensional matrix if found before dimensioning), it must be declared before use.

E.g.

```
 DIMENSION G(20)
 CHANGE "GOOFY" TO G
 FOR I=0 TO 5
    PRINT G(I),
 NEXT
 PRINT
 G(2)=ASC(L)
 G(4)=ASC(R)
 CHANGE G TO GO$
 PRINT GO$
```

yields the following output:

```
  5             71            79            79            70
  89
 GLORY
```

If the array is too short for the string, an error message is printed:

```
 DIMENSION G(2)
 ...

 ? No room for string in line 2.
 CHANGE "GOOFY" TO G
                   ^
```

CLEAR

The CLEAR statement is used to clear the screen. Depending on the system, if CLEAR is used on a Unix console, it is equivalent to the shell command 'clear' and takes no arguments, and if used on the Windows® propmt window is roughly equivalent to the command 'cls'. On terminal apps the clearing is not a real clearing, but an insertion of void lines until the screen appear empty. By scrolling up the window, the previous content of the screen is shown again. On real terminal this is not possible, so the clearing is effective.

The CLEAR statement does not reset the text colors set by COLOR.

CLOSE#/CLOSE:

The CLOSE statement closes the channels specified in the argument line; if the file was not opened, a warning is printed. Syntax

```
 CLOSE <c1>, <c2>, ...
```

where <c1>, <c2> are channel stream in the range 1÷9.

CLOSE may be used with the suffixes # and :, but they are only adornments. The CLOSE statement proper function needs the channel number only, which may be given as a pure number; no control is done to see if the suffix matches the real file state (e.g. using CLOSE# on a random access file). Besides, CLOSE does not check if a random access file is numeric or string.

E.g.

```
CLOSE #1,3,:7
```

closes channel 1, 3 and 7 (the pictures suggest that channel 1 and 3 were sequential, while channel 7 was random, but, as stated before, this may not reflect the real case, because CLOSE acts on the channel number only).

A note is worthy here: CLOSE is not really necessary, if the program gets to an end, because **tbas**, as a safety measure, closes all the opened files before quitting, and thus all opened channels (and file connections) are safely and automatically closed, even in case of interrupts and abrupt stops. CLOSE is necessary only when you have to reopen the channel with another file connected. In any case, I suggest to CLOSE all opened files, because other dialects may not be so tolerant, in case you have to run your programs elsewhere, and this is anyway a good programming style.

COLOR

The COLOR statement sets the screen text color to the RGB values given as arguments, as in the example:

```
COLOR red,green,blue
```

The colors values range from 0 (absence) to 255 (full color). They are resized automatically to the bound limits if beyond. The RGB code let the user to define more than 16 million colors (virtually any color).

Some strings can be used to define color constants; they are:

```
NAME       RGB VALUES
----------------------
BLACK      0,0,0
GRAY       190,190,190 (also as GREY)
DARKGRAY   49,79,79
RED        255,0,0
DARKGREEN  0,128,0
GREEN      50,205,50
YELLOW     255,255,0
BLUE       0,0,255
LIGHTBLUE  70,130,180
MAGENTA    255,0,255
CYAN       0,255,255
WHITE      255,255,255
```

as in the following example:

```
COLOR RED
PRINT "I'm RED!"
```

The modifier REVERSE will set the inversion of colors between foreground and background; the modifier ADJUST will restore current foreground and background:

```
COLOR RED
COLOR REVERSE
PRINT "I'm RED-Reversed!"
COLOR ADJUST
PRINT "I'm RED-Only"
```

If no arguments are given, the COLOR statement alone will reset colors to default.

The CLS statement, in addition to clearing the screen, resets also the color to the default states (foreground and background). Use CLEAR if you wish to clear the screen and maintain the color states you have set.

Remember that, in any case, when exiting, **tbas** will reset colors to the default terminal state.

CLOSE:

See the CLOSE# statement.

CLS

The CLS statement is used to clear the screen. It's a different thing than CLEAR, which uses standard shell commands. CLS also resets the screen coordinates to (1,1) and reset the text colors states (foreground and background) to default.

COMMAND

The COMMAND statement sends the argument string that follows (any format) to the underlying shell as a shell command. Synopsis:

```
COMMAND <s>
COMMAND (<s>)
```

where <s> is any legal string (literal, string variable, string array element, string expression); parentheses are optional.

Note: this is a potentially dangerous command, since no control is done upon the argument string, and thus this can lead to problems or user system errors. It must be used with care.

COMMON

The COMMON statement is used to pass variables to a chained program; COMMON is followed by the list of variables you want to pass, separated by comma or semicolon; syntax:

```
 COMMON <var>, <var>, ...
```

where <var> terms are an unordered list of variables (simple or arrays).

E.g.

```
 COMMON A7,name$;C2,array(),E$
```

Arrays must be specified with the () extension. If you want to pass all variables, use

```
 COMMON ALL
```

(which corresponds to OPTION RESET OFF) and if you want to remove all previous COMMON declarations, use

```
 COMMON NONE
 COMMON CLEAR
```

(which are equivalent and perform the same task of OPTION RESET ON).

A note is worthwhile here. COMMON declarations are retained through the chain; if for instance program A sets variable name$ as COMMON, and then calls program B by CHAIN, program B will of course see name$; if program B on its turn calls program C by CHAIN, program C will see name$ as well.

E.g.

```
 100 COMMON A,D()
 110 CHAIN "PROG3"
```

If you have a variable called ALL, and you need to pass it through CHAIN, the call

```
 COMMON ALL
```

is interpreted not as "make variable ALL common" but as "make all variables common". Thus, to pass the single ALL variable, put it not as first in the list by adding a comma:

```
 COMMON ,ALL
```

The same for NONE or CLEAR.

CONST/CONSTANT

The CONST statement (which may be also typed as CONSTANT) is used to declare constants, that is elements that cannot change their content. Their names may be of any length up to 56; they must begin with a letter or the underscore and may be followed by letters, numbers, underscores, dots or question marks; names ending with $ identify string constants; legal names are:

```
a, i4$, Gross_total, title$, _w
```

The case of the letter does not count: 'c' and 'C' are the same constant.

Once declared, constants are unmodifiable; they can be used for storing special values that do not change for the whole program. Since assignments to constants is impossible, their instantiation must be given on the declaration line:

```
CONST <nv> =<n> [AS INTEGER], ...
CONST <sv> =<s>, ...
```

where <nv> is any legal name for a numeric constant (not ending in $), <sv> is any legal name for a string constant (ending in $), <n> is any numerical expression (made of literal numbers, functions, variables and array terms), while <s> is any string expression (made of literal strings, string functions, variables and string array terms). An error is raised in case the assignment is missing or invalid.

A note is worthwhile here: you can duplicate constants:

```
CONST name=24, name=48
```

This statement is perfectly legal, but you must be aware that the second name is the first encountered in the search, and thus the statement:

```
PRINT name
```

prints 48. The first 'name' instance is, at all effects, unreachable until the end of program, or until you execute

```
ERASE name
```

In this case the hidden 'name' constant becomes visible again.

When used inside a SUB, constants declared for the first time are local, and are destroyed when the SUB exits, unless the EXPORT statement is used.

The AS INTEGER specification lets the numerical constant to store an integer values (see DECLARE for further notes about this).

The AS INTEGER specification may be avoided by appending the % character to the name:

```
CONSTANT i%
```

In this case, i% is declared (implicitly) as an integer constant. A note is worthwhile, here: the % character is part of the name, so that 'i' and 'i%' are two different constants.

CONSTANT

See the CONST statement.

CONTINUE (without arguments)

The CONTINUE statement provides a useful way in the WHEN ERROR error treatment routines (USE or HANDLER section - see the WHEN ERROR statements) to pass back the program control to the line following the one that caused the error condition or where the user interrupt occurred. It is the RESUME equivalent of the ON ERROR routines.

CONTINUE without arguments must be used only in a WHEN ERROR handler. See the WHEN ERROR statements for further details.

CONTINUE (with arguments)

The CONTINUE statement for loops and decisional structures (with an argument) works in a slightly different way than EXIT; its purpose is to jump to the end of the outer cycle in the list of its arguments (just as in EXIT), to restart it over. An example is thousand words:

```
WHILE ...
   FOR ...
       DO ...
           ...
           IF <cond> THEN
               CONTINUE FOR
           END IF
       LOOP
   NEXT ...' CONTINUE resumes here
WEND
```

When CONTINUE FOR is met, it forces the interpreter to jump to NEXT, to perform the next FOR iteration (or stop if limit is reached) and to unload all the inner cycles (e.g. DO) and decision structures (e.g. IF).

The number of arguments of CONTINUE must be all in the reverse order with respect to the open cycles or, if unambiguous, the last only suffices.

COPY

The COPY statement copies the first argument array onto the second. Syntax:

```
COPY <nv1> TO <nv2> [IN <n>]
COPY <sv1> TO <sv2> [IN <n>]
```

When the first array is a vector, and the second a table, the copy happens in the column specified by the optional IN keyword; if no IN keyword is used, the column 0 is used as the destination.

The first array must have any of its dimensions lower or equal to the correpondent dimension of the second array. If lower, the remaining values in the destination array are left untouched.

The following program shows how COPY is used:

```
 DECLARE VEC(2), ARR(3,3), B(4,4)
 DECLARE I,J
 MAT ARR=CON     ' set columns 1-3 as 'ones'
 VEC(0)=24   ' some scattered values for the vector
 VEC(1)=48
 VEC(2)=96
 COPY ARR TO B
 FOR I=0 TO 4
    FOR J=0 TO 4
        PRINT B(I,J),
    NEXT J
    PRINT
 NEXT I
 PRINT
 COPY VEC TO ARR IN 2
 FOR I=0 TO 3
    FOR J=0 TO 3
        PRINT ARR(I,J),
    NEXT J
    PRINT
 NEXT I
```

```
  0             0             0             0             0
  0             1             1             1             0
  0             1             1             1             0
  0             1             1             1             0
  0             0             0             0             0

  0             0             24            0
  0             1             48            1
  0             1             96            1
  0             1             1             1
```

If the two arrays A and B have indentical row and column dimensions, the two statements:

```
 COPY A TO B
```

and

```
 MAT B=A
```

perform the same task. COPY has a more general usage, while MAT perform a redefinition of the destination array when possible (conforming to the

DEC protocol). Moreover, COPY uses the full dimension range starting from base 0, while MAT starts from 1.

DATA

(freely adapted from the LBMAA-A-D DEC document)
The DATA statement is used to store information in the source code. The READ statement can be used to assign to the listed variables those values which are obtained from a DATA statement. Neither statement is used without the other.

Before the program is run, the interpreter takes all of the DATA statements in the order they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data engine supplies the next available datum.

There are two independent data blocks: numeric and string, each item is separated by comma and each data block has its own counter. Thus, data may be any type of number or strings, under the following laws:

- strings may be written as 'naked strings', separated by commas (but they must begin with a letter, can include spaces and cannot contain the comma character itself) and as 'quoted strings', enclosed by double quotes "" and which may contain any character in any order (included the comma); note: the space does not separate naked strings.

- numbers may be written as 'literal' numbers, separated by commas (and including the sign, the E exponent and the dot) and as 'formulas', provided they begin with a number and not with a letter (or they will be taken as naked strings).

E.g.

```
 DATA THIS,IS,A NAKED,STRINGS,LIST
 DATA "This is a quoted string, enclosed in quotes","This also"
 DATA -1,0,1E3,1.2E-4
 DATA 1.2*4.567*3.1415,1*sqr(2)
```

Notice that the string "A NAKED" is considered a unique string and that last formula, which is sqr(2), is written as "1 * str(2)" in order to begin it with a number.

If you have any strings beginning with numbers, like for instance 12MONTHS, you must enclose it in quotes.

The location of DATA statements is arbitrary, as long as they occur in the correct order. A common practice is to collect all DATA statements and gather them in the same place, before any running statement or just before the END statement.

If the data engine runs out of data, the program emits the classic OUT OF DATA error message. See the NO DATA statement about this subject and how to circumvent this effect.

The following is a very personal observation: the DATA statement is unique among programming languages, and has shown a very great power to store information within the source, while modern languages tend to differentiate between data and code. This is good, but to a boy who is learning how to compute this or that, the DATA statement provides a very practical way to write down data items once and let the program use them without the need to INPUT or build external files with data in them. Believe me when I say that DATA and READ are the first statements that I learnt.

DECLARE/DIM/DIMENSION

The DECLARE statement (which may be also typed as DIM and DIMENSION) is used to declare variables and arrays. Names may be of any length up 56; they must begin with a letter or the underscore and may be followed by letters, numbers, underscores, dots or question marks; names ending with $ identify string terms; legal names are:

```
 a, i4$, Gross_total, title$, _w, done?, first.score
```

The case of the letter does not count: 'a' and 'A' mean the same variable.

Variable and array types are dynamic-size strings and string arrays (up to 255 characters at most) and integer/floating-point numbers and numerical arrays; they must be declared only if OPTION EXPLICIT is ON; if no instantiation is provided, strings are set to the null string and numbers are all set to 0 (zero); the statement for declaring variables and arrays is DECLARE or DIM follow the syntax:

```
 DECLARE <nv> [=<n>] [AS INTEGER|AS REAL], ...
 DECLARE <sv> [=<s>] [AS STRING], ...
 DECLARE <nv>([<n1>[,<n2>]]) [AS INTEGER|AS REAL], ...
 DECLARE <sv>([<n1>[<n2>]]) [AS STRING], ...

 DIM <nv> [=<n>] [AS INTEGER|AS REAL], ...
 DIM <sv> [=<s>] [AS STRING], ...
 DIM <nv>([<n1>[,<n2>]]) [AS INTEGER|AS REAL], ...
 DIM <sv>([<n1>[,<n2>]]) [AS STRING], ...
```

where <nv> is any legal name for numeric variable (not ending in $), <sv> is any legal name for string terms (ending in $), <n> is any numerical expression (made of literal numbers, variables and array terms), while <s> is any string expression (made of literal strings, variables and string array terms). <n1> and <n2> are integer values that define the dimension of the array (1 or 2 dimensions).

If the array is instantiated without specifying its dimensions:

```
 DECLARE arr()
```

a matrix with dimension 11x11 is created (index 0 to 10 for both dimensions).

A note is worthwhile here: you can duplicate names:

```
DIM name=24, name=48
```

This statement is perfectly legal, but you must be aware that the second name is the first encountered in the search, and thus the statement:

```
PRINT name
```

prints 48. The first 'name' instance is, at all effects, unreachable until the end of program, or until you execute

```
ERASE name
```

In this case the hidden 'name' variable yielding 24 becomes visible again.

When used inside a SUB, variables declared or used for the first time are local, and are destroyed when the SUB exits unless the EXPORT statement is used.

The AS INTEGER specification sets the numerical variable to store and yield only integer values; at present integers are not really integers, but an integer rendition of the value in memory. Due to the double C format, integers may maintain their properties and full digits only if lower than 999,999,999,999; greater values are likely to loose some decimals, while the magnitude is retained.

The AS INTEGER specification may be avoided if the % character is appended to the name:

```
DECLARE i%
```

In this case, i% is declared (implicitly) as an integer. A note is worthwhile, here: the % character is part of the name, so that 'i' and 'i%' are two different variables (in particular, if 'i' is declared by DECLARE i AS INTEGER, both are integer, but the second is more readable in the source code, and moreover, may be easily used in old programs that use the % specification without the need to declare these variables).

NOTE: The AS STRING and AS REAL specifier may be added respectively to a string or non-integer declaration, without loss of generality: these specifier are basically ignored, but **tbas** controls if AS STRING is used with a numerical variable or vice versa if AS INTEGER/AS REAL are used with a string variable. Please note that AS STRING and AS REAL are optional and can be safely ignored: they are added to the language syntax only for making the declarations clearer, if needed.

Using vectors and matrices

The matrix features are historically one of the more ancient features of the BASIC language, surprisingly conceived in version II (1964) of the Dartmouth BASIC, as a Cardboard add-on to the default BASIC compiler.

tbas offers the same flexibility of its ancestor, with these characteristics:

- undefined arrays called before initialization are pre-dimensioned 10x10, that is as 11x11 elements matrices (indices range for MAT statements runs from 1 to 10 for each dimension). This implicit dimensioning appeared as-is in 1964 version of the Dartmouth BASIC, and is disliked by many structured programmers. If you feel to, use OPTION EXPLICIT to force the dimensioning of all the vectors, matrices and variables you need. Note that all vectors must be always declared because, if not, they'd be implicitly dimensioned as matrices.

- vectors (unidimensional arrays) are declared as

```
DIM <var>(<n>)
```

and are interpreted and printed as vertical arrays. These are the kind of vectors that can be multiplied with a matrix (through the MAT * statement).

- horizontal vectors (unidimensional arrays) are declared as

```
DIM <var>(0,<n>)
```

and are interpreted and printed as horizontal arrays. A note is worthwhile here: if you don't dimension a horizontal array and start calling one of its elements as, e.g., A(0,4), **tbas** won't dimension a horizontal array but an 11x11 matrix. . Note also that horizontal arrays are accessible even in case of OPTION BASE 1.

- matrices (bidimensional arrays) are declared as

```
DIM <var>(<n1>,<n2>)
```

- indexes start from 0 if elements are accessed through variables calls, and start from 1 if elements are accessed through the MAT statements. Thus, if OPTION BASE 1 is used, the MAT statements are not affected, but the individual array elements calls are.

If a matrix is referenced by one value only - e.g. say an element of matrix F(10,10) is referenced by F(3) - **tbas** interprets it as F(0,3), and operates on the correspondent value; the matrix is *not* transformed in a vector. This is not the case of horizontal vectors, which must be referenced by the both of their indexes (the first always null) or an error will be raised.

DEF FN

(freely adapted from the LBMAA-A-D DEC document)

The construct DEF FN lets the user define (in a very old-style way) 26+26 numeric functions. The name of the defined function must be three letters, the first two of which are FN; e.g. FNA, FNb, FNZ. Each DEF statement introduces a single function definition, and FNA is a different function than FNa (notice the upper and lower characters). For example, if you repeatedly use the function EXP(-X^2)+5, introduce this function by the following:

```
DEF FNE(X)=EXP(-X^2)+5
```

and call for various values of the function by FNE(.1), FNE(3.45), FNE(A+2) and so forth. This statement saves a great deal of time when you need values of the function for a number of different values of the variable (I must underline here that the SUB statement is a proper enhanced substitute for DEF, which is maintained for backward compatibility).

The DEF statement may occur anywhere in the program, and the expression to the right of the equal sign may be any formula that fits on one line. It may include any combination of other functions, also some defined by other DEF statements (and even itself in a recursion).

The variables in the argument list (X in the example) are to be considered local to the function; the global variables corresponding to the local ones are retained and are made available out of the DEF calculation. All other global variables mean themselves in the calculation.

As in the following examples, each defined function may have zero, one, two or more numeric or string variables:

```
10  DEF FNB(name$,Y)=3*len(name$)*Y-Y^3
105 DEF FNC(X,Y,Z,W)=FNB(X,Y)/FNB(Z,W)
530 DEF FNA=3.1415*R^2
```

In the definition of FNA, the current value of R is used when FNA occurs. Similarly, if FNR is defined by the following:

```
70  DEF FNR(X)=SQR(2+LOG(X)-EXP(Y*Z)/(X+SIN(2*Z)))
```

you can ask for FNR(2.7) and give new values to Y and Z before the next use of FNR, because they are global.

The method of having multiple line DEFs is illustrated by the function shown below (that implements the MAX function):

```
10  DEF FNM(X,Y)
20  LET FNM=X
30  IF Y<=X THEN 50
40  LET FNM=Y
50  FNEND
```

The absence of the equal sign (=) in line 10 indicates that this is a multiple line DEF. In line 50, FNEND terminates the definition, The pseudo-variable FNM serves as a temporary variable holding the result of computation. The lines between DEF and FNEND (included) are not executed on run-time but only if the FNM() function is invoked..

Multiple line DEFs may not be nested, and jumps outside the range DEF-FNEND are forbidden (and raise an error); also you cannot jump into a multiline DEF-FNEND.

GOSUB and RETURN statements are not allowed in multiple line DEFs.

Note: even if string arguments are accepted in a DEF FN structure, the numerical nature of DEF FN does not change, and the result is always a numerical value. If you need a string result, use a SUB string structure, which is more versatile.

DIM

See the DECLARE statement.

DIMENSION

See the DECLARE statement.

DO UNTIL..LOOP

The construct DO UNTIL/EXIT/LOOP is not classical in BASIC, but it's widely accepted in all modern BASICs. **tbas** accepts the following syntax:

```
DO UNTIL <cond>
   ...
   ... EXIT DO
   ...
LOOP
```

In this form, the 'do until-loop' cycle is perfectly equivalent to the 'while-end while' cycle with the condition negated (NOT <cond>). The 'do until-loop' cycle is characteristic because it's never executed if the condition is true from the start.

DO WHILE..LOOP

The construct DO WHILE/EXIT/LOOP is not classical in BASIC, but it's widely accepted in all modern BASICs. **tbas** accepts the following syntax:

```
DO WHILE <cond>
   ...
   ... EXIT DO
   ...
LOOP
```

In this form, the 'do while-loop' cycle is perfectly equivalent to the 'while-end while' cycle. The 'do while-loop' cycle is characteristic because it's never executed if the condition is false from the start.

DO..LOOP UNTIL

The construct DO/EXIT/LOOP UNTIL is not classical in BASIC, but it's widely accepted in all modern BASICs. **tbas** accepts the following syntax:

```
DO
   ...
   ... EXIT DO
   ...
```

```
LOOP UNTIL <cond>
```

The 'do-loop until' cycle is characteristic because it's executed at least once, because the check is done at the end.

DO..LOOP WHILE

The construct DO/EXIT/LOOP WHILE is not classical in BASIC, but it's widely accepted in all modern BASICs. **tbas** accepts the following syntax:

```
DO
   ...
   ... EXIT DO
   ...
LOOP WHILE <cond>
```

The 'do-loop while' cycle is characteristic because it's executed at least once, because the check is done at the end.

ELSE

See the IF..THEN statement.

ELSE IF..THEN

See the IF..THEN statement. As a side note, ELSE IF may also be written ELSEIF, for compatibility with other environments.

END/STOP/SYSTEM

The termination of a program may happen anywhere, when the END statement is found, and is implicit when the listing gets to the end; in this case, the END statement is optional.

ABORT, STOP and SYSTEM perform a similar action; the difference in using END, ABORT, STOP or SYSTEM is here summarized:

- END means "regular termination"; it returns the code for EXIT with SUCCESS.
- ABORT means "anticipated irregular termination"; it returns the code 4 (EXIT with ABORT in **tbas** lingo). If followed by a string, this string is printed before exiting (it may be used to precise the error condition, for example).
- STOP means "anticipated termination"; it returns the code 2 (EXIT with STOP in **tbas** lingo: 2 is not EXIT with SUCCESS, but the end is regular).
- SYSTEM corresponds to STOP, but it prints also "READY." on the terminal, signifying (in BASIC fashion) that **tbas** has ended its duty and returns control to the Operating System. Isn't it cute?

ABORT, STOP and END may be typed in the interactive session, while SYSTEM cannot, because there is a session command SYSTEM with the same name that is executed in its place (and terminates the session).

END HANDLER

See the HANDLER statement.

END IF

See the IF..THEN statement.

END SELECT

See the SELECT statement.

END SUB

See the SUB statement.

END WHEN

See the WHEN ERROR IN and the WHEN ERROR USE statements.

END WHILE

See the WHILE statement.

ENTER

See the INPUT statement.

ERASE

The ERASE statement is used to erase arrays and variables from memory, with the following syntax:

```
 ERASE <var>, <var>, ...
```

where the <var> terms are an unordered list of variables (simple or arrays).

Its general action is to erase references for variables and arrays, and to free arrays memory. After ERASE, the variables in the list are at all effect not existing, and if the OPTION EXPLICIT statement is used, their name cannot be used again, or an error is raised.

EXEC

The EXEC statement executes the string arguments that follows, assuming they are a BASIC statements; a statement executed by EXEC must end its duty on a single line; The syntax is:

```
 EXEC <s1>,<s2>, ...
```

where <s1>, <s2>, ... are string arguments.

Its main duty is to execute the COMMAND$ string, as input from the user after the file name in console mode, or after '--' in the interactive session under the RUN command, or execute program lines read from a sequential file. Example:

```
REM THIS PROGRAM 'EXECUTES' FIRST LINE OF PROGRAM INSTR
OPEN "INSTR" AS INPUT
LINEINPUT#1,A$
EXEC A$
```

EXIT

Sometimes, it is desirable to exit from a cycle before its natural ending (for instance when a specific condition is satisfied); in these cases, the EXIT statement is a great tool.

Its syntax is very simple: it is followed by the keywords DO, WHILE or FOR in a list separated by commas or semicolon. The specified keywords cause the interruption of the most inner cycle of the declared type. An example:

```
FOR ...
   WHILE ...
       WHILE ...
           DO ...
                ...
                IF <cond> THEN EXIT WHILE
           LOOP
       END WHILE
   END WHILE    ' EXIT resumes here
NEXT ...
```

The EXIT WHILE statement, for instance, cause the program to resume to the line just after the inner END WHILE, and to close the DO/LOOP inner cycle, as long as all the IF and SELECT decision structures that were open in the meantime.

If there are two cycles with the same name, to exit from the outer, it is sufficient to specify both in the EXIT line:

```
            IF <cond> THEN EXIT WHILE, WHILE
```

The first FOR will jump out of inner WHILE, the second one cycle outer, no matter what cycles and decision structures are in the middle.

To follow other BASICs syntax, you can write in inverse order the closing cycles: this is of no importance for **tbas**; for instance:

```
WHILE ...
   FOR ...
       DO ...
           ...
           IF <cond> THEN
               EXIT DO;FOR;WHILE
```

```
            END IF
        LOOP
    NEXT ...
END WHILE
```

This syntax is perfectly equivalent to

```
                EXIT WHILE
```

but it makes the program compatible with other interpreters and compilers; anyway, **tbas** will recognize all wrong sequences or the attempt to unload some cycle that was not open.

See also BREAK.

ERRPRINT

The ERRPRINT statement is like PRINT, with the difference all the output is redirected to the standard error channel rather than the standard output.

Unlike OPTION ERRORSTREAM, which modifies the default output for all subsequent statements until the end of the program, or until an OPTION ERRORSTREAM OFF is used, ERRPRINT has effect only on the current print arguments.

NOTE: If ERRPRINT is used on a file channel, like in ERRPRINT#1, it behaves like PRINT#1, because the printing on file is not redirected to stderr. If a channel is opened as the standard screen output, it is not redirected as well; the redirection is established only for channel 0 which is the console output channel.

EXIT HANDLER|EXIT WHEN

The EXIT HANDLER statement serves as a quick exit from a WHEN ERROR section, both in the IN or USE version. The effect of EXIT HANDLER is to jump immediately to the statement that follows the WHEN ERROR clause set, properly closing it.

If EXIT HANDLER is used out of a WHEN ERROR structure, an error is raised.

See the WHEN ERROR statement for further details.

EXIT SUB

See the SUB statement.

EXIT WHEN

The EXIT WHEN statement serves as a quick exit from a WHEN ERROR IN section. The effect of EXIT WHEN is to jump immediately to the END WHEN statement, and properly close the WHEN ERROR IN section.

EXIT WHEN has the same effect of EXIT HANDLER, but EXIT WHEN may be used only in the USE section of WHEN ERROR IN structures, while EXIT HANDLER may be used in any handler (inner or outer).

See the EXIT HANDLER statement for further details.

EXPORT

The EXPORT statement makes variables/arrays of a SUB routine available to the SUB calling section. Syntax:

```
EXPORT <var>, <var>, ...
```

where <var> terms are an unordered list of variables (simple or array).

When a SUB routine starts, it inherits the variables and the arrays spaces of the calling entity, and the SUB can use these values at need, since they are considered 'global'; every new variable declared or used in the body of the SUB is considered 'local'; when the SUB ends, all local variables are deleted; if you want to pass these local variables back to the calling entity, you have to use the EXPORT statement, which reads the variables in the comma-separated list that follows and marks them as exportable.

E.g.:

```
EXPORT a,b$,c(),d$()
```

Arrays are exported as whole, and they must be marked with the '()' token afterwards (or they will be interpreted as variables). If you try to EXPORT a global variable, nothing happens (these variable needs no exporting); if you try to EXPORT an undeclared variable or array, this variable/array is created empty and exported.

The EXPORT statement has a one-level effect; look at this example:

```
sub high(F)
       export K
       return F*F
       K=3
end sub

sub low(H)
       J=2*high(H)
   print "G=";G,"J=";J,"K=";K
end sub

rem MAIN
low(4)s
print "G=";G,"J=";J,"K=";K
```

The 'low' subroutine is called by the MAIN section; it calls the subroutine 'high' on its turn; by calling 'high', 'low' receives the K variable that 'high' EXPORTs, but this K will not be passed back to MAIN,

when low terminates.

The output is:

```
 G= 0          J= 32         K= 3
 G= 0          J= 0          K= 0
```

Notice here that G is created local in SUB 'low' (because 'low' is called before the first G instance is found in MAIN), destroyed, and recreated in the main body as global; J is a local variable that is not passed back to main; this is recreated by main along with G and K when the last line in executed.

The statement

```
 EXPORT ALL
```

may be used to export all local variables (numerical and string variables, numerical and string arrays).

If you have a variable called ALL, and you need to pass it through EXPORT, the call

```
 EXPORT ALL
```

is interpreted not as "export variable ALL" but as "export all variables". Thus, to export the single ALL variable, put it not as first in the list by adding a comma:

```
 EXPORT ,ALL
```

EXPORT may appear anywhere in the SUB, provided it is executed before exiting, and more than one instance may be present.

A note is worthwhile: if you use EXPORT to export an uninstantiated variable, this variable is created empty and marked for export; any subsequent DECLARE will create another variable that hides the one created by EXPORT. To avoid this, use EXPORT as the last instruction or so.

FILE#/FILE:

(freely adapted from the LBMAA-A-D DEC document)

Opening a file means establishing a connection between **tbas** and a file on the file system, through which data is passed back and forth. Data is read from file in read mode and returned as number or string values; data is written to file in write mode, in form of number or string values.

The FILE statement has the following syntax:

```
 FILE arg1, arg2,... argn
```

where each argument has the form:

```
[#|:]N[,|:]"string formula"
```

The number N may be any numeric formula identifying the channel number; the file id # for sequential access or : for random access is optional, and if omitted, a sequential access file is opened.

Again, the file name may be followed by %, $ (with a number) or or nothing, to define the type of the file and (if a string random file) the record length directly within the file name.

The FILE statement does not permit the enclosing quotes to be omitted when the string formula argument is a constant; this is because a statement like FILE:1,B$ would cause an ambiguity, since B$ can be a variable (B$) with a file name in it, or a random access file B with string id ($).

If FILE finds the required channel already open, it disconnects the relative file freeing the channel, and reopens the channel connecting it to the new file; the old file type is immaterial.

Don't be misguided by the fact FILE and FILES have a similar name. FILE is executed on the run, when encountered, while FILES is executed before the running phase, and is ignored in run-time.

FILE:

See the FILE# statement.

FILES

(freely adapted from the LBMAA-A-D DEC document)
Opening a file means establishing a connection between **tbas** and a file on the file system, through which data is passed back and forth. Data is read from file in read mode and returned as number or string values; data is written to file in write mode, in form of number or string values.

The FILES statement has the following syntax:

```
FILES "name<type>" [:|#]<ch1>,... "name<type>" [:|#]<ch2>
```

Arguments are separated either by commas or by semicolons. Since it is executed in the preprocessing phase, the string identifying the file name must not be a variable, but has to be a proper string; if there are no spaces or special characters, the double quotes are optional.

The <type> token may be a percent sign %, a dollar sign $, optionally followed by an integer, or it can be omitted; if a percent % is specified, the file is assumed to be a random access numeric file; a dollar $ indicates a random access file string file, and the optional following number specifies the number of characters in string field, conventionally corresponding to the longest string in the record. This field is the same for all strings in the same file. A maximum of 256 characters and a minimum of one character can be specified. If the number is

omitted from the dollar sign access type and the file does not presently exist, a default length of 34 characters is established, and if the file does exist, the length with which the file was previously written is established.

Channels in the FILES statement are assigned consecutively to the argument of all the FILES statements in the program (that must not address mode than 9 files, of course). If an argument is omitted, the channel for the missing argument is skipped. For example:

```
FILES ,, A;,B
FILE C,D,
FILE E
```

assigns file A to channel 3, file B to channel 5, file C to channel 6, D to channel 7 and E to channel 9.

Don't be misguided by the fact that FILE and FILES have a similar name. FILE is executed on the run, when encountered, while FILES is executed in the preprocessing phase, and is ignored in run-time (that is, FILES opened channel are open whe program starts).

FILLPAGE#

This command starts to fill up the required channel page with empty lines until the end of page and prints the page number on last line, if required.

The FILLPAGE statement has the form:

```
FILLPAGE <c1>, <c2>, ...
```

where the arguments <c> have the form [#]N. The # character is optional, since there is no ambiguity, because FILLPAGE must be used only with sequential files.

E.g.

```
100 OPEN "Report.txt" for OUTPUT as #3
110 OPEN "Summary.txt" for OUTPUT as #6
120 PRINT #3,....
130 PRINT #6,....
......
400 FILLPAGE #3,6
```

If no arguments are provided, the statement works on the terminal. If no channel is specified in a queue, the terminal is set. e.g.

```
100 FILLPAGE ,#3
```

works on terminal and on channel #3. The command

```
100 FILLPAGE #3,,#6
```

works on channel #3, the terminal and channel #6. Since the page filling is performed immediately, this command may be repeated on the same command line:

```
100 FILLPAGE 3,3,3
```

This examples shows how to fill up the current page with void lines and then insert two void pages (each with its own page number if PAGE #3 was used) on file on channel #3.

FNEND

See the DEF FN statement.

FOR..TO..STEP|BY

The construct FOR/TO/BY|STEP/EXIT FOR/NEXT is the classical BASIC cycle (apart perhaps for the EXIT FOR); it's a fixed-sized cycle, with limits known from the start. **tbas** accepts the following syntax:

```
FOR <var> = <n1> TO <n2> BY|STEP <n3>
    ...
    ... EXIT FOR
    ...
NEXT [<var>]
```

The cycle uses <var>=<n1> as the first value, evaluates the body of the cycle, and when NEXT is met, <var> value is checked against <n2>, and if greater the cycle ends. Otherwise it adds to <var> the value <n3> if specified, or 1 if not specified, and re-executes the FOR cycle. The values used for the determination of the cycle are taken in the FOR line, and are not changed furthermore. As such, the FOR cycle is not dynamic: if variables that change their value in the cycle itself are used, the confrontation with the cycle limits remain the ones that were valid when the FOR line was evaluated.

The NEXT statement may be written as NEXT alone, and in this case, the cycle var is the <var> of the current 'for-next' cycle; or the index variable may be specified, and in this case it must match the current 'for-next' cycle index; if there are more than one nested cycles, they can be closed in one instruction, specifying the cycle vars in the reverse order:

```
FOR <var1> ...
   FOR <var2> ...
      FOR <var3> ...
         ...
NEXT <var3>,<var2>,<var1>
```

E.g.

```
FOR X=1 TO 10
   FOR Y=1 TO 3
      FOR K=0 TO 2
```

```
          ...
 NEXT K,Y,X
```

If the starting condition is false, or the keywords BY or STEP are used with wrong signs, the cycle is never executed, with no warning message.

FREE#

See the SCRATCH# statement.

FREE:

See the SCRATCH# statement.

GET#

The GET# statement reads a character from the file opened on the channel id after the # symbol, and attributes it to the variable that follows; if the variable is numeric, the character is stored as its ASCII representation, and if the variable is a string variable, the character is stored as a one-char string; the syntax is:

```
 GET arg, <sv>
 GET arg, <nv>
```

where argument arg has the form:

```
 [#]<n> for sequential files only
```

<n> is a channel identifier, whose integer part is taken, in the range 1÷9, identifying an open channel; <sv> is any string variable or string array element, and <nv> is any numeric variable or numeric array element. Since GET# must be used on sequential files only, the # character is optional.

E.g.

```
 REM THIS PROGRAM SIMULATES cat
 OPEN "prices.dat" FOR INPUT AS #3
 WHILE NOT EOF #3
    GET #3, A$
    PRINT A$;
 END WHILE
 CLOSE #3
```

E.g. if you have the file "test.txt" with this content:

```
 AAAA
 BBBB
 CCCC
 DDDD
```

the following program

```
OPEN "test.txt" FOR INPUT AS 2
WHILE NOT EOF(2)
       GET#2, F
       PRINT F,
WEND
PRINT
CLOSE 2
```

has the following output:

```
 65            65            65            65            10
 66            66            66            66            10
 67            67            67            67            10
 68            68            68            68            10
```

GOSUB

The GOSUB statements jumps unconditionally to a line number corresponding to the label following it (see GOTO for the label types system).

The label identifies a specific program chunk that executes a defined task (this zone is called subroutine); when the task is completed, the statement RETURN must be used, in order to get back to the calling GOSUB and execute the next instruction. E.g.

```
    ...
    GOSUB 200
    ...' RETURN resumes here
    STOP
    ...
    ...
200 <subroutine>
210 <subroutine>
220 RETURN
```

The program flow jumps from GOSUB to the line marked with 200, executes all the code from there until the first RETURN, then resumes back to the statements following GOSUB; since the subroutine section cannot be executed directly, because the RETURN statement would cause an error message to appear (there would be no GOSUB address to get back to), a STOP (or any END statement) is to be put *before* the line labelled with 200, to isolate the subroutine form the rest of the program.

GOTO/GO TO

The GOTO statement jumps unconditionally to the line number corresponding to the label following it; there are two types of labels available: numeric and alphanumeric; the first type makes **tbas** compatible with all BASIC programs written for line-numbered interpreters and compilers:

```
80 GOTO 100
90 ....
100PRINT "That's it!"
```

The second uses modern alphanumeric labels:

```
        GOTO print_exit
        ...

 print_exit:
        PRINT "Quitting."
        STOP
```

NOTE: A jump from within a SUB outside a SUB or from within a loop outside the same loop is not strictly prohibited, but the programmer needs to be aware of what it's being done (and maybe jump back to restore pointers). This freedom has some costs: a jump outside a loop may leave some inner reference not updated, and thus causing some bad behaviour. If you need to definitely exit the loop, use EXIT (see), or define the whole critic section inside a SUB and define clearly all exit points. This last solution is the core of the structured programming theory.

Using labels

Labels, as seen in the previous paragraph, are of two kind: numeric labels,

```
 125 PRINT
```

that must precede the statement, and alphanumeric labels, whose name is constituted by letters, numbers, underscores, dots and question marks, but must begin with a letter or underscore, not with a number. E.g.

```
 exit_point:
```

Alphanumeric labels must be declared on their own line (they must not be followed by a statement, or they will not be recognized) and must be ended with a colon, which is not part of the label name. Any statement using labels can simply use this name as a jump address:

```
 GOTO exit_point
```

A note is worthwhile, here: some statements require a numeric label to reside on their own line in order to be found, e.g.

```
 100 DATA 1,2,3
 110 : Value=####
 ...
 ...
 200 RESTORE 100
 300 PRINT USING 110, T
```

Such statements derive from old traditional BASICs, requiring expressively only numeric labels (they are mainly the PRINT USING format lines, beginning with colon and the DATA lines which must be RESTOREd to a specific line, as in the example).

Numeric and alphanumeric labels are instead available for all the statements that use or set an address, like GOTO, GOSUB, IF-THEN/GOTO, ON

THEN/GOTO, ON GOSUB, ON ERROR, ON ATTENTION, NO DATA.

HANDLER

See the WHEN ERROR USE statement.

IF..THEN

The construct IF/ELSE IF/ELSE/END IF lets the user perform an action (whatever it is) depending on a particular state (the condition <cond>, which may be true or false).

IF..THEN has the following syntaxes:

Multiline IF structure

```
IF <cond> THEN
   ...
   ...
ELSE IF <cond> THEN
   ...
   ...
ELSE
   ...
   ...
END IF
```

Only END IF is required, after its own IF clause, and there may be as many ELSE IF clauses as needed and one ELSE clause; nothing must follow THEN or ELSE, in order to avoid interpreting it as an 'if-then' followed by a command (both for IF-THEN and for ELSE-IF-THEN). Of course, IF clauses may be nested, up to 999 levels.

Jump-to-label IF structure

```
IF <cond> THEN|GOTO <l>
```

If an IF-THEN is followed by a line number or an existing alphanumeric label, a jump to that labeled line will occur in case <condition> is true. GOTO may be used in place of THEN in this case, and of course both may be used (this case falls in the next case). This is the dear old way of treating jumps, with the difference that **tbas** uses unordered labels and not ordered numbered lines as jump objects.

Command-mode IF structure

```
IF <cond> THEN <statement> [ELSE <statement>]
```

If an IF-THEN is followed by a direct statement (i.e. a statement that ends its duty on the same line), this will be executed directly, e.g.

```
IF <cond> THEN PRINT "RIGHT!"
```

is a legal and working line; you cannot put here a SELECT decision structure, nor any loop structure nor any WHEN ERROR structure, because

in this case the preprocessor won't be able to locate the command after THEN.

If an IF-THEN is followed by another IF, this is a mere substitution for two joined conditions; e.g:

```
 IF <cond1> THEN IF <cond2> THEN
    ...
 END IF
```

is a substitution for

```
 IF <cond1> AND <cond2> THEN
    ...
 END IF
```

which has a more 'logic' feeling. Of course this can be iterated more than twice. Anyway, both structures may used in **tbas**, so choose the one you like.

If the IF-THEN has an ELSE clause of one direct statement, it may be written directly following the THEN statement, without colons, commas or the like.

```
 IF <cond> THEN <stattrue> ELSE <statfalse>
```

The ELSE clause <statfalse> is performed in case <cond> returns false.

Remember that the IF..THEN..ELSE clause is a simple two-way logic statement to execute one of two clauses; if one of these clauses is another IF..THEN..ELSE clause, the ELSE part always refers to the first IF in the queue, but this may not be what you mean; for instance:

```
 IF A=0 THEN IF B=0 THEN PRINT "AB=0" ELSE PRINT "B<>0"
```

This line is equivalent to

```
 IF A=0 THEN ..................... ELSE PRINT "B<>0"
```

or, in a structured exposition:

```
 IF A=0 THEN
    IF B=0 THEN
       PRINT "AB=0"
    END IF
 ELSE
    PRINT "B<>0"
 END IF
```

If you are in trouble in such multiple logic statements, use structured IFs: they are safer...

IMAGE

The IMAGE statement must be used in conjunction with a numeric label, and its purpose is to define an image line that contains characters and markers suitable for the USING feature. The first space after IMAGE (if present) is not taken in account for the image line; this is true only for the first blank: the second on and the rest of the string are part of the image line (see the 'colon' statement).

Tick comments are not allowed in an image line, because the string formatter would use the tick by itself.

See the USING statement for details.

E.g. the program

```
 10 IMAGE Result is ####.##
    h=24.24
    PRINT USING 10,h
```

returns

```
 Result is   24.24
```

INCLUDE

The INCLUDE statement alters the standard input source (by default the keyboard) and sets the new source from the file given as a string argument (possibly with the whole path, if necessary), e.g.:

```
 INCLUDE "from.txt"

 INPUT "First value"; a$
 PRINT a$
 INPUT "Second value"; a$
 PRINT a$
 INPUT "Third value"; a$
 PRINT a$
 INPUT "Fourth value"; a$
 PRINT a$
```

This program sets the default input from file "from.txt"; the reading proceeds from the first character to the next, and the End-Of-Line is taken as the Enter key, so that any single input item stands on its own line.

When the input is over (the End-Of-File is met), the standard input from keyboard is restored. If the previous program uses the following three-lines data file "from.txt":

```
 24
 48
 96
```

The following output appears:

```
 First value ?
 24
 Second value ?
 48
 Third value ?
 96
 % Termination of INCLUDE file. Restoring default input in line 9.
 Fourth value ?
```

The three lines could fit well the first three input items, but when the fourth is required, there are no more lines in the file, so **tbas** resets input to the default and waits for a manual input, issuing a warning.

Please note that the file input items are not printed (while the user input is typed by the user and is so visible), but the output flow remains conceptually the same.

Note also that only INPUT and MATINPUT are influenced by this statement; INKEY$ and INPUT$ (which require a keyboard input) ignore the INCLUDE settings and continue to expect their input from keyboard.

If the file specified as argument should not exist, an error message is shown.

INPUT/ENTER

(freely adapted from the LBMAA-A-D DEC document)
The INPUT statement lets the user input data from keyboard. This is particularly useful if the program has a general library usage (e.g. the calculation of the volume of the sphere), and the user must supply her/his own data (in the example, the sphere radius). Data may be entered by an INPUT statement (it may be typed also as ENTER), which act as READ but accepts numbers of alphanumerical data from the terminal keyboard. Syntax:

```
 INPUT ["<prompt>"],<var1>, <var2>, ...
 ENTER ["<prompt>"],<var1>, <var2>, ...
```

where <prompt> is a literal string and <var1>, <var2>, ... are numeric or string variables.

E.g.

```
 INPUT X,Y
```

When **tbas** encounters this statement (if the OPTION PROMPT is ON, it types a question mark followed by a grace space), it waits for the user input. The user must then type all the values or strings needed, in the same order as they appear according to the variables types, separated by commas. All variables will be set to the values input by keyboard.

If you input more values than needed, **tbas** ignores the excess; if you,

on the other hand, type fewer items than that the statement requires, **tbas** will wait for other answers on a new line of input (in case OPTION PROMPT is ON, a question mark is printed, for the user's ease). This is repeated until all values are entered.

E.g.

```
 INPUT X,Y,Z
 PRINT X*X,Y*Y,Z*Z
 ? 23,56
 ? 67
 529           3136          4489
```

INPUT accepts a simple quoted string (the <prompt>) before the variables list, separated by semicolon or comma, as in the example:

```
 INPUT "Enter your name:";N$
```

Old BASIC compilers used to adopt a PRINT statement before the INPUT; in this case, the PRINT statement is terminated by a semicolon, to let the question mark appear just after the string:

```
 PRINT "YOUR VALUES OF X,Y AND Z ARE:";
 INPUT X,Y,Z
```

which shows:

```
 YOUR VALUES OF X,Y AND Z ARE:? 23,56,67
```

Note: if the final comma does not count as null value, all intermediate commas may be used to set some variable to the null value; see the following examples:

```
 ENTER "Enter your name:";N$,M$
 PRINT "|";N$;"|", "|";M$;"|"

 Enter your name: ? john,lennon
 |john|        |lennon|

 Enter your name: ? ,lennon
 ||            |lennon|

 Enter your name: ? john,
 ?lennon
 |john|        |lennon|
```

Seen? The comma before the second string instructs **tbas** to consider the first input as a null value, the comma as last is interpreted as a missing datum, which is re-asked. To enter all null values, enter as much commas as the variables number, as in the example:

```
 Enter your name: ? ,,
 ||            ||
```

The first null string stands before the first comma, the second null string before the second comma, and there is no missing data, so the rest is ignored.

The prompt may be iterated:

```
 INPUT "The first variable is ";v1, "The second variable is";v2
 PRINT v1,v2
```

yields:

```
 The first variable ? 45
 The second variable ? 78
  45            78
```

INPUT#/INPUT:/READ#/READ:

(freely adapted from the LBMAA-A-D DEC document)
The INPUT and READ statements for files read data items from files and have the following syntax:

For sequential access files:

```
 READ arg, <var>, <var>, ...
 INPUT arg , <var>, <var>, ...
```

where argument arg has the form:

```
 #<c>
```

For random access files:

```
 READ :N, <var>, <var>, ...
 INPUT :N, <var>, <var>, ...
```

where argument arg has the form:

```
 :<c>
```

<c> is a channel identifier in the range 1÷9. The value is truncated to an integer if necessary. At least one variable must be present in each READ or INPUT statement. The delimiter following <c> can be a comma or a colon. The variables are separated by a comma or semicolon.

The variables in a READ or INPUT statement for a sequential access file can be string or numeric or a mixture of both. The variables in a READ or INPUT statement for a random access file can be string or numeric, but not both, because a given random access file cannot contain both string and numeric data items.

READ and INPUT statements for sequential access files differ from one another in the following way. The READ statement expects each line of data in the file to begin with a line number, which it then skips. That is, the line number is not treated as data. If a line number is not

present, an error message is issued. The INPUT statement, on the other hand, does not expect a line number on each line of data. If one is present, it is read as data. It is illegal to use both INPUT and READ statements to read from the same sequential access file unless the file has been restored between the two types of statements. An attempt to mix READ and INPUT statements for sequential access files results in a fatal error message. Read and INPUT statements for random access files are completely equivalent. They both begin reading at the item that the pointer for the file specifies, and continue reading sequentially until all of the variables have been filled. It is legal to use both READ and INPUT statements to input from the same random access file.

If the user attempts to read beyond the last item in either a sequential access or a random access file, a fatal error message is issued. In a random access file, it is possible to have items that have not been written but that are within the file (because some subsequent item has been written). If such an item is in a numeric file and is read, a value of zero is input. If such an item is in a string file, a string containing no characters is returned.

INPUT:

See the INPUT# statement.

JUMP

The JUMP statement is used only within an ON ERROR, ON ATTENTION and NO DATA structures, provided the error that caused the jump to the handling section really appeared. The syntax is:

```
JUMP <n>
JUMP(<n>)
```

where <n> is a physical file line number. You cannot anyway JUMP to line zero or to a line which is greater than the last line number of the file. Moreover, the JUMP argument is a physical address line, a real line in the BASIC program; thus it must obey to two rules:

a) it must be a number (not a numeric label); its value may be the result of an expression, and its integer part is taken.

b) it must be one of the lines of the program.

The JUMP statement was conceived to be used in conjunction with NXL()/ASM()/ESM()/ERL(), functions that return the physical line number where the error/interrupt occurred.

LET

The LET statement introduces an assignment:

```
LET A=24
```

Its scope is merely to be compatible with older BASIC programs, but it

was optional even in the DEC-20 BASIC and supposedly in other coeval environments, and so in **tbas**.

The LET statements, as said earlier, can be grouped in series, being an assignment:

```
A=0: LET B=23: V=C=4
```

is a legal and working line.

Note: the LET statement should be used in all assignments that may be cause some misunderstandings when a variable name 'looks' like a statement; for instance:

```
DECLARE DATA
DATA=45
```

leads surely to confusion, because DATA is a reserved word; if you must use DATA as a variable name (even if any legal programming language bans the usage of such names), to assign 'DATA' use LET:

```
LET DATA=45
```

The previous assignment 'looks' like a DATA statement, the last is a LET statement, and cannot be confused. This said, the only word you really must not use as a name is LET. Guess why...

LIBRARY

The LIBRARY statement loads in memory all the libraries (i.e. subroutines) of the BASIC program given as arguments; this means that all SUBs and DEFs of the argument program are stored in memory as if they were written in the source file in the same place of the LIBRARY statement; the syntax is:

```
[OPEN AS] LIBRARY <f>
OPEN <f> AS LIBRARY
```

where <f> is the name of the BASIC library file. Only SUBs and DEFs are loaded, neither comments nor the main statements. In the first form, the words "OPEN AS" are optional.

For example, if file "lib.bas" has the following lines:

```
SUB log2(X)
   return log(X)/log(2)
END SUB
```

and your program "your.bas" has the lines:

```
LIBRARY "lib.bas"
INPUT "Value=";Y
PRINT log2(Y)
```

The third line will work as expected, because it is as if the log2() function was previously defined in the same file.

The LIBRARY statement is not an OPEN statement, even if the name can be misleading. Thus, if you write

```
 OPEN "lib.bas" AS LIBRARY
```

you don't have to use the CLOSE statement, because it does not use any of the available channels.

NOTE: the LIBRARY statements must follow some rules:

* if the library to be added via LIBRARY has all numbered lines, it is suitable both for the console mode (without interaction) and the interactive mode. The only fact you should care of is that the library and the program (and all libraries already added) have no same line numbers: this could lead to execution of statements which use a wrong line number destination address.

* if the library to be added via LIBRARY has no numbered lines, this won't cause any problem to the console mode. In the interactive mode, the library will be added with lines numbered following the last existing program line number (or last existing last-loaded-library line number). The library won't be visible with LIST, but GLIST will show the complete program. Using SAVE at this point, will save the program without libraries.
  If you should want to save the program *and* the loaded libraries, use GSAVE (or SAVE WITH LIBS). The libraries will be saved along with the current program (queued), and all lines containing a LIBRARY statement will be commented out, to avoid the reloading of the same libraries at the next LOAD + RUN.

* if the library to be added via LIBRARY has only few numbered lines (destination for GOTO/GOSUB or IMAGE lines, for instance), which for tbas are perfectly legal, since they are considered simply labels, they will be loaded retaining their number. This again could lead to wrong execution of statements which use a wrong line number destination address. Worse, any renumbering could lead to wrong order.
  Summing up: it's better that the libraries to be added to tbas via LIBRARY are either all numbered (with numbered-lines range-check for the interactive mode) or completely unnumbered; the partial enumeration, instead, must be used with extreme care.

LINE INPUT

See the LINPUT statement.

LINE INPUT#

See the LINPUT# statement.

LINE PRINT

See the LPRINT statement.

LINE READ#

See the LINPUT# statement.

LINPUT/LINE INPUT/ACCEPT

The line reader statement LINPUT (which can be also written LINE INPUT and ACCEPT) works exactly like INPUT, but it reads the whole line of the user keyboard input and return it as a string or as a number, according to the variable in the argument list:

```
LINE INPUT School_name$
```

If a numeric variable is required, only the first number in the line is converted and returned as a number. If an alphanumeric string should precede any number, zero is returned.

A literal string may follow the statement (and followed by a comma), to print a prompt message:

```
ACCEPT "Enter your full name", Name$
```

The file must end with a linefeed. Linefeed is the end of the line marker; if it lacks, LINPUT cannot know that the line has ended. This may cause failure in reading last line of a file, for instance. So take care to hit Enter after last line and eventually leave a blank line as last.

LINPUT#/LINE INPUT#/LREAD#/LINE READ#/ACCEPT#

The line reader statements LINPUT# (which can be also written LINE INPUT# and ACCEPT#) and LREAD# (which can be also written LINE READ#) work exactly like INPUT# and READ#, with the same differences and behaviours or INPUT and READ#, but they read the whole line of a sequential access file and return it as a string or as a number, according to the variable in the argument list:

```
LINPUT arg, <var>
LINE INPUT arg, <var>
ACCEPT arg, <var>

LREAD arg, <var>
LINE READ arg, <var>
```

where <var> is any numeric or string variable and argument arg has the form

```
#<c>
```

<c> is a channel identifier in the range 1÷9. E.g.

```
LINE INPUT #3, Name$
```

If a numeric variable is required, only the first number in the line is converted and returned as a number. If an alphanumeric string should precede any number, zero is returned.

LREAD# and LINPUT# apply only to sequential files, and if used with random access files an error message is issued.

LOCATE

The LOCATE statement sets the screen coordinates to the values given as arguments, as in the example:

```
LOCATE row,col
```

The first argument is the row, followed by the column value; they both start from 1 and the upper left corner has coordinates (1,1). The coordinates proceeds rightward for columns, and downward for rows.

If the row or column values are less than 1, they are automatically reset to 1. The row and column upper limits instead are not resized by **tbas**, but consider that the terminal you are using might trim or resize these values; moreover, if the column value exceeds screen width, the terminal might use more screen lines to fit the request and if the row value is greater than the screen height, a scroll might occur, with unpredictable results.

If one value only is given as argument, it is interpreted as the columns value, using as row the last value (or 1 if never set):

```
LOCATE col
```

If no arguments are given, the LOCATE statement alone will set coordinates to the upper left corner located at (1,1), and thus the two statements here reported are equivalent:

```
LOCATE
LOCATE 1,1
```

LPRINT/LINE PRINT

The LPRINT statement (that may be written also as LINE PRINT) works exactly like the PRINT statement, but its purpose is to begin the redirection of the output to the printer (and thus on the temporary file) and not to the terminal. If no QUEUE statement is used, the first LPRINT statement encountered (or the first after the last ROUTE statement) will set a temporary file with a provisional name, that will be deleted after the routing to printer is ordered (unless OPTION ROUTING OFF is used).

With this regard, the QUEUE statement used alone or a LINE PRINT have the same effect. See also the chapter "QUEUE and ROUTE in details".

LREAD#

See the LINPUT# statement.

MARGIN/NO MARGIN

(freely adapted from the LBMAA-A-D DEC document)
Normally, the right output margin for the terminal and sequential access files is 72 characters. Whenever a sequential access file is assigned to a channel by a FILES, FILE or OPEN statement, the file's output margin is automatically set to 72 characters. At the beginning of and also at the end of program execution, the terminal output margin is set to 72 characters. There is no margin in a random access file.

The MARGIN and MARGIN ALL statements allow the user to set the right output margin for the terminal or any sequential access file from 3 to 132 characters. The NO MARGIN and NO MARGIN ALL statements allow the user to reset the terminal and sequential access files to the default margin of 72.

The form of the MARGIN statement is:

```
 MARGIN <c1>, <c2>, ...
```

where each <c> argument has the form:

```
 #N, numeric formula
```

The arguments can be separated by commas or semicolons. N is the channel specifier and the # character is mandatory. The numeric formula specifies the margin size; it is truncated to an integer. Either a comma or a colon can be used to separate the channel number from the margin size. If only the margin size is present in the argument, that argument refers to the terminal.

The form of the MARGIN ALL statement is:

```
MARGIN ALL numeric formula
```

This statement sets the sequential access files on open sequential channels 1 through 9 to the margin specified by the numeric formula, the value of which is truncated to an integer before the margin is set. The terminal is not affected by the MARGIN ALL statement.

The MARGIN statement has no effect on random access files or on channels that have no files assigned to them, and causes an error if used in such cases. Consequently, the MARGIN ALL statement is a convenient way to set a margin for all sequential access files currently assigned to channels.

The margins set by the MARGIN and MARGIN ALL statements apply only to output. The margin for input lines for both the terminal and sequential access files is not affected by these statements; it is always 255 characters. An attempt to input a line longer than 255 characters results in an error message.

A margin set by a MARGIN or MARGIN ALL statement takes effect as soon as a new line of output is begun for the terminal or the sequential access file. Although the right margin can be set to any number between 3 and 132 characters, the margin for lines output by WRITE statements must be at least 7 characters to allow for the line number and its following tab. If the margin is less than 7 characters for a line-numbered file, an error message is issued by the first WRITE statement referencing the file.

The form of the NO MARGIN statement is:

```
NO MARGIN <c1>, <c2>, ...
```

where each <c> argument has the form:

```
[#]N
```

where N is the channel specifier and the # channel id is optional. If an argument is omitted, the terminal is specified; for example:

```
NO MARGIN #6,,2
```

refers to the terminal and the files on channels 2 and 6. Since the NO MARGIN statement is assumed to have at least one argument, the statement NO MARGIN without arguments refers to the terminal only.

The form of the NO MARGIN ALL statement is:

```
NO MARGIN ALL
```

The NO MARGIN ALL statement resets all of the open sequential access files on channels 1 through 9 to the default margin of 72, but does not affect the terminal. Like the MARGIN and MARGIN ALL statements, NO MARGIN and NO MARGIN ALL statements have no effect on channels that have random access files or no files assigned to them. Consequently, the NO MARGING ALL statement is a convenient way to set all of the sequential access files currently assigned to channels in nopage mode.

MAT statements

---

tbas has inherited the powerful MAT statements that were in decb (and in the DEC BASIC); they date back to 1966, to the Dartmouth BASIC version III and even earlier, to 1964 as a cardboard addition to version II. They can save a lot of work, in loading data, printing data, elaborating data. Something that is not present in modern languages. (Remember, I’ve never said that...)

Arrays in MAT statements are treated with BASE=1, that is the index 0 in never considered. This is because in math texts all indices start from 1, and this is reflected in the programming point of view of the MAT statements, which I find easy and handy. And, no need to say, this is the very same behaviour of the DEC-20 compiler, to which **tbas** is inspired.

This means that if you declare array (2,4), MAT statements use a 2x4 matrix, but you know that the real memory space (included the 0 indices) is a 3x5 matrix.

If you should need the whole memory space, you can write your own MAT statements and seize them at need.

Matrix input/output

Here's the full explanation of the statements that do the input/output.

MAT INPUT

The MAT INPUT statement lets the user to input values from keyboard. Syntax:

```
 MAT INPUT <t1>, <t2>, ...
```

where <t> items are arrays in the form of vectors and matrices, specified by means of the name only. If the following form is used:

```
 MAT INPUT t1(M,N)
```

the array (vector or matrix) is redimensioned to the specified values, provided the array was previously defined and instantiated with a memory space greater than or equal to the required. The nature of the array can also be changed (vector to matrix or vice versa), by using the proper number of arguments.

If the input line is not sufficient to hold all the numbers to be input, add an ampersand at the end of the line, and the input will continue. If you don't put the ampersand, **tbas** will assume the rest of the values are null.

If you input more values than needed, the excess is ignored.

NUM will hold the number of input values after the statement has executed, included the excess, and this is useful to check if the correct number of values has been input.

E.g.

```
 DIM F(3,4)
 MAT INPUT F
 PRINT NUM
 [user input:
  ? 1,2,3&
  ?4,5,6,7&
  ?1,2,3,4,5&
  ]
  12
```

tbas is conscious of the number of values needed, and it stops asking new values if the right number of values has been input. In the example, the last ampersand is optional because 3x4=12 values were input and **tbas** stops asking new values.

MAT INPUT#/MAT READ#

The MAT INPUT#/MAT READ# statements lets the user to input values from a file. Syntax:

```
MAT INPUT# <c>, <t1>, <t2>, ...
MAT READ# <c>, <t1>, <t2>, ...
```

where <c> is an input channel, <t1>, <t2>, ... are arrays in the form of vectors and matrices, specified by means of the name only. If the following form is used:

```
MAT INPUT# t1(M,N)
MAT READ# t1(M,N)
```

the array (vector or matrix) is redimensioned to the specified values, provided the array was previously defined and instantiated with a memory space greater than or equal to the required. The nature of the array can also be changed (vector to matrix or vice versa), by using the proper number of arguments.

MAT READ# expects a line number before the list for each line input (see INPUT# and READ# for details), while MAT INPUT# does not.

The function reads all necessary numbers from the file, but if there is no more data, the rest is filled with zeroes.

tbas is tolerant on the format of files; the input data may be separated by comma or blanks, in any number of lines.

NUM will hold the number of input values after the statement has executed, included the excess, and this is useful to check if the correct number of values has been input.

E.g. suppose you have the file "base.bas" containing the following lines in textual form (any spaced distance between two consecutive numbers):

```
1 2 3 4 5
2 -2 2 -2 2
5    6
```

```
OPEN "base.bas" FOR INPUT AS 1
DIM F(3,4)
MAT INPUT #1, F
MAT PRINT F
PRINT NUM
```

```
 1             2             3             4
 5             2            -2             2
-2             2             5             6

 12
```

Now, suppose you have the basic input file:

```
100 1 2 3 4 5
110 2 -2 2 -2 2
120 5    6
```

(it is the same of the previous example, but line-numbered). The same program becomes:

```
OPEN "base.bas" FOR INPUT AS 4
DIM F(3,4)
MAT READ #4, F
MAT PRINT F
PRINT NUM

 1             2             3             4
 5             2            -2             2
-2             2             5             6

 12
```

with identical results.

The MAT INPUT# and MAT READ# statements were built to be capable of reading matrices data written respectively by MAT PRINT# and MAT WRITE#, so that a program may read, write, re-read, re-write and so forth, without the intervention of the user on the results file.

MAT PRINT/MAT WRITE

The MAT PRINT (which may be typed as MAT WRITE also) prints the content of the arrays list to the screen. Syntax:

```
MAT PRINT <t1>, <t2>, ...
MAT WRITE <t1>, <t2>, ...
```

where <t1>, <t2>, ... are arrays in the form of vectors and matrices, specified by means of the name only.

There is no difference between MAT PRINT and MAT WRITE.

Numerical arrays

The picture or vectors, horizontal vectors and matrix is different; see the following examples:

```
DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
DIMENSION vec(6), table(5,3), hvec(0,5)
MAT READ vec, hvec
RESTORE
MAT READ table
MAT PRINT vec
MAT PRINT table
MAT PRINT hvec

 1
```

```
 2
 3
 4
 5
 6

 1             2             3
 4             5             6
 7             8             9
 10            11            12
 13            14            15

 7             8             9             10            11
```

Vertical vectors are printed vertically, horizontal vectors are printer horizontally, matrices are printed in full wide format, the first index identifying the rows number (5 in the example) and the second the columns number (3 in the example).

The default view can be changed by the modifiers semicolon ';' and comma ',' like the usual PRINT counterpart.

The semicolon prints a compact form:

```
MAT PRINT vec;
MAT PRINT table;
MAT PRINT hvec;

 1  2  3  4  5  6

 1  2  3
 4  5  6
 7  8  9
 10  11  12
 13  14  15

 7  8  9  10  11
```

Vertical vectors are turned horizontal, and printed compact.

The comma prints a wider form:

```
MAT PRINT vec,
MAT PRINT table,
MAT PRINT hvec,

 1             2             3             4             5
 6

 1             2             3
 4             5             6
 7             8             9
 10            11            12
 13            14            15
```

```
  7             8             9             10            11
```

Vertical vectors are turned horizontal but printed wider, the rest is not changed.

You can mix the output:

```
 MAT PRINT vec, table; hvec

  1             2             3             4             5
  6

  1  2  3
  4  5  6
  7  8  9
  10  11  12
  13  14  15

  7  8  9  10  11
```

If you omit the last modifier in a list, it is assumed equal to the previous.

```
 MAT PRINT table; hvec

  1  2  3
  4  5  6
  7  8  9
  10  11  12
  13  14  15

  7  8  9  10  11
```

Note: since any modifier upsets the vertical vectors, and any list in MAT PRINT must be separated by comma or semicolon, in order to print a vertical vector you have to use MAT PRINT to print only that particular vertical vector.

String arrays

String matrices behave the same, but they follow a different path in compact forms, because of the OPTION SPACING statement. If it's enabled (OPTION SPACING ON), strings are spaced, and the division between them is guaranteed. If it's disabled (OPTION SPACING OFF) they are packed, and the division must be provided by strings themselves.

E.g.

```
 OPTION SPACING ON
 DATA PRIMA,SECONDA,TERZA,QUARTA,QUINTA
 DIMENSION vec$(5)
 MAT READ vec$
 MAT PRINT vec$;
```

```
 PRIMA SECONDA TERZA QUARTA QUINTA

 OPTION SPACING OFF
 DATA PRIMA,SECONDA,TERZA,QUARTA,QUINTA
 DIMENSION vec$(5)
 MAT READ vec$
 MAT PRINT vec$;

 PRIMASECONDATERZAQUARTAQUINTA
```

Note: since any modifier upsets the vertical vectors, and any list in MAT PRINT must be separated by comma or semicolon, in order to print a vertical vector you have to use MAT PRINT to print only that vertical vector.

By using OPTION NULLS OFF, the MAT PRINT statement for string arrays gains a special feature: since void strings are (of course) void, the void string is identified by means of the phantom string mask "| |" (eleven spaces enclosed by bars), to set a place for them empty strings; the same with option -N at start.

E.g.

```
 OPTION NULLS OFF
 DIM arr$(3)
 MAT arr$=NUL$
 arr$(2)="HELPFUL"
 MAT PRINT arr$

 |           |
 HELPFUL
 |           |
```

If OPTION NULLS OFF is not used, or if OPTION NULLS ON is used (or ignored, since ON is the default) the 'real' empty string is printed.

```
 OPTION NULLS ON
 DIM arr$(3)
 MAT arr$=NUL$
 arr$(2)="HELPFUL"
 MAT PRINT arr$


 HELPFUL
```

MAT PRINT#/MAT WRITE#

The MAT PRINT#/MAT WRITE# print the content of the arrays list to the screen. Syntax:

```
 MAT PRINT#<c>, <t1>, <t2>, ...
 MAT WRITE#<c>, <t1>, <t2>, ...
```

where <c> is an output channel and <t1>, <t2>, ... are arrays in the form of vectors and matrices, specified by means of the name only.

These statements work in the same vein of the MAT PRINT statement for screen, but they direct the output to a textual file. The usual conventions for comma and semicolon hold.

The difference between MAT PRINT# and MAT WRITE# is the same of PRINT# and WRITE#: the second writes the line number before the text (starting from 1000 and advancing by 10), while the first does not.

E.g.

```
 DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
 OPEN "mat.bas" FOR NEW OUTPUT AS #3
 DIMENSION vec(6), table(5,3), hvec(0,5)
 MAT READ vec, hvec
 RESTORE
 MAT READ table
 MAT PRINT#3, vec
 MAT PRINT#3, table
 MAT PRINT#3, hvec
 CLOSE #3
```

This program creates the file "mat.bas" on the file system with the following content:

```
  1
  2
  3
  4
  5
  6

  1             2             3
  4             5             6
  7             8             9
  10            11            12
  13            14            15

  7             8             9             10            11
```

MAT READ

The MAT READ statement lets the user to input values from DATA lines. Syntax:

```
 MAT READ <t1>, <t2>, ...
```

where <t1>, <t2>, ... are arrays in the form of vectors and matrices, specified by means of the name only. If the following form is used:

```
 MAT READ t1(M,N)
```

the array (vector or matrix) is re-dimensioned to the specified values, provided the array was previously defined and instantiated with a memory space greater than or equal to the required. The nature of the array can also be changed (vector to matrix or vice versa), by using the proper number of arguments.

The DATA lines must hold all of the numbers needed to populate the MAT READ arrays, or the classic OUT OF DATA error will be raised.

RESTORE may be used to re-read efficiently the DATA items.

NUM is not updated by the MAT READ statements, since the control of the data is on the source itself.

E.g.

```
 DATA 1,2,3,4,5,6,7,8,9
 DIMENSION table(3,3)
 MAT READ table
 MAT WRITE table

  1             2             3
  4             5             6
  7             8             9
```

Matrix calculus

To understand the following statements, I define the following:

- a 'result' matrix is the matrix that will contain the result of the calculation, and its name will follow the MAT statement
- an 'object' matrix is a matrix in the body of the statement, composed in various forms.

tbas uses internal storing for the intermediate results, so that the result matrix may be one of the object matrices, for a safe result. For instance:

```
 MAT A = A + B
```

is a legal statement, that calculates the sum of A and B and stores back the result in A.

MAT + (plus)

The MAT + statement (mat plus) is used to sum to matrices and store the result in the result matrix. Syntax:

```
 MAT <t1> = <t2> + <t3>
```

where <t1> is the result matrix and <t2> and <t3> the matrices to sum, specified by means of the name only.

The dimensions of the two matrices to sum must match. If the result matrix has different dimensions, but nonetheless it can host the result matrix, it is resized accordingly.

E.g.

```
 DECLARE res1(1,3),res2(1,3)
 DATA 1,2,3,4,5,6,7,8,9
 MAT READ res1, res2
 MAT PRINT res1; res2
 MAT res=res1 + res2     ' res is declared 10x10 and resized
 PRINT
 MAT PRINT res;

  1  2  3

  4  5  6


  5  7  9
```

MAT - (minus)

The MAT - statement (mat minus) is used to subtract matrices and store the result in the result matrix. Syntax:

```
 MAT <t1> = <t2> - <t3>
```

where <t1> is the result matrix and <t2> and <t3> the matrices to subtract, specified by means of the name only.

The dimensions of the two matrices to subtract must match. If the result matrix has different dimensions, but nonetheless it can host the result matrix, it is resized accordingly.

E.g.

```
 DECLARE res1(1,3),res2(1,3)
 DATA 1,2,3,4,5,6,7,8,9
 MAT READ res1, res2
 MAT PRINT res1; res2
 MAT res=res1 - res2     ' res is declared 10x10 and resized
 PRINT
 MAT PRINT res;

  1  2  3

  4  5  6


 -3 -3 -3
```

MAT * (times)

The MAT * statement (mat times) is used to multiply matrices and store the result in the result matrix. Syntax:

```
 MAT <t1> = <t2> * <t3>
```

where <t1> is the result matrix and <t2> and <t3> the matrices to subtract, specified by means of the name only.

The dimensions of the two matrices to subtract must fit the matrix multiplication criteria, so that t1 row = t2 row, t1 colums = t3 column, and t2 column = t3 row. If the result matrix has different dimensions, but nonetheless it can host the result matrix, it is resized accordingly.

E.g.

```
 DECLARE res(3,3),res1(2,3),res2(3,2)
 DATA 1,2,3,4,5,6,7,8,9,10,11,12
 MAT READ res1, res2
 MAT PRINT res1; res2
 MAT res=res1 * res2     ' note: res was declared 3x3
 PRINT
 MAT PRINT res;          ' now it is 2x2

  1  2  3
  4  5  6

  7  8
  9  10
  11  12


  58  64
  139  154
```

Note: the result matrix cannot be one of the object matrices when dimensions play a fundamental role, but in case all matrices are square and with the same dimensions, the rule still hold.

E.g.

```
 DECLARE res(2,2),obj(2,2)
 DATA 1,2,3,4,5,6,7,8,9
 MAT READ res, obj
 MAT PRINT res; obj
 MAT res = res * obj
 PRINT
 MAT PRINT res;

  1  2
  3  4

  5  6
  7  8


  19  22
  43  50
```

MAT K (constant)

The MAT K statement is used to multiply a matrix by a constant and store the result in the result matrix. Syntax:

```
MAT <t1> = (<n>) * <t2>
```

where <t1> is the result matrix and <t2> the object matrix, specified by means of the name only. The constant value <n> (any legal numeric expression) must be enclosed in parentheses, because it may be a math expression.

The dimensions of the two matrices should match, but if the result matrix has different dimensions and nonetheless it can host the result matrix, it is resized accordingly.

E.g.

```
DECLARE res(4,3),res1(3,3)
DATA 1,2,3,4,5,6,7,8,9,10,11,12
MAT READ res1
MAT PRINT res1
MAT res=(sqr(2)) * res1 ' note: res was a 4x3
MAT PRINT res           ' now it is a 3x3

 1            2            3
 4            5            6
 7            8            9

 1.41421      2.82843      4.24264
 5.65685      7.07107      8.48528
 9.89949      11.3137      12.7279
```

MAT CON/MAT UNITY

The MAT CON statement (that may be written more conveniently as MAT UNITY, since I haven't understood yet what CON stands for) sets an arrays to all ones. Syntax:

```
MAT <t1> = CON[(<n1>[,<n2>])]
MAT <t1> = UNITY[(<n1>[,<n2>])]
```

where <t1> is the result matrix, specified by means of the name only.

The CON/UNITY may be followed by matrix dimensions enclosed in parentheses that resizes the matrix as needed.

E.g.

```
DIM g(3,3)
MAT g=CON
MAT PRINT g

 1            1            1
```

```
 1             1             1
 1             1             1

DIM g(3,3)
MAT g=CON(2,3)
MAT PRINT g

 1             1             1
 1             1             1
```

The resize procedure can change the state of the array; for instance, an array declared as a table can be turned as a full vector:

```
DIM g(3,3)
MAT g=CON(3)
MAT PRINT g

 1
 1
 1
```

The resize value must define a memory space which is contained in the original size of the array, that is the original space created with the DIM statement cannot be changed (unless the RESIZE statement is used). An attempt to resize to anything that requires more space raises an error:

```
DIM g(3,3)
MAT g=CON(50)
MAT PRINT g

? Array/matrix size exceeds current memory space in line 2.
MAT g=CON(50)
           ^
```

MAT IDN/MAT IDENTITY

The MAT IDN statement (that may be written more conveniently as MAT IDENTITY) sets the result matrix to the identity matrix, with ones on the principal diagonal and zero elsewhere. Syntax:

```
MAT <t> = IDN[(<n>[,<n>])]
MAT <t> = IDENTITY[(<n>[,<n>])]
```

where <t1> is the result matrix, specified by means of the name only.

E.g.

```
DIM arr(4,4)
MAT arr=IDN
MAT PRINT arr

 1             0             0             0
 0             1             0             0
```

```
  0             0             1             0
  0             0             0             1
```

The matrix must be square, or an error is raised.

E.g.

```
 DIM arr(4,3)
 MAT arr=IDN
 MAT PRINT arr

 ? Matrix is not square in line 2.
 MAT arr=IDN
           ^
```

The IDN/IDENTITY may be followed by matrix dimensions enclosed in parentheses that resizes the matrix as needed by giving two dimensions, but they must be two equal values in order to guarantee that the result matrix is square, and it must be verified that the memory space can host the entire resizing.

E.g.

```
 DIM arr(10,10)
 MAT arr=IDN(3,3)
 MAT PRINT arr

  1             0             0
  0             1             0
  0             0             1
```

E.g.

```
 DIM arr(3,3)
 MAT arr=IDN(10,10)
 MAT PRINT arr

 ? Array/matrix size exceeds current memory space in line 2.
 MAT arr=IDN(10,10)
                 ^
```

MAT INV/MAT INVERT

The MAT INV statement (which can be typed also as MAT INVERT) inverts a square matrix, and stores it in the result matrix. Syntax

```
 MAT <t1> = INV(<t2>) [ELSE ...]
 MAT <t1> = INVERT(<t2>) [ELSE ...]
```

where <t1> is the result matrix and <t2> is the matrix to invert, specified by means of the name only.

If the inversion is not possible, a warning appears, but if the ELSE clause is specified, the warning does not appear, and what follows ELSE

is performed instead (a jump or a command, just like the IF..THEN statement).

In either cases, the system variable DET will contain the determinant (null in case of impossible inversion).

E.g.

```
 DECLARE mat(3,3),res(3,3)
 DATA 1,2,-3,4,5,6,-7,8,9
 MAT READ mat
 MAT PRINT mat
 MAT res=INV(mat)
 MAT PRINT res
 PRINT "Determinant=";DET

  1             2            -3
  4             5             6
 -7             8             9

  8.33333E-3    0.116667     -0.075
  0.216667      3.33333E-2    0.05
 -0.186111      6.11111E-2    8.33333E-3

 Determinant=-360
```

If the previous program uses instead the following DATA line:

```
 DATA 1,2,3,4,5,6,7,8,9
```

The result would be:

```
  1             2             3
  4             5             6
  7             8             9

 % Singular matrix inverted in line 5.
  0             0             0
  0             0             0
  0             0             0

 Determinant= 0
```

If an ELSE clause is specified (in this case a jump is performed, but this is not the only possible case, and please note that GOTO is pleonastic here) the result would be:

```
 DECLARE mat(3,3),res(3,3)
 DATA 1,2,3,4,5,6,7,8,9
 MAT READ mat
 MAT PRINT mat
 MAT res=INV(mat) ELSE GOTO isSingular
 MAT PRINT res
 PRINT "Determinant=";DET
```

```
 STOP

isSingular:
 PRINT "Matrix is singular (determinant null)"
 STOP

  1             2             3
  4             5             6
  7             8             9

 Matrix is singular (determinant null)
```

See also DET.

A note about the inversion algorithm: it based on the Gauss factorization with partial pivoting, an algorithm that satisfies the majority of cases, but if the matrix is **very** sparse, the result may be not precise. In any case, I couldn't find any problem so far. Let me know if you find a matrix that is badly inverted...

MAT NUL$/MAT NULL$

The MAT NUL$ statement (that may be written also as MAT NULL$) sets a string array to all void strings. Syntax:

```
 MAT <t1> = NUL$[(<n1>[,<n2>])]
 MAT <t1> = NULL$[(<n1>[,<n2>])]
```

where <t1> is the result string array, specified by means of the name only.

E.g.

```
 DIM g$(3,3)
 MAT g$=NUL$
 MAT PRINT g$

 |           | |           | |           |
 |           | |           | |           |
 |           | |           | |           |
```

The NUL$/NULL$ may be followed by matrix dimensions enclosed in parentheses that resizes the matrix as needed.

```
 DIM g$(3,3)
 MAT g$=NUL$(4)
 MAT PRINT g$

 |           |
 |           |
 |           |
 |           |
```

The resize procedure can change the state of the array; for instance, an

array declared as a vector can be turned as a matrix:

```
DIM g$(50)
MAT g$=NUL$(3,3)
MAT PRINT g$

|          | |          | |          |
|          | |          | |          |
|          | |          | |          |
```

The resize procedure must define a memory space which is contained in the original size of the array, that is the original space created with the DIM statement cannot be changed (unless the RESIZE statement is used). An attempt to resize to anything that requires more space raises an error:

```
DIM g$(3,3)
MAT g$=NUL$(50)
MAT PRINT g$

? Array/matrix size exceeds current memory space in line 2.
MAT g$=NUL$(50)
            ^
```

MAT TRN/MAT TRANSPOSE

The MAT TRN statement (that may be written in full as MAT TRANSPOSE) takes the object array, transpose it and stores it in the result matrix. Syntax:

```
MAT <t1> = TRN(<t2>)
MAT <t1> = TRANSPOSE(<t2>)
```

where <t1> is the result matrix and <t2> is the matrix to transpose, specified by means of the name only.

E.g.

```
DIM arr(2,7),res(10,10)
MAT arr=CON
MAT PRINT arr
MAT res=TRANSPOSE(arr)
MAT PRINT res

 1            1            1            1            1
 1            1
 1            1            1            1            1
 1            1

 1            1
 1            1
 1            1
 1            1
 1            1
```

```
 1             1
 1             1
```

The result matrix is sized implicitly if needed when its dimension can host the result matrix values, otherwise an error is raised.

A vector can be used in this regard, but the transpose action consists only in the reversing the vertical/horizontal state of the array:

```
DIM arr(7),res(10,10)
MAT arr=CON
MAT PRINT arr
MAT res=TRANSPOSE(arr)
MAT PRINT res

 1
 1
 1
 1
 1
 1
 1

 1             1             1             1             1
 1             1
```

MAT ZER/MAT ZERO

The MAT ZER statement (that may be written also as MAT ZERO) sets an arrays to all zeroes. Syntax:

```
MAT <t1> = ZER[(<n1>[,<n2>])]
MAT <t1> = ZERO[(<n1>[,<n2>])]
```

where <t1> is the result matrix, specified by means of the name only.

The ZER/ZERO may be followed by matrix dimensions enclosed in parentheses that resizes the matrix as needed.

E.g.

```
DIM g(3,3)
MAT g=ZER
MAT PRINT g

 0             0             0
 0             0             0
 0             0             0

DIM g(3,3)
MAT g=ZER(2,3)
MAT PRINT g

 0             0             0
```

```
 0             0             0
```

The resize procedure can change the state of the array; for instance, an array declared as a vector can be turned as a matrix:

```
DIM g(50)
MAT g=ZER(4,4)
MAT PRINT g

 0             0             0             0
 0             0             0             0
 0             0             0             0
 0             0             0             0
```

The resize procedure must define a memory space which is contained in the original size of the array, that is the original space created with the DIM statement cannot be changed (unless the RESIZE statement is used). An attempt to resize to anything that requires more space raises an error:

```
DIM g(3,3)
MAT g=ZER(50)
MAT PRINT g

? Array/matrix size exceeds current memory space in line 2.
MAT g=ZER(50)
           ^
```

Example of a three-equations system solution

As an illuminating example of general usage of the MAT function, let's suppose I have to solve the following equation system:

$$\begin{cases} x + 2y = 3 \\ 3x - 7y + 2z = -1 \\ y - 4z = 0 \end{cases}$$

I first of all collect data of the coefficients matrix and the knowns vector (the one after the equal sign):

Coefficients matrix:

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & -7 & 2 \\ 0 & 2 & -4 \end{bmatrix}$$

Knowns vector:

$$K = \begin{bmatrix} 3 \\ -1 \\ 0 \end{bmatrix}$$

Unknowns vector:

$$X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Then, I proceed to store these data in a DATA line

```
DATA 1,2,0,3,-7,2,0,2,-4
DATA 3,-1,0
```

and after that I dimension what I need by now: a 3x3 matrix and two 3-elements vectors, one for the knowns, one for the unknowns; then I load values and print them:

```
DIM A(3,3),K(3),X(3)
MAT READ A,K
MAT PRINT A,K
```

The equations system can be written in matrices calculus as this:

$$[A][X] = [K]$$

Now, if I pre-multiply both expressions by

$$[A]^{-1}$$

I get

$$[A]^{-1}[A][X] = [A]^{-1}[K]$$

and for a fundamental property of matrices calculus I get

$$[I][X] = [A]^{-1}[K]$$

$$[X] = [A]^{-1}[K]$$

So all I have to do is get the inverse of matrix A and store it in a matrix, say B. I need to dimension matrix B (square 3x3) and operate an inversion instruction:

```
DIM B(3,3)
MAT B=INV(A)
```

Simple. Now the problem is to multiply the inverted matrix (3x3) by the knowns vector (3 values), to obtain the unknowns 3-values vector X

```
MAT X=B*K
```

Almost done. To verify the result I have to calculate the error. Let's calculate the original expression (using a temporary vector K1 to hold the results):

$$[A][X]$$

```
DIM K1(3)
MAT K1=A*X
```

The error is of course the difference between the K1 vector and the original knowns vector; I use another vector E to hold the error:

```
DIM E(3)
```

```
 MAT E=K1-K
```

Let’s print the results:

```
 PRINT "Solutions are:"
 MAT PRINT X
 PRINT "Error vector:"
 MAT PRINT E
```

The smaller is the error, the higher is the accuracy of the solution.

Another validity test is to multiply matrix A with its inverse; the result should be of course the identity matrix, and as long as the principal diagonal contains numbers approaching 1 leaving numbers approaching zero elsewhere, the found inverse is good.

```
 DIM I(3,3)
 MAT I=B*A
 PRINT "Test for Identity matrix:"
 MAT PRINT I
```

The final program follows, with all statements gathered and commented:

```
 REM main data
 DATA 1,2,0,3,-7,2,0,2,-4
 DATA 3,-1,0
 DIM A(3,3),K(3),X(3),B(3,3),K1(3),E(3),I(3,3)
 MAT READ A,K
 REM print original data
 MAT PRINT A,K
 REM inversion and calculation of solution
 MAT B=INV(A)
 MAT X=B*K
 REM errors calculation
 MAT K1=A*X
 MAT E=K1-K
 REM printing solutions
 PRINT "Solutions are:"
 MAT PRINT X
 PRINT "Error vector:"
 MAT PRINT E
 REM test for identity matrix
 MAT I=B*A
 PRINT "Test for Identity matrix:"
 MAT PRINT I
```

The output is:

```
  1             2             0
  3            -7             2
  0             2            -4

  3            -1             0
```

```
 Solutions are:
  1.33333
  0.833333
  0.416667

 Error vector:
  0
 -4.44089E-16
  0

 Test for Identity matrix:
  1              1.38778E-16   0
  0              1             0
  0              0             1
```

You now see how powerful is the MAT system of BASIC!

NEXT

See the FOR..NEXT statement.

NO DATA

The NO DATA statement is a shortcut for the statement:

```
 ON ERROR 110
```

(being 110 the error code of OUT OF DATA), and is used to catch the error condition of the running out of data in READ/DATA statements routines; it can be used in these ways:

```
 NO DATA [GOTO|THEN] <label>
```

This form traps error condition 110 redirecting to line with label <label>, which should contain code to treat this special error condition. GOTO and THEN are optional.

```
 NO DATA
```

This form disables the NO DATA feature.

If NO DATA is used more than once, only the most recent instance reference will be stored and used.

NO DATA can be used even within 'WITH ERROR IN' and 'WITH ERROR USE' structures because, in the error treating routing (**tbas** internals), 'WHEN ERROR' statements have precedence over ON ERROR and are evaluated earlier.

E.g.

```
        ...
        NO DATA 400
        READ T
        ...
```

```
 400    PRINT"SUPPLY MORE VALUES, PLEASE"
        STOP
```

NO MARGIN

See the MARGIN statement.

NO PAGE

See the PAGE# statement.

NO PAGE ALL

See the PAGE# statement.

NO PAGENUM#

See the PAGENUM# statement.

NO PAGENUM ALL

See the PAGENUM# statement.

NO QUOTE ALL

See the QUOTE# statement.

NO QUOTE#

See the QUOTE# statement.

ON ATTENTION

The ON ATTENTION statement is an attempt to treat user interrupt conditions basing on the ON ERROR style. Its usage is simple:

```
 ON ATTENTION [GOTO|THEN] <label>
```

This form traps user interrupts, redirecting to line with label <label> in case any occurs.

```
 ON ATTENTION
```

This form disables the ON ATTENTION feature.

If ON ATTENTION is used more than once, only the most recent reference will be used.

ON ATTENTION can be used with 'WHEN ERROR IN' and 'WHEN ERROR USE' structures because 'WHEN ERROR' routines are evaluated earlier in the error treating routines (**tbas** internals).

ON ERROR

The ON ERROR statement and its related statements constitute a simple attempt to treat error conditions. Its usage is simple:

```
ON ERROR [GOTO|THEN] <label>
```

This form traps all errors, redirecting to line with label <label> in case any occurs.

```
ON ERROR <err> [GOTO|THEN] <label>
```

This form traps error with code <err>, redirecting to line with label <label> in case error <err> occurs, but proceeds with usual error messages for all the remaining error types.

```
ON ERROR
```

This form disables all the ON ERROR features.

If ON ERROR is used more than once (with reference to specific error codes or all), only the most recent will be used.

ON ERROR can be used with 'WITH ERROR IN' and 'WITH ERROR USE' structures because 'WHEN ERROR' routines are evaluated earlier in the error treating routines (**tbas** internals).

ON..GOSUB

The ON GOSUB statement lets the program flow executing different subroutines depending on the control value (a numeric variable appearing between ON and GOSUB): the control value must have values ranging from 1 and N, where N is the last numerable address given; any value beyond these limits cause an error message to be shown and the program stops.

If the keyword OTHERWISE is used after the addresses queue, all values outside the range does not cause error, but rather will execute the subroutine at the address written after OTHERWISE. The syntax follow these rules:

```
ON <nv> GOSUB <a1>,<a2>,<a3>...[, OTHERWISE <a4>]
```

where <nv> is a numeric value ranging from 1 to N and the OTHERWISE clause is optional. <a1>, <a2>.. are numeric or alphanumeric labels identifying execution lines.

The addresses must start a subroutine that must be terminated with the RETURN statement (see the GOSUB statement). The RETURN statement (that closes the subroutine at each various addresses), will resume execution from the line following the ON GOSUB statement.

E.g.

```
ON gross GOSUB review, store, send, otherwise re_input
```

The example is made in the assumption that review, store, send and re_input are defined labels, and that gross takes values that run from 1 to 3 in the general usage and outside this range for the 'otherwise' case.

A more traditional example is:

```
 100 ON B3 GOSUB 1000,2000,3000
```

ON..THEN/GOTO

The ON..THEN/GOTO statement lets the program flow take different directions depending on the control value (a numeric variable appearing between ON and THEN or GOTO, perfectly interchangeable): the control value must have values ranging from 1 and N, where N is the last numerable address given; any value beyond these limits cause an error message to appear and the program stops.

If the keyword OTHERWISE is used after the addresses queue, all values outside the range does not cause error, but rather will direct the flow to the address written after OTHERWISE. The syntax follow these rules:

```
 ON <nv> GOTO <a1>,<a2>,<a3>...[, OTHERWISE <a4>]
```

where <nv> is a numeric variable whose value ranges from 1 to N and the OTHERWISE clause is optional. <a1>, <a2>.. are numeric or alphanumeric labels identifying execution lines.

E.g.

```
 ON index GOTO calc, print, exit, otherwise set_error
```

The example is made in the assumption that calc, print, exit and set_error are defined labels, and that index takes values that run from 1 to 3 in the general usage and outside this range for the 'otherwise' case.

A more traditional example is:

```
 25 ON A GOTO 100,200,300
```

OPEN

Opening a file means establishing a connection between the running program and a file on the file system, through which data is passed back and forth. Data is read from file in read mode and returned as number or string values; data is written to file in write mode, in form of number or string values (see also FILE and FILES).

The OPEN statement has the following syntax:

```
 OPEN [READONLY] "f" FOR [NEW] <id>[,] AS [RANDOM|READONLY] [FILE]
[:|#]<n1>[,] [TYPE NUM|STR<n2>]
 OPEN "name" AS LIBRARY
```

"f" may be any file name (string or string value); <id> may be INPUT or OUTPUT; <n1> must be any valid channel number 1÷9 not yet open; if the word NEW is found and <id> is OUTPUT, then the file is erased when opened. NEW is ignored in case it is used with INPUT (there's no meaning in opening a file for input and erase it before starting reading!); if RANDOM is found before the channel, the type is set as RANDOM; if FILE is found before the channel, it is ignored; if <n1> is preceded by # (optional) the sequential access type is established and RANDOM must not be used, if <n1> is preceded by : (optional) RANDOM is optional and the random access file is established; if TYPE is used, the random type follows (numeric or string); if TYPE NUM or if TYPE is not used (and RANDOM is set elsewhere), the numeric type is assumed; if TYPE STR is used the string type is assumed; if <n2> is used, <n2> is taken as the record length for the random string type; if <n2> is not used, the default string length of 34 characters is assumed.

If no FOR <id> is found, FOR INPUT is assumed. Thus, file must exist, while if no AS <n1> is found (e.g. OPEN "name" FOR OUTPUT), channel is assigned according to the first available channel (but this channel is not given back to the user, so the user may ignore it); if no FOR and no AS are used (e.g. OPEN "name") INPUT <id> and the first available channel are assumed. Thus, file must exist.

Unlike FILE and FILES, OPEN treats one file at each invocation.

The READONLY flag sets the input mode to treat the file as read-only, that is no writing to it is possible. This ensures that sensible files are not corrupted by accidental programming mistakes.

Examples are:

```
 OPEN "data" FOR INPUT AS #1
 OPEN "rett%" FOR INPUT
 OPEN "rett" FOR INPUT AS RANDOM FILE 1, TYPE NUM
 OPEN "names$56" FOR OUTPUT AS 2
 OPEN "names" FOR OUTPUT AS RANDOM FILE 2, TYPE STR 56
 OPEN READONLY "names" AS #3
 OPEN "names" AS READONLY #5
```

The second and third examples are equivalent; the fourth and fifth examples are equivalent. The sixth and seventh examples use the same structure with two different channels, but in any case file "names" is opened for INPUT as readonly.

The simpler usage for reading data from a file is

```
 OPEN "data"
```

The file "data" is opened in sequential mode for INPUT in the first available channel starting from 1, and if this is the only statement used for opening a file, the channel will be the number 1.

For what about the LIBRARY token, see the LIBRARY statement.

OPTION

The OPTION statement evaluates some extra-BASIC option, which can influence the interpreter, not the language. This option is built to maximize the compatibility effect, showing no errors in case of unrecognized option or unresolved/unfitting arguments.

The available options and their default value are:

OPTION	default value
OPTION ANGLE DEGREES|RADIANS	RADIANS
OPTION BASE 0|1	0
OPTION CAPS ON|OFF(*)	OFF
OPTION CASE ON|OFF(*)	OFF
OPTION COMPARISON RELATIVE|ABSOLUTE	RELATIVE
OPTION DEBUG ON|OFF(*)	OFF
OPTION DIFFERENCE 0..n|OFF	6|OFF
OPTION ECHO ON|OFF(*)	ON
OPTION ERRORSTREAM ON|OFF (*)	OFF
OPTION EXPLICIT ON|OFF(*)	OF
OPTION FORMAT AMERICAN|EUROPEAN	AMERICAN
OPTION HEADER ON|OFF (*)	OFF
OPTION NULLS ON|OFF (*)	ON
OPTION PRECISION 0..n|OFF	0=OFF
OPTION PROMPT ON|OFF(*)	ON
OPTION RAWPRINT ON|OFF(*)	OFF
OPTION RESET ON|OFF(*)	ON
OPTION ROUTING ON|OFF(*)	ON
OPTION SPACING ON|OFF(*)	OFF
OPTION TAB 0..n|OFF	14
OPTION TIMING ON|OFF (*)	OFF
OPTION VINTAGE ON|OFF (*)	OFF
OPTION WARNINGS ON|OFF*	ON
OPTION ZERO 0..n|OFF*	0=OFF

Statements marked with (*) can be activated by means of the first identifier alone or with the ON specifier (e.g. OPTION SPACING is equivalent to OPTION SPACING ON).

OPTION VINTAGE is set by the preprocessor unit, and thus can be put anywhere in the listing, but remember that any further option may override its settings; OPTION HEADER too is set by the preprocessor unit, but it can be overridden.

OPTION ANGLE

This option sets the default angle measure in radians or degrees. The statement accepts also the simplified forms OPTION RADIANS and OPTION DEGREES, with obvious purposes.

OPTION BASE 0|1

This option sets the lowest array index to 0 if OPTION BASE 0 is used, to 1 if OPTION BASE 1 is used; in any case, matrices are not affected, since they work always with index 1.

OPTION BASE is more compatible with others BASICs if you use only 0 or 1, but **tbas** interprets the numerical code in the TRUE/FALSE sense, that is 0 set base 0, not-zero sets base 1. This enables using variables and mathematical expressions as a numerical code for OPTION BASE.

OPTION CAPS ON|OFF

This option enables/disables writing output in capital letters; neither the inner memory state of strings nor the PRINT USING statement nor the INPUT and MAT INPUT statements are affected by this option (strings are input as-is); the effect is only on the output, i.e. to what appears on the screen.

OPTION CASE ON|OFF

This option enables/disables case sensitiveness in case of string comparison.

OPTION COMPARISON RELATIVE|ABSOLUTE

This option enables the overriding of the trimming of trailing spaces ahead and behind the strings. As from the DEC-20 tradition, two strings are usually compared after the trailing spaces are removed (RELATIVE mode), but this feature is not common to other BASIC dialects so, by activating OPTION COMPARISON ABSOLUTE, the trimming is not taken in account, and comparison is made as-is. Note that the == operator is transparent to this option, since it acts always in ABSOLUTE mode.

OPTION DEBUG ON|OFF

This option enables/disables the debug features. It's worthwhile noting here that this option may enable/disable debugging on specific sub-sections of a BASIC program, while option -d at the command line starts debugging from the preprocessing phase to the end of execution (or to the first OPTION DEBUG OFF statement).

OPTION DIFFERENCE <n>|OFF

This option sets the quasi-equal feature (|= operator for numbers), which evaluates to TRUE when two numbers difference in absolute value is lesser than 10^(-<n>). <n> may be any numerical formula result, whose positive integer part is used; a zero value or OFF cause the setting of the default value of 6.

OPTION ECHO ON|OFF/OPTION ECHO|NOECHO

This option enables/disables the keyboard echoing during the input. OPTION ECHO is equivalent to OPTION ECHO ON, and OPTION NOECHO is equivalent to OPTION ECHO OFF. The NOECHO option does not prevent the <Enter> key (pressed after an INPUT) to print a New Line, as not to go counter-intuitive for the program user.

OPTION ERRORSTREAM ON|OFF

This option redirects the default output to standard error (stderr) when ON is used, and restores the default output to standard out (stdout) if OFF is used. The expression "OPTION ERRORSTREAM" (without argument) is equivalent to "OPTION ERRORSTREAM ON".

NOTE: the redirection established by OPTION ERRORSTREAM has effect only for channel 0 which is the screen output channel; file channels are not affected.

OPTION EXPLICIT ON|OFF

This option enables the explicit declaration of all variables and arrays in a program; if activated, the program structure becomes robust, because no variable is out of control. The default state is off, to guarantee a total backward compatibility with DEC-20 programs. The expression "OPTION EXPLICIT" (without argument) is equivalent to "OPTION EXPLICIT ON".

OPTION FORMAT AMERICAN|EUROPEAN

This option sets American format (which enables the dot as decimal separator and the comma for triple unities - e.g. 123,456.789 for 123456.789) or European format (which enables the comma for the decimal separator and the tick for triple unities - e.g. 123'456,789). This option is applicable only to the USING feature; the standard numeric representation always uses the dot for decimal separator and no divisions in triple unities.

OPTION HEADER ON|OFF

This option controls the printing of the program name and time and data before any statement execution.
Note: since the header should be printed before any other output, OPTION HEADER is preprocessed and executed before any statement. Thus, it may be put anywhere in the listing.

If used from the Interactive session, ON (or an empty argument) will enable the header, while OFF will disable it.

OPTION NULLS ON|OFF

This option controls the printing of the empty string in the MAT PRINT statement. Usually empty strings in MAT PRINT are simply printed empty (length zero), but if strings are columned, this may break the output (printing empty lines or unaligned strings).

The OPTION NULLS OFF disables the nulls effect: the empty string in MAT PRINT is substituted by a series of eleven spaces enclosed in bars:

```
|          |
```

The empty string is made visible, and cannot break the output.
Note: the PRINT statement is not influenced by the OPTION NULLS statement, and empty strings are printed... as empty.

OPTION PRECISION <n>|OFF

This option sets the numerical output precision for decimal numbers; <n> may be any numerical expression whose integer result is taken as argument.
- with PRECISION OFF the default output is set ("automatic" display);
- if PRECISION = 0 the default output is also set ("automatic" display);
- if PRECISION is in the range 1-PRECLIM, then the number is printed

with that precision; in current implementation, PRECLIM=16.
See also the companion statement SET DIGITS.

OPTION PROMPT ON|OFF

This option enables/disables the displaying of the "?" prompt with INPUT and MAT INPUT statements.

OPTION RAWPRINT ON|OFF

Normally, the output is controlled by **tbas**, in the sense that it takes care of new lines, carriage returns and so forth. But the programmer may need something different; for instance, he may want to control the output by himself. He thus needs a raw printing mode.

This option sets the raw printing mode; the comma and semicolon still continue to work, but the raw mode does not take care of the limit of the screen, and thus the programmer must take care of carriage returns.

This option enables also a limited escape character recognition, according to this table:

alarm beep	\a	ASCII 7
backspace	\b	ASCII 8
form feed	\f	ASCII 12
newline	\n	ASCII 10
carriage return	\r	ASCII 13
horizontal tab	\t	ASCII 9
vertical tab	\v	ASCII 11
backslash	\\	ASCII 92
single quote	\'	ASCII 39
double quote	\"	ASCII 34

Note: this option has no effect on MAT PRINT (because matrices follow their own printing way, and thus they gather and print items in the standard way) and on the PRINT USING statements (because they work on single output lines with their own protocols).

OPTION RESET ON|OFF

This option controls the resetting of memory between two consecutive CHAIN calls. See the CHAIN statement for details.

OPTION ROUTING ON|OFF

This option enables/disables the routing of the temporary file (created by QUEUE) to printer. In case OFF is specified, the content of the temporary file is not directed to the printer, and is made available to the user as a textual file under the name '**tbas**_temp<n>.txt', where <n> starts from 1 and is incremented by 1 at each invocation of ROUTE. See the QUEUE, ROUTE and LINE PRINT statements for details.

OPTION SPACING ON|OFF

This option controls the printing of compact string matrices, emitting or not a single space to separate two consecutive strings. Default is OFF (that is string are packed conscutively without spaces).

OPTION TAB <n>|OFF

This option controls the default tab value regulating the printing with comma; a value of <n> in the range from MARGINMIN to MARGINMAX (both defined in tbas.h) sets the tab value to <n>; if <n> is zero or OPTION TAB OFF is used, tab is set to the default value; if <n> is less than MARGINMIN or greater than MARGINMAX tab value remains unmodified. In the current default implementation, MARGINMIN is 3, and MARGINMAX is 132, while the tab value is set to 14.

OPTION TIMING ON|OFF

This option controls the printing of the execution time in seconds of the running program. In case of CHAINed program, the total of the execution of all programs is printed.

No arguments or the argument ON will enable timing, while OFF will disable it.

OPTION VINTAGE

This option tries to rend the behaviour and aspect of programs in monitors of Early Seventies; it sets the following parameters:

ANGLE RADIANS,	BASE 0,	CAPS ON,
CASE OFF,	COMPARISON RELATIVE,	DEBUG OFF,
DIFFERENCE OFF,	ECHO ON,	EXPLICIT OFF,
FORMAT AMERICAN,	HEADER ON,	PRECISION OFF,
PROMPT ON,	RAWPRINT OFF,	RESET ON,
ROUTING OFF,	SPACING OFF,	TAB 14,
TIMING ON,	ZERO OFF.	

Note: since the vintage features should be enabled before any calculation or output, OPTION VINTAGE is preprocessed and executed before any statement. Thus, it may be put anywhere in the listing, but as such any of its settings can be overridden by further OPTION statements.

OPTION WARNINGS ON|OFF

This option enables/disables the printing of warnings (those prepended by '%') that may be undesired in some cases (for instance when one wants to implement his own error messages).

The regular error messages (those prepended by '?') are not influenced by OPTION WARNINGS.

A last advice: don't forget the 'S' in WARNINGS!

OPTION ZERO <n>|OFF

This option enables/disables the zero feature, which prints as zero all numbers that as absolute value are lesser than

OPTION ZERO <n>|OFF

This option enables/disables the zero feature, which prints as zero all numbers that as absolute value are lesser than

$$10^{-n}$$

(e.g. OPTION ZERO 8 prints as zero a number like 2.3E-9), and enables the signed figure 'inf' to be printed in place of the number at the extremes of evaluable numbers.

<n> may be any numerical formula result, whose positive integer part is used; OFF or a value of zero disables the feature. The 'inf' feature is not affected by <n>, of course.

If OPTION ZERO is used without any parameters, the default value of 6 is assumed.

PAGE ALL

See the PAGE# statement.

PAGE#/PAGE ALL/NO PAGE#/NO PAGE ALL

(freely adapted from the LBMAA-A-D DEC document)
Normally, output to the terminal or to sequential access files is not divided in pages; that is, it is in nopage mode. Whenever a sequential access file is assigned to a channel by an OPEN, FILES or a FILE statement, it is automatically set in nopage mode.

The PAGE and PAGE ALL statements allow the user to set a page size to any positive number of lines both for the terminal and for sequential access files. The NO PAGE and NO PAGE ALL statements allow the user to set the terminal and sequential access files to nopage mode. Nopage and page modes are meaningless for random access files. The form of the PAGE statement is:

```
PAGE argl, arg2, . . . argn
```

where each argument has the form:

```
#<c>, numeric formula
```

The arguments can be separated by commas or semicolons. <c> is the channel specifier and the # character is mandatory. The numeric formula is truncated to an integer and used to specify the page size. Either a comma or a colon can be used to separate the channel number from the page size. If only a page size is present in an argument, that argument refers to the terminal.

The form of the PAGE ALL statement is:

```
PAGE ALL [,;] numeric formula
```

This statement sets the sequential access files on channels 1 through 9 to a page size specified by the numeric formula; however, the terminal

is not affected. The value of the numeric formula is truncated to an integer before the page size is set. The comma, semicolon or nothing separates the ALL token and the numeric formula.

The PAGE statement has no effect on random access files or on channels that have no files assigned to them, and cause an error if used in such cases. Consequently, the PAGE ALL statement is a convenient way to set a page size for all of the sequential access files currently assigned to channels.

The page has at least one line. If a PAGE or PAGE ALL statement specifies a page size of zero or less than zero, an error message is issued.

The form of the NO PAGE statement is:

```
NO PAGE arg1, arg2, . . . argn
```

where each argument has the form:

```
[#]<c>
```

where <c> is the channel specifier and the # channel id is optional. If an argument is omitted, the terminal is specified; for example:

```
NO PAGE #3,,2
```

refers to the terminal and the files on channels 2 and 3. Since the NO PAGE statement is assumed to have at least one argument, the statement NO PAGE without arguments refers to the terminal only.

The form of the NO PAGE ALL statement is:

```
NO PAGE ALL
```

The NO PAGE ALL statement sets all of the open sequential access files on channels 1 through 9 in nopage mode, but does not affect the terminal. Like the PAGE and PAGE ALL statements, NO PAGE and NO PAGE ALL statements have no effect on channels that have random access files or no files assigned to them. Consequently, the NO PAGE ALL statement is a convenient way to set all of the sequential access files currently assigned to channels in nopage mode.

PAGENUM ALL

See the PAGENUM# statement.

PAGENUM#/PAGENUM ALL/NO PAGENUM#/NO PAGENUM ALL

The PAGENUM and PAGENUM ALL statements allow the user to enable the printing of the "PAGE #<>" string at the center of last line of the page, during the printing of pages, as the last printed output, even in case of a <PA> or FILLPAGE statement. The NO PAGENUM and NO PAGENUM ALL statements allow the user to disable the feature. Nopage number and page number modes are meaningless for random access files and an error is

generated in case a random file or a not open file is addressed. The form of the PAGENUM statement is:

```
PAGENUM arg1, arg2, . . . argn
```

where each argument has the form:

```
[#]<c>
```

The arguments can be separated by commas or semicolons. <c> is the channel specifier and the # character is optional (the statement alone sets the feature for the terminal).

The form of the PAGENUM ALL statement is:

```
PAGENUM ALL
```

and is directed to all open sequential file channels 1 to 9.

The NO PAGENUM and NO PAGENUM ALL statements disable the feature. Again, the statement NO PAGENUM alone is addressed to the terminal, and the NO PAGENUM ALL is directed to all open sequential file channels 1 to 9.

PIPE

PIPE opens a dedicated virtual channel on the execution process of the bash/DOS command that follows, contained into a string; the virtual channel receives the output of the command, which is stored in an indefinite-length buffer (capable of holding any string output, depending on the system for its maximum dimension); this buffer is reset at each PIPE invocation.

Note: the channel is established only once, during the PIPE execution, and it’s immediately closed afterwards, when the command is complete and the output is returned.

The typical usage is:

```
PIPE "cat aaa"
PRINT PIPE$
```

See PIPE$ for more about its usage.

IMPORTANT NOTE: while PIPE works smoothly with Linux and UNIX machines, and while it works also in Windows with CygWin if run within CygWin bash, this statement does not work in Windows if used from the DOS Prompt.

PREPEND#

PREPEND sets the file in write mode and brings the file pointer to the start-of-file, so that any further printing is inserted before current first line. When a sequential file is opened by FILES, FILE or OPEN in OUTPUT mode, and the file exists at that time, it is automatically set

in append mode and the file pointer is set to the end-of-file; so PREPEND is necessary if you want to add text before the existing text; it is necessary also if your file was opened in INPUT mode (not READ-ONLY), and you want to turn it to OUTPUT mode to add some lines to it. Prepending to a random access file has no meaning.

The PREPEND statement has the form:

```
PREPEND arg1, arg2, ...argn
```

where the arguments have the form:

```
[#]<c> for sequential files
```

where <c> is a channel identifier in the range 1÷9. The # character is optional, since there is no ambiguity, because PREPEND must be used only with sequential files.

At least one argument must be present in an PREPEND statement.

E.g.

```
100 OPEN "Yesterday-report.txt" for INPUT as #3
110 OPEN "Summary.txt" for OUTPUT as #6
120 PREPEND #6
```

sets channel 6 (which was opened in write mode), to prepend-mode; from now on, everything printed to channel 6 will be printed before the existing text. Of course, OPEN must not use the NEW flag and SCRATCH# must not be used before PREPEND#.

Note: the PREPEND statement uses a temporary file to host local information, a file which is deleted after the process is done. The file is a textual file and has a standard name, so it should be compatible with any Operating System. There are two issues:
- the local current running directory must have write privileges, as with any write mode process, but since BASIC programs can be called from within any directories, you must assure to be in a write-enabled directory to run a program which uses PREPEND#
- the PREPEND# statement must not be used in series on the same channel: only one instance of it must be active for each channel; this means that you cannot PREPEND twice or more; but if you should to, instead of writing:

```
PREPEND #1
...
PREPEND #1
....
```

which prints a warning and does not execute the second PREPEND statement, write:

```
PREPEND #1
...
```

```
UPDATE #1
PREPEND #1
....
```

UPDATE forces the closing of the PREPEND feature and the second PREPEND# statement is executed as desired.

PRINT/WRITE

The PRINT statement (that may also be written as WRITE) is used to print terms on the terminal (screen) and has the following form:

```
PRINT list of formulas and delimiters
```

The formulas in the list can be string or numeric or both. The TAB function can be used. The delimiters can be commas, semicolons, or <PA> delimiters; they have the following meanings:

- the comma advances the printing cursor to the next tab position;

- the semicolon concatenates two formulas (and is optional);

- the <PA> delimiter advances to the next page is the page system is enabled (see the PAGE statement and its companions).

When a comma or a semicolon is left as last in the item list, its state is preserved for the next PRINT statement (that is, a semicolon concatenates the next print item, the comma puts the next item on the same line, in the next tab position).

Both comma and semicolon may be iterated, but only the comma produces visible results (advancing to the next tab position without printing).

In normal printing mode, if printing next item means overpassing the terminal length, the output is printed to the next line; in raw printing mode, there is no control over the screen length; such control must be provided by the programmer.

If used for printing on the terminal (PRINT or WRITE, no channel specified), the two statements are completely equivalent. In this case, WRITE is a mere synonym of PRINT, as it does not print any line number preceding the output.

The normal mode of output for WRITE and PRINT statements to the terminal is noquote mode. In noquote mode, strings are not enclosed in quotes. Also, strings are concatenated if they are output with a semicolon separating them. The terminal is automatically set in noquote mode. In order to write terms in quote mode, the QUOTE or QUOTE ALL statement for terminal must be used, both of which are described in following paragraphs. When the terminal is in quote mode, **tbas** accepts WRITE and PRINT statements that are in the usual form but it makes whatever small changes that are necessary to the formatting in order to preserve the integrity of the data items.

E.g.

```
 name$="JULIUS CAESAR"
 A=B=24
 C=D=102
 E=SQR(3)
 PRINT "THE BRUTUS LOTTERY'S WINNER IS ";name$
 PRINT A,B,C,D;E;F
 PRINT A,,3.4*sqr(2)

 THE BRUTUS LOTTERY'S WINNER IS JULIUS CAESAR
  24            24            102           102  1.73205  0
  24                          4.80833

 PRINT "Bruce is a good boy, good companion!",34*9.87/SQR(2),-1
 QUOTE
 PRINT "Bruce is a good boy, good companion!",34*9.87/SQR(2),-1

 Bruce is a good boy, good companion!              237.291       -1
  "Bruce is a good boy, good companion!"           237.291        -1
```

See the QUOTE statement for further details.

PRINT#/WRITE#/PRINT:/WRITE:

(freely adapted from the LBMAA-A-D DEC document)
The WRITE and PRINT statements write data items to files. The behaviour of such statement depend on the file type on which it's going to operate.

Using sequential files

The WRITE and PRINT statements for sequential access files have the following forms:

```
 WRITE #<c>, list of formulas and delimiters
 PRINT #<c>, list of formulas and delimiters
```

where <c> is the channel specifier. The delimiter following <c> can be a comma or a colon; it can be omitted if the list is omitted. The formulas in the list can be string or numeric or both. The TAB function can be used. The delimiters can be commas, semicolons, or <PA> delimiters; they have the same meanings that they have in the PRINT statement for the terminal. WRITE and PRINT statements for sequential access files differ from one another in the following way. The WRITE statement begins each line of output with a line number followed by a tab. The first line in the file is numbered 1000 and subsequent line numbers are incremented by 10. The PRINT statement, on the other hand, does not begin lines with line numbers. It is illegal to use both WRITE and PRINT statements to write to the same sequential access file unless the file has been erased (by means of the SCRATCH command) between the two types of statements. An attempt to mix WRITE and PRINT statements result in a fatal error message. Files created by WRITE statements are normally read by READ statements. Files created by PRINT statements are normally read by INPUT statements.

The normal mode of output for WRITE and PRINT statements for sequential access data files is noquote mode. In noquote mode, strings are not enclosed in quotes even if they contain characters that the READ and INPUT statements see as delimiters. Also, strings are concatenated if they are output with a semicolon separating them. Noquote mode is the mode used when writing a text file. Noquote is the default mode; a sequential access file is automatically set in noquote mode wher it is assigned to a channel by a FILE or FILES statement. However, noquote mode is not suitable when writing pure data files because the integrity of the data is not maintained. In order to write a pure data file, the file must be set in quote mode. This can be done by the QUOTE or QUOTE ALL statement, both of which are described in following paragraphs. When a file is in quote mode, **tbas** accepts WRITE and PRINT statements that are in the usual form but it makes whatever small changes that are necessary to the formatting in order to preserve the integrity of the data items.

PRINT and WRITE subtleties

The PRINT# and WRITE# statements, if used with channel numbers from 1 to 9, bear a little difference: while PRINT# prints the output as is, the WRITE# statement prints a line number before any new line of output; this line number starts from 1000 and steps up by units of 10 for each output. This process is paired with the READ# statement, that expects a line number before any new line, and discards it, while INPUT# does not expect any line number, and if one is found, it’s reads as a plain number.

An important note: if you start using one statement (WRITE# or PRINT#), you must go on with the same command, or an error will be raised; this is because the two types must not mix, or the resulting file will not be readable by READ# or INPUT# respectively.

Using random access files

The WRITE and PRINT statements for random access files have the forms:

```
WRITE :<c>, formula, formula, . . . formula
PRINT :<c>, formula, formula, . . . formula
```

where <c> is the channel specifier. The delimiter following the channel specifier can be a comma or a colon. At least one formula must be present in each statement. The formulas are separated from one another by a comma. In a given statement, all of the formulas must be string or all of them must be numeric because a random access file is either string or numeric but not both.

WRITE: and PRINT: statements for random access files are exactly equivalent; they both begin writing to the record that the pointer for the file specifies, and continue writing sequentially until all of their arguments have been written. It is legal to use both WRITE and PRINT statements to write to the same random access file.

PRINT:

See the PRINT# statement.

PROGRAM/TITLE

The PROGRAM statement (that may be written also as TITLE and as _TITLE) is an informative statement that must be located in the first line of the program. It is followed by any characters sequence (that may be enclosed in quotes) that is printed, along with two empty lines. All blanks after PROGRAM are ignored . E.g.:

```
 PROGRAM Test for PRINT
 PRINT SQR(45)

 Test for PRINT


  6.7082
```

In case it is met elsewhere, PROGRAM raises a syntax error.

PUT#

The PUT# statement writes a character to an opened file, according to the following syntax:

```
 PUT arg, <sv>
 PUT arg, <nv>
```

where argument arg has the form:

```
 [#]<c> for sequential files only
```

<c> is a channel identifier, whose integer part is taken, in the range 1÷9, identifying an open channel; <sv> is any string variable or string array element, and <nv> is any numeric variable or numeric array element.

If PUT# is followed by a string variable, it converts the first character of the string to its ASCII code and writes it to file; otherwise, it writes the numerical value, converted to ASCII code (that is reduces to the range 0÷255, to file.

See also GET#.

E.g.

```
 REM THIS PROGRAM SIMULATES cp
 OPEN "prices.dat" FOR INPUT AS #3
 OPEN "prices.cop" FOR NEW OUTPUT AS #6
 WHILE NOT EOF #3
    GET #3, A$
    PUT #6, A$;
 WEND
```

Another example: if you have the file "test.txt" with this content:

```
 AAAA
 BBBB
 CCCC
 DDDD
```

the following program

```
 OPEN "test.txt" FOR INPUT AS 2
 OPEN "test2.txt" FOR NEW OUTPUT AS 4
 PRINT "File test.txt contains the ASCII values:"
 WHILE NOT EOF(2)
    GET#2, F
    PRINT F,
    IF F<>CR THEN
        PUT#4, F+INT(RND*10)
    ELSE
        PUT#4, F
    END IF
 WEND
 PRINT
 PRINT "File test2.txt contains instead:"
 RESTORE #4
 WHILE NOT EOF(4)
    GET#4, A$
    PRINT A$;
 WEND
 CLOSE 2,4
```

has the following output:

```
 File test.txt contains the ASCII values:
  65            65            65            65            10
  66            66            66            66            10
  67            67            67            67            10
  68            68            68            68            10
 File test2.txt contains instead:
 HAED
 ECDF
 CDLE
 IHHM
```

Notice that I try to preserve the carriage return by comparing it with CR.

QUEUE

Modern printers cannot be used as line-printers, as the old pin-printers did on punched paper-sheets. Modern printers (ink-jet or laser type) output one sheet per time.

So **tbas** does not really work with printers; rather, it redirects, when required, the output to a temporary file, which is printed when the

ROUTE statement is met or when the program ends.

The QUEUE statement enables the redirection of all subsequent PRINT statements to print on the temporary file rather than on the screen. The syntax is:

```
QUEUE
QUEUE <filename>
```

If used alone, QUEUE creates a temporary file in the system; this file will be deleted after the sending to printer or the end of the program (unless the OPTION ROUTING OFF is used).

If followed by a string value (a variable or a string in double quotes), this will be taken as the temporary file name (with path if necessary) and this will not be deleted after printing.

See the chapter "QUEUE and ROUTE in details".

QUOTE ALL

See the QUOTE# statement.

QUOTE#/QUOTE ALL/NO QUOTE#/NO QUOTE ALL

(freely adapted from the LBMAA-A-D DEC document)
The default mode for output to sequential file access data files or to the terminal is no-quote mode. The QUOTE and QUOTE ALL statements allow the user to change the mode of the terminal and sequential access files to quote mode. Quote mode changes the way that the data items are written into the files or onto the terminal. In quote mode, strings are enclosed in double quotes by **tbas** if they contain blanks, tabs, or commas; a leading blank is output immediately before strings and negative numbers; and a double quote character cannot be output by the user. If such an attempt is made to output a double quote character, an error message is issued. Also a data item cannot be longer than the maximum amount of space available on a screen line. If an attempt is made to output a data item longer than this, a fatal error message results. In no-quote mode, the data item would be split across two or more lines. These modifications to the normal formatting are sufficient to insure that the integrity of the data is maintained.

The opposite of quote mode is noquote mode, which can be set by the NO QUOTE and NO QUOTE ALL statements. Noquote mode is the default mode for the terminal and sequential access files. Whenever a sequential access file is assigned to a channel by an OPEN, FILES or a FILE statement, it is automatically set in noquote mode. NO QUOTE and NO QUOTE ALL statements are only necessary if the user wishes to change a file from quote to noquote mode. When creating a pure data file, in addition to setting the file in quote mode, it is good practice to separate the formulas in the WRITE or PRINT statements with semicolons to pack the data items close together.

The form of the QUOTE statement is:

QUOTE argl, arg2, . . . argn

where each argument has the form #<c>

where <c> is the channel specifier. The :<c> argument causes an error. Since QUOTE is assumed to have at least one argument, the statement QUOTE without arguments refers to the terminal. The form of the QUOTE ALL statement refers to open sequential channels 1 through 9, but not to the terminal.

When a channel is referenced in a QUOTE or QUOTE ALL statement and that channel has a sequential access file currently assigned to it, output to the file is done in quote mode. If a sequential access file is not presently assigned to the channel, nothing is done and no error message is returned, and in this case the quote state is not changed.

The form of the NO QUOTE statement is the same as that of the QUOTE statement, except that the word NO QUOTE is substituted for the word QUOTE.

The use of the QUOTE ALL or NO QUOTE ALL statement is a convenient way to set all sequential access files currently assigned to channels in the appropriate mode, since the statements will not return error messages about or affect unassigned channels or the terminal and will not damage any of the random access files currently assigned to channels.

Quote or noquote mode can be set even if the file is in read mode because these modes have no effect on input. They will affect the output if the file is subsequently put in write mode. If the mode is changed from quote to noquote or vice versa, the change takes effect immediately (but only for open sequential files).

RANDOMIZE

The RANDOMIZE statement recalculates the seeds for the Random Numbers Generator.. The algorithm depends on current time, and so you may RANDOMIZE in two near instants and get quite different seeds.

See the RND function.

READ

(freely adapted from the LBMAA-A-D DEC document)
The READ statement is used to load all information stored in the source code by DATA. READ assigns to the listed variables those values which are obtained from a DATA statement. Neither statement is used without the other.

Before the program is run, the interpreter takes all of the DATA statements in the order they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data engine supplies the next available datum.

E.g.

```
 DATA 1,"Main St.",45,KINGSTON
 READ N,addr$,weight,town$
```

See DATA for further details.

READ#

See the INPUT# statement.

READ:

See the INPUT# statement.

REDIM

The REDIM statement redimensions existing arrays, with the following syntax:

```
 REDIM <t1>(<n1>,<n2>),<t2>(<n3>,<n3>),...
```

where <t> are names of numeric or string arrays, and <n> are the new row and column values you want to set the array to.

REDIM tries to conserve memory if possible. Actually, if the new row and column values determine a space which can be contained in the original array space, a simple redefinition of inner parameters is done and no new memory is involved. If the new row and column values determine a space which is larger that the existing space, a new space is created (if possible), the old space is copied in the new space and the array is completely redefined; in such a case, the original space is freed and the array appears and behaves as if it was defined as such.

E.g.

```
 DECLARE arr(20,15)
 ...
 REDIM arr(15,10) ' the memory is retained
 ...
 REDIM arr(30,20) ' new memory is created
```

The first DECLARE sets up a memory space of 21x16=336 items; the first REDIM uses only 16x11=176, and thus the original 336 items space is retained and the array is resized to use only 176/336 of the items.

The second REDIM needs a total of 31x21=475 items, which cannot fit the original space; thus, new space is granted, and the array is set up to 475 items; it's at all effect as if arr() was created as:

```
 DECLARE arr(30,20)
```

in the first line.

If the memory expansion is not possible (the OS does not accept this resizing), a message appears and an error is raised.

RESET

See the RESTORE statement.

RESET#

See the RESTORE# statement.

RESET:

See the RESTORE# statement.

RESTORE/RESET

The RESTORE statement for DATA (which can be typed also as RESET and must not to be confused with the RESTORE statement for files) permits resetting the DATA pointer and lets READing data in DATA statements of a program more than once. Whenever RESTORE is encountered in a program, **tbas** restores the data block pointer to the first number and to the first string in the correspondent first DATA statements. A subsequent READ statement then starts reading the data block all over again. There are several syntax formats:

```
RESTORE
RESTORE *
RESTORE $
RESTORE <n>
RESTORE <n>*
RESTORE <n>$
```

RESTORE alone resets both strings and numbers blocks, but if RESTORE is followed by * (asterisk), it restores the numbers block only, and if followed by $ (dollar sign) it restores the strings block only. This lets using different data blocks in different file positions.

If RESTORE is followed by a number, it restores the data block to label, if found. If after the number an asterisk is found, **tbas** restores only numbers, if there is a dollar sign, restores only strings, if there's nothing, tries to restore both.

If you use the number format with * or $, and if the line label does not exist, an error is raised. If you specify a number only, the resetting is tried on both blocks, but remember that if the label does not exist, the RESTORE has no effect on the respective block. Let's examine an example program.

E.g. if I have the following DATA block:

```
10 REM VARIOUS DATA BLOCK
20 DATA 1,2,3,4, "TITLE"
25 DATA"FIRST STRING",0
26 DATA"SECOND STRING"
30 DATA 5,6,7,8
```

If some values are previously read (suppose two numeric items, just to move up the job):

```
 READ E,T
 PRINT E,T
```

I get of course:

```
  1             2
```

You can try the following examples successively, and see what changes; in the following the piece of code and the relative output is shown; first of all a simple restoring:

```
 RESTORE
 READ R$,R
 PRINT R$,R

 TITLE          1
```

Now both blocks point to second item; using now:

```
 RESTORE*
 READ R$,R
 PRINT R$,R

 FIRST STRING   1
```

The numeric block was reset. Now string block points to item 3 and numeric block points to item 2; using now:

```
 RESTORE$
 READ R$,R
 PRINT R$,R

 TITLE          2
```

The string block was reset. Now string block points to item 2 and numeric block points to item 3. Let’s try now to reset to a specific line number:

```
 RESTORE 30*
 READ R$,R
 PRINT R$,R

 FIRST STRING   5
```

Notice that the first item in line 30 is 5. With strings:

```
 RESTORE 26$
 READ R$,R
 PRINT R$,R

 SECOND STRING  6
```

The numeric block was not reset. Now look at this: I reset to line 25 where there are both a numeric and a string item:

```
 RESTORE 25
 READ R$,R
 PRINT R$,R

 FIRST STRING   0
```

If you specify a generic line label where there is only type of items, the other does not change:

```
 RESTORE 26
 READ R$,R
 PRINT R$,R

 SECOND STRING  2
```

Or, on the other side, if you try to reset both blocks to line 30 (where there are numbers only), the string pointer remains unchanged but...:

```
 RESTORE 30
 READ R$,R
 PRINT R$,R
 ? Out of DATA in line 45.
 READ R$,R
        ^
```

Strings ran out... The complete program is:

```
 10 REM VARIOUS DATA BLOCK
 20 DATA 1,2,3,4, "TITLE"
 25 DATA"FIRST STRING",0
 26 DATA"SECOND STRING"
 30 DATA 5,6,7,8
    READ E,T
    PRINT E,T

    RESTORE
    READ R$,R
    PRINT R$,R

    RESTORE*
    READ R$,R
    PRINT R$,R

    RESTORE$
    READ R$,R
    PRINT R$,R

    RESTORE30*
    READ R$,R
    PRINT R$,R
```

```
    RESTORE 26$
    READ R$,R
    PRINT R$,R

    RESTORE 20
    READ R$,R
    PRINT R$,R

    RESTORE 26
    READ R$,R
    PRINT R$,R

    RESTORE 30
    READ R$,R
    PRINT R$,R
```

And the complete output is:

```
  1             2
 TITLE          1
 FIRST STRING   1
 TITLE          2
 FIRST STRING   5
 SECOND STRING  6
 TITLE          1
 SECOND STRING  2
 ? Out of DATA in line 38.
 READ R$,R
        ^
```

A NO DATA statement would help, here... If I specify an label that does not exist:

```
 RESTORE 100
```

both indexes remain unchanged, but if I specify a type:

```
 RESTORE 100$
 ? RESTORE couldn't find string DATA line 100 in line 7.
 RESTORE 100$
             ^
```

RESTORE#/RESTORE:/RESET#/RESET:

(freely adapted from the LBMAA-A-D DEC document)

The RESTORE statement for files (which can be typed also as RESET and not to be confused with the RESTORE statement for DATA) has the form:

```
 RESTORE argl , arg2, ...argn
```

where the arguments have the form:

```
 #<c> for sequential files
 :<c>   for random access files
```

where <c> is a channel specifier. At least one argument must be present in a RESTORE statement.

Restoring a sequential access file sets the file in read mode. Reading will start at the beginning of the file. When a sequential access file is opened by a FILES, FILE or OPEN statement and the file exists at that time, it is automatically set in read mode; it is not necessary to restore it. It is only necessary to restore a sequential access file if it has been set in write mode and the user wishes to set it to read mode in the same program.

Restoring a random access file simply sets the pointer for the file to the first record in the file. When a random access file is opened on a channel by FILES, FILE or OPEN statement, its pointer is automatically set to point to the first record of the file.

Note: a difference with the DEC-20 does exist for RESTORE; while the DEC-20 could write

```
RESTORE 4
```

to mean "restore sequential file on channel 4", **tbas** would interpret it as "restore DATA pointer to label 4"; this means that for **tbas** the # character is necessary to distinguish between the two modes, and thus the **tbas** equivalent to previous expression is

```
RESTORE #4
```

RESUME

When ON ERROR/ON ATTENTION conditions are set, normally the redirection is done to a piece of code able to treat the error condition. In this case, after the error treatment, the RESUME statement is useful to pass back the program control to the line following the one that caused the error condition or where the user interrupt occurred. It is the CONTINUE equivalent of the WHEN ERROR routines.

E.g.

```
 ...
 ON ERROR 400
 ...
 OPEN "shiva.bas" FOR INPUT  ' note: shiva.bas does not exist
 ...                         'RESUME gets back here
 ...
 STOP
 400 REM ERROR TREATMENT CODE:
     REM solve for shiva.bas not existing
     REM by creating a new void "shiva.bas" file
     OPEN "shiva.bas" FOR NEW OUTPUT AS #9
     CLOSE #9
     REM re-execute the opening
     OPEN "shiva.bas" FOR INPUT
     RESUME
```

RESUME cannot be used independently of ON ERROR/ON ATTENTION and if **tbas** encounters a RESUME in legal ON ERROR/ON ATTENTION pieces of code but with no error condition raised or no user interrupt, an imperative error message will be printed and execution will stop.

RETRY

The RETRY statement provides a useful way in the WHEN ERROR error treatment routines (USE or HANDLER section - see the WHEN ERROR statements) to re-execute the faulty code, provided the error condition was solved in the handler (or this would cause a loop).

RETRY must be used only in a WHEN ERROR handler. See the WHEN ERROR statements for further details.

RETURN (without argument)

The RETURN statement without argument closes the subroutine invoked by a GOSUB and returns back to the instruction that follows the one that called it.

This statement, as said, has no arguments. More than one RETURN may be used in the same subroutine, and the first executed will set the return phase.

It's illegal to use RETURN out of a GOSUB call.

RETURN (with argument)

The RETURN statement with argument registers the return value for the function defined through SUB and marks it as 'function'. The absence of a RETURN statement in a SUB marks it as 'procedure'.

It does not automatically return from the SUB. See SUB for details.

ROUTE

The ROUTE statement sends the content of the routing temporary file to printer, and disables the QUEUE statement, setting regular PRINT to screen again (unless LPRINT or another QUEUE are used).

When the ROUTE statement is called, the temporary file created by the user, as said, remains available under the given name, but if the OPTION ROUTING OFF is set, in case of a temporary file not created by the user (QUEUE was called without arguments), the content of the temporary file is copied back to the file '**tbas**_temp.txt" in the current directory, before its deletion. The user can so use the output file (made available as a textual file) without the effective and actual use of a printer.

The ROUTE statement is not really necessary in case all the output must be redirect to printer, because at the end of the execution, **tbas** checks if there is an active queueing process, and if the case executes an implicit ROUTE to send the temporary file content to printer.

See also the chapter "QUEUE and ROUTE in details".

SCRATCH#/SCRATCH:/FREE#/FREE:

(freely adapted from the LBMAA-A-D DEC document)
The SCRATCH statement (that may be written as FREE) has the form:

```
SCRATCH arg1, arg2, ...argn
FREE arg1, arg2, ...argn
```

where the arguments is the same as of the FILE statement. Each argument has the form:

```
[#]<c> for sequential files
:<c>   for random access files
```

where <c> is a channel specifier. At least one argument must be present in a SCRATCH statement. Scratching a sequential access file erases it and sets it in write mode. Writing will start at the beginning of the file. Scratching a random access file simply erases it and sets the pointer to the first record in the file.

SCRATCH:

See the SCRATCH# statement.

SELECT

The construct SELECT/CASE/CASE DEFAULT/END SELECT is not classical in BASIC, but it's widely accepted in all modern BASICs. **tbas** accepts the following syntaxes:

Value driven SELECT

```
SELECT <formula>
   <stat1>
   <stat2>
   ....
   CASE [=] <value1> [TO value]
        <statement>
        <statement>
        ...
   CASE [=] <value2> [TO value]
        <statement>
        <statement>
        ...
   ...
   CASE DEFAULT|CASE ELSE
        <statement>
        <statement>
        ...
END SELECT
```

The <formula> return type determines the type value; if <formula> is a

string expression, the return value will be a string, and the CASE sections will compare strings equality (numbers must be enclosed in double quotes, or filtered by the STR$() function); otherwise, <formula> is a numeric expression supposed to return a number, and thus CASE sections will compare number equality (remember that, for number storing reasons, the equality is not guaranteed with floating-point decimals). CASE may be followed by an optional '=' sign.

The DEFAULT/ELSE section, optional, encloses all the unevaluated CASEs and serves as a catch-all case.

Statements following the SELECT line (<stat1> etc..), not strictly related to the CASEs, are simply executed, with no effect on the <formula>.

The CASE comparison may be grouped in ranges; see the following example in case of numeric expression:

```
CASE 1 TO 10
   PRINT "low"
CASE 11 TO 20
   PRINT "high"
```

This is a mere substitution for a succession of statements like CASE 1 ..., CASE 2 ..., and so on. Even strings can be ranged:

```
CASE "Algiers" to "Geneve"
```

Strings are compared character by character unless they are not equal or there are not any characters left to compare; the example includes all the cases where words are greater or equal than "Algiers" (this means they have the same letters or some more, e.g. "Algiers1" and "Algiert" are both greater than "Algiers" and thus in the range) or words that are less or equal than "Geneve" (this means they have the same letters or some less, e.g. "Geneva" or "Genev" are both lower than "Geneve", and thus in the range); any string with intermediate letters is in the range, no matter how long it is (e.g. "Festival of Stonehenge" is in the range, even if very long).

Mute SELECT

In case you don't specify any variable after SELECT, it begins a 'mute' process; this version analyses all CASEs to check for the first truth value found; this brings versatility, because truth values may be originated by many diverse comparisons, and not only by using the variable in the SELECT clause:

```
SELECT
       CASE <true>
               ...
       CASE <true2>
               ...
       CASE <true3>
               ...
       CASE DEFAULT|CASE ELSE ' (all remaining cases)
```

```
 END SELECT
```

The truth value may be originated by a number comparison, a number alone (with the usual convention that 0 is false and not-0 is true), or a string comparison of any variable/array item, that is the mute select has the widest range of possibilities.

NOTE: ranges cannot be used in a mute select.

SET DIGITS

The SET DIGITS statement sets the printer precision of numerical values, with the following syntax:

```
 SET DIGITS <n>
 SET DIGITS(<n>)
```

where <n> is any integer value in the range 0÷16.

- if <n> < 0 the default output is also set ("automatic" display);
- if <n> is in the range 0-PRECLIM, then the number is printed with that precision; in current implementation, PRECLIM=16. In particular, if <n> = 0, all numbers will be printed as rounded integers.

E.g.

```
 PRINT SQR(2)
  1.41421

 SET DIGITS 10
 PRINT SQR(2)
  1.4142135624

 SET DIGITS -1  'restores automatic display
```

Note: the real effective precision of numbers when <n> approaches 16 depends on C and the gcc compiler.

See also the companion statement OPTION PRECISION, and the chapter "Number format".

SET

(freely adapted from the LBMAA-A-D DEC document)
SET acts on random files only, setting the file pointer to the argument data. It has the following syntax:

```
 SET argl, arg2, . . . argn
```

where the arguments can be separated by commas or semicolons. Each argument has the form:

```
 [:]<c>, numeric formula
 [:]<c>: numeric formula
```

where <c> is the channel specifier. The colon preceding the channel specifier can be omitted because SET is only used for random access files. Each SET statement must have at least one argument. When the SET statement is executed, the pointer for the file on the specified channel is moved so that it points to the item in the file that is specified by the numeric formula, which has been truncated to an integer. If the numeric formula after truncation is less than or equal to zero, an error message is issued. The items in the file are numbered sequentially; the first item in the file is 1, the second 2, and so forth. The next statement in the program that reads from or writes to the random access file will read or write the item to which the pointer was set, provided that the pointer has not been moved again by a subsequent SET statement or another statement.

STOP

See the END statement.

[DECLARE] SUB

The first BASIC compilers/interpreters had a very primitive way of dealing with subroutines, a way that required great skill and a lot of calculations: the GOSUB/RETURN structure.

tbas is completely compatible with these instructions, provided the GOSUB target has a proper label. The jump is unconditioned, and when a RETURN is found (the first found), a back jump is automatically done to the instruction following GOSUB.

But **tbas** has more. You can build any kind of subroutines, with any number of arguments and one return value. The structure is:

```
[DECLARE] SUB <name>(<arg list>)
   EXPORT <var list> | ALL
   ....
   ... EXIT SUB
   ...
   RETURN <val>
   ...
END SUB
```

The token DECLARE is optional and serves uniquely for compatibility with other dialects.

If <name> is followed by $, the return value (if any) must be a string, otherwise it is a number, or it can be none. If at least one RETURN is found in the SUB listing during the pre-parsing phase, the subroutine is marked as a function, otherwise it is a procedure. The difference is subtle: functions can enter any calculations and condition tests, but cannot be directly executed; procedures, on the contrary, can be executed as any statement, but cannot be part of any calculation. If you omit the RETURN in a function, **tbas** won't complain for this, but if you use it in a calculation it will. If you put the RETURN in a procedure, this last will be turned to function, and again **tbas** won't emit a sigh,

but if you try to use it as a statement it will.

The case of the letter does not count: 'name' and 'NAME' refer to the same SUB.

<arg-list> is a list of variables arguments (not related with the variables of the main program), that yield the variables/arrays contents to be passed to the SUB. Arrays are specified by means of the name followed by '()', no other special character is used: the programmer must declare arrays before passing them to the SUB, otherwise 10x10 matrices will be used in their place in the body of the SUB. If an argument name is followed by $, **tbas** expects a string type there in the call, otherwise it expects a number type.

The RETURN doesn't need to be the last statement in the SUB and you can have any number of them. Its purpose is to register the return value; e.g.

```
SUB myfunc(F, M())
   RETURN -1
   ...
   IF ... THEN EXIT SUB
   ...
   IF ... THEN EXIT myfunc
   ...
   RETURN F*M(I,J)
   ...
   ...
END SUB
```

In the previous example, myfunc is set to return -1 as a default value, and if you anticipate the exit by means of EXIT SUB, because of some specific conditions, no problem, the returned value is registered; if you get past the EXIT condition and reach last RETURN, the return value will be changed to the calculated value; executing all remaining instructions won't affect at all the registered value, unless another RETURN is used. Incidentally, if you use RETURN as the last instruction, **tbas** BASIC will behave as any other language, returning the given value.

I want to remark here that the RETURN statement does not mean 'return' in the sense of 'get it back now' but in the sense of 'register'. That's why it has not to be the last instruction, and when met, it does not cause immediate return (like C, for instance). To use it in the sense of 'get it back now' RETURN should be followed by EXIT SUB or END SUB. But use it as you like.

Please notice here the behavior of EXIT, which, if is followed by SUB, ends current SUB, but if the programmer wants to make the sub name explicit, he may also write:

```
EXIT myfunc
```

where myfunc is the SUB name. If myfunc is a string function, the '$' character must explicitly be written:

```
 EXIT myfunc$
```

All the variables declared or used for the first time in the SUB subroutines are local. See elsewhere for the local/global variables system and the EXPORT statement. If you use recursive calls to a SUB, it's safer to DECLARE all local variables inside the SUB, so that all recursive calls will set these local variables by their own. On the contrary, all SUBs from second recursive call on, will see the local variables of the first call as global, because these variables were declared before the following recursive calls.

If the SUB has no arguments, it can be defined and called with or without parentheses:

```
 SUB myfunc
        ....
 END SUB

 myfunc          ' naked
 myfunc()        ' empty parentheses
```

When EXIT SUB is found, the SUB will execute END SUB, and resume to the next instruction after the one where the SUB was called.

Note: variables and arrays passed to SUBs are passed by value; that is, the expression passed as argument is evaluated and the correspondent value is assigned to the variable argument; arrays are copied in full. There is no implemented way to pass variables by address or reference.

Note: The statements GOSUB and RETURN (without argument) are not allowed in SUBs.

SWAP

The SWAP statement exchanges the content of the two variables that follows. Syntax:

```
 SWAP <nv1>, <nv2>
 SWAP <sv1>, <sv2>
```

where <nv> and <sv> are respectively numerical and string variables or arrays elements. Condition for the good working are:

- the two arguments must be variables, not literal numbers
- the two arguments must be coherent, that is both numeric or both string-type.
- you cannot exchange whole arrays, but only array elements.

Examples:

```
 DECLARE I%=35,J=56
 PRINT I%,J
 SWAP I%,J
 PRINT I%,J
```

```
 35            56
 56            35

DECLARE I$="ANNE",J$="SYLVIE"
PRINT I$,J$
SWAP I$,J$
PRINT I$,J$

ANNE          SYLVIE
SYLVIE        ANNE
```

SYSTEM

See the END statement.

TITLE

See the PROGRAM statement.

UPDATE

The UPDATE statement updates the file-state in the argument channel, ensuring data flushing. UPDATE also closes any PREPEND# state on the channel arguments, so that a new fresh PREPEND# may be set anew afterwards.

UPDATE may be used with the suffixes # and :, but they are only adornments. The UPDATE statement proper function needs the channel number only, which may be given as a pure number; no control is done to see if the suffix matches the real file state (e.g. using UPDATE# on a random access file), because UPDATE does not check if file is random or sequential or if a random access file is numeric or string.

The syntax is elementary:

```
UPDATE arg1, arg2, ...argn
```

where the arguments have the form:

```
[#]<c>  for sequential files
[:]<c>    for random access files
```

where <c> is a channel specifier; the # character for sequential files is optional.

UPDATE does not report any error message or warning in case the channel in the argument queue is not open, or if no flushing was done, because its action is transparent.

E.g.

```
UPDATE #2,:4,6
```

Update sequential channels 2 and 6 and random access channel 4.

USE

See the WHEN ERROR IN statement.

PRINT with USING specifics

PRINT USING and USING let the user print a string with specifics established according to some rules. First of all the specifics are contained in a quoted string, in a string variable or in an image line; here are examples of all of them:

```
PRINT USING "####",A

form$="####"
PRINT USING form$,A

10:####
PRINT USING 10, A
```

The second and third formats are reusable.

There are two specific formatters types, one for numbers, two for strings.

Any character that is not a formatting character or is not in the positions specified by the formatting protocols, stands by itself. This means that if any kind of text is inserted in the formatting string, this text will be written in the output queue.

As an independent statement, USING can be used to establish a common format for all subsequent PRINT statements with the same formatting rules of PRINT USING (to disable, use USING alone or USING OFF; also a PRINT USING statement disables the USING setting).

```
USING "###.#######" ' SETS USING
A=3.4
PRINT SQR(A)
PRINT A*A
PRINT A
USING OFF
PRINT SQR(A)
PRINT A*A
PRINT A
```

to get the following input:

```
   1.8439089
  11.5600000
   3.4000000
  1.84391
  11.56
  3.4
```

The first three PRINT use the USING format, the USING OFF disables the

feature, the next three PRINT use the default format.

If you set a format, only items matching the format can be printed; if for instance you set for:

```
 USING "###.###"
 PRINT "HELLO"
```

you get the error

```
 ? Attempt to output a number to a string field or vice versa.
```

So use it with care.

Number format

A number format is introduced by at least two characters among '#' and '.'; if only one occurrence of these characters is found, it is treated as the character itself. E.g.

```
 "###.###"
```

is a valid numeric format value. A '#' character signals a digit position, the '.' dot sign signals the position of the dot in the decimal representation. It has to be noted that the sign has always a place, and that the zero unity is always expressed, thus the first two positions should be always two '#' signs, but **tbas** won't complain if there's only one or none.

If no '.' dot is in the image file, the format is called "integer representation". Only one dot may be put in a numeric format, or the second dot will start another number format.

Some other characters may influence the output of numbers: '^^^^', '-', '0', '$', '*', ','

Four '^^^^' circumflex characters signals the exponential format, but this is not available for integer representations, where they stand for themselves; the '^^^^' characters are counted in the image representation to take in account the 'E', the sign ('+' or '-'), and two digits (for the exponent). Any lesser quantity of circumflexes (e.g. ^^^) stand for themselves, while any greater quantity of circumflexes cause the exceeding characters (after the first four) to stand for themselves. It is worthwhile noting here that the exponential format fills the entire integer space image format, adapting the exponent. For instance, see the following program:

```
 PRINT USING "######.#####^^^^",123456789
 PRINT USING "#####.#####^^^^",123456789
 PRINT USING "####.#####^^^^",123456789
```

which prints:

```
 12345.67890E+04
 1234.56789E+05
```

```
 123.45679E+06
```

(the first empty space is for the sign): the exponent changes because the integer space changes.

The '-' minus character, put at the end of the number format, signals that a negative number is printed with a trailing ending minus (bank account format). Positive numbers are printed without sign.

If a numeric image begins with two or more '0' zero characters (math scopes) or '*' asterisk signs (used when printing checks, for instance), the number is output with leading zeroes or asterisks characters filling the unused positions in the output field (i.e. the number does not fill entirely the output field). If a numeric image begins with two or more '$' dollar signs, the number is output with a floating dollar sign in front (unfortunately, for those who live in countries where other money formats are in use, like the GB Pound, the Euro or the Yen, it must be noted that the ASCII table - made by Americans - has only the dollar sign; trying to use extended characters may result in violation of **tbas** interpreter capability. If you have an extended-ASCII-capable terminal, try using your own money character in a fixed position). The zero, asterisk and dollar sign formats cannot output directly a minus sign; thus, in case the number may also be negative, use the '-' trailing minus sign, to have the minus printed at the end. The exponential sign '^^^^' is not available with '0', '*' and '$'.

One or more commas in the integer part of the numeric image specification cause the digits in the integer part to be separated every three unities (hundreds, thousands, etc.) separated by commas. The comma itself counts as a character in its own, so that you have not to use extra characters. However, for what said before, a place should be taken in account for the leading minus in case of negative numbers, and the comma is not available as a numeric image starter character.

Note: the precision of a number printed through USING is subject to the floating point rules of the C-double format. After the 15th decimal position the precision is lost; this means that if you try printing something like:

```
 PRINT USING "####################",1 + 2.3E18
```

you will get

```
 23000000000000000000
```

and the last digit (the appended 1) is lost; there is too much distance in magnitude between 2.3E18 and 1. The maximum available magnitude, basing on the current C-double format storing capabilities, is 14, that is

```
 PRINT USING "####################",1 + 2.3E14
```

returns correctly

```
     230000000000001
```

but higher magnitudes lose any significant digit beyond that. Remember this limitation when you've got to print long integers. Roughly, the limit is between 2^49, which is precise, and 2^50 which loses some decimals.

The same can be said for the decimal part of decimal numbers; if you try to print something like:

```
 PRINT USING "##.##################",2/3  ' 18 decimals
```

you'll get

```
 0.666666666666666963
```

that is, anything beyond the 15th decimal is rubbish, literally (you may find different rubbish numbers on your computer). Remember this limitation when you've got to print long decimal parts (incidentally, this is the reason why OPTION PRECISION is limited to 16 as the highest argument value).

Summing up: the precision of a number output is confined in the following magnitude range: 1E14÷1E-15 (while the limit in the number storing is the magnitude range 1E76÷1E-78).

String format

There are two types of string formatting protocols: the DEC protocol and the **tbas** protocol. The first is the same of the DEC-20 BASIC, with the same rules. The second is derived from more recent BASIC compilers and interpreters, with some enhancements.

The DEC Protocol

The DEC protocol uses the apostrophe ' to introduce a string format, followed by a series of letters among C, L, R or E (the letters in a string image specification must be contiguous). The letters in a single string output must be of the same type; the extra characters will represent themselves. The apostrophe itself counts as a character position (the first character in the string), and need not be followed by a specification letter: if used alone, only the first letter will be printed.

The letter C as in 'CCCCCCC causes the string to be centered in the output field, and trimmed if its length exceeds the output field length.

The letter L as in 'LLLLLLL causes the string to be left-justified in the output field, and trimmed if its length exceeds the output field length.

The letter R as in 'RRRRRRR causes the string to be right-justified in the output field, and trimmed if its length exceeds the output field length.

The letter E as in 'EEEEEEE causes the string to be left-justified in the output field, and if its length exceeds the output field length, the field is expanded accordingly.

The tbas protocol

The **tbas** protocol's string image field is introduced by one instance of '<' or '>' and followed by as many '#' characters as needed (the same used for the numeric output fields), optionally closed by one instance of '<' or '>'. The starting characters '<' and '>' count as a character position, but cannot appear alone: at least one '#' must follow. This lets use the '<' or '>' character in a text (for instance in arrows like -->). This means that one-character string extraction is not available, but of course this does not mean you cannot print one-character strings!

The format <#######> causes the string to be centered in the output field, and trimmed if its length exceeds the output field length.

The format <######## causes the string to be left-justified in the output field, and trimmed if its length exceeds the output field length.

The format >######## causes the string to be right-justified in the output field, and trimmed if its length exceeds the output field length.

The format <#######< causes the string to be left-justified in the output field, and if its length exceeds the output field length, the field is expanded accordingly.

The string formats of two different protocols can coexist in the output field, provided each is referenced by its own string.

Using PRINT USING and WRITE USING

Apart for the strange title of this paragraph, the PRINT USING/WRITE USING statements may be used with the following syntax:

```
 On terminal:
 PRINT USING <n>, <var>, <var>...
 PRINT USING <s>, <var>, <var>...
 WRITE USING <n>, <var>, <var>...
 WRITE USING <s>, <var>, <var>...
```

If used for printing on the terminal (no channel specified), the PRINT USING and WRITE USING statements are completely equivalent. In this case, WRITE USING is a mere synonym of PRINT USING, as it does not print any line number preceding the output, as the WRITE statement does.

```
 To files:
 PRINT USING# <c>, <n>, <var>, <var>...
 PRINT USING# <c>, <s>, <var>, <var>...
 PRINT <c>, USING# <n>, <var>, <var>...
 PRINT <c>, USING# <s>, <var>, <var>...
 WRITE USING# <c>, <n>, <var>, <var>...
 WRITE USING# <c>, <s>, <var>, <var>...
 WRITE <c>, USING# <n>, <var>, <var>...
 WRITE <c>, USING# <s>, <var>, <var>...
```

The PRINT USING# and WRITE USING# statements instead, if used with channel numbers from 1 to 9, bear a little difference, the same found between PRINT# and WRITE#: while PRINT USING# prints the output as is,

the WRITE USING# statement (as the WRITE# statements does), prints a line number before any new line of output, with the same behaviour of WRITE#.

E.g.
```
 20 :####.##  ####.##
 PRINT USING #1, 20, length, velocity
 PRINT #1, USING 20, length, velocity

 frm$="'LLLLL"
 WRITE USING #4, frm$, "JULE"
 WRITE #4, USING frm$, "JULE"
```

An important note: if you start using one statement (WRITE USING# or PRINT USING#), you must continue with the same command, or an error will be raised; this is because the two types must not mix their output, or the resulting file will not be readable by READ# or INPUT# respectively.

WAIT

The WAIT statement is used to stop the execution of a program for a certain amount of seconds, according to the following syntax:

```
 WAIT <n>
 WAIT
```

If the argument is present, wait for <n> of seconds, otherwise wait for 3 seconds. The <n> value is cut to an integer value before its usage.

Note: in current implementation, WAIT simply blocks the execution of the program (no threads nor subprocesses).

WEND

See the WHILE statement.

WHEN ERROR IN

The WHEN ERROR structure provides a very advanced way for errors-catching techniques. It was a feature present in the evolution of the DEC-20 BASIC (called TRUE BASIC) and was supposed to supply a smarter way than that available with ON ERROR and its companion.

Note: some errors are not catchable (unrecoverable). See the Errors list section for details.

WHEN ERROR IN with Internal Handler

The first form involves the internal handler:

```
 WHEN ERROR IN
    ...
    ... <supposed faulty code>   ' RETRY re-executes this line
    ...                          ' CONTINUE move to this line
```

```
 USE
    ... <code treating error>
    ... RETRY|CONTINUE|EXIT HANDLER|EXIT WHEN
    ...
 END WHEN
```

This is useful for code that is supposed to fail for some reasons (for example, a zero value at denominator, or an unopened file) and the handler is specific and not usable elsewhere. The process involves these steps:

- the code after WHEN ERROR IN is executed;
- if no error condition is raised, when the code reaches USE, it jumps to END WHEN and continues.
- if an error is raised, the program counter jumps to the piece of code after USE, that is supposed to treat the error condition
- in any case, after END WHEN, the code resumes to what follows the WHEN ERROR section.

WHEN ERROR IN structures may be used in a SUB, provided they are entirely contained in the SUB, from WHEN ERROR IN to USE to END WHEN.

WHEN ERROR USE with External Handler

The second form involves an external handler:

```
 WHEN ERROR USE <handler_name>
    ...
    ... <supposed faulty code>   ' RETRY re-executes this line
    ...                          ' CONTINUE move to this line
    ...
 END WHEN
 ...
 <other code>
 ...
 HANDLER <handler_name>
    ... <code treating error>
    ... RETRY|CONTINUE|EXIT HANDLER|EXIT WHEN
    ...
 END HANDLER
```

The handler's name follows the usual laws (a letter or underscore followed by letters, numbers or underscores). This is useful for code that is supposed to fail for some reasons and the handler is re-usable elsewhere. The process involves these steps:

- the code after WHEN ERROR USE is executed and the HANDLER name is stored;
- if no error condition is raised, when the code reaches END WHEN it continues
- if an error is raised, the program counter looks for the HANDLER with the specified name, and jumps to the piece of code after HANDLER, that is supposed to treat the error condition; the HANDLER code may reside elsewhere in the listing
- in any case, after END WHEN, the code resumes to what follows the WHEN

ERROR section
- the section HANDLER/END HANDLER is invisible to the program counter which, in case of normal execution, never executes it.

If the handler is not found, an unrecoverable error is raised.

See also the RETRY, CONTINUE, EXIT HANDLER and EXIT WHEN statements.

WHEN ERROR USE structures cannot be used in a SUB, because of the detached nature of the HANDLER, which is a structure that must remain independent from any other structure.

WHILE

The construct WHILE/EXIT/END WHILE|WEND is not classical in BASIC, but it's widely accepted in all modern BASICs. **tbas** accepts the following syntax:

```
WHILE <cond>
   <statement>
   <statement>
   ... EXIT WHILE
   <statement>
END WHILE|WEND
```

The 'while-end while' cycle is characteristic because it is never executed if the condition is false from the start.

Note: WEND can be used in place of END WHILE (added for compatibility issues).

WRITE

See the PRINT statement.

WRITE#

See the PRINT# statement.

WRITE:

See the PRINT# statement.

2.8. BASIC functions

Here's the complete list of all **tbas** functions. All functions ending with $ return strings, all the others return numbers.

2.8.1. Math functions

Functions that accept or return an angular measure, work with radians by default; this can be changed by OPTION ANGLE DEGREES, that turns the

default behaviour of such function to degrees, both in input and output.

<PA>

This is not a real function, rather it's an implicit command to the screen to fill up current page with empty lines until the end of page and print the page number on last line, if required.

This command (in this atypical form) comes directly from the DEC-10/DEC-20 BASIC, and adapted to **tbas**. Its function is also performed by the statement FILLPAGE (see).

ABS

This function returns the absolute value of the number in argument. In math the absolute value of x is often written as $|x|$.

Domain: $x \in \Re$
Range: $y \in \Re : y \geq 0$

E.g.

```
PRINT ABS(-3.4)
 3.4
PRINT ABS(3.4)
 3.4
```

ACOS

See the COS family functions.

ACOSEC/ACSC

See the COSEC family functions.

ACOSECH/ACSCH

See the COSECH family functions.

ACOSH

See the COSH family functions.

ACOT

See the COT family functions.

ACOTH

See the COTH family functions.

ALPHA

The ALPHA function takes a string arguments, and returns TRUE if it composed only by the ASCII characters from 'a' to 'z' or from 'A' to 'Z', i.e. not numbers or punctuation characters.

ASEC

See the SEC family functions.

ASECH

See the SECH family functions.

ASIN

See the SIN family functions.

ASINH

See the SINH sine family functions.

ATAN/ATN

See the TAN family functions.

ATANH/ATNH

See the TANH family functions.

BIN

The BIN function converts the string given as argument (interpreted as a binary number) to its normal decimal representation. Conversion proceeds from right to left, because the lowest bit is on the right (see BIN$).

E.g.

```
 PRINT BIN("11100")
  28
```

Of course the following holds:

```
 PRINT BIN(BIN$(17))
  17
```

If the string contains digits which are not a binary value (0 or 1), the conversion stops and a warning is printed, and the value converted so far is returned.

E.g.

```
PRINT BIN("11201")
% BIN/OCT/HEX argument was not exhausted in line 1.
 1
```

CEIL/CEILING

The CEIL function (that may be written as CEILING as well) maps a real number to the smallest following integer. More precisely, CEIL(x) is the smallest integer not less than x. If the argument is an integer value, it remains unchanged.

The CEIL function may be better understood if one tries to see the interval between two integers as a skyscraper story:

```
---------- 3            ---------- -2

  2.4                     -2.4
---------- 2            ---------- -3
```

The higher integer is the higher story. Now, any intermediate value stands in a room with one floor and one ceiling, and if the number is the ceiling itself, it does not change.

E.g.

```
PRINT CEILING(-2)
-2
PRINT CEILING(-2.4)
-2
PRINT CEILING(0)
 0
PRINT CEILING(3)
 3
PRINT CEILING(2.4)
 3
```

The COS family functions

The COS function returns the cosine of the angular measure given as argument.

Domain: $x \in \Re$
Range: $y \in \Re : -1 \leq y \leq 1$

The inverse of the COS function is performed by the ACOS function, that returns the angular measure of the cosine value given as argument; being the cosine function not injective, the ACOS returns values in a limited range:

Domain: $x \in \Re : -1 \leq x \leq 1$
Range: $y \in \Re : 0 \leq y < \pi$

E.g.

```
OPTION ANGLE DEGREES
PRINT COS(135)
-0.707107
PRINT ACOS(-0.707107)
 135.
```

In this case, the dot after 135 means the number is not a perfect integer, because the input is not precisely the COS of 135°.

The COSEC family functions

The COSEC function (that may be also typed as CSC) returns the cosecant of the angular measure given as argument; the COSEC function is defined as the reciprocal of the sine function.

Domain: $x \in \Re$
Range: $y \in \Re : y \leq -1 \text{ or } y \geq 1$

For x=0 or $x=\pm\pi$ the returned value is ±INF, and no error message is issued.

The inverse of the COSEC function is performed by the ACOSEC function (that may be written also as ACSC) that returns the angular measure of the cosecant value given as argument; being the cosecant function not injective, the ACOSEC returns values in a limited range:

Domain: $x \in \Re : x \leq -1 \text{ or } x \geq 1$
Range: $y \in \Re : -\pi/2 < y < \pi/2$

with the obvious consequence that if x=±INF the returned value is close to zero or to π, but the exact value is never returned because the greatest value managed by **tbas** is not really ∞.

E.g.

```
OPTION ANGLE DEGREES
PRINT COSEC(34)
 1.78829
PRINT ACOSEC(1.78829)
 34.
```

In this case, the dot after 34 means the number is not a perfect integer, because the input is not precisely the COSEC of 34°.

The COSECH family functions

The COSECH function (that may be typed also as CSCH) returns the hyperbolic cosecant of the number given as argument. If the argument is null (or quasi-null), a signed INF is returned. The COSECH function is defined as the reciprocal of the hyperbolic sine function SINH.

Domain: $x \in \Re - \{0\}$
Range: $y \in \Re - \{0\}$

The inverse of the COSECH function is performed by the ACOSECH function (that may typed as ACSCH as well). A signed INF is returned in case x approaches 0, positive if x>=0, negative otherwise.

Domain: $x \in \Re - \{0\}$
Range: $y \in \Re - \{0\}$

E.g.

```
PRINT COSECH(1)
 0.850918
PRINT ACOSECH(0.850918)
 1.
```

In this case, the dot after 1 means the number is not a perfect integer, because the input is not precisely the COSECH of 1.

The COSH family functions

The COSH function returns the hyperbolic cosine of the value given as argument.

Domain: $x \in \Re$
Range: $y \in \Re : y \geq 1$

The inverse of the COSH function is performed by the ACOSH function, that returns the value of the hyperbolic cosine given as argument. Since the COSH function is symmetric, the ACOSH function returns values in the positive branch of the x-axis. An error is raised if the argument is out of domain.

Domain: $x \in \Re : x \geq 1$
Range: $y \in \Re : y \geq 0$

E.g.

```
PRINT COSH(1)
 1.54308
PRINT ACOSH(1.54308)
 0.999999
```

In this case, the returned value is not a perfect integer, because the input is not precisely the COSH of 1.

The COT family functions

The COT function returns the cotangent of the angular measure given as argument. If the argument is outside the domain, it is reported to the domain interval, since the return value does not change. If the argument is close to $\pm\pi$ or to 0, the big returned value is not precise; in this case, assume that the big returned number is the infinite value ∞.

Domain: $x \in \Re : \pi < x < 0$
Range: $y \in \Re$

The inverse of the COT function is performed by the ACOT function, that returns the angular measure of the cotangent value given as argument; being the cotangent function not injective in the whole x-axis, the ACOT returns values in a limited range:

```
Domain: x ∈ ℜ
Range:  y ∈ ℜ : -π/2 < y ≤ π/2
```

E.g.

```
OPTION ANGLE DEGREES
PRINT COT(88)
 3.49208E-2
PRINT ACOT(3.49208E-2)
 88.
```

In this case, the dot after 88 means the number is not a perfect integer, because the input is not precisely the COT of 88°.

The COTH family functions

The COTH function returns the hyperbolic cotangent of the value given as argument. A signed INF is returned in case the argument value approaches zero.

```
Domain: x ∈ ℜ - { 0 }
Range:  y ∈ ℜ - { ]-1,1[ }
```

The inverse of the hyperbolic cotangent is performed by the ACOTH function, that returns the measure of the hyperbolic cotangent value given as argument. If value is 1 or -1, the returned value is INF, with the same sign of the argument. IF value is outside the domain, an error condition is raised.

```
Domain: x ∈ ℜ - { ]-1,1[ }
Range:  y ∈ ℜ - { 0 }
```

E.g.

```
PRINT COTH(1.2)
 1.19954
PRINT ACOTH(1.19954)
 1.19999
```

Here, the returned value is not equal to 1.2 because the input value given to ACOTH is not precisely COTH(1.2) but a mere approximation.

DEG/DEGREES

The DEG function (that can be typed also as DEGREES) returns the angle measure given as argument in radians, converted to degrees with sign.

E.g.

```
 PRINT DEG(PI)
 180
```

The DEG function does not change the OPTION ANGLE state.

Please, pay attention using PI in such cases: if you have enabled the OPTION ANGLE DEGREES the result is affected, because PI returns 180, and the function DEG interprets it as 180 radians.

E.g.

```
 OPTION ANGLE DEGREES
 PRINT DEG(PI)
  10313.2
```

DET

The function DET (not to be confused with the DET variable) returns the determinant of the square matrix in argument. The matrix is identified by means of the name only (dimensions are internally stored).

The matrix is inverted by means of a proper inversion process, but no attribution is made to any variable: the process is executed only to retrieve the determinant. If the argument matrix is singular, 0.0 is returned.

The inversion does not need to be previously performed (as with the DET variable), in order to obtain the determinant.

E.g.

```
 DECLARE mat(3,3),res(3,3)
 DECLARE res
 DATA 1,2,-3,4,5,6,-7,8,9
 MAT READ mat
 MAT PRINT mat
 res=DET(mat)
 PRINT res

  1             2            -3
  4             5             6
 -7             8             9

 -360
```

DIV

The DIV function performs the Number Theory division, which assumes that the remainder of the integer division must always be positive for any signs of the operands; this implies that sometimes (namely when the dividend is negative), the result must be increased by one (with sign) to have the remainder always positive, according to the following examples:

```
  PRINT DIV(17,5)     ' ==> 3 x 5 + 2 = 17
```

```
 3
PRINT DIV(-17,5)    ' ==> -4 x 5 + 3 = -17
-4
PRINT DIV(-17,-5)   ' ==> 4 x -5 + 3 = -17
 4
PRINT DIV(17,-5)    ' ==> -3 x -5 + 2 = 17
-3
```

In case the divisor is null, a warning appears, and the result is the ∞ value with the sign of the dividend.

Note: There is no infix operator that performs this calculation.

DOT

The DOT function returns the dot product between two vectors. The two vectors (not matrices) given as arguments must be specified by means of the name only, and they must have the same dimensions.

The two vectors do not need to be of the same type: one horizontal vector and one vertical may be used for calculating the dot product.

Error messages are returned in case either argument is a matrix (two dimensions), or if it is an undeclared vector, or the dimensions of the two vectors differ.

E.g.

```
DECLARE vec1(6), vec2(6)
DECLARE res
DATA 1,2.3,6.7,7,8.2,-4.9
DATA -3,2.4,-1.28,5,6,7
MAT READ vec1,vec2
MAT PRINT vec1;vec2 ' print vectors horizontally
res=DOT(vec1,vec2)
PRINT res

 1  2.3  6.7  7  8.2 -4.9

-3  2.4 -1.28  5  6  7

 43.844
```

ERFC

The ERFC returns the complementary error function of X, computing the difference of the error function from 1.0; the error function is known as erf() in most languages, and its value may be obtained by 1-erfc().

EXP

The EXP function returns the result of the neperian number 'e' to the power of the number given as argument. In the range of the possible results, any argument greater than 176.752531 returns the infinite ∞

value.

Domain: $x \in \Re$
Range: $y \in \Re : y \geq 0$

E.g.

```
PRINT EXP(2)
 7.38906
PRINT EXP(346)
 5.78960E+76
```

EVAL

The EVAL function returns a numeric value from a string input, so there is not really a Domain or a Range to be specified. The only rule is that the string must hold a ground expression; for instance, "3.4" or "2*SQR(34)" are perfect strings being ground formulas.

E.g.

```
PRINT EVAL("3.4")
 3.4

PRINT EVAL("SQR(3) + SQR(2)")
 3.14626
```

In general, any number, math function or any combination of these is good as a EVAL() argument, but this combination must be 'ground', that is with no variables. Variables are not parsed in EVAL (unlike EXPR). So "SQR(2)" is legal, while "SQR(Y)", with Y=2, is not legal (in practice, any number that can be typed on a pocket calculator is legal as well as any formula that can be calculated by typing on a pocket calculator).

E.g.

```
Y=3
PRINT EVAL("SQR(Y)")
? EVAL argument does not contain a ground number in line 2.
PRINT EVAL("SQR(Y)")
                ^
```

An error message is shown in case the string argument is empty.

E.g.

```
A$=""
PRINT EVAL(A$)
? VAL/EVAL/EXPR argument is empty in line 2.
PRINT EVAL(a$)
            ^
```

If the string does not *begin* with any number or ground formula, an error message is shown.

E.g.

```
A$="Pounds 345"
PRINT EVAL(A$)
? EVAL argument does not contain a ground number in line 2.
PRINT EVAL(A$)
     ^
```

Note: in case the formula is composed of ground terms only, EXPR and EVAL are perfectly equivalent. See also the VAL and EXPR function.

EXPR

The EXPR function returns a numeric value from a string input, so there is not really a Domain or a Range to be specified. The only rule is that the string must hold a valid math expression, with or without any variables; for instance, "2.8" or "G*SQR(Y)" are a perfect strings being regular math formulas.

E.g.

```
PRINT EXPR("2.8")
 2.8

Y=3
PRINT EXPR("SQR(Y) + SQR(2)");
 3.14626
```

An error message is shown in case the string argument is empty.

E.g.

```
A$=""
PRINT EXPR(A$)
? VAL/EVAL/EXPR argument is empty in line 2.
PRINT EXPR(A$)
            ^
```

If the string does not **begin** with any number or formula, but with a letter or underscore, a value correspondent to that variable name is returned, and this variable follows usual declaration laws.

E.g.

```
PRINT EXPR("X") ' implicit declaration of variable X
 0
```

If the string does not **begin** with any number or formula or variable, it is an (usual) illegal expression.

E.g.

```
A$="%"
PRINT EXPR(A$)
```

```
 ? Illegal expression in line 2.
 PRINT EXPR(A$)
               ^
```

Note: in case the formula is composed of ground terms only, EXPR and EVAL are perfectly equivalent. See also the VAL and EVAL functions.

FRAC/FP

The function FRAC (that may be typed FP as well) returns the factional part of the decimal number given as argument.

```
 Domain: x ∈ ℜ
 Range:  y ∈ ℜ
```

E.g.

```
 PRINT FP(3.56)
  0.56
 PRINT FP(-3.56)
 -0.56
```

FREE

The function FREE (also as FREE(X) with X dummy) returns the number of free programming slots, as an integer type. Remember that memory is allocated, when the line is stored, by the Operating System, so there is enough space for programs as long as the Operating System can allocate memory; the value indicates the number of free programming lines available (of any length) that you can still use.

GAMMA

The GAMMA function returns the gamma function of the value in argument.

```
 Domain: x ∈ ℜ - { non-positive integers }
 Range:  y ∈ ℜ
```

The gamma function, which is a sort of expansion of the factorial to real number, can be defined for all numbers (except the non-positive integers) by the following function:

$$\Gamma(t) = \int_0^{\infty} x^{t-1}\, e^{-x}\, dx$$

In case of non-positive integers, the NAN value is returned.

E.g.

```
 PRINT GAMMA(2.3)
  1.16671
 PRINT GAMMA(-3)
  NAN
```

Note: NAN is not returned by **tbas**, rather is a value that is returned by the C compiler and the underlying Operating System used to generate these examples. You may see 'nan' or 'Nan', or even something else, here...

HEX

The HEX function converts the string given as argument (interpreted as a hexacimal number) to its normal decimal representation. Conversion proceeds from left to right, as usual (see HEX$).

E.g.

```
 PRINT HEX("FFF")
  4095
```

Of course the following holds:

```
 PRINT HEX(HEX$(17))
  17
```

If the string contains digits which are not a hexadecimal value (0 to 9 and A to F), the conversion stops and a warning is printed, and the value converted so far is returned.

E.g.

```
 PRINT HEX("FFGF")
 % BIN/OCT/HEX argument was not exhausted in line 1.
  255
```

IDIV

The IDIV function performs the Classic Number Theory division, which assumes that the remainder of the integer division may be either positive or negative for any signs of the operands; the result is the one expected by natural thinking, according to the following examples:

```
  PRINT IDIV(17,5)     ' ==> 3 x 5 + 2 = 17
   3
  PRINT IDIV(-17,5)    ' ==> -3 x 5 - 2 = -17
  -3
  PRINT IDIV(-17,-5)   ' ==> 3 x -5 - 2 = -17
   3
  PRINT IDIV(17,-5)    ' ==> -3 x -5 + 2 = 17
  -3
```

In case the divisor is null, a warning appears, and the result is the ∞ value with the sign of the dividend.

Note: The \ operator performs the same operation in infix mode.

INT/FLOOR/INTEGER

The INT function (that is called also FLOOR or in full INTEGER) maps a real number to the largest previous following integer. More precisely, INT(x) or FLOOR(x) is the largest integer not greater than x. If the argument is an integer value, it remains unchanged.

The INT/FLOOR function, as with the CEIL function, may be better understood if one tries to see the interval between two integers as a skyscraper story:

```
---------- 3            ---------- -2

  2.4                     -2.4
---------- 2            ---------- -3
```

The higher integer is the higher story. Now, any intermediate value stands in a room with one floor and one ceiling, and if the number is the floor itself, it does not change.

E.g.

```
 PRINT INT(-2)
 -2
 PRINT FLOOR(-2.4)
 -3
 PRINT INT(0)
  0
 PRINT INTEGER(3)
  3
 PRINT FLOOR(2.4)
  2
```

INV

The INV function is a wrapper for the inversion of x, which is 1/x. The two processes are equivalent. INV was introduced for compatibility issues.

E.g.

```
 PRINT INV(3.4)
  0.294118
 PRINT INV(0)
 % Division by zero in line 25.
  5.78960E+76
```

IP

The function IP returns the integer part of the decimal number given as argument.

Domain: $x \in \Re$
Range: $y \in \mathbb{N}$

E.g.

```
 PRINT IP(3.56)
  3
 PRINT IP(-3.56)
 -3
```

LBOUND/LDIM

The LBOUND function (available also as LDIM) returns the lower index value for a given array in argument, with optional indication of the required dimension.

The request is in the form:

```
 LBOUND(V[,N])
```

where V is the array name and N (optional) is the dimension. If omitted, dimension 1 is assumed. At present, N may be 1 or 2; if different, an error message is shown.

Note: you can check for the second dimension of a vector, because internally it is always null, and thus this operation returns always zero.

The array is identified by its letters, if () is added, it is skipped.

Note: the lower index for all vectors and matrices is always the BASE index, of course, which is 0 or 1.

E.g.

```
 DECLARE vec1(6), mat1(2,18)
 PRINT LBOUND(vec1(),1),LDIM(mat1,1)
 PRINT LBOUND(vec1,2),LDIM(mat1(),2)
  0             0
  0             0
```

LEN/LENGTH

The LEN function (that may be typed as LENGTH as well) returns the length of the string in argument; the string may be a literal string, enclosed in double quotes, or a variable, or a string formula.

E.g.

```
 PRINT LEN("ANNA")
  4
 H$="WIND"&STR$(24)
 PRINT LENGTH(H$)
  6
 PRINT LEN("ANNA "+H$)
  11
```

LGAMMA

The LGAMMA returns the natural logarithm of the absolute value of the gamma function of the value in argument. With respect to GAMMA, the Domain is the complete $\Re$ field.

```
Domain: x ∈ ℜ
Range:  y ∈ ℜ
```

E.g.

```
PRINT LGAMMA(-3)
 5.78960E+76
PRINT LGAMMA(1.25)
-9.82718E-2
```

LOG/LOGE/LN

The function LOG (which may be typed also as LN – which is a modern name present in many scientific calculators – and LOGE – for LOGarithm of E – of more ancient tradition) returns the natural logarithm in neperian base (the neperian number, which is 'e', has the a value near to 2.71828) of the number given as argument.

```
Domain: x ∈ ℜ : x > 0
Range:  y ∈ ℜ
```

If the argument is null, the negative infinite $-\infty$ is returned, with a warning.

E.g.

```
PRINT LOG(1)
 0

PRINT LN(0)
% LOG/LOG10 of zero in line 2.
-5.78960E+76
```

If the argument is negatibe, an error message is shown and execution stops.

E.g.

```
PRINT LOGE(-1)
? LOG/LOG10 argument is negative in line 3.
PRINT LOGE(-1)
             ^
```

LOG10/CLOG/LGT

The function LOG10 (which may be typed also as CLOG and LGT) returns the logarithm in base 10 of the number given as argument.

```
Domain: x ∈ ℜ : x > 0
Range:  y ∈ ℜ
```

If the argument is null, the negative infinite -∞ is returned, with a warning.

E.g.

```
PRINT LOG10(2)
 0.30103
```

If the argument is negatibe, an error message is shown and execution stops.

E.g.

```
PRINT CLOG(0)
% LOG/LOG10 of zero in line 2.
-5.78960E+76

PRINT LGT(-1)
? LOG/LOG10 argument is negative in line 3.
PRINT LGT(-1)
           ^
```

MAX

The MAX function returns the maximum value present in the variable-length list of arguments, which may be literal numbers or numeric variables in quantity from 1 to any value that fits the length of the programming line. The comparison takes in account the sign, so that -1 is greater than -3 and 2 is greater than -1.

E.g.

```
jack=111
PRINT MAX(2,-4,jack,10)
 111
```

In case no argument is given, 0 (zero) is returned. E.g.

```
PRINT MAX()
 0
```

MIN

The MIN function returns the minimum value present in the variable-length list of arguments, which may be literal numbers or numeric variables in quantity from 1 to any value that fits the length of the programming line. The comparison takes in account the sign, so that -3 is lesser than -1 and -1 is lesser than 2.

E.g.

```
jack=11
PRINT MIN(2,-4,jack,10)
-4
```

In case no argument is given, 0 (zero) is returned. E.g.

```
PRINT MIN()
 0
```

MOD

The MOD function returns the remainder of the Classic Number Theory division, which assumes that the remainder of the integer division may be either positive or negative for any signs of the operands; the result is the one expected by natural thinking, according to the following examples:

```
PRINT MOD(17,5)     ' ==> 2 + 3 x 5 = 17
 2
PRINT MOD(-17,5)    ' ==> -2 + -3 x 5 = -17
-2
PRINT MOD(-17,-5)   ' ==> -2 + 3 x -5 = -17
-2
PRINT MOD(17,-5)    ' ==> 2 + -3 x -5 = 17
 2
```

In case the divisor is null, a warning appears, and the result is zero.

Note: The %% operator performs the same operation in infix mode.

NUMBER

The NUMBER function takes a string arguments, and returns TRUE if it composed only by the ASCII characters from '0' to '9', i.e. not alphabetical characters or punctuation characters.

OCT

The OCT function converts the string given as argument (interpreted as an octal number) to its normal decimal representation. Conversion proceeds from left to right, as usual (see OCT$).

E.g.

```
PRINT OCT("777")
 511
```

Of course the following holds:

```
PRINT OCT(OCT$(17))
 17
```

If the string contains digits which are not an octal value (0 to 7), the conversion stops and a warning is printed, and the values converted so

far is returned.

E.g.

```
 PRINT OCT("7787")
 % BIN/OCT/HEX argument was not exhausted in line 1.
  63
```

PICT

The PICT function takes a string arguments, and returns TRUE if it composed only by punctuation characters in the range 32-47, 58-64, 91,96 or 123-127, i.e. not numbers or alphabetical characters.

RAD/RADIANS

The RAD function (that can be typed also as RADIANS) returns the angle measure given as argument in degrees converted to radians, with sign.

E.g.

```
 PRINT RAD(-420)
 -7.33038
```

The RAD function does not change the OPTION ANGLE state.

Please, pay attention using PI with this function, because if you have not enabled the OPTION ANGLE DEGREES the result is affected, because PI returns 3.141592, and the function RAD interprets it as 3.141592 degrees.

E.g.

```
 OPTION ANGLE RADIANS
 PRINT RAD(PI)
  5.48311E-2
```

REMAINDER

The REMAINDER function returns the remainder of the Number Theory division, which assumes that the remainder of the integer division is always positive for any signs of the operands; the result is not the one expected by natural thinking, according to the following examples:

```
  PRINT REMAINDER(17,5)     ' ==> 2 + 3 x 5 = 17
   2
  PRINT REMAINDER(-17,5)    ' ==> 3 + -4 x 5 = -17
   3
  PRINT REMAINDER(-17,-5)   ' ==> 3 + 4 x -5 = -17
   3
  PRINT REMAINDER(17,-5)    ' ==> 2 + -3 x -5 = 17
   2
```

In case the divisor is null, a warning appears, and the result is zero.

Note: There is no infix operator that performs this calculation.

REAL

The REAL function is a no-op, and simply returns the argument (which is a regular numeric expression). This function was introduced to enhance compatibility with other BASIC languages which use a stronger type recognition, in order to return any expression result (which may be an integer or boolean value) as a common real value. **tbas** does so implicitly, so the REAL function is not really necessary, but if you happen to use it (true, Bruce?) **tbas** gently reacts as expected and does not emit any error message or warning.

RND

The RND function (one of the oldest BASIC functions, actually) returns a pseudo-random number in the range [0.0, 1.0], that is limits excluded.

The random number is generated by an algorithm which starts from a fixed seed, making any further output predictable. To change the seed, use the RANDOMIZE statement, which chooses a new seed basing on current time and date, making further output unpredictable (at least for human beings).

RND may be used with a (dummy) argument or alone, as a pseudo-variable.

E.g.

```
 PRINT RND
 PRINT RND(0)
  0.786499
  2.07520E-2
```

Read the chapter "The pseudo-random number generator" for information about the generation process.

ROUND/ROF/FIX

The ROUND function (that may be typed ROF or FIX, as of older BASIC traditions) truncates the decimal number in the first argument to a fixed number of decimals specified by the second argument. This last digit is rounded according to the next digit that follows, so that if this is 0 to 4, the last digit remains the same, if it is 5 to 9, the last digit is increased by one.

```
 PRINT ROUND(2.35,1)
  2.4
```

If the second argument is not specified, 0 is assumed, and the truncation returns a rounded integer:

```
 PRINT ROF(2.67)
  3
```

If the second argument is lesser than zero, zero is returned.

```
PRINT FIX(2.67,-3)
 0
```

The SEC family functions

The SEC function returns the secant of the angular measure given as argument; the SEC function is defined as the reciprocal of the cosine function COS.

Domain: $x \in \Re$
Range: $y \in \Re : y \leq -1 \text{ or } y \geq 1$

For $x=\pm\pi/2$ the returned value is ±INF, and no error message is issued.

The inverse of the SEC function is performed by the ASEC function) that returns the angular measure of the secant value given as argument; being the secant function not injective, the ASEC returns values in a limited range:

Domain: $x \in \Re : x \leq -1 \text{ or } x \geq 1$
Range: $y \in \Re : 0 \leq y < \pi$

with the obvious consequence that if x=±INF the returned value is close to $\pm\pi/2$, but the exact value is never returned because the greatest value managed by **tbas** is not really ∞.

E.g.

```
OPTION ANGLE DEGREES
PRINT SEC(2)
 1.00061
PRINT ASEC(1.00061)
 2.00075
```

The returned value 2.00075 is approximate, because the result of SEC(2) was cut.

The SECH family functions

The SECH function returns the hyperbolic secant of the value given as argument; the SECH function is defined as the reciprocal of the hyperbolic cosine function COSH.

Domain: $x \in \Re$
Range: $y \in \Re : 0 \leq y \leq 1$

The inverse of the SECH function is performed by the ASECH function, which returns the value of the hyperbolic secant given as argument. Since the SECH function returns values in the range [0,1], with asymptotes in 0, the ASECH function returns INF when x approaches 0, and since it is a symmetric function, it returns values always in the positive branch.

Domain: $x \in \Re : 0 < x \leq 1$

Range: $y \in \Re : y \geq 0$

E.g.

```
OPTION ZERO
PRINT SECH(100)
 0
PRINT ASECH(0)
 inf
```

SGN/SIGN

The SGN function (that may be written conveniently as SIGN as well) returns -1 if the value given as argument is negative, returns 0 if it is exactly zero and returns 1 if it is positive. The argument may be any valid mathematical expression returning a value which is interpreted as an integer.

Domain: $x \in \Re$
Range $y \in \mathbb{N} : y = -1$ or 0 or 1

E.g.

```
PRINT SGN(-3)
-1
PRINT SIGN(3-3)
 0
PRINT SIGN(3)
 1
```

SHL

The SHL function (SHL stands for SHift Left) shifts the first argument for the number of positions specified by the second argument towards left. Both arguments are required.

It's worthwhile noting here that shifting left means increasing the value of the number, and is equivalent to multiplying the left argument by 2 to the power of the second argument.

E.g.

```
FOR I=1 TO 6
   PRINT SHL(4,I)
NEXT

 8
 16
 32
 64
 128
 256
```

If the shift value is higher than the distance from the greatest integer

(in shifting steps), the number overflows (with no warnings) and zero is returned from here on.

E.g.

```
 FOR I=145 TO 155
    PRINT SHL(25600,I)
 NEXT

 -9.39524E+8
 -1.87905E+9
  5.36871E+8
  1.07374E+9
 -2.14748E+9
  0
  0
  0
  0
  0
  0
```

SHR

The SHR function (SHR stands for SHift Right) shifts the first argument for the number of positions specified by the second argument towards right. Both arguments are required.

It’s worthwhile noting here that shifting right means decreasing the value of the number, and is equivalent to dividing the left argument by 2 to the power of the second argument.

E.g.

```
 FOR I=1 TO 6
    PRINT SHR(256,I)
 NEXT

  128
  64
  32
  16
  8
  4
```

If the shift value is higher than the distance from zero (in shifting steps), zero is returned from here on.

E.g.

```
 FOR I=1 TO 6
    PRINT SHR(256,I)
 NEXT

  6
```

```
 3
 1
 0
 0
 0
```

The SIN family functions

The SIN function returns the sine of the angular measure given as argument.

Domain: $x \in \Re$
Range: $y \in \Re : -1 \leq x \leq 1$

The inverse of the SIN function is performed by the ASIN function, that returns the angular measure of the sine value given as argument; being the sine function not injective, the ASIN returns values in a limited range:

Domain: $x \in \Re : -1 \leq x \leq 1$
Range: $y \in \Re : -\pi/2 < y < \pi/2$

E.g.

```
OPTION ANGLE DEGREES
PRINT SIN(135)
 0.707107
PRINT ASIN(0.707107)
 45.
```

In this case, the dot after 45 means the number is not a perfect integer, because the input is not precisely the SIN of 45°.

The SINH family functions

The SINH function returns the hyperbolic sine of the value given as argument.

Domain: $x \in \Re$
Range: $y \in \Re$

The inverse of the SINH function is performed by the ASINH function, which returns the value of the hyperbolic sine given as argument.

Domain: $x \in \Re$
Range: $y \in \Re$

E.g.

```
PRINT SINH(99)
 4.94452E+42
PRINT ASINH(4.94452E+42)
 99.
```

In this case, the dot after 99 means the number is not a perfect integer, because the input is not precisely the SINH of 99.

SQR/SQRT

The SQR function (that may be written as SQRT as well) returns the square root of the number given as argument. Unlike the DEC-20 and the Dartmouth families BASICs, a negative argument raises an error, because it's out of the domain of the function.

```
Domain: x ∈ ℜ : x ≥ 0
Range:  y ∈ ℜ : y ≥ 0
```

E.g.

```
PRINT SQR(3)
 1.73205
PRINT SQR(-3)
? SQRT of a negative number in line 64.
```

SUM

The SUM function returns the sum of the variable-length list of arguments, which may be literal numbers or numeric variables in quantity from 1 to any value that fits the length of the programming line.

E.g.

```
jack=11
PRINT SUM(2,-4,jack,10)
 19
```

In case no argument is specified, 0 (zero) is returned.

E.g.

```
PRINT SUM()
 0
```

The TAN family functions

This TAN function returns the tangent of the angular measure given as argument. If the argument is outside the domain, it is reported to the domain interval, since the return value does not change. If the argument is close to $\pm\pi$, the big returned value is not precise; in this case, assume that the big returned number is the infinite value ∞.

```
Domain: x ∈ ℜ : π/2 < x < π/2 (angular measure)
Range:  y ∈ ℜ
```

The inverse of the TAN function is performed by the one-argument ATAN function (which can be written ATN as well) that returns the angular measure of the cotangent value given as argument; being the cotangent function not injective in the whole x-axis, the ATN function returns

values in a limited range:

```
Domain: x ∈ ℜ
Range:  y ∈ ℜ : -π ≤ y ≤ π (angular measure)
```

The ATAN function works also with two arguments: in this case, ATAN(y,x) will compute the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value. It is used mostly to convert from rectangular (x,y) to polar (r,theta) coordinates that must satisfy the following criteria:

```
x = r*cos theta
y = r*sin theta
```

In general, conversions to polar coordinates are computed with the following general formulae:

```
r := sqrt(x*x+y*y)
theta := atan2(y,x)
```

where x and y are the signed values of the sine and cosine of the trigonometric circle.

E.g.: usage of TAN and ATN(x)

```
OPTION ANGLE DEGREES
PRINT TAN(135)
-1
PRINT ATN(-1)
-45
```

In this case -45 is an exact value, because the result -1 is a definite integer value.

E.g.: usage of ATN(y,x)

```
OPTION ANGLE DEGREES
Angle=30
X=COS(Angle)
Y=SIN(Angle)
R=SQR(X^2+Y^2)
Theta=ATN(Y,X)
PRINT "R=";R,"Theta=";Theta
R= 1          Theta= 30
```

The TANH family functions

The TANH function returns the hyperbolic tangent of the value given as argument.

```
Domain: x ∈ ℜ
Range:  y ∈ ℜ : -1 < y < 1
```

The inverse of the TANH function is performed by the ATANH function

(which may be written as ATNH as well), that returns the measure of the hyperbolic tangent given as argument. Since the TANH function returns values in the range [-1,1], with asymptotes in -1 and 1, the ATANH function return a signed INF in case of x=-1 or x=1, and raises an error in case the value is beyond domain.

Domain: $x \in \Re : -1 \leq x \leq 1$
Range: $y \in \Re$

E.g.

```
PRINT TANH(0.95)
 0.739783
PRINT ATANH(0.739783)
 0.95
```

TEN

The TEN function returns the result of 10 to the power of the number given as argument. In the realm of the possible result, any argument greater than 76.76264889 returns the infinite ∞ value.

Domain: $x \in \Re$
Range: $y \in \Re : y \geq 0$

E.g.

```
PRINT TEN(2)
 100
PRINT TEN(346)
 5.78960E+76
```

UBOUND/UDIM

The UBOUND function (available also as UDIM) returns the highest index value for a given array in argument, with optional indication of the required dimension.

The request is in the form:

```
UBOUND(V[,N])
```

where V is the array name and N (optional) is the dimension. If omitted, dimension 1 is assumed. At present, N may be 1 or 2; if different, an error message is shown.

Note: you can check for the second dimension of a vector, because internally it is always null, and thus this operation returns always zero.

The array is identified by its letters, if () is added, it is skipped.

E.g.

```
DECLARE vec1(6), mat1(2,18)
PRINT UBOUND(vec1(),1),UDIM(mat1,1)
PRINT UBOUND(vec1,2),UDIM(mat1(),2)
 6             2
 0             18
```

VAL

The VAL function returns a numeric value from a string input, so there is not really a Domain or a Range to be specified. The only rule is that the string must hold a number in digits, with the following rules

- a minus/plus sign
- the integer part
- one dot for separation of integer and fraction parts
- the letter E for the introduction of the exponentiation format, w
- a prepended minus/plus sign both for the exponent number
- the exponential part (which must be written if E is used)

Not any other representation; for instance, "-3.4E+02" is a perfect string that looks like a number.

Each part is optional, but one among the integer part or one dot + the fractional part must be used. The figure E02 alone is not seen as a number but as a variable name.

E.g.

```
PRINT VAL("-3.4E+02")
-340

PRINT VAL("12P");
% VAL argument not exhausted in line 19.
 12
```

An error message is shown in case the string argument is empty.

E.g.

```
A$=""
PRINT EVAL(A$)
? VAL/EVAL/EXPR argument is empty in line 2.
PRINT EVAL(A$)
             ^
```

If the string does not *begin* with any number or ground formula, zero is returned and the warning of not exhausted argument is printed.

E.g.

```
A$="Pounds 345"
PRINT VAL(A$)
% VAL argument not exhausted in line 2.
 0
```

See also the EVAL and EXPR function.

VMAX

The VMAX function returns the maximum value present in the array/matrix given as the only argument without parenthesis.

E.g.

```
 DATA 1,3,-4,7,1,-100
 DIM hec(6)
 MAT READ hec
 PRINT VMAX(hec)
  7
```

In case no argument is given, VMAX returns the maximum value (possibly updated) of the last processed VMAX vector/matrix. E.g.

```
 PRINT VMAX()  ' processes hec again
  7
```

In case there is no previous processed VMAX vector/matrix, zero is returned.

VMIN

The VMIN function returns the minimum value present in the array/matrix given as the only argument without parenthesis.

E.g.

```
 DATA 1,3,-4,7,1,-100
 DIM hec(6)
 MAT READ hec
 PRINT VMIN(hec)
 -100
```

In case no argument is given, VMIN returns the minimum value (possibly updated) of the last processed VMIN vector/matrix. E.g.

```
 PRINT VMIN()  ' processes hec again
 -100
```

In case there is no previous processed VMIN vector/matrix, zero is returned.

VSUM

The VSUM function returns the sum of the items of the array/matrix given as the only argument without parenthesis.

E.g.

```
 DATA 1,3,-4,7,1,-100
```

```
DIM hec(6),
MAT READ hec
PRINT VSUM(hec)
-92
```

In case no argument is given, VSUM returns the sum of the items of the (possibly updated) last processed VSUM vector/matrix. E.g.

```
PRINT VSUM()  ' processes hec again
-92
```

In case there is no previous processed VSUM vector/matrix, zero is returned.

2.8.2. Error functions

tbas has some functions dealing with error states. They normally return zero, but in case of an error or an interrupt occurred, they become instantiated with some data related to the error or interrupt. Hence, they may be used for proper error-treating subroutines.

I must notice here that the error functions are built in such a way to avoid the generation of errors on their own, or this would interfere with the error routines, possibly generating infinite loops or wrong error attributions. So, avoid wrong attributions (strings for numbers, or the like): this would cause an immediate stop of your program.

ALN

The ALN function returns the line number label where the user interrupt occurred.

In case of no interruptions, or in case the line had no label, it returns zero.

The ALN function accepts a numerical dummy argument in parentheses, but this is not required. In case the argument is missing, as in ALN(), it is interpreted as a null argument.

E.g.

```
PRINT ALN
 1035
PRINT ALN(4)
 1035
```

ASL

The function ASL returns the line number where the user interrupt occurred, i.e. the physical line number that was in execution when the user pressed CTRL-C.

In case of no interruptions, it returns zero.

The ASL function accepts a numerical dummy argument in parentheses (for compatibility issues), but this is not required. In case the argument is missing, as in ASL(), it is interpreted as a null argument.

E.g.

```
 PRINT ASL
  23
 PRINT ASL(4)
  23
```

ELN

The ELN function returns the line number label where a recoverable error occurred.

In case of no errors, or in case the line had no label, it returns zero.

The ELN function accepts a numerical dummy argument in parentheses, but this is not required. In case the argument is missing, as in ELN(), it is interpreted as a null argument.

E.g.

```
 15 CAUSE ERROR 44
    PRINT ELN

  15
```

ERL/ESL

The ERL function (that may be typed also as ESL, for historical compatibility) returns the physical line number where a recoverable error occurred.

In case of no errors, it returns zero.

The ERL/ESL functions accept a numerical dummy argument in parentheses (for compatibility issues), but this is not required. In case the argument is missing, as in ERL() or ESL(), it is interpreted as a null argument.

E.g.

```
 PRINT ERL
  17
 PRINT ESL(4)
  17
```

ERR/ESM

The ERR function (that may be typed also as ESM for historical issues) returns the numerical code correspondent to the error occurred.

In case of no errors, it returns zero.

The ERR/ESM functions accept a numerical dummy argument in parentheses (for compatibility issues), but this is not required. In case the argument is missing, as in ERR() or ESM(), it is interpreted as a null argument.

E.g.: simulation of error 44

```
    ON ERROR 30
    CAUSE ERROR 44
        STOP
 30 PRINT ERR,
    PRINT ERR(4),
    PRINT ESM,
    PRINT ESM(4)

  44            44            44            44
```

ERR$/ESM$

The ERR$ function (which may be typed also as ESM$) returns a string with the error message, and depending on the argument:

- if the argument is negative or null, it returns the string of current occurred error/interrupt. In case the argument is missing, as in ERR$() or ESM$(), it is interpreted as a null argument.
- if the argument is a proper error code, it prints the string of that error, the same the system would print.

The string is printed as-is, included the tokens that identify the complementary values (%d for an integer, %s for a string, %c for a character), which are not rendered by ERR$.

E.g.: simulation of error 0 for printing a default error message

```
    ON ERROR 30
    PRINT "SIMULATION OF AN ERROR"
    CAUSE ERROR
    REM This line is the first error-free
    PRINT "Safe stop."
    STOP

 30 PRINT ERR$(0)
    JUMP(NXL(ESL))

 SIMULATION OF AN ERROR
 ? Enabled error condition
 Safe stop.
```

NXL

The NXL function returns the physical line number of line following the one where the error occurred.

NXL accepts as argument an address (the physical line number of the file) and returns next executable line (which is, supposedly, the first line free from the error condition) and passes this address to JUMP. If the argument is zero, a warning is printed.
Typically, its argument is ESL, which contains the physical line number where the error occurred, so that:

```
 JUMP NXL(ESL)
```

jumps to the line that followed the faulty line, simulating a RESUME. See the JUMP statement for further details.

E.g.: simulation of error 44 for getting to first error-free line

```
    ON ERROR 30
    PRINT "SIMULATION OF ERROR 44"
    CAUSE ERROR 44
    REM This line is the first error-free
    PRINT "Safe stop."
    STOP

 30 PRINT NXL(ESL)
    JUMP(NXL(ESL))

 SIMULATION OF ERROR 44
  4
 Safe stop.
```

In case a non-numerical argument is given as argument, NXL() will return zero.

2.8.3. String functions

String function always (well, quite always) return a string, but their arguments may be strings or numbers.

Note: in counting string characters, the first position is 1 (first character), and not 0 (like in C). So a string length corresponds to the position of the last character, which is common sense.

ASC/ASCII/ORD

The ASC function (that may be typed ASCII and ORD as well) returns the ASCII code of the character in argument; Syntax:

```
 ASC(<c>)
```

The argument <c> can be:

- a character (only in the range 0-127) which stands for itself.
- a string in any form (enclosed in double quotes, a variable, a string function): in this case the first character of the result string is returned.
- a default token (see below)

Note: the ASCII table represented by ASC() is in the range 0-255.

E.g.

```
 PRINT ASC(A)
  65
 PRINT ASC(0)
  48
 PRINT ASC("Anna")
  65
 PRINT ASCII("")
  0
 PRINT ORD( )    ' the blank space
  32
```

The ASC function recognizes a number of tokens identifying the standard ASCII control codes; they are:

```
 NUL = 0     The NULL character
 SOH = 1     Start of Heading
 STX = 2     Start of Text
 ETX = 3     End of Text
 EOT = 4     End of Transmission
 ENQ = 5     Enquiry
 ACK = 6     Acknowledge
 BEL = 7     Bell
 BS = 8      Backspace
 HT = 9      Horizontal Tabulation
 LF = 10     Line Feed
 VT = 11     Vertical Tabulation
 FF = 12     Form Feed
 CR = 13     Carriage Return
 SO = 14     Shift Out
 SI = 15     Shift In
 DLE = 16    Data Link Escape
 DC1 = 17    Device Control 1
 DC2 = 18    Device Control 2
 DC3 = 19    Device Control 3
 DC4 = 20    Device Control 4
 NAK = 21    Negative Acknowledge
 SYN = 22    Synchronous Idle
 ETB = 23    End of Transmission Block
 CAN = 24    Cancel
 EM = 25     End of Medium
 SUB = 26    Substitute
 ESC = 27    Escape
 FS = 28     File Separator
 GS = 29     Group Separator
```

```
RS = 30     Record Separator
US = 31     Unit Separator
SP = 32     The space character
DEL = 127   Delete
```

The tokens must not be typed within double strings (as I could be tempted to do) because in this case it is a string, and the ASC function returns the ASCII code of the first character in the string. E.g.

```
PRINT ASC(FF)
 12
PRINT ASC(DEL)
 127
PRINT ASC(RS)
 30
PRINT ASC(NUL)
 0
```

See https://www.cs.tut.fi/~jkorpela/chars/c0.html for details.

Note: you may have noticed that this function does not return a string, but a number; so, why is it here? Well, in my distorted mind, the ASC function is not a math function, and it deals with characters, so I put it here. You don't agree? You have all reasons, but this function will remain here. You will forgive me.

BIN$/BINOF$

The BIN$ function (that may be written also BINOF$) returns a string with the binary representation of the number in argument. The syntax is:

```
BIN$(<n1>[,<n2>])
```

The first argument <n1> contains the number to be converted to its binary string form (in normal decimal form). The second argument <n2> is optional; if <n2>=0, the number is filled with leading zeroes (to complete the 32 bits pattern); if <n2>=1 (or any not null value), the number is printed in compact form. If the second argument is absent, the number is always printed in compact form.

A negative number is always printed in full 32 bits.

The bit 0 is on the right. Use the REV$ function to reverse it in case you want bit 0 on the left.

E.g.

```
PRINT BIN$(13)
1101
PRINT BIN$(-13)
11111111111111111111111111110011
PRINT BIN$(13,0)
00000000000000000000000000001101
PRINT BIN$(13,1)
1101
```

Of course the following holds:

```
PRINT BIN$(BIN("1101"))
1101
```

CAP$

The CAP$ returns string in argument with lower characters and first letter capitalized. Separators are blanks and dashes. Syntax:

```
CAP$(<s>)
```

E.g.

```
a$="PICCOLA QUESTIONE DA CHIARIRE"
b$="member of the roman-catholic church"
c$="john LENNON onO"
print cap$(a$)
print cap$(b$)
print cap$(c$)

Piccola Questione Da Chiarire
Member Of The Roman-Catholic Church
John Lennon Ono
```

CHR$

The CHR$ returns a one-length string containing the character corresponding to the ASCII code in argument. It is someway the reverse of the ASC function. Syntax:

```
CHR$(<n>)
```

E.g.

```
PRINT CHR$(9);CHR$(65);CHR$(235);CHR$(91)
        Aë[
```

Note: the picture of characters in the 128-255 range depends of the coding of your terminal, and you may see a different output.

ENV$

The ENV$ function returns a string with the content of the environment variable passed as a string argument. Syntax:

```
ENV$(<s>)
```

E.g.

```
PRINT ENV$("PWD")
/Volumes/BINBACKUP/tbas
[or whatever is your working directory]
```

Note: on systems different from the UNIX family (or the UNIX emulated versions, like Cigwin), this statement may work badly.

FORMAT$

The FORMAT$ function accepts a format string <s> and a value <n> as arguments and returns a string containing the value formatted, according to the laws of PRINT USING, adopting string <s> as format string. Syntax:

```
FORMAT$(<s>,<n>)
```

E.g.

```
PRINT FORMAT$("=###.#########",sqr(2))
=  1.414213562

PRINT FORMAT$("##.###^^^^",sqr(3))
 1.732E+00
```

If the string should contain more than one number format, only the first is used, and the rest ignored. If the string should contain a string format, an error is raised. E,g,

```
PRINT FORMAT$("## ##.###^^^^",sqr(3))
 1

PRINT FORMAT$("<#######>",sqr(3))
? Attempt to output a number to a string field or vice versa in line 1.
PRINT FORMAT$("<#######>",sqr(3))
```

FREE$

The function FREE$ (also as FREE$(X) with X dummy) returns a string containing the number of free programming slots. Remember that memory is allocated, when the line is stored, by the Operating System, so there is enough space for programs as long as the Operating System can allocate memory; the value indicates the number of free programming lines available (of any length) that you can still use.

HEX$/HEXOF$

The HEX$ function (that may be written also as HEXOF$) returns a string with the hexadecimal representation of the number in argument. The syntax is:

```
HEX$(<n>)
```

where <n1> is the number to be converted to its hexadecimal string form (in normal decimal form).

A negative number is always printed in full 32 bits (8 bytes).

The byte 0 is on the right. Use the REV$ function to reverse it in case

you want bit 0 on the left.

E.g.

```
PRINT HEXOF$(13)
D
PRINT HEXOF$(-13)
FFFFFFF3
PRINT HEXOF$(-1)
FFFFFFFF
```

INKEY$

The INKEY$ function stops program execution and waits for a key. When the key is pressed on the keyboard (no need to press ENTER), the correspondent ASCII character is immediately returned as a one-char string, without echo. The function has the syntax

```
INKEY$[(<n>)]
```

The optional argument <n> defines the time in seconds after which INKEY$ returns the null string and the program can continue.

In the following example, the [text in brackets] represents text typed on the keyboard, and not part of the input or output.

E.g.

```
A$=INKEY$
PRINT "|";A$;"|"

[d]
|d|
```

E.g.

```
key$=inkey$(3)
if key$="" then
   print "Key null"
else
   print "Key: ";key$
end if

[g before time expires]
Key: g

[after time expires]
Key null
```

Note: the timing feature is enabled only for UNIX and the UNIX-family; on Windows® and the old Max OS 9 the timing feature is disabled.

IMPORTANT NOTE: in Windows, due to the compatibility gcc/API libraries, INKEY$ may echo the typed character. This is not due to **tbas**.

INPUT$

The INPUT$ function stops program execution and waits for a text to be typed by the used and ended by ENTER, after which the text is returned as a string. The function has the syntax:

```
INPUT$[(<n>)]
```

The optional argument <n> defines the imposed string length; in this case, after the required number of characters, the string is returned and the program can continue.

In the following example the [text in brackets] represents text typed on the keyboard, and not part of the input or output.

E.g.

```
A$=INPUT$
PRINT "|";A$;"|"

[Typing is good + ENTER]
|Typing is good|
```

E.g.

```
key$=input$(3)
print "Key: ";key$

[g t s without ENTER]
Key: gts
```

Note: if OPTION ECHO is used, the typed characters are echoed (default). If OPTION NO ECHO is used, there is no echo and the typing remains hidden.

INSTR

The INSTR returns the numeric position of string <s2> in string <s1>, starting at position <n> if specified, or else at first character of <s1> in case the position argument is not specified. Syntax:

```
INSTR([<n>,]<s1>,<s2>)
```

If the position value <n> is lower than zero, it is set to zero, and if it's greater than length of <s1>, it is set equal to length of <s1>.

E.g.

```
A$="LAVABEN"
B$="AVA"
PRINT INSTR(A$,B$)
 2

A$="LAVABEN"
```

```
B$="AVA"
PRINT INSTR(3,A$,B$)
 0
```

If either of the comparing strings is null, the returned value is always zero.

E.g.

```
A$="LAVABEN"
B$=""
PRINT INSTR(A$,B$)
 0

A$=""
B$="AVA"
PRINT INSTR(A$,B$)
 0
```

The case comparison is influenced by OPTION CASE; if ON the search of INSTR will find "A" and "a" as different. If OFF, the two will be matched as equal.

LEFT$

The LEFT$ function returns the left substring of string <s> in the first argument; the length of the returned string is controlled by the second argument <n>. Syntax:

```
LEFT$(<s>,<n>)
```

E.g.

```
PRINT LEFT$("John Lennon",4)
John
```

If <n> is negative, an error is raised. If it is null, the empty string is returned.

LOWER$/LCASE$

The LOWER$ function (that may be typed also as LCASE$) returns the string in argument with all lower characters in the range a÷z. Syntax:

```
LOWER$(<s>)
LCASE$(<s>)
```

E.g.

```
PRINT LOWER$("THIS MORNING")
this morning
```

LPAD$

The LPAD$ function returns the string <s> in first argument left-padded with the number of spaces specified in the second argument <n> spaces. An error condition is raised if number is negative. Syntax:

```
LPAD$(<s>,<n>)
```

E.g.

```
A$="IAN"
B$=LPAD$(A$,5)
PRINT "|";B$;"|"
|  IAN|
```

If string is longer, it is returned without padding. E.g.

```
A$="COMMONWEALTH"
B$=LPAD$(A$,5)
PRINT "|";B$;"|"
|COMMONWEALTH|
```

If string is null, the statement is equivalent to SPACE$(<n>). E.g.

```
A$=""
B$=LPAD$(A$,5)
PRINT "|";B$;"|"
|     |
```

MID$

The MID$ function returns the substring of string <s> in the first argument, starting from the position <n1> defined by the second argument, for the number of characters <n2> contained in the third argument. Syntax:

```
MID$(<s>,<n1>[,<n2>])
```

This function differs from SEG$ because in MID$ the numerical arguments specify the starting character and the length of selection, while in SEG$ they specify the starting and ending characters of selection.

If <n1> is null or negative, an error is raised.

Third argument is optional; if not specified, length is set to the length of the string from n1 to the end of string. If it is null, the null string is returned. If it is negative, an error is raised.

E.g.

```
PRINT MID$("Christopher Cross",5, 4)
stop

PRINT MID$("John Lennon",6)
```

```
Lennon

PRINT MID$("John Lennon",6,6)
Lennon

PRINT MID$("John Lennon",6,0)
[the empty string]

PRINT MID$("Christopher Cross",56,32)
[the empty string]
```

OCT$/OCTOF$

The OCT$ function (that may be written also as OCTOF$) returns a string with the octal representation of the number in argument. The syntax is:

```
OCT$(<n>)
```

where <n1> is the number to be converted to its octal string form (in normal decimal form).

A negative number is always printed in full 32 bits (at most 11 triplets).

The byte 0 is on the right. Use the REV$ function to reverse it in case you want bit 0 on the left.

E.g.

```
PRINT OCT$(13)
15
PRINT OCT$(-13)
37777777763
PRINT OCT$(-1)
37777777777
```

PIPE$

The PIPE$ string function returns the content of the buffer string previously retrieved by the PIPE statement (see). The returned value of PIPE$ can be a unique string (trimmed if greater than 255 characters) or a slice of the whole string: in this case, a further execution of PIPE$ will yield the next slice, and so on.

PIPE$ can be used in two modes: PIPE$ without arguments return the whole output at once; e.g.:

```
PRINT PIPE$
```

PIPE$(N) with a numeric argument will return next N-wide slice, with N lower or equal to 255; e.g.:

```
PRINT PIPE$(128)
```

Note that if the argument of N is null or negative, the whole output is returned, because an empty slice is meaningless. Remember also that if N is too small, the string may not be fully restored. Use values that can satisfy you the best.

To fully understand its usage, look carefully at the following program:

```
SUB OUTPIPE(C$,W)      ' C$ = SHELL COMMAND TO BE EXECUTED, W = SLICE
   PIPE C$             ' THE PIPE STATEMENT OPENS THE COMMAND
   DO
      A$=PIPE$(W)      ' THE PIPE$ FUNCTION GIVES ONE SLICE AT A TIME
      PRINT A$;
   LOOP WHILE A$<>""   ' NO MORE SLICES; THE EMPTY STRING IS RETURNED
   PRINT
END SUB

OUTPIPE("ls -la",40)
```

The SUB OUTPIPE() uses PIPE to open the C$ string argument as a bash command; next, the DO-LOOP cycle reads consecutive slices from the channel opened by PIPE using the function PIPE$, and prints the returned value in a packed form; the cycle stops when the last call to PIPE$() returns the empty string (signalling the end of output); the effect of the packed form (the appended semicolon) is that the output is fully rebuilt and it appears as if the bash command "ls -la" was executed from the BASIC source!

POS

The POS returns the numeric position of string b$ in string a$, starting at position <n> if specified, or else at first character of a$ in case the position argument is not specified. Syntax:

```
POS(a$,b$[,<n>])
```

If the position value <n> is lower than zero, it is set to zero, and if it’s greater than length of a$, it is set equal to length of a$.

E.g.

```
A$="LAVABEN"
B$="AVA"
PRINT POS(A$,B$)
 2

A$="LAVABEN"
B$="AVA"
PRINT POS(A$,B$,3)
 0
```

If either of the comparing strings is null, the returned value is always zero.

E.g.

```
A$="LAVABEN"
B$=""
PRINT POS(A$,B$)
 0

A$=""
B$="AVA"
PRINT POS(A$,B$)
 0
```

The case comparison is influenced by OPTION CASE; if ON the search of POS will find "A" and "a" as different. If OFF, the two will be matched as equal.

REV$

The REV$ function returns the reverse string given as argument. Syntax:

```
REV$(<s>)
```

E.g.

```
PRINT REV$("The Beatles")
seltaeB ehT

PRINT CAP$(REV$("The Rolling Stones"))
Senots Gnillor Eht

PRINT REV$(BIN$(43))
110101
```

RIGHT$

The RIGHT$ function returns the right substring of string <s> in the first argument; the length of the returned string is controlled by the second argument <n>. Syntax:

```
RIGHT$(<s>,<n>)
```

E.g.

```
PRINT RIGHT$("John Lennon",6)
Lennon
```

If <n> is negative, an error is raised. If it is null, the empty string is returned.

RPAD$

The RPAD$ function returns the string <s> in first argument right-padded with the number of spaces specified in the second argument <n> spaces. An error condition is raised if number is negative. Syntax:

```
RPAD$(<s>,<n>)
```

E.g.

```
A$="IAN"
B$=RPAD$(A$,5)
PRINT "|";B$;"|"
|IAN  |
```

If string is longer, it is returned without padding. E.g.

```
A$="COMMONWEALTH"
B$=RPAD$(A$,5)
PRINT "|";B$;"|"
|COMMONWEALTH|
```

If string is null, the statement is equivalent to SPACE$(<n>). E.g.

```
A$=""
B$=RPAD$(A$,5)
PRINT "|";B$;"|"
|     |
```

RPT$/REPEAT$

The RPT$ function (that may be written in full also as REPEAT$) builds a string made of multiple occurrences of the first argument, for the number of items specified by the second argument. Syntax:

```
RPT$(<s>,<n>)
```

E.g.

```
PRINT RPT$("Watch out! ",3)
Watch out! Watch out! Watch out!
```

Note: this statement makes very easy passing the maximum string length limit, causing an error. So pay attention when building such strings. E.g.

```
d$=RPT$("H",350)

? String formula > 255 characters in line 18.
d$=RPT$("H",350)
                  ^
```

See also the SPACE$ statement (from a different tradition), that has a similar scope.

SEG$

The SEG$ function returns the substring of string <s> in the first argument, starting from the position <n1> defined by the second argument, and ending in the position <n2> defined in the third argument. Syntax:

```
SEG$(<s>,<n1>[,<n2>])
```

This function differs from MID$ because in SEG$ the numerical arguments specify the starting and ending characters of selection, while in MID$ they specify the starting character and the length of selection.

If <n1> is null or negative, an error is raised.

Third argument <n2> is optional; if not specified, the end position is set to be the last character of string <s>; if <n2> is null or lower than <n1>, the null string is returned.

E.g.

```
PRINT SEG$("Christopher Cross",5,8)
stop
PRINT SEG$("Christopher Cross",13)
Cross
PRINT SEG$("Christopher Cross",4,0)
[the null string]
PRINT SEG$("Christopher Cross",4,3)
[the null string]
```

SPC$/SPACE$

The SPC$ function (that may be typed also in full as SPACE$) returns a string of spaces, long as the value defined in the argument. Syntax:

```
SPC$(<n>)
SPACE$(<n>)
```

If <n> is negative, it is set to zero. If greater than the greatest string length, it is set equal to the greatest string length (255 characters in current version).

If <n> is zero, the null string is returned.

E.g.

```
PRINT "A";SPC$(4);"B"
A    B
PRINT "A";SPACE$(40);"B"
A                                       B
PRINT "A";SPC$(0);"B"
AB
```

STR$/NUM$

The STR$ function (that may be typed also as NUM$) evaluates the numeric formula <f> in the argument, and returns a string containing the result; this is not a number, but a string, and follows the rules of the strings printing (no space ahead or after in no quote mode). Syntax:

```
STR$(<f>)
NUM$(<f>)
```

E.g. notice the difference in the output of the following two lines:

```
PRINT SQR(2)
 1.41421

PRINT STR$(SQR(2))
1.41421
```

In the first case the output is a number, in the second case the output is the string ['1' '.' '4' '1' '2' '1'].

STRING$

The STRING$ function builds a string made of multiple occurrences of the second argument, for the number of items specified by the first argument. Syntax:

```
SPACE$(<n>,<c>)
SPACE$(<n>,<s>)
```

If the second argument is a number, it is interpreted as an ASCII code, and the built string will consist of <n> copies of the correspondent character:

```
PRINT STRING$(10,78)    ' 78 is the ASCII code of N
NNNNNNNNNN
```

If the second argument is a string result in any form, this string is the string to be multiplied.

```
PRINT STRING$(3,"Watch out! ")
Watch out! Watch out! Watch out!
```

Note: this statement makes very easy passing the maximum string length limit, causing an error. So pay attention when building such strings. E.g.

```
d$=STRING$(350,"H")

? String formula > 255 characters in line 18.
d$=STRING$(350,"H")
                 ^
```

See also the RPT$/REPEAT$ statements (from a different tradition), that have a similar scope.

TRIM$

The TRIM$ function returns the string in argument without trailing spaces, ahead and behind. The trailing spaces are blanks and tabulations. Syntax:

```
TRIM$(<s>)
```

E.g.

```
A$="   Avenue    "
PRINT "|";A$;"|"
PRINT "|";TRIM$(A$);"|"

|   Avenue    |
|Avenue|
```

UPPER$/UCASE$

The UPPER$ function (that may be typed also as UCASE$) returns the string in argument with all upper characters in the range A÷Z. Syntax:

```
UPPER$(<s>)
UCASE$(<s>)
```

E.g.

```
PRINT UPPER$("this morning")
THIS MORNING
```

USR$

The USR$ function returns a default string, which in the default version is "LS:<tonibin>". This string is customizable in tbas.h, before compiling. Syntax:

```
USR$[(<n>)]
```

(dummy argument if given). This function does exist only for compatibility issues with the Dartmouth tradition.

2.8.4. File functions

The syntax of these functions, as homage to the various dialects, accept a wide variety of formats.

```
EOF             - return true on end of file (random or seq)
END             - return true on end of file (random or seq)
EXISTS          - return read/write permissions on argument file
MORE            - return false on end of file (random or seq)
LOC             - return current record pos of a random access file
LOF             - return last record pos of a random access file
```

EOF/END

EOF and END both return true if the End-Of-File is reached on the channel in argument. The syntax is (x ranges from 1 to 9):

```
EOF x          END x
EOF (x)        END (x)
EOF #x         END #x
EOF # (x)      END # (x)
EOF (#x)       END (#x)
EOF :x         END :x
EOF : (x)      END : (x)
EOF (:x)       END (:x)
```

EXISTS

EXISTS is a function that analyzes the file whose name is given as argument and returns specific codes that help in using the file or address the program in other directions if file does not exists; example:

```
IF EXISTS("/Volumes/BACKUP/track.tst")>0 THEN ....
```

The string in argument may be a literal string enclosed in double quotes or a string variable of any kind (string variable or array string variable).

Return values are:

- if the file does not exist, -8 is returned;
- if the string is empty, -4 is returned, along with a warning
- if the file exists but is not a file (a directory or a block file) -4 is returned
- if the file exists but is not accessible by **tbas**, 0 is returned
- if the file exists and is accessible in read mode only, 1 is returned
- if the file exists and is accessible in write mode only, 2 is returned
- if the file exists and is accessible in read and write mode, 3 is returned (that is 1+2)

The sign of the return values help in using EXISTS:

- if the return value is < 0, the file does not exist, or it's not a file, or there is an irregular file name in the request
- if the return value is ≤ 0, no read/write operations are possible
- if the return value is strictly 0, the file does exist, but is not accessible (for example it is on an unaccessible drive)

The bitwise & operator can be used in this context for more specific controls:

- to check for read access, use 1 & EXISTS(file$)
- to check for write access, use 2 & EXISTS(file$)
- to check for read & write access, use 3 & EXISTS(file$)

where file$ contains the file name.

LOC/LOF

LOC returns the number of the record to which the pointer of the random access file is currently pointing to, where as LOF returns the number of the last record in the random access file. The syntax is

```
LOC x      LOF x
LOC (x)    LOF (x)
LOC :xLOF :x
LOC : (x)  LOF : (x)
LOC (:x)   LOF (:x)
```

All the functions END, EOF, MORE, LOC and LOF are built as to ignore the parentheses position; all the function END, EOF and MORE assume sequential access file if no identifier is written (# or :), while LOC and LOF assume random access file if no : is written, and if # is used with LOC and LOF, an error message is printed.

MORE

MORE is the inverse of EOF, and returns true if the End-Of-File is not reached on the channel in argument. The syntax follows:

```
MORE x
MORE (x)
MORE #x
MORE # (x)
MORE (#x)
MORE :x
MORE : (x)
MORE (:x)
```

2.8.5. Time functions

There are many functions that deal with time, and all use the decimal time format. The 'decimal time' is a time format in decimals (the number returned by CLK); it is in two formats:

- the time format, composed by hh.dddd, where hh is the hour, and dddd are the sum of minutes/60 and seconds/3600.
- the date format, composed by yy.mmdd or yyyy.mmdd, where yy are last two year's digits (the century retrieved is the current one), yyyy is the whole year (it must be greater than 1600, or it will be set to 1600), mm is the month (01-12) and dd is the day (01-31).

CLK

The CLK function return current decimal time. Syntax:

```
CLK[(<n>)]
```

(dummy argument if given). CLK may be used without parentheses and argument.

E.g. supposing it's 11 hour, 40 minutes and 37 seconds:

```
 PRINT CLK
  11.6769
```

(which is about the 67% of the 11th hour).

DATE

The DATE function returns current date in decimal format. Syntax:

```
 DATE[(<n>)]
```

(dummy argument if given). DATE may be used without parentheses and argument.

E.g. supposing it's January 15, 2015:

```
 PRINT DATE
  15.0115
```

DAY

The DAY function returns a date day number. The argument of the function DAY, if zero, causes current day value to be returned; if not zero, it is interpreted as a decimal date format and the correspondent day value is returned.

If used without parentheses and argument, DAY returns current day value.

E.g. supposing it's August 26, 2014

```
 PRINT DAY
  26

 PRINT DAY(2013.0414)    ' decimal date of April 14, 2013
  14
```

DAY does not consider at all the year's value and thus does not check if a wrong day is used (for instance using February 29 in a not-leap year).

HOUR

The HOUR function returns a hour number. The argument of the function HOUR, if zero, causes current hour value to be returned; if not zero, it is interpreted as a decimal time format and the correspondent hour value is returned.

If used without parentheses and argument, HOUR returns current hour value.

E.g. supposing it's 14:49:52

```
 PRINT HOUR
  14

 PRINT HOUR(13.0414) ' decimal time of 13:02:29
  13
```

MINUTE

The MINUTE function returns a minute number. The argument of the function MINUTE, if zero, causes current minute value to be returned; if not zero, it is interpreted as a decimal time format and the correspondent minute value is returned.

If used without parentheses and argument, MINUTE returns current minute value.

E.g. supposing it's 14:49:52

```
 PRINT MINUTE
  49

 PRINT MINUTE(13.0414)   ' decimal time of 13:02:29
  2
```

MONTH

The MONTH function returns a date month. The argument of the function MONTH, if zero, causes current month value to be returned; if not zero, it is interpreted as a decimal date format and the correspondent month value is returned.

If used without parentheses and argument, MONTH returns current month value.

E.g. supposing it's August 26, 2014

```
 PRINT MONTH
  8

 PRINT MONTH(2013.0414)  ' decimal date of April 14, 2013
  4
```

MONTH does not consider at all the year's value and thus does not check if a wrong day is used (for instance using February 29 in a not-leap year).

MONTH$

The MONTH$ function returns a string containing a month's name. The syntax is:

```
 MONTH$
```

```
MONTH$(<n1>)
MONTH$(<n1>,<n2>)
```

In the second form, <n1> must be a decimal date number, and in this case the correspondent month string is returned. If <n1> is null, current month string is returned.

E.g.

```
PRINT MONTH$(14.0225)
February

PRINT MONTH$(0) ' Today is January 1st, 2016
January
```

MONTH$ alone is a synonym of MONT$(0).

The second optional function argument <n2> is an integer value that specifies how many letters of the month must be printed. Values range from 1 to 3 (in the example January is assumed):

```
PRINT MONTH$(0,1)
J

PRINT MONTH$(0,2)
Ja

PRINT MONTH$(0,3)
Jan
```

If <n2> is negative or null, the whole string is returned. If it's greater than 3, it's automatically resized to 3.

MONTH$ does not consider at all the year's value and thus does not check if a wrong day is used (for instance using February 29 in a not-leap year).

SECOND

The SECOND function returns a second number. The argument of the function SECOND, if zero, causes current second value to be returned; if not zero, it is interpreted as a decimal time format and the correspondent second value is returned.

If used without parentheses and argument, SECOND returns current second value.

E.g. supposing it's 14:49:52

```
PRINT SECOND
 52

PRINT SECOND(13.0414)   ' decimal time of 13:02:29
 29
```

TIM

The TIM function returns the elapsed execution time in milliseconds from the running program start up to the TIM call; the TIM call includes all the programs in the current CHAIN so far.

Two independent TIM calls define an interval of execution that does not include the user delay in case of keyboard input.

During an interactive session, TIM is reset at each RUN, but it's not reset at the end, continuing advancing. It can so be used to test some user time intervals.

E.g. supposing you take some seconds to press the key 't'

```
T1=TIM
A$=INKEY$
PRINT A$
T2=TIM
PRINT "TIME:";(T2-T1)

t
TIME: 1
```

(you may see different values, here, because the TIM function depends on the computer clock).

E.g. clock measure (to show how slow my computer is!)

```
FOR I=1 to 10
   FOR J=1 to 10000
   NEXT
   PRINT TIM
NEXT

 272
 547
 850
 1108
 1409
 1679
 1954
 2238
 2513
 2800
```

It takes 2800 milliseconds (that is 2.8 seconds) to perform the test. I'd like to know how fast modern computers are!

Note: I wrote this program with my old eMac with Mac OS X 10.4; now I use a laptop quad core (which is 10x faster), an Intel 4xCore i3 @ 2.00 GHz, with OpenSuse Linux, an OS I adore. So I guess this part should be rewritten, but I won't do it. It reminds me of some recent past.

TIME$

The TIME$ string returns a standard time string in the form "HH:MM:SS". Syntax:

```
TIME$[(<n>)]
```

(dummy argument if given).

E.g.

```
PRINT TIME$
11:39:04
```

WEEKDAY$

The WEEKDAY$ function returns a string containing the week day's name of the decimal date in the argument. The syntax is:

```
WEEKDAY$
WEEKDAY$(<n1>)
WEEKDAY$(<n1>,<n2>)
```

In the second form, <n1> must be a decimal date number starting from January 1st, 1583 (see the note at YEAR), and in this case the correspondent week day string is returned. If <n1> includes February 29 and the year is leap, all's good, but if you use February 29 in a not-leap year, the day is automatically converted to March 1st. Rules are:

-if <n1> is negative or null, current week day string is returned.

E.g.

```
PRINT WEEKDAY$(0)   ' Today is January 1st, 2016
Friday
```

WEEKDAY$ alone is a synonym of WEEKDAY$(0).

- if <n1> is in the range 1÷99, it is interpreted as belonging to the normalized century (see YEAR);

E.g.

```
PRINT WEEKDAY$(14.0225) ' decimal date of February 25, 2014
Tuesday

PRINT WEEKDAY$(45.0225) ' decimal date of February 25, 1945
Sunday
```

- if <n1> is in the range 100÷1582, a warning is printed and the year is set to 1583;

E.g.

```
PRINT WEEKDAY$(123.1212)
% Illegal decimal year -- returning 1583 in line 3.
Monday
```

– otherwise the year itself in the form yyyy is returned;

E.g.

```
PRINT WEEKDAY$(2014.0225)   ' decimal date of February 25, 2014
Tuesday
```

– in case you use leap years and not-leap years:

```
PRINT WEEKDAY$(2012.0229)   ' date of Febrary 29, 2012 (leap)
Wednesday

PRINT WEEKDAY$(2013.0229)   ' date of Febrary 29, 2013 (not-leap)
Friday

PRINT WEEKDAY$(2013.0301)   ' date of March 1st, 2013 (not-leap)
Friday
```

You see the second and third examples refer to the same day, because February 29, 2013 was converted to March 1st, 2013 before calculation.

The second optional function argument <n2> is an integer value that specifies how many letters of the week day must be printed. Values range from 1 to 3 (in the example Friday is assumed):

```
PRINT WEEKDAY$(0,1)
F

PRINT WEEKDAY$(0,2)
Fr

PRINT WEEKDAY$(0,3)
Fri
```

If <n2> is negative or null, the whole string is returned. If it’s greater than 3, it’s automatically resized to 3.

YEAR

The YEAR function returns a date year. The year may be any year since 1583[2];

The argument is evaluated according to the following rules, taking as argument the year part of the decimal date input value (that is the integer part):

– if the argument is negative or null, or absent, current year is

[2] Why 1583? This is the first year after the Gregorian calendar institution ruled by decree in 1582 in the papal bull "Inter Gravissimas" by Gregory XIII, so I assume this is the first useful value for the modern times.

returned;

E.g.

```
 PRINT YEAR
 PRINT YEAR(0)
 PRINT YEAR(-31.1212)
```

All these return 2016 (if this is the current year).

- if the argument is in the range 1÷99, the normalized century plus the argument is returned; the normalized century is the interval between -85 and +15 years with respect to the current year, and not the pure current century.

E.g.

```
 PRINT YEAR(22.1214) ' decimal date of December 14 of '22
  2022

 PRINT YEAR(45.1214) ' decimal date of December 14 of '45
  1945
```

The normalized century has a turning point: for instance, in 2016, the turning point is 31 (=16+15), that is:

```
 PRINT YEAR(30.1212) ' decimal date of December 12 of '30
  2030

 PRINT YEAR(31.1212) ' decimal date of December 12 of '31
  1931
```

This turning point varies from year to year. So take this in account when using YEAR.

- if the argument is in the range 100÷1582, it is set to 1583, and a warning is printed.

E.g.

```
 PRINT YEAR(123.0315)
 % Illegal decimal year -- returning 1583 in line 6.
  1583
```

- otherwise, the year itself in the form yyyy is returned.

```
 PRINT YEAR(3023.0330)   ' decimal date of March 30, 3023
  3023
```

Note: since the year zero causes current year to appear, to select the year 2000 you must use the full four-digits year, because 00 would be of course seen as zero.

2.9. System constants and variables

The following are system constants or variables whose content is not user-writable; some are simply the 'constant' version of functions that use a dummy value - e.g. TIM and TIM(N) are equivalent - while others work as if the argument is zero - e.g. MONTH is equivalent to MONTH(0). Those ending with $ have string content.

> **Beware: you can redefine these system variables as new user variables, constants or sub names, but remember that tbas** does not control such names on declaration, and if you don't really know what you are doing, you could make impossible using them, or worse cause segmentation error.

CLK

return current time in decimals

CLK$

return current hour in the format HH.MM.SS

COMMAND$

return a string containing the parameters following the file name in the **tbas** invocation from console. COMMAND$ is a global string holding all the parameters and arguments that follow the file name in the console invocation of **tbas**; in the interactive session, return what follows the '--' after RUN (see).
E.g. the following program 'COUNTTO.BAS':

```
 LET LIM = VAL(COMMAND$)
 FOR I = 1 TO LIM
     PRINT I,
 NEXT
 PRINT
 PRINT "I COUNTED TO ";COMMAND$;" AS YOU COMMANDED"
```

is invoked like this:

```
 $ ./tbas COUNTTO.BAS 10
```

and returns as expected:

```
 1            2            3            4            5
 6            7            8            9            10
 I COUNTED TO 10 AS YOU COMMANDED
```

Another specific usage of COMMAND$ is to furnish arguments to the EXEC statement (see).

e.g. the following program 'EXECTO.BAS':

```
 LET A=24
```

```
 PRINT "THE STATEMENT YOU INPUT=";COMMAND$;"; IT IS EXECUTED AS:"
 EXEC COMMAND$
```

is invoked like this:

```
 $ ./tbas EXECTO.BAS "PRINT A*A"
```

and returns as expected:

```
 THE STATEMENT YOU INPUT=print a*a; IT IS EXECUTED AS:
  576
```

The conversion to lower characters is done to prevent any misinterpretations, because **tbas** is case insensitive.

Note: the string interpreted in COMMAND$ is the union of all the strings that follow the commands in command mode; besides, COMMAND$ accepts escaping (it's bash that accomplish this task):

```
 $ ./tbas EXECTO.BAS "PRINT\"Hello,\";A*A"
```

which produces

```
 THE STATEMENT YOU INPUT=print"Hello,";a*a; IT IS EXECUTED AS:
 Hello, 576
```

and the following

```
 $ ./tbas EXECTO.BAS "PRINT" "A*A"
```

which produces

```
 THE STATEMENT YOU INPUT=print  A*A; IT IS EXECUTED AS:
  576
```

Note: beware: under the interactive session, the COMMAND$ is filled with what followed the '--' (dash-dash) token after RUN, but in this case, the string must not be enclosed in quotes. See RUN for more notes.

NOTE: Windows users may pay attention to use COMMAND$ with a prepended @ to any string <str> argument, which causes Windows to return the content of the file named <str> (if any available) in place of the string <str> itself, and if this is not the intended behaviour, this may cause some possible problems to the program execution.

CR

(constant) return the end-of-line ASCII code

DATE

return the current date in decimal format

DATE$/DAT$

return the current date in the format of a string as YY/MM/DD.

DAY

return the current day

DIR$

return the current working directory

DET

return the determinant after a matrix inversion

EPS

(constant) return the least computable number

FALSE

(constant) return always the false condition 0

FREECHANNEL

return the index of first available channel (i.e. not yet opened by OPEN, FILE or FILES); if none is available, return 0

HOUR

return the current hour in the range 0..23

INF

(constant) return the higher significant number

LASTCHANNEL

return the index of the last channel successfully opened by OPEN, FILE or FILES; if none is opened, return 0

LASTF

return the last value calculated through a DEF FN

MAYBE

return a random TRUE or FALSE value

MINUTE

return the current minute in the range 0..59

MONTH

return the current month in the range 1..12

NUM

return the number of values input so far by MAT INPUT

PI

(constant) return the Greek Pi

PROGRAM$

(constant) return the name of the current program under execution. In the Interactive session, PROGRAM$ return the name of the file LOADed/SAVEd as last

RND

return a random number in the range 0..1, limits not included.

SECOND

return the current second in the range 0..59

STREAM

return 0 if the current stream is the default (stdout), or 1 if the current stream is the error stream (stderr)

TIM

return the number of milliseconds since program start

TRUE

(constant) return always the true condition -1

YEAR

return the current year as a four-digits number.

Note: FALSE and TRUE have been added for more elegant 'eternal' cycles (cycles that are broken only by some internal condition) like:

```
WHILE TRUE
```

```
    ....
 END WHILE
```

or

```
 DO
    ....
 LOOP UNTIL FALSE
```

Note: UNIX, Mac and Windows® machine differ in the usage of the Carriage Return; the CR constant should be able to overpass this problem; if you check against CR rather than 10 or 13 (or both), you should be able to make compatible programs. This has to be tested on several machines before, though... a task beyond my power.

2.10. Manual Interrupt

When the user with to stop an execution (because of an infinite cycle, or during an INPUT or whatever is needed), the behaviour depends on the running execution:

- during the console execution (the default one) the execution is stopped with a message:

tbas stopped after user interrupt at line NNN (labelled as LLL)

where NNN is the physical line number of the program listing. If the line has a label, it is added to the output as LLL.

- during the interactive session, to stop execution, a CTRL-C alone is not enough. The correct sequence that stops execution and also re-enables the session operability is CTRL-C + ENTER. If a CONTINUE is used after a CTRL-C alone, the command is not operative and another invocation is needed (the ENTER you typed after the command re-enables the session, but the command is lost). The message that appears is the following:

```
 CTRL-C stop at line LLL
```

where LLL is the line number where the stop occurred (the label number and not the physical number).

2.11. QUEUE and ROUTE in details

The QUEUE and ROUTE statements are an aid in using **tbas** for directing printing to a printer. But this surely needs some further explanation...

2.11.1. Printer setup

To address the temporary file to printer, **tbas** uses a system UNIX command, called 'lpr'. As long as you can print from console through lpr, **tbas** is supposed to work. I chose lpr because it's ubiquitous in UNIX and its family (for Windows® read further).

But if your system uses a different command to send files to printer, say 'myprinter' (a fake name), do the following:

- open file tbas.h with an editor

- search for the definition of ROUTESTRING

- change the text in double quotes from "lpr" to "myprinter"

- save and recompile.

Then test the following examples, and if they should not work, let me know. I'll try a different approach.

2.11.2. Routing with the default file name

The first case is the simpler. You don't choose any option, just use what's built-in. OPTION ROUTING ON is not really needed, because it's the default state for the OPTION ROUTING statement.

E.g.

```
 OPTION ROUTING ON

 REM start the process of queuing to printer
 QUEUE

 REM print lines (to temporary file)
 PRINT "First line"
 PRINT "Second line"

 REM close queue: previous lines are sent to printer
 ROUTE
```

The temporary file is sent to printer and removed.

If the LPRINT (aka LINE PRINT) statement is used without any QUEUE statement, it works in the very same way of the QUEUE without argument, because it sets a temporary system file as output destination, that will be deleted in case of ROUTE or program ending (see LPRINT for details).

2.11.3. Off-routing with the default file name

The second case requires the disabling of the router.

E.g.

```
 OPTION ROUTING OFF

 REM start the process of queuing to printer
 QUEUE

 REM print lines (to temporary file)
 PRINT "First line"
 PRINT "Second line"

 REM close queue: previous lines are not sent to printer
 REM and are saved to file
 ROUTE
```

The queue closing does not involve the printer, because the routing was set off. Rather, the output is saved onto a file with the name **tbas**_tempN.txt, where N is a progressive number.

The first time the program is executed, you will find file **tbas**_temp1.txt in current directory, containing the printer output. It's a textual file, that can be safely added to an existing word processor file (troff, Lyx, Word, OpenOffice Writer...). If you run the program a second time, **tbas** recognizes the existence off tabs_temp1.txt and so advances the counter once more, to create file **tbas**_temp2.txt. And so forth. There is no risk to overwrite existing print files.

2.11.4. Routing with a user file name

The third case, restoring OPTION ROUTING ON, requires the user to provide a file name for the temporary file. This is the only mode that provides both the printing to printer **and** the saving on the output file.

E.g.

```
OPTION ROUTING ON

REM start the process of queuing to printer
QUEUE "ggg.txt"

REM print lines (to temporary file)
PRINT "line #1"
PRINT "line #2"

REM close queue: previous lines are sent to printer
REM but the file won't be deleted for further usage
ROUTE
```

After the printing is done, you can find the file "ggg.txt" in current directory, containing the very same lines of the printed output.

2.11.5. Off-routing with a user file name

The fourth case requires both giving a file name for the printer queue and disabling the routing to printer.

E.g.

```
OPTION ROUTING OFF

REM start the process of queuing to printer
QUEUE "hhh.txt"

REM print lines (to temporary file)
PRINT "line #1"
PRINT "line #2"

REM close queue: previous lines are not sent to printer
REM but the file won't be deleted for further usage
ROUTE
```

After the printing is done, you can find the file "hhh.txt" in current directory, containing the printed lines.

2.12. The pseudo-random number generator

In older times the official tests for random numbers generation were beyond conception, and so the random generator of those BASICs was not certainly officially tested, but nonetheless the algorithm was good enough for any BASIC user.

The algorithm behind the pseudo-random numbers generation in **tbas** is simple and naive (you can check its sources for the technical matters), but has shown, in spite of this, a very good behaviour. I have found that this algorithm follows a two-sequences range: the first is a variable length sequence that depends on starting values and that is unique, the second is a recurring cycle with a periodicity of 8532532 events. Since the initial sequence has shown a length variability from about 100000 (1E5) to 7000000 (7E6) items, there may be 8.6 million to over 17 million useful pseudo-random numbers, before the cycle repeats!

This said, the algorithm is not to be used for serious cryptography or important cyphering/deciphering programs, because it does not guarantee unquestionable results for a huge amount of pseudo-random numbers. If you need such a generator, build your own: it’s a very interesting intellectual challenge, believe me!

2.13. Limits

tbas has its own default setup that the programmer should know, in order to design her/his own programs and interpret correctly the results:

- arrays use 0 as the lowest index; MAT statements, however, use 1 as the lowest index; this is done for historical reasons, and also because indexes in matrix theory books start from 1 (a11, a12, etc...); in any case, if you refuse this point of view, you can build your own printing procedure, using a SUB that you can size to your needs. You can enable arrays to use 1 as the lowest index through OPTION BASE 1; this operation does not affect matrices.
- the greatest positive interpretable number is INF=5.7896044482E76 (2^255)
- the lowest negative interpretable number is MINF=-5.7896044482E76 (-2^255)
- the smallest number, in absolute value, is EPS=8.6361685763E-78 (2^-256); any number in absolute value smaller than this is interpreted as zero.
- the highest magnitude in printing the integer part of a number with USING is E14
- the smallest magnitude in printing the decimal part of a number with USING is E-15
- the string length is up to 255 characters wide (plus the terminator)
- a file line is up to 255 characters wide (plus the terminator)
- file streams numbers can be any in the range 1 to 9

- EXP() works with any real up to 176.752531, which is the upper boundary; any greater number cause overflowing, and return INF.
- TEN(), similarly, works with any real up to 76.76264889, which is the upper boundary; any greater number cause overflowing, and return INF.

2.14. Piping tbas

If some parameter is passed to **tbas** through a pipe, like in this example:

```
$ echo some text | tbas
```

the global effect is to pass the text "some text" to the BASIC parser; the passed text is interpreted as a linear BASIC statement (that is, a statement that exhausts its duty on a single line and that does not require any preprocessing). The information passed through a pipe disables option -i, and the interactive session does not start.

The total effect can be overridden by the usage of option -p, which has the same usage:

```
$ tbas -p "some text"
```

3. The interactive session

If option –i is used, **tbas** will start an interactive session; if the file name is given, it will be loaded in memory[3], otherwise you will start a bare session. During a session, **tbas** changes its behaviour and becomes **a numbered-lines interpreter**. This transformation requires that the program must be ordered (according to the line numbers) before execution, and this may require some (hopefully negligible) time. The nature of **tbas** does not change: line numbers are still labels, in the sense given in the rest of this manual, but they acquire a deeper and more important role, and can be used in the sense of the original DEC Basic Shell. The only difference is that the interactive session does not make use of the special markers @ and #, which are reserved to the normal unnumbered-lines programs. This is because the interactive session expects lines begin with a number, and the @ and # markers are not numbers...

When the session starts (after a banner), the cursor is set to the first character of the input line; in practice there is no prompt in the default situation. If you prefer any prompt, use option --prompt=<str> which will set string <str> as the prompt. Bash variables are correctly interpreted. See **tbas** man page.

When the session starts, the current directory (where you launched **tbas** from) is set as current into **tbas** session, so that CAT will list programs located here. If you want to change the current directory use CD; this is a simple command that accepts a directory name and tries to open it and set it as current.

The other options remain active, during the session, but out of the session itself; for instance, the Header and Timing options –H and –t appear when RUN is invoked, but not in the session commands evaluation.

During the session, **tbas** accepts three types of strings from the user; they are all exposed in the following. The **tbas** team is working on the expansion of this set.

3.1. Shell commands

Shell commands are direct commands interpreted by **tbas**. They can have arguments, but they are not BASIC statements: they cannot be used into the sources. They are not an emulation of an existing older BASIC: they come from some interaction between the **tbas** team members, whose ideas were mixed and formed the starting base for this set.

Here they are:

<num> CCCC	store CCCC at line <num> as a program line
APROPOS CCCC	print detailed help of command CCCC
AUTO	start automated input from line n with step s
BYE	end session
CAT [ext]	list files in current directory

[3] If the program given after –i is not a numbered one, it is converted to a numbered one (implicitly) starting from 100 with step 10, as if CONVERT was used into the interactive session as the first command.

CD name	change current directory to "name"
CLEAN	erase all variables references
CONTINUE	continue the program interrupted with CTRL-C
CONVERT name	convert unnumbered file "name" and load it
DUMP	dump the ASCII codes of the program lines
FIND CCCC	print program lines which contain CCCC
GLIST	lists program and libraries zero-padded unindented
GOODBYE	end session (also GOO, hommage to DEC-BASIC)
GOTO n	equivalent to RUN n, but without clearing variables
GSAVE [name]	save current program + libraries on file "name"
HELP	print this brief help
LIBS	print loaded libraries report
LIST [n-m]	list program from n to m; either can be omitted
LOAD name	load file "name" erasing previous content
MEMORY	memory status report (it can also be typed as MEM)
MERGE name	load a program adding it to existing memory
MONITOR	temporary exit to shell. Type exit to return to BASIC
NEW	erase program memory (also SCRATCH)
PROMPT CCCC	set CCCC as the current prompt; it may be iterated
PWD	print working directory
QUIT	end session
RENAME CCCC	rename current program
RENUM n,s	renumber all lines starting from n, using step s
RESET	restore normal color in case inversion remains active
RUN [n]	execute program by clearing variables (RUNQ exits)
RUNQ [n]	execute program by clearing variables; end session
SAVE [name]	save current program on file "name"
SHELL CCCC	send CCCC to the underlying environment
SYSTEM	end session (it can also be typed as SYS)
TRACE	enable or disable debugging
TYPE name	print content of "name", located in current directory
?	fast synonym of PRINT

Shell commands are case insensitive, and arguments are maintained as-is. The input is governed by the readline library, which adds the commands history and file search, making the **tbas** interactive session a pleasure to use. Commands history does not replicate the same commands, so that you can find what you're searching for faster. The readline facility is programmed so that the word completion of the first typed characters is bound to the command set, while the arguments are file-system related, to enjoy both worlds of word completion.

The commands arguments can be avoided only if unambiguous; all string arguments (the *name* above) can be enclosed in double quotes, if desired, but this is not required (this may be a requirement for Windows® users, though).

There are also facilities for mistyping (something that happens to me very very often) and abbreviation; LSIT and LAOD are correctly interpreted as LIST and LOAD, respectively, avoiding me some nerve crisis; **L** alone is interpreted as LIST (without arguments); the same for **R** as RUN, **S** as SYSTEM and **Q** or **B** as QUIT or BYE.

The question mark ? is a synonym of the BASIC statement PRINT that can

be used from the interactive session (were you a Commodore 64 user like me?). Thus, for instance:

```
 ? A$,B
```

will print the content of A$ and B comma-spaced, as if the statement was

```
 PRINT A$,B
```

The question mark itself is not a BASIC statement.

A NOTE ABOUT APROPOS

APROPOS is a tool for getting more help for a specific shell command, whose name is passed as argument (the output is the very exact text of the manual).

Note: APROPROS cannot be used for the BASIC language statements.

Example:

```
  APROPOS PWD
```

will print the help text of the command PWD. BASIC statements or unrecognized words raise an error:

```
  APROPOS FOR
```

will print the error message:

```
  What?
```

As a special added feature, APROPOS will print the text in red and won't break words, reformatting the text (left aligned) to assure an easier reading.

A NOTE ABOUT ASPECT

ASPECT is used to change the visual aspect of the line numbers labels during LIST. It has two alternative arguments, which can be omitted:

```
  ASPECT ZERO[ES]  set the zero character as filler
  ASPECT 0         set the zero character as filler (0 = zero)
  ASPECT SPACE[S]  set the space as filler
  ASPECT           set the space as filler (default)
```

The tokens in square brackets [] are optional.

Once the command is used, the option is maintained until another ASPECT is invoked or until the end of program.

A NOTE ABOUT AUTO

AUTO is used to input program lines with automatically-filled line

numbers, with a user start value 'n' and a user step value 's', which are the arguments of the command.

AUTO can omit any or all of its arguments; the comma is mandatory to define the step:

```
AUTO            begin from number 100 with step 10
AUTO n          begin from number n with step 10
AUTO ,s         begin from number 100 with step s
AUTO n,s        begin from number n with step s
```

Any input substitutes the previous line with the same line number, if any. To stop the process, type the underscore as the first character of the line. The CTRL-C or the CTRL-D won't work in AUTO mode. AUTO does not clean the memory, to let the programmer add a specific section of the program only.

A NOTE ABOUT BYE

BYE (which may be abbreviated as B) terminates the session (see also SYSTEM and QUIT). As an homage to the DEC-BASIC environment, also GOOD-BYE and the abbreviated GOO are seen as BYE. LOGOFF also can be used, with the same effect of BYE.

A NOTE ABOUT CAT

CAT (which can be spelled also as CATALOG or DIR) is used to print the current directory content. It can omit its argument, which is a file extension; in case of no arguments, CAT will list all files (directories first), otherwise it will list files with the same extension as the argument, in case sensitive mode; to list in case insensitive mode, append * after the extension; to list directories only, use * alone:

```
CAT             list all directories and all files
CAT bas         list bas files (not BAS!)
CAT BAS         list BAS files (not bas!)
CAT TXT*        list TXT and txt files
CAT txt*        list TXT and txt files
CAT *           list directories only
```

A NOTE ABOUT CD

CD is used to set to change the current directory.

CD can omit its argument, in which case it gets back to the original directory (where tbas was started). The figures '..' and '~' (jump to parent directory in the tree and jump to the $HOME directory) are available, but only as unique strings, not as the starting characters of the destination string. Examples:

```
CD              change to the default directory
CD bas/         change to the directory bas/ in current directory
CD ..           change to parent directory in the tree
CD ~            change to the user's main directory ($HOME)
```

The jump can occur only into directories whose privileges are owned by the user.

With the '..' figure, since CD is a simple tool and to avoid errors, change directories one at a time; for instance;

```
CD ../Download
```

won't work, because '../Download' is appended to the current directory, building an erroneous target directory, and thus failing. The previous task (and all similar tasks) can be substituted by the sequence:

```
CD ..
CD Download
```

with the same results.

A NOTE ABOUT CLEAN

CLEAN clears all variables and arrays. At all effects, it's like a program has never been executed. CLEAN is called implicitly before each RUN statement.

A NOTE ABOUT CONTINUE

CONTINUE (which may be also written as CONT) can, in some measures, continue a program interrupted with CTRL-C; the continuation may not accomplish all tasks interrupted by the user (for instance, an interruption during INPUT does not continue inside the INPUT), but cycles indices, variables and the running structure is maintained and the program takes over the interruption.

A NOTE ABOUT DELETE

DELETE (which may be typed as DEL also) is used to delete lines from the program in memory.

The argument is required. The dash is mandatory to define ranges:

```
DELETE n         delete line n
DELETE -n        delete from first to n
DELETE n-        delete from n to the last
DELETE n-m       delete from n to m (included)
```

A NOTE ABOUT DUMP

DUMP is another variant of LIST which lists the program lines in their ASCII numeric codes, followed by their textual representation. It can omit any or all of its arguments; the dash is mandatory to define ranges:

```
DUMP             dump all lines
DUMP n           dump line n
DUMP -n          dump from first to n
```

```
DUMP n-          dump from n to the last
DUMP n-m         dump from n to m (included)
```

DUMP is useful in case of textual files imported from other Operating Systems, or in case you want to know, in a non-destructive way, if one blank is a space or a tab, or if there is some undesired escape sequence in the code. For instance, the line

```
 10 PRINT A
```

is dumped as:

```
--> Line 10 {1}
80 82 73 78 84 32 65
PRINT A
```

The number inside the curly brackets {} is the physical line number in the text.

This command was conceived for debugging purposes, and at first I planned to keep it for me only, but ultimately I decided that if it is useful for me, it may useful for you too.

A NOTE ABOUT FIND

FIND (which can be typed also as SCAN or WHERE) will search the argument string in all lines of the program in memory (including libraries, if RUN was invoked at least once), printing all matching lines with the label and the program line; lines are printed in order, with no indentation. No message appears in case of no match.

A NOTE ABOUT GLIST

GLIST is a variant of LIST, and has the following features:

- it uses the zero-padded line numbers format
- it lists also the hidden extern libraries (unnumbered lines)
- it lists without indentation

GLIST has a special power (confronted with LIST) because it can see under the hood, and see the complete unfolding of the current program. It has no fancy amenities like indentation. It was designed for getting the whole memory in a textual representation that was understandable in a glance, included the hidden libraries.

This command was conceived for debugging purposes, and at first I planned to keep it for me only, but ultimately I decided that if it is useful for me, it may useful for you too.

A NOTE ABOUT GOTO

GOTO is used to start a program from a specific line number, not necessary the first; as such, GOTO is used like RUN, with the difference its argument is mandatory, and follows the same directives of the one for

RUN, with the non-trivial exception that variables and arrays are maintained. This can be used to restart a program stopped with STOP, for instance (within certain limits, see also CONTINUE).

A NOTE ABOUT HELP

HELP prints a list of commands of the interactive shell and a brief explanation of their scope.

If HELP is followed by an interactive session command name, it is interpreted as a mere synonym of APROPOS.

```
HELP             show the help list
HELP <comm>      synonym of APROPOS <comm>
```

A NOTE ABOUT INDENT

INDENT is used to set the indentation level used by LIST and SAVE; the user is free to set her/his preferred value, which must be a literal number; if OFF or 0 (zero) is used, the indentation is canceled; if INDENT is invoked alone, the default value of 3 is restored:

```
INDENT n          set n as the indentation level
INDENT 0          set no indentation (zero-level)
INDENT OFF        set no indentation (zero-level)
INDENT            reset value to default 3
```

The default value may be changed in the customizable section of tbas.h.

A NOTE ABOUT LIBS

The LIBRARY BASIC statement, when invoked from within a program during a console execution, simply adds the SUBs/DEFs of the library to the current execution program. The library is not removed, after the program's end, and its functions can be used from the command line, but they are not part of the program (LIST won't show them).

This feature is useful to run more instances of the library-loader program, without multiple additions of the library itself to the current listing (which instead, if saved, will save only the program and not the libraries).

Whenever you want to know which libraries are loaded and active, you can use LIBS, which will list all loaded libraries (SUBs and DEFs) and their count: the output is the same of the debugger, without color inversion.

A NOTE ABOUT LIST

LIST is used to print the program in memory to screen.

LIST can omit any or all of its arguments; the dash is mandatory to define ranges:

```
LIST             list all lines
```

```
LIST n          list line n
LIST -n         list from first to n
LIST n-         list from n to the last
LIST n-m        list from n to m (included)
```

Line numbers are printed right aligned and space-padded in a 5-digits format. This assures that the contents of lines like 10, 6775 and 20000 are aligned. If you prefer a full zero-padded picture, use option --full at start or use the command ASPECT (see). Program lines are indented according to the usual laws (tabulation is set to 3 characters, which is a common BASIC set).

The 0 (zero) postfix instructs LIST to print without indentation; the ranges are the same as those of LIST, e.g.:

```
LIST0           list all lines
LIST0 n         list line n
LIST0 -n        list from first to n
LIST0 n-        list from n to the last
LIST0 n-m       list from n to m (included)
```

LIST does not list libraries loaded through LIBRARY statements. See GLIST.

A NOTE ABOUT LOAD

LOAD (which may be typed also as OLD) is used to load a BASIC program in memory, whose name (possibly with path) is passed as argument. The program memory is cleaned before the load.

A NOTE ABOUT MEMORY

MEMORY (which may be typed also as MEM) prints a memory report, useful to test if the program grows as desired and to check the available space, or to see the variables state.

A NOTE ABOUT MERGE

MERGE is used to load a BASIC program in memory, in the very same vein of LOAD, adding the program to the one already in memory; if some lines of the MERGEd file have the same line numbers of some already in memory, these will be overwritten (in brief: MERGE overwrites). And, as you may have already imagined, MERGE works just as LOAD in case of empty memory.

MERGE erases the libraries both from program and from memory, to avoid overlapping. To restore the libraries in memory, RUN once and they will be restored.

A NOTE ABOUT MONITOR

MONITOR starts a UNIX shell session. The user's preferred shell must be specified in 'tbas.h' (bash is set as default). Once in the shell, you can perform all the stuff you need, and when you type 'exit' you will be redirected back to the environment of tbas.

It is a short-hand command for

```
SHELL bash
```

or whatever shell you want to start.

A NOTE ABOUT NEW

NEW (which can be written also as SCRATCH) performs a complete program erasure. Variables are maintained and a new program can now be entered or loaded, letting great console interaction with the existing variables if needed.

A NOTE ABOUT PROMPT

PROMPT can be used to change the session prompt on-the-fly. Use any string, enclosed in quotes or not.

```
PROMPT >
```

will set the 'greater than' symbol as the prompt.

A NOTE ABOUT PWD

PWD (acronym for Print Working Directory) prints the name of current directory on the screen. It's a UNIX heritage.

A NOTE ABOUT QUIT

QUIT (which may also be abbreviated as Q) terminates the session (see also BYE and SYSTEM).

A NOTE ABOUT RELEASE

RELEASE erases the inner file name reference of the program in memory. It's main usage is to preserve the program file against accidental SAVE commands that could overwrite the source. It has no arguments:

```
RELEASE
```

Using SAVE at this point causes an error, because the inner file name reference is empty, but you know your file is safe. See also SAVE and RENAME.

A NOTE ABOUT RENAME

RENAME changes the system default file name with the one specified as argument:

```
RENAME newname
```

SAVE can be used effectively now, because the file name is stored internally. An error message is printed in case no arguments are given. See also SAVE and RELEASE.

A NOTE ABOUT RENUM

RENUM (which can also be spelled as RENUMBER, RES or RESEQUENCE) renumbers the program lines, using the same routine of CONVERT, updating inner references for jumps (to labels or line numbers) and colon labels.

RENUM can omit any or all of its arguments, the first being the new number that the first line will assume, and the second the step distancing the line numbers; default values are 100 and 10 respectively:

```
RENUM             renumber program from 100 with step 10
RENUM 500         renumber program from 500 with step 10
RENUM ,40         renumber program from 100 with step 40
RENUM 200,20      renumber program from 200 with step 20
```

A NOTE ABOUT RESET

The RESET command (which differs from the RESET statement) restores the default color and state of the screen; if you forget to restore some color, and the session becomes unresponsive (because you cannot see what you type), hit the Enter key and type RESET.

A NOTE ABOUT RUN

RUN executes the program in memory or in the file system, clearing the variables and arrays content and references:

```
RUN            run the program in memory
RUN n          run the program in memory from line with label n
RUN CCCC       load CCCC in memory and run
RUN CCCC n     load CCCC in memory and run it from line with label n
```

RUN can omit its arguments, and the running of the current program will start from the line with the lowest number or, if the numeric argument is given, the start will be from the next line number greater than or equal to the argument itself. If the string CCCC is used (with or without double quotes), this is equivalent to

```
LOAD CCCC
RUN [n]
```

If after RUN and the standard arguments the '--' (dash-dash) token is written, what follows is gathered into COMMAND$; the string after '--' must not be enclosed in quotes and quotes work according to BASIC (because the input is not driven by bash, but by tbas itself).

RUNQ is a version of RUN (with the very same options) that quits after RUN. It's a short-hand for RUN + BYE (if the running is the last action required, and for example, no SAVE is necessary).

RUN can be typed also as START (any parameter is ignored).

A NOTE ABOUT SAVE

SAVE is used to store to disc the program currently in memory; it will append the extension .bas if no BASIC extension (.bas or .BAS) is provided. Syntax:

```
  SAVE                    save current program with default system name
  SAVE newname            save current program with a new default name
  SAVE  WITH  LIBS          save  current  program  along  with  extern
libraries
  SAVE WITH LIBS newname  save current program with a new default name
                          along with extern libraries
  GSAVE                    save  current  program  along  with  extern
libraries
  GSAVE newname           save current program with a new default name
                          along with extern libraries
```

SAVE can omit its argument (a file name), but only if the name was previously stored someway; for instance, if a file was invoked as a tbas argument, like in

```
  tbas -i asc.bas
```

the SAVE command, used without arguments, will act on the argument file (asc.bas in the example). RENAME can be useful here.

The argument of SAVE will set the file name:

```
  SAVE newname
```

The argument from now on now may be omitted, because the file name is saved internally at the first time.

SAVE won't save the loaded libraries (those that are loaded through LIBRARY), unless the labels WITH LIBS are used:

```
  SAVE WITH LIBS newname
  GSAVE newname
```

these versions of SAVE will save in the source also the loaded extern libraries; to avoid confusion with a possible next LOAD + RUN, all LIBRARY statements are commented out. GSAVE stands for Global-SAVE and is a synonym of SAVE WITH LIBS.

Note: the first BASIC environments (e.g. Dartmouth, DEC) used to save programs using a 5-digit format, that is setting the zero padded format. This ensured a better lines reading (spaces could - not in tbas! - compromise the correct loading). As a homage to that philosophy and also to ensure that programs saved by tbas are correctly loaded by other BASICs, SAVE uses the 5-digit zero-padded format, with the proper automatic indentation. Programs, even the simplest, take a professional aspect, believe me.

A NOTE ABOUT SHELL

SHELL parses all the text that follows (it may be included in double

quotes or in single quotes, if needed) and passes it to the underlying shell (bash, DOS Prompt, etc.); as for COMMAND (of the BASIC language), I must advise you that this is a potentially dangerous command. So use it with care.

Double quotes are removed, before passing the string to bash, so that variables (like $HOME) are passed and recognized by bash. Single quotes are not removed, to let the string being interpreted as conventionally established for single quotes. Example:

```
SHELL ls
```

will list the directory content using bash!

A NOTE ABOUT SYSTEM

SYSTEM (which may also be written as SYS and abbreviated as S) terminates the session (see also BYE and QUIT). SYSTEM may also be typed as MONITOR, as a homage to the DEC-BASIC environment.

An optional string argument (without quotes, in the style of a shell command, with optional arguments), if given, is passed as a shell command to the underlying Operating System after tbas has ended, and suppressing the job termination lines. This is another difference with BYE or QUIT.

Incidentally, if you don't want the job termination lines to appear, type SYSTEM followed by a white space. The job lines won't be printed, but no command will be sent to the underlying shell.

A NOTE ABOUT TRACE

TRACE, used as general command, is a mere duplicate of the OPTION DEBUG ON/OFF BASIC command, which can be invoked from console; it has been added because the names TRACE/UNTRACE are historically attested in other BASIC environments:

```
TRACE             enable debugging
TRACE ON          enable debugging
TRACE OFF         disable debugging
UNTRACE           disable debugging
```

Anyway, in tbas TRACE has a couple of interesting features.

If followed by a variable index (a number), it shows the variable type and content:

```
TRACE 1
```

tbas answers immediately (in inverted colors if not vetoed):

```
varname [#1], type numeric, content = 100.000000
```

where 'varname' is the name of the variable with index 1. Of course,

since this is a 'reading' into the existing variable database, the program must be run at least once to 'find' the variable contents. An error message appears if the variable index is out of range.

If followed by a variable name, instead, tbas store the name into an inner register; when the variable changes content (through an assignment) the variable debug line is printed during the running of program:

```
TRACE A$
```

will print (again, in inverted colors if not vetoed and every time A$=... is met):

```
A$ [#3], type string, value = |Anna|
```

No error messages appear if the variable is not found. Values (for numeric and strings variables) correspond to the last assigned value (i.e. the string is printed 'after' the assignment). Strings are enclosed into the bars || to mark possible trailing spaces.

The features of TRACE here explained cannot be used for vectors or matrices.

A NOTE ABOUT TYPE

TYPE is used to print a textual program in the file system to standard output; there is no loading, and memory is not affected. If the file name given as argument is followed by * (which, in case of double quotes, must be 'inside' the quotes), the printing is in screens, and you advance to next screen with the ENTER key (screens are sized for a 25-rows monitor); to stop the flow before the natural end, type CTRL-C and then ENTER. Examples:

```
TYPE cript.bas
```

will type the whole program.

```
TYPE cript.bas*
```

will type the program in screens.

END OF NOTES (this line is necessary for the generation of the APROPOS help, so don't remove);

3.2. Numbered lines

If the shell command begins with a digit, it's interpreted as an input line; the line number is converted and saved apart (as a label), and the rest of the string is stored in memory (and should be a proper BASIC line, but not necessarily: it may be a WRITE# file). For instance:

```
 10 PRINT "Hello world!"
```

The string is maintained as-is, so capitalized words are not converted (they are converted internally, for unambiguous execution, but this is

hidden to the user).

If the BASIC statement is null, the line is deleted; for instance

```
 10
```

will delete the previous input line (see also DELETE).

If you retype a line with an existing number, the last will substitute the previous content. For instance:

```
 10 PRNT "Hello"
 20 END
```

Oops, I made an error! Let's retype:

```
 10 PRINT "Hello"
 LIST

 10 PRINT "Hello"
 20 END
```

The introduction of a numbered line erases the loaded libreries, both from program and from memory. To restore the libraries, execute RUN at least once and they will be restored.

I hope you feel acquainted with this procedure, which is the very same of all those wonderful BASIC machines of the Eighties (Commodore 64, Sinclair ZX81, Acorn BBC and others) which literally built the Computer Science knowledge of us born in the Sixties!

3.3. BASIC commands

If the session command is not a shell command, and does not begin with a digit, **tbas** assumes it is a direct BASIC statement, and tries to execute it.

```
 PRINT "Hello"
 Hello
```

Assignments work as expected:

```
 A=24:LET J=2.3
```

This is useful for changing the variables contents before re-entering the execution with GOTO. Remember that when you type RUN (with or without argument) the variables contents are reset.

Not all BASIC language statements can be used this way; for instance, DEF, SUB, cycle and decisional structures cannot be invoked within the shell. In detail, you can execute only commands that execute in one line and don't require preprocessing (so you cannot use DATA/READ, SUB/DEF, FILES, FOR/NEXT, WHENERROR/HANDLER, WHILE/UNTIL, SELECT/CASE, IF/THEN/ELSE in the structured format).

A note about IF/THEN/ELSE: if this structure is followed by a direct BASIC statement or by a valid assignment, the statement is available also in the interactive session; for example:

```
 IF A=0 THEN B=0 ELSE PRINT C
```

works as expected.

while LIBRARY is theoretically callable from the interactive session, its usage would prevent programming (i.e. adding lines or LOADING programs), since the order of the inner instructions set would be crashed, so it has been banned. In order to use LIBRARY in the sense it was conceived for, write a program which loads the libraries you would load from console, and run it, or LOAD or write the program and use LIBRARY **after** first RUN, when the libraries will be correctly loaded and usable from the command line, but remember that any line addition (via LOAD or at this point erases the libraries from memory, and with LOAD or CONVERT from the program too. To restore the libraries, reLOAD the program and execute RUN at least once, and they will be restored in memory.

4. Errors messages and codes

To obtain the list of codes and error messages hardwired in your current version of **tbas**, it's sufficient to type:

```
$ ./tbas --errors-list
```

Remember: the %s figure means that some words (strings) are substituted during the message printing on run-time, the %d figure means that a number is substituted, the %c figure means that a character is substituted.

The error codes and strings in the list can be used in designing your error-tracking system (with WHEN ERROR, WHEN ERROR IN, ON ERROR and the like).

A foreword: in case you encounter the "? MEMORY ERROR" abrupt stop, it means the program has tried to write a memory zone out of its range. This may be due to errors in **tbas**, and you are invited to signal this to me immediately. The same if you find the segmentation error message (SIGSEGV, an error not detectable by **tbas**).

The SIGSEGV error may also be caused by a function which uses recursion, when it has overpassed the system stack depth, which is a system setting. When tbas executes, it enlarges the stack depth to the maximum available in the system, to ebable the highest recursion level, and restores the previous level when exiting. Unfortunately, this trick does not guarantee you from a last and final SIGSEGV error: if this were to happen, your computer has reached its computational power. Remember that recursive functions fill memory rather quickly; unless they are absolutely necessary, it's preferable to convert them into iterative algorithms, which can be performed safer and possibly faster.

IMPORTANT: If you signal bugs, malfunctions, recursion stops and inconsistencies, please include the version of tbas you are using, the name and version of your Operating System, the name and version of the compiler you used to compile **tbas**, the source file (or the piece of code) that caused the error, and also write down in few words what you expected and what you got instead.

Thanks in advance.

4.1. Unrecoverable Errors

Most of the errors can be recovered/intercepted by the error treatment routines, but some cannot (with ON ERROR, WHEN ERROR and the like). This is because either they are system errors or they occur in the error treatment routines themselves. They are mainly in the range 0÷24.

The error 0 has a special meaning (see CAUSE ERROR).

When a recoverable error happens in an error treatment routine, the system does not spread the error, and prints the error message before stopping regularly. This is because an error occurring in an error treatment routine cannot generate an error by itself or it may be re-caught in a (potentially) infinite loop.

5. Syntax Coloring for tbas

Here is a description of some aids in programming for vim/gvim and emacs you can use. Enjoy!

5.1. vim/gvim

In the package there's a file called **tbas**.vim, which may help vim/gvim users to write **tbas** BASIC programs with a colored-syntax. Follow these instructions in order to install it:

1. Look for the file filetype.vim in the user vim directory (Programs/vimfiles/ for Windows® and $HOME/.vim/ for UNIX), open it with a text editor and add the following lines to it (create this file empty if it doesn't exist):

```
" tbas
au BufNewFile,BufRead *.bas     setf tbas
```

2. copy **tbas**.vim in syntax/; you may create this directory if it does not exist (Programs/vimfiles/syntax for Windows® and $HOME/.vim/syntax for UNIX).

3. You're done. Now, when you open a .BAS file with vim you will see the proper statements colored by function;

4. To set the **tbas** format manually, type:

```
:set syn=tbas
```

And you're done.

5. And now? Happy programming.

5.2. emacs

Well, I'm no expert in emacs. I use gvim. Is there anyone amomg you who wants to send an emacs syntax coloring file suitable for **tbas**, with installing instructions and usage? I'd add it to the package. Thanks in advance.

6. Summing up

Sometimes, in developing software, you meet fantastic people; you already know the ones I myself have met: Ian, who proved to be a friend and a great tester, discovering tons of bugs since the decb times (with immense patience); Bruce Axtens, who discovered a lot of bugs in the first Alpha version of **tbas** and suggested many good additions (with great passion); Tom Lake, who proposed the interactive session, and tested the whole thing (with great enthusiasm; a real supporter). They are the **tbas** team. I thank them all a lot.

Thanks also to Marcus Cruz, who suggested a couple of interesting improvements.

If you want to see some samples, go to

https://trello.com/b/VQtE921A/sub-and-programs

and use the programs there. All are checked. If you cannot enter the site of Trello, ask Bruce, who is the creator of the site of tbas c/o Trello.

Two things are still to be said: the first is that I hope you like **tbas**. The second is **it's GPL: enjoy! ♥**

Table of Contents

1. Introduction 1
1.1. **tbas** compilation 2
1.1.1. Linux/UNIX 2
1.1.2. Windows and CygWin 2
1.1.3. Windows and other compilers 3
1.2. **tbas** DESIGN 3
2. **tbas** features 4
2.1. Conventions 4
2.2. Syntax rules 4
2.3. Special structures 6
2.3.1. Consecutive assignments 6
2.3.2. Multiple assignments 7
2.3.3. Left-assignments 8
2.3.4. Comments and text markers 9
2.3.5. The PRAGMA feature 10
2.3.6. Literal string format 10
2.3.7. Special markers 11
2.4. Language operators 12
2.4.1. Math operators 13
2.4.2. Relational operators 13
2.4.3. Logical operators 14
2.4.4. Bitwise operators 15
2.4.5. Operators Priority 16
2.5. Numbers picturing 16
2.6. Files types 16
2.7. BASIC Statements 19
: (colon) 19
ABORT 19
ACCEPT 19
ACCEPT# 19
APPEND# 20
BREAK 20
CALL 21
CASE 21
CASE DEFAULT 21
CASE ELSE 21
CAUSE ERROR 22
CHAIN 22
CHANGE 23
CLEAR 24
CLOSE#/CLOSE: 24
COLOR 25
CLOSE: 26
CLS 26
COMMAND 26
COMMON 27
CONST/CONSTANT 27
CONSTANT 29
CONTINUE (without arguments) 29
CONTINUE (with arguments) 29

COPY 29
DATA 31
DECLARE/DIM/DIMENSION 32
    Using vectors and matrices 33
DEF FN 34
DIM 36
DIMENSION 36
DO UNTIL..LOOP 36
DO WHILE..LOOP 36
DO..LOOP UNTIL 36
DO..LOOP WHILE 37
ELSE 37
ELSE IF..THEN 37
END/ABORT/STOP/SYSTEM 37
END HANDLER 38
END IF 38
END SELECT 38
END SUB 38
END WHEN 38
END WHILE 38
ENTER 38
ERASE 38
EXEC 38
EXIT 39
ERRPRINT 40
EXIT HANDLER|EXIT WHEN 40
EXIT SUB 40
EXIT WHEN 40
EXPORT 41
FILE#/FILE: 42
FILE: 43
FILES 43
FILLPAGE# 44
FNEND 45
FOR..TO..STEP|BY 45
FREE# 46
FREE: 46
GET# 46
GOSUB 47
GOTO/GO TO 47
    Using labels 48
HANDLER 49
IF..THEN 49
    Multiline IF structure 49
    Jump-to-label IF structure 49
    Command-mode IF structure 49
IMAGE 51
INCLUDE 51
INPUT/ENTER 52
INPUT#/INPUT:/READ#/READ: 54
INPUT: 55
JUMP 55
LET 55
LIBRARY 56

LINE INPUT 57
LINE INPUT# 57
LINE PRINT 58
LINE READ# 58
LINPUT/LINE INPUT/ACCEPT 58
LINPUT#/LINE INPUT#/LREAD#/LINE READ#/ACCEPT# 58
LOCATE 59
LPRINT/LINE PRINT 59
LREAD# 60
MARGIN/NO MARGIN 60
MAT statements 61
    Matrix input/output 62
        MAT INPUT 62
        MAT INPUT#/MAT READ# 63
        MAT PRINT/MAT WRITE 64
            Numerical arrays 64
            String arrays 66
        MAT PRINT#/MAT WRITE# 67
        MAT READ 68
    Matrix calculus 69
        MAT + (plus) 69
        MAT - (minus) 70
        MAT * (times) 70
        MAT K (constant) 72
        MAT CON/MAT UNITY 72
        MAT IDN/MAT IDENTITY 73
        MAT INV/MAT INVERT 74
        MAT NUL$/MAT NULL$ 76
        MAT TRN/MAT TRANSPOSE 77
        MAT ZER/MAT ZERO 78
        Example of a three-equations system solution 79
NEXT 82
NO DATA 82
NO MARGIN 83
NO PAGE 83
NO PAGE ALL 83
NO PAGENUM# 83
NO PAGENUM ALL 83
NO QUOTE ALL 83
NO QUOTE# 83
ON ATTENTION 83
ON ERROR 84
ON..GOSUB 84
ON..THEN/GOTO 85
OPEN 85
OPTION 87
    OPTION [ANGLE] RADIANS|DEGREES 87
    OPTION BASE 0|1 87
    OPTION CAPS CAPS ON|OFF 88
    OPTION CASE ON|OFF 88
    OPTION COMPARISON RELATIVE|ABSOLUTE 88
    OPTION DEBUG ON|OFF 88
    OPTION DIFFERENCE <n>|OFF 88
    OPTION ECHO ON|OFF/OPTION ECHO|NOECHO 88

OPTION ERRORSTREAM ON|OFF 88
OPTION EXPLICIT ON|OFF 89
OPTION FORMAT AMERICAN|EUROPEAN 89
OPTION HEADER ON|OFF 89
OPTION NULLS ON|OFF 89
OPTION PRECISION <n>|OFF 89
OPTION PROMPT ON|OFF 90
OPTION RAWPRINT ON|OFF 90
OPTION RESET ON|OFF 90
OPTION ROUTING ON|OFF 90
OPTION SPACING ON|OFF 90
OPTION TAB <n>|OFF 91
OPTION TIMING ON|OFF 91
OPTION VINTAGE 91
OPTION WARNINGS ON|OFF 91
OPTION ZERO <n>|OFF 91
OPTION ZERO <n>|OFF 92
PAGE ALL 92
PAGE#/PAGE ALL/NO PAGE#/NO PAGE ALL 92
PAGENUM ALL 93
PAGENUM#/PAGENUM ALL/NO PAGENUM#/NO PAGENUM ALL 93
PIPE 94
PREPEND# 94
PRINT/WRITE 96
PRINT#/WRITE#/PRINT:/WRITE: 97
Using sequential files 97
PRINT# and WRITE# subtleties 98
Using random access files 98
PRINT: 98
PROGRAM/TITLE 99
PUT# 99
QUEUE 100
QUOTE ALL 101
QUOTE#/QUOTE ALL/NO QUOTE#/NO QUOTE ALL 101
RANDOMIZE 102
READ 102
READ# 103
READ: 103
REDIM 103
RESET 104
RESET# 104
RESET: 104
RESTORE/RESET 104
RESTORE#/RESTORE:/RESET#/RESET: 107
RESUME 108
RETRY 109
RETURN (without argument) 109
RETURN (with argument) 109
ROUTE 109
SCRATCH#/SCRATCH:/FREE#/FREE: 110
SCRATCH: 110
SELECT 110
Value driven SELECT 110
Mute SELECT 111

SET DIGITS 112
SET 112
STOP 113
SUB 113
SWAP 115
SYSTEM 116
TITLE 116
UPDATE 116
USE 117
PRINT with USING specifics 117
Number format 118
String format 120
The DEC Protocol 120
The **tbas** protocol 121
Using PRINT USING and WRITE USING 121
WAIT 122
WEND 122
WHEN ERROR IN 122
WHEN ERROR IN with Internal Handler 122
WHEN ERROR USE with External Handler 123
WHILE 124
WRITE 124
WRITE# 124
WRITE: 124
2.8. BASIC functions 124
2.8.1. Math functions 124
<PA> 125
ABS 125
ACOS 125
ACOSEC/ACSC 125
ACOSECH/ACSCH 125
ACOSH 125
ACOT 125
ACOTH 125
ALPHA 126
ASEC 126
ASECH 126
ASIN 126
ASINH 126
ATAN/ATN 126
ATANH/ATNH 126
BIN 126
CEIL/CEILING 127
The COS family functions 127
The COSEC family functions 128
The COSECH family functions 128
The COSH family functions 129
The COT family functions 129
The COTH family functions 130
DEG/DEGREES 130
DET 131
DIV 131
DOT 132
ERFC 132

EXP 132
EVAL 133
EXPR 134
FRAC/FP 135
FREE 135
GAMMA 135
HEX 136
IDIV 136
INT/FLOOR/INTEGER 137
INV 137
IP 137
LBOUND/LDIM 138
LEN/LENGTH 138
LGAMMA 139
LOG/LOGE/LN 139
LOG10/CLOG/LGT 139
MAX 140
MIN 140
MOD 141
NUMBER 141
OCT 141
PICT 142
RAD/RADIANS 142
REMAINDER 142
REAL 143
RND 143
ROUND/ROF/FIX 143
The SEC family functions 144
The SECH family functions 144
SGN/SIGN 145
SHL 145
SHR 146
The SIN family functions 147
The SINH family functions 147
SQR/SQRT 148
SUM 148
The TAN family functions 148
The TANH family functions 149
TEN 150
UBOUND/UDIM 150
VAL 151
VMAX 152
VMIN 152
VSUM 152
2.8.2. Error functions 153
ALN 153
ASL 153
ELN 154
ERL/ESL 154
ERR/ESM 154
ERR$/ESM$ 155
NXL 156
2.8.3. String functions 156
ASC/ASCII/ORD 156

BIN$/BINOF$ 158
CAP$ 159
CHR$ 159
ENV$ 159
FORMAT$ 160
FREE$ 160
HEX$/HEXOF$ 160
INKEY$ 161
INPUT$ 162
INSTR 162
LEFT$ 163
LOWER$/LCASE$ 163
LPAD$ 164
MID$ 164
OCT$/OCTOF$ 165
PIPE$ 165
POS 166
REV$ 167
RIGHT$ 167
RPAD$ 167
RPT$/REPEAT$ 168
SEG$ 168
SPC$/SPACE$ 169
STR$/NUM$ 169
STRING$ 170
TRIM$ 170
UPPER$/UCASE$ 171
USR$ 171
2.8.4. File functions 171
EOF/END 171
EXISTS 172
LOC/LOF 173
MORE 173
2.8.5. Time functions 173
CLK 173
DATE 174
DAY 174
HOUR 174
MINUTE 175
MONTH 175
MONTH$ 175
SECOND 176
TIM 177
TIME$ 178
WEEKDAY$ 178
YEAR 179
2.9. System constants and variables 181
CLK 181
CLK$ 181
COMMAND$ 181
CR 182
DATE 182
DATE$/DAT$ 182
DAY 183

DIR$ 183
DET 183
EPS 183
FALSE 183
FREECHANNEL 183
HOUR 183
INF 183
LASTCHANNEL 183
LASTF 183
MAYBE 183
MINUTE 183
MONTH 184
NUM 184
PI 184
PROGRAM$ 184
RND 184
SECOND 184
STREAM 184
TIM 184
TRUE 184
YEAR 184
2.10. Manual Interrupt 185
2.11. QUEUE and ROUTE in details 185
2.11.1. Printer setup 185
2.11.2. Routing with the default file name 186
2.11.3. Off-routing with the default file name 186
2.11.4. Routing with a user file name 187
2.11.5. Off-routing with a user file name 187
2.12. The pseudo-random number generator 188
2.13. Limits 188
2.14. Piping **tbas** 189
3. The interactive session 190
3.1. Shell commands 190
3.2. Numbered lines 202
3.3. BASIC commands 203
4. Errors messages and codes 205
4.1. Unrecoverable Errors 205
5. Syntax Coloring for **tbas** 206
5.1. vim/gvim 206
5.2. emacs 206
6. Summing up 207