NAME

bas − BASIC interpreter

SYNOPSIS

bas [**−b**] [**−d**] [**−l** *file*] [**−r**] [**−u**] [*program* [*argument*...]]
bas [**−−backslash−colon**] [**−−do−repeat**] [**−−lp** *file*] [**−−restricted**] [**−−uppercase**] [*program* [*argument*...]]
bas −h|−−help
bas −−version

DESCRIPTION

Introduction

Bas is an interpreter for the classic dialect of the programming language BASIC, as typically found on microcomputers of the eighties. If no file is given, it reads optionally numbered lines from standard input. Non-numbered lines are executed immediatly (direct mode), numbered lines are stored in ascending order. The line number must be a positive integer. All statements are compiled before the program is run, which catches syntactic and other errors. Keywords and variable names are not case sensitive.

If a program with unnumbered lines is loaded, storing or deleting numbered lines in direct mode is not possible. You must use **renum** to renumber the program first or **edit** to modify the entire program inside an editor. If a numbered program is loaded, typing a line number with an otherwise empty line deletes that line same the **delete** does.

If a *program* is given, it is loaded, compiled and run; **bas** will exit if program execution **stop**s, **end**s or just hits the end of the program. If the first line of a program file begins with **#!**, it is ignored by the interpreter, but kept when saving a program.

The name of a loaded program is converted to an absolute pathname, so loading or saving it again uses the originally specified file even if the current directory was changed since.

Statements

Each line of a BASIC program contains one or more of the statements below. Multiple statements are grouped by a colon (**:**) in between them. Brackets (**[** and **]**) are converted to parentheses when loading a program for compatibility with dialects that use them e.g. to indicate array indices.

variable[(*argument*{**,** *argument*})] = *expression*
: Assign the value of the function to the variable.

[**call**] *function*[(*argument*{**,** *argument*})]
: Call the *function* or procedure. The ANSI syntax requires the keyword **call**, whereas other BASIC dialects don't. In **bas**, **call** is optional.

chdir *directory$*
: Change the current directory to *directory$*.

clear
: Set all numerical variables to 0 and all string variables to the empty string. All arrays lose their dimension and geometry. All files are closed.

close [**#***channel%*{**,#***channel%*}]
: Close the specified channels. If no channels are given, all channels but the standard input/output channel **#0** are closed.

cls
: Clear the screen. Not all terminals support this. The rarely used keyword **home** will be converted to **cls** when loading a program.

color [*foreground*][**,**[*background*][**,**[*border*]]]
: All parameters must be between 0 and 15. If your terminal type supports ANSI colour codes, the foreground colour will be set to the given value. The background colour can only be set to one of the lower 8 colours, selecting a higher colour silently uses the matching lower colour (e.g. red instead of light red). The border colour is always ignored. The following colours are available: black (0), blue (1), green (2), cyan (3), red (4), magenta (5), brown (6), white (7), grey (8), light blue (9), light green (10), light cyan (11), light red (12), light magenta (13), yellow (14), bright

white (15).

copy *from$* **to** *to$*
Copy a file.

data *input-data*{**,***input-data*}
Store data to be accessed by **read**. The *input-data* has the same format as data that is read by **input** statements. Data can not be stored in direct mode. This statement is ignored during execution. Abbreviation: **d.**

dec *lvalue*{**,***lvalue*}
Decrement *lvalue*.

def fn*function*[(*parameter*[**,***parameter*...])]
Define a function. Function identifiers always start with **fn**. A single line function ends and returns its value using **=***expression*. A multi line function ends and returns its value with **fnend**. Additionally, a multi line function may return a value using **fnreturn** *expression* before reaching its end. Functions can not be declared in direct mode. Note: Multi line functions are not supported by all BASIC dialects. Some dialects allow white space between the **fn** and the rest of the identifier, because they use **fn** as token to mark function identifiers for both declaration and function call. **Bas** removes that space when loading a program.

defdbl *variable*[**−***variable*]
Declare the global *variable* as real. Only unqualified variables (no type extension) can be declared. Variable ranges can only be built from single letter variables. Declaration is done at compile time.

defint *variable*[**−***variable*]
Declare the global *variable* as integer.

defstr *variable*[**−***variable*]
Declare the global *variable* as string.

def proc*function*[(*parameter*[**,***parameter*...])]
Define a procedure (function that does not return a value). Procedure identifiers always start with **proc** if defined this way. A procedure ends with **end proc**. This is the BBC BASIC syntax. Procedures can not be declared in direct mode.

delete [*from*][**−|,**][*to*]
Delete the specified line or range of lines. Unlike **list**, the lines must exist.

dim *variable* **(***dimension%*{**,***dimension%*}**)**
Dimension the array *variable*. If the array variable already exists, it must first be **erase**d. The *dimension%* specifies the upper index of the last element, not the number of elements! The lower index is specified by **option base**, it is zero by default.

display *filename$*
Display the contents of *filename$* on standard output, like *cat*(1) does.

do

exit do

loop Repeat the loop body in between **do** and **loop** until **exit do** ends looping.

do until *condition*

exit do

loop Unless the *condition* is true or **exit do** ends looping, repeat the loop body in between **do while** and **loop**. This is equivalent to **while not/wend**.

do while *condition*

exit do

loop While the *condition* is true or **exit do** ends looping, repeat the loop body in between **do while** and **loop**. This is equivalent to **while/wend**.

do

exit do

loop until *condition*
Repeat the loop body in between **do** and **loop until** until the *condition* is true or until **exit do** ends looping. This is equivalent to **repeat/until**.

edit [*line*]
Save the program to a temporary file, start an external editor on that file and load the program again from the file. If a *line* is given, the editor is told to start on that line. **Bas** knows the calling conventions for a number of common editors, but if your favourite is not among them or does not support that feature, the line will be ignored. The editor is specified by the environment variable **VISUAL**, or **EDITOR** if **VISUAL** is not set. If that is not set as well, *vi*(1) is used.

end End program execution. If the program was started from direct mode, return to direct mode, otherwise the interpreter terminates. Although allowed by BASIC itself, **bas** only allows **return** statements to be followed by a colon and a comment, because anything else would be unreachable. In interactive mode, a diagnostic message is printed to indicate the program did not just terminated after the last line.

environ *entry$*
Modify or add an environment *entry$* of the form *variable*=*value*.

erase *variable*{,*variable*}
Erase the array *variable*.

function *function*[(*parameter*[,*parameter*...])]
Define a function (ANSI BASIC style). A function ends using **end function**. The name of the function is a local variable inside the function and its value is returned as function result when program execution reaches the end of the function. Functions can not be declared in direct mode.

field [#]*channel%*,*width* **as** *lvalue$* {,*width* **as** *lvalue$*}
Allocate *width* bytes in the record buffer to the *lvalue$*. The total number of allocated bytes must not exceed the record length. The same record buffer can be allocated to different lvalues by using multiple field statements. Fielded lvalues must be set with **lset** and **rset**. Simple assignments to them will cause different storage to be allocated to them, thus not effecting the random access buffer.

files [*pattern$*]
Run **ls −l** with the file specification as argument. Only the meta characters ***?[]** are passed through.

for *lvalue* = *expression* **to** *expression* [**step** *expression*]

next *lvalue*{,*lvalue*}
The **for** loop evaluates first the **to** and the **step** expression, storing their values for all loop executions, and afterwards performs the initial variable assignment. Then it executes the statements inside the loop, if the variable is lower or equal than the **to** value (or greater than for negative steps). The **next** statement verifies if the variable already reached the **to** value. If not, it increments if by the evaluated **step** value and causes a new repetition. A missing **step** *expression* is treated as **step 1**. The **next** statement may be followed by a list of lvalues. Due to the dynamic variable geometry, the lvalues themselves are only checked for belonging to the same variable as those in **for**. If no lvalues are given, the innermost loop is terminated. For loops can be left by **exit for**. Note: That statement is not offered by all BASIC dialects. Most dialects restrict loop variables to scalar variables.

get [**#**]*channel%* [**,***record*]

Read the record buffer of *channel%* from the file it is connected to, which must be opened in **random** mode. If a *record* number is given, the record is read there instead of being read from the current record position. The first record is 1.

get [**#**]*channel%***,**[*position*]**,***lvalue*

Read the *lvalue* from the specified channel, which must be opened in **binary** mode. If a *position* is given, the data is read there instead of being read from the current position. The first position is 1.

goto *integer*

Continue execution at the specified line. If used from direct mode, the program will first be compiled. The older two word **go to** will be converted into the newer **goto**. Although allowed by BASIC itself, **bas** only allows **goto** statements to be followed by a colon and a comment, because anything else would be unreachable. This also concerns assigned **goto** statements.

gosub *integer*

Execute the subroutine at the specified line. The older two word **go sub** will be converted into the newer **gosub**. If used from direct mode, the program will first be compiled. The **return** statement returns from subroutines. Abbreviation: **r.** Although allowed by BASIC itself, **bas** only allows **return** statements to be followed by a colon and a comment, because anything else would be unreachable.

if *condition* [**then**] *statements* [**else** *statements*]

If the *condition* evaluates to a non-zero number of a non-empty string, the statements after **then** are executed. Otherwise, the statements after **else** are executed. If the **then** or **else** statements are directly followed by an integer, **bas** inserts a **goto** statement before the number and if the condition is directly followed by a **goto** statement, a **then** is inserted.

if *condition* **then**

elseif *condition* **then**

else

end if If the **then** statement is at the end of a line, it introduces a multi-line construct that ends with the **end if** statement (note the white space between **end** and **if**). This form can not be used in direct mode, where only one line can be entered at a time. Abbreviations for **then** and **else**: **th.** and **el.**

image *format*

Define a format for **print using**. Instead of using string variables, print formats can be defined this way and referred to by the line number. The *format* can be given as a string literal, which allows leading and trailing space, or without enclosing double quotes. **Bas** converts the second form to a quoted string. This statement is ignored during execution. **Note**: No two dialects share the syntax and semantics for numbered print formats, but many offer it one way or another. This statement allows you to adapt much existing code with small changes, but probably differs from most dialects in one way or another.

inc *lvalue*{**,***lvalue*}

Increment *lvalue*.

input [**#***channel%***,**][**;**][*string*[**;**|**,**|**:**]]*lvalue*{**,***lvalue*}

The **input** statement prints the optional prompt *string* and a trailing question mark (**?**). After, it reads comma separated values and assigns them to the given variables. If too few values are typed in, missing values will be requested with the prompt **??**. An empty value for a numeric variable means zero. If too much input data is given, a warning is printed. If a channel other than **#0** is specified, no question marks or error messages will be printed, instead an error is returned. A semicolon before the prompt will not move the cursor to a new line after pressing RETURN. If the prompt is followed by a comma, colon or no punctuation at all, no question mark will be printed after the prompt. **Note**: Some dialects allow a string expression instead of the *string*.

kill *filename$*

Delete a file.

[**let**] *lvalue*{**,***lvalue*} **=** *expression*

Evaluate the *expression* and assign its value to each *lvalue*, converting it, if needed. *Lvalues* are variables or array variable elements. All assignments are performed independently of each other.

line input [**#***channel%***,**][*string***;|,**]*lvalue$*

The **line input** statement prints the optional prompt *string*, reads one line of input and assigns unmodified it to the *lvalue$*. Using a comma instead of a semicolon makes no difference with this statement.

[**l**]**list** [*from*][**−|,**][*to*]

List (part of) the program text. Control structures will automatically be indented. If the parameter *from* is given, the listing starts at the given line instead of the beginning. Similarly, *to* causes the listing to end at line *to* instead of the end of the program. The given line numbers do not have to exist, there are merely a numeric range. The syntax variant using a minus sign as separator requires that the first line is given as a literal number. This statement may also be used inside programs, e.g. for **list erl**. **llist** writes the listing to the lp channel.

load [*file$*]

Load the program *file$* (direct mode only). The name may be omitted to load a program of the name used by a previous **load** or **save** statement.

local *variable*{**,***variable*}

Declare a variable local to the current function. The scope ranges from the declaration to the end of the function.

locate *line*,*column*

Locate the cursor at the given *line* and *column*. The first line and column is 1. Not all terminals support this.

lock [**#**]*channel%*

Wait for an exclusive lock on the file associated with the *channel%* to be granted.

lset *variable$***=***expression*

Store the left adjusted *expression* value in the storage currently occupied by the *variable$*. If the storage does not suffice, the *expression* value is truncated, if its capacity exceeds the length of the *expression* value, it is padded with spaces.

rset *variable$***=***expression*

Store the right adjusted *expression* value in the storage currently occupied by the *variable$*, padding with spaces from the right if the storage capacity exceeds the length of the *expression* value.

mat *variable***=***matrixVariable*

Matrix variables are one or two-dimensional array variables, but the elements at index 0 in each dimension are unused. The *variable* does not have to be dimensioned. Note: If it is, some BASIC dialects require that its number of elements must be equal or greater than that of the *matrixVariable*, which is valid for all matrix assignments. The *variable* will be (re)dimensioned to the geometry of the *matrixVariable* and all elements (starting at index 1, not 0) of the *matrixVariable* will be copied to *variable*.

mat *variable***=***matrixVariable*[**+|−|*** *matrixVariable*]

The *variable* will be (re)dimensioned as for matrix assignments and the matrix sum (difference, product) will be assigned to it. Note: Some BASIC dialects require that result matrix *variable* must not be a factor of the product, e.g. **a=a*a** is illegal in those dialects.

mat *variable***=(***factor***)*** *matrixVariable*

Assign the scalar product of the *factor* and the *matrixVariable* to *variable*.

mat *variable*=**con**[(*rows*[,*columns*])]
: Assign a matrix whose elements are all **1** to *variable*. If dimensions are specified, the matrix *variable* will be (re)dimensioned. A missing number of *columns* (re)dimensions the variable with 2 columns, including column 0.

mat *variable*=**idn**[(*rows*[,*columns*])]
: Assign a matrix whose diagonal elements are **1** and remaining elements are **0** to *variable*. Some dialects can only generate square matrices and use only one argument to specify both rows and columns.

mat *variable*=**inv**(*matrixVariable*)
: Assign the inverse of the *matrixVariable* to *variable*, (re)dimensioning it if needed. Only two-dimensional square matrixes can be inverted.

mat *variable*=**trn**(*matrixVariable*)
: Assign the transposed elements of *matrixVariable* to *variable*, (re)dimensioning it if needed. Note: Some BASIC dialects require that *variable* and *matrixVariable* are different. Only two-dimensional matrixes can be transposed.

mat *variable*=**zer**[(*rows*[,*columns*])]
: Assign a matrix whose elements are all **0** to *variable*.

mat input [**#***channel%*,]*variable*[(*rows*[,*columns*])]{, *variable*[(*rows*[,*columns*])]}
: This statement reads all elements of a matrix *variable* without row or column 0 from the specified channel (or standard input, if no channel is given). For two-dimensional matrices, the elements are read in row order. Elements are separated with a comma. If the channel is **#0**, the prompt **?** is printed until all elements are read.

mat print [**#***channel%*[,]][**using** *format*;]*matrixVariable*{;|, *matrixVariable*}[;|,]
: Print the given *matrixVariable*, optionally using the **using** format string or line (see **print using** below) for formatting the matrix elements. If no format string is used, a following comma prints the elements in zoned format (default), whereas a semicolon prints them without extra space between them. The output starts on a new line, unless the output position is already at the beginning of a new line. A blank line is printed between matrix variables.

mat read *variable*[(*rows*[,*columns*])]{, *variable*[(*rows*[,*columns*])]}
: Read constants from **data** statemets and assign them to the elements of the matrix *variable*.

mat redim *variable*(*rows*[,*columns*]){, *variable*(*rows*[,*columns*])}
: Resize a matrix *variable*. The matrix must not exist before, in which case it will be created. If it does exist, it must be of the same dimension, but it may be smaller or larger. Truncated elements will be permanently lost, new elements will be set to **0** for numeric and **""** for string variables. Identical positions in the old and the new matrix keep their value. Note: Some BASIC dialects require that the matrix variable must exist before, some only forbid to grow matrix variables beyond their original dimension and some keep the values at the same storage position, which appears as if they got shuffled around when changing the size and as if previously lost values reappear.

mat write [**#***channel%*[,]]*matrixVariable*{;|, *matrixVariable*}[;|,]
: Write the values of the given *matrixVariable* to the specified channel or to standard output if no channel is given. Different values are separated by commas and a newline is written at the end of a line. Strings will be written enclosed in double quotes and positive numbers are not written with a heading blank.

**mid$**(*lvalue$*,*position%*[,*length%*])=*value$*
: Replace the characters starting at the given *position%* inside *lvalue$* with the characters from *value$*. An optional *length%* limits how many characters of *lvalue$* are replaced. The replacement will not go beyond the length of *lvalue$*. The string length always stays unchanged and excess characters will not be copied. Trying to copy more characters than given is ignored and does not cause an error. Note: Not all BASIC dialects support this statement.

mkdir *directory$*
Create a *directory$*.

name *oldname$* **as** *newname$*
Rename the file *oldname$* to *newname$*.

new Erase the program to write a new one (direct mode only). All files are closed and all variables removed.

on *choice%* **goto** *line*{**,***line*}
If the integral value of *choice* is 1, execution continues at the first specified *line*, if 2, on the second, etc. If the value falls outside the range for which lines are given, execution continues at the next statement.

on *choice%* **gosub** *line*{**,***line*}
This is similar to **on goto**, but a **gosub** is executed instead of the **goto**.

on error goto 0
If executed in the context of an exception handler, re-throw the last exception that happened. Otherwise disable exception handling.

on error *statements*
Register the *statements* as exception handler to catch any thrown exceptions. Exception handlers inside procedures are always local: If a procedure aborts by an unhandled exception, that exception may be caught by its caller. If the *statements* do not abort the program or jump elsewhere, execution continues at the next line. Note: This more general form differs from traditional interpreters that require **on error goto**.

on error off
Disable exception handling.

open *mode$***,**[**#**]*channel%***,***file$*[**,***length*]
Open the *file$* through the *channel%*. The mode must be **"i"** for input, **"o"** for output, **"a"** for appending output or **"r"** for random access. Opening the file for random access requires the record *length* to be specified. This syntax is used by MBASIC and some other interpreters.

open *file$* [**for input**|**output**|**append**|**random**|**binary**] [**access read**|**write**|**read write**] [**shared**|**lock read**|**lock write**] **as file** [**#**]*channel%* [**len=***length%*]
Open the *file$* through the *channel%*. Files opened in **input** mode must already exist, whereas the other methods create them as needed. If the file is opened for random access and no record *length* is specified, a record length of 1 is used. This is the ANSI BASIC syntax found in more modern programs. The **binary** mode is similar to **random** mode, but there is no fixed record length: Data is read and written directly using **get** and **put** without using **field**. If no open method is specified, the file is opened as *random*. Optionally, a file access mode can be specified.

The file locking implementations vary greatly between dialects: Some implementations offer independent locks for reading and writing, others offer shared locks (usually used for many readers) and exclusive locks (usually used for writers). Additionally, locks may be advisory/cooperative or mandatory. Most dialects use exclusive locks of highest protection by default. **Bas** implements POSIX shared/exclusive locks, which are usually advisory, and offers the following:

shared any process can read or write file

lock read
shared lock, **open** fails if file is locked exclusively

lock write
exclusive lock

default no lock is taken, same as **shared**

Programs using locks may fail if the dialect they were written for had different lock semantics!

option base *base*
: Specify the lowest array index for **dim** (zero by default). Note: Many BASIC dialects enforce the base to be 0 or 1, further they require the base to be specified only once and before creating any arrays. **Bas** allows to set an individual base for any array, but all **mat** functions require the bases of their operands to be equal and to be 0 or 1.

option run
: Ignore terminal interrupts (usually control c) and XON/XOFF flow control (control s/control q).

option stop
: Accept terminal interrupts (usually control c) to stop a program and XON/XOFF flow control (control s/control q) to stop and resume terminal output.

out *address*,*value*
: Write the value to the I/O port *address*. Direct port access is not available in the portable version.

poke *address*,*value*
: Write the value to the memory *address*. Direct memory access is not available in the portable version.

[**l**]**print** [**#***channel%*[**,**]][**using** *format***;**]{*expression*|**tab(***position***)**|**spc(***length***)**|**;**|**,**}
: Evaluate the expressions and print their values to the integral expression *channel%*. If no channel is given, the standard output channel **#0** will be used. The statement **lprint** prints to the printer channel and no other channel can be specified. The **using** format string or line may contain the following characters:

  _ Print the following character instead of interpreting it as formatting command.

  ! Print the first character of a string.

  **** Print two more characters of a string as there are spaces between the backslashes.

  & Print a string without any formatting. Note: Some BASIC dialects use **&** characters to specify the string width. A single **&** would only print the first character in those dialects. In other dialects, an ampersand represents one digit of the numeric format, padding the number with zeroes.

  + A plus at the beginning or end of a numeric format causes the sign to be printed at the beginning or the end.

  – A minus at the end of a numeric format prints a trailing minus after negative numbers and a space else.

  , A comma inside the integral part of a numeric format inserts a comma before each three-digit group of the integral part of the number. It also represents one digit in the format. Although one comma suffices, it makes formats more readable to insert a comma every three digits.

  # Each hash sign represents one digit of the numeric format. If there are fewer digits in the integral part of the value, it is preceded by spaces.

  ^ Each caret represents one digit of the exponent. At least three carets are required, because the exponent is leaded by an **E** and the epxonent sign is always printed. The number is printed in the numeric format asked for by hash signs with the exponent adjusted accordingly, e.g. printing **5** using **###.##^^^^** results in **500.00E-002**.

  ***** Like a hash sign, but the number will not be preceded by spaces, but by asterisks.

  0 Like a hash sign, but the number will not be preceded by spaces, but by zeroes.

  . The dot specifies the position of the decimal point between a pound/asterisk sign group for the integral value and an optional pound sign group for the precision of the fractional part.

$ A dollar sign prefixes the number with a dollar. Further dollar signs increase the numeric width like **#** and *****. If the dollar sign stands in front of all padding, it will precede it, otherwise it will be printed after any padding.

any other character
Any other character is printed literally and separates different numeric fields of a multi-field format.

If no format is given, positive values are printed with a heading space, negative values are printed with a heading minus, the precision is set as required and the number is followed by a space. **print** without **using** will advance to the next line if the value of the expression no longer fits into the current line.

A semicolon concatenates the output while a comma puts the values in columns. A trailing semicolon suppresses printing a trailing newline. The pseudo function **tab**, which must only be used within **print** statements, spaces to the specified print position (column) with 0 being the leftmost position. If the current print position is already beyond *value*, it does nothing. If *value* is beyond the output width, advancing the position stops there. The pseudo function **spc** is similar to **tab**, but it prints as many spaces as specified by its argument. Abbreviation: **?** or **p.**

put [**#**]*channel%* [**,***record*]
Write the record buffer of *channel%* to the file it is connected to, which must be opened in **random** mode. If a *record* number is given, the record is written there instead of being written to the current record position.

put [**#**]*channel%***,**[*position*]**,***value*
Write the *value* to the specified channel, which must be opened in **binary** mode. If a *record* number is given, the data is written there instead of being written to the current position.

randomize [*number%*]
Seed the random number generator. If no argument is given, it will be initialised with a random number.

read *lvalue*{**,***lvalue*}
Read constants from **data** statements and assign them to the *lvalue*s.

rem *arbitrary text*
This statement introduces comments.

rename *from$* **to** *to$*
Rename a file.

' *arbitrary text*
This is an alternative form of comments, which can directly follow statements without a colon. An exclamation mark instead of the quotation mark is also recognised and converted to a quotation mark.

renum [*first*[**,***increment*]]
Renumber the program. The *first* line number and the line number *increment* can be optionally given. If omitted, a value of 10 will be used for both.

repeat

until *condition*
Execute the loop body and repeat doing so if the *condition* is not zero. The loop body will be executed at least once. Abbreviation: **rep.**

restore [*line*]
Restore the data pointer to the first **data** statement for reading data again. An optional line number restores the pointer to the first **data** statement in that line. Abbreviation: **res.** Note: Some BASIC dialects allow to specify a line without a **data** statement and search beginning from that line for one. This implementation does not allow that, because it is more often an error than used as a feature.

resume *line*
End an exception handler and continue execution at the specified line. This is only needed if you intend to re-throw exceptions by on **on error goto 0**. Although allowed by BASIC itself, **bas** only allows **resume** statements to be followed by a colon and a comment, because anything else would be unreachable.

run [*line*|*file$*]
Compile the program, clear all variables, close all files and start program execution. If a file is specified, the file is loaded first and run from the beginning. If a line is specified, execution starts at the given line.

save [*file$*]
Save the program to the given *file$* (direct mode only). The name may be omitted to save the program under the name used by a previous **load** or **save** statement.

select case *selector*

case *match*{**,** *match*}

match = *expression* [**to** *expression*] | **is** *relop expression*

case else

end select
Execute the statements after the first **case** statement that matches the *selector* expression, then skip to the *end select* statement. A single *expression* matches its value, **to** matches the range between the first and the second *expression* including the limits, and **is** compares the *selector* using the relational operator with the *expression*. The *case else* branch always matches if none of the above did. If the *selector* does not match any branch, control passes to the statement following **end select**. **Note**: Some BASIC dialects treat this case as an error.

shell [*command$*]
If a *command$* is given, it is executed as child process of **bas** as bourne shell command. If used without a *command$*, the shell specified by the environment variable **SHELL** (defaults to the bourne shell if not set) is started without arguments.

sleep *pause*
The program pauses for *pause* seconds. If your system allows it, fractional seconds can be used.

stop Stop the program. Apart from printing where the program stopped, this is identical to **end**.

sub*function*[(*parameter*[**,***parameter*...])]
Define a procedure (function that does not return a value). A procedure ends with **subend**; the alternative forms **sub end** and **end sub** are converted to **subend** when loading programs. A procedure can be left by **subexit**; again the alternative forms **sub exit** and **exit sub** and converted to **subexit** when loading programs. Procedures can not be declared in direct mode. This is the ANSI syntax.

swap *lvalue1***,***lvalue2*
Swap the contents of *lvalue1* and *lvalue2*. Both must be of identical type.

system [*status%*]
Exit from **bas**. Alternatively, **bye** may be used. An optional exit status between 0 and 255 may be specified, otherwise the status 0 will be returned.

tron Enable tracing by printing the line number of each executed program line.

troff Disable program tracing.

truncate [**#**]*channel%*
Truncate the file after the current position. The file must be opened with write access.

unlock [**#**]*channel%*
Release any locks on the file associated with the *channel%*.

unnum Remove all line numbers that are not needed, which is the the opposite to **renum**. This command is specific to **bas**, although a similar command is found in Bytewater BASIC.

wait *address*[**,**]*mask*[**,**]*select*
Wait until the I/O port *address* (XORed with *select*, if specified) masked out using *mask* is not equal zero. Direct port access is not available in the portable version.

while *expression*

wend While the *expression* is not zero, the loop body, ended by **wend**, will be repeatedly executed.

width [**#***channel%*[**,**]] [[*width%*][**,***zone%*]]
Set the channel *width%*. After *width%* characters have been printed to the channel, a newline character is automatically sent to it for starting a new line. A *width%* of zero sets the channel width to infinite. Optionally, the *zone* width can be specified. Note: Some dialects use this, others use the **zone** statement.

write [**#***channel%*[**,**]]{*expression*[**,**|**;**]}
Write the values of the given expressions to the specified channel or to standard output if no channel is given. Different expressions are separated by commas and a newline is written at the end of the list. Strings will be written enclosed in double quotes and positive numbers are not written with a heading blank.

xref Output a list of all functions, global variables, **GOSUB** and **GOTO** statements and the line numbers where they are referenced.

zone [**#***channel%*[**,**]]*width%*
Set the channel zone *width%*. A comma in PRINT advances to the next print zone, similar to a tabulator.

Expressions and Functions

Expressions consist of operators or functions that act on integer, real (floating point) or string values. Beside decimal notation, integer values can be written as hexadecimal values by prefixing them with **&h** and as octal values by prefixing them with **&o**. String constants may contain paired double quotes to specify double quote characters inside strings. If the constant is terminated by the end of the line, the trailing double quote can be omitted. Numeric constants with the suffix **#** or **!** are always regarded as floating point constants, **bas** ignores the precision specification, because it does not offer different precisions. Integer constants may be followed by the suffix **%**. If an integer literal is outside the integer value range, it is treated as a floating point literal.

The table below shows the available operators with decreasing priority. The operator **=>** is converted to **>=**, **=<** is converted to **<=** and **><** is converted to **<>** when programs are loaded.

operator	meaning
^	exponentiation
−	unary negation
+	unary plus
*****	multiplication
/	floating-point division
****	integer division (equal to fix(a/b))
mod	modulo
+	addition, string concatenation
−	substraction
>	greater than
>=	greater than or equal to
=	equal to
<>	not equal to
<=	less than or equal to
<	less than
not	binary complement
and	binary and
or	binary or
xor	binary exclusive or
eqv	binary equivalent
imp	binary implication

Besides operators, various builtin functions can be used in expressions. The dollar character (**$**) denotes that the argument must be of the type string. The actual parameters of functions, both builtin and user-defined, as well as subroutines, are passed by value. Note: Modern (not old) ANSI BASIC passes actual parameters by reference. Many classic dialects don't offer call by reference and **bas** follows that direction. Arguments to functions and subroutines must be enclosed in parentheses. Note: Some dialects allow to omit them, which introduces ambiguity in some cases.

abs(*x***)** Return the absolute value of *n*.

asc(*string$***)**
Return the numeric value of the first character of the *string*.

atn(*x***)** Return the arctangent value of *x*.

bin$(*n%***)**
Return a string containing the binary conversion of *n%*.

bin$(*n%***,***digits%***)**
Return a string containing the binary conversion of *n%* with the specified number of *digits%*.

chr$(*value%***)**
Return a string of length 1 that contains the character with the given *value%*.

cint(*x***)** Return the integral value value nearest to *x* (rounded upwards).

code(*string$***)**
Return the numeric value of the first character of the *string*. This is the same as **asc(***string***)**, used by dialects that took non-ASCII systems into consideration.

command$
Return extra command line arguments after the program name, separated by spaces. The program name is not part of the return value. Note: This function is implemented for compatibility and does not deal with arguments with embedded spaces.

command$(*n%***)**
Return the *n%*th argument passed to the program, starting with 1. The first returned argument (index 0) is the program name.

commandcount
Return the number of command line arguments (the highest index that may be used as argument for **command$()**. Classic BASIC interpreters loop through arguments until **command$()** returns an empty string, which is wrong, because empty arguments are valid and may be followed by non-empty arguments.

cos(*x_rad***)**
Return the cosine value of *x_rad*.

cvd(*x$***)**
Convert a string value generated by **mkd$(***x***)** back to a floating point value. The string characters contain the bytes of a C double precision value. The string length and the byte encoding is machine dependent and not portable.

cvi(*x$***)** Convert a string value back to an integral value. The string characters contain the bytes of a signed little endian number and the sign bit of the last byte determines the sign of the resulting number.

cvs(*x$***)** Convert a string value generated by **mks$(***x***)** back to a floating point value. The string characters contain the bytes of a C single precision value. The string length and the byte encoding is machine dependent and not portable.

date$ Return the date as a 10-character string in the form *mm−dd−yyyy*.

dec$(*x***,format$)**
Convert *x* to a string according to the **print using** *format$*.

deg(*radians***)**
Convert radians to degrees.

det Return the determinant of the last matrix inverted.

edit$(***string$*,***code%***)**
Return the result of editing the *string$* as indicated by the *code%*. The following editing codes are available:

1	discard parity bit
2	discard all spaces and tabs
4	discard all carriage returns, line feeds, form feeds, deletes, escapes and nulls
8	discard leading spaces and tabs
16	convert multiple spaces and tabs to one space
32	convert lower case to upper case
64	convert left brackets to left parentheses and right brackes to right parentheses
128	discard trailing spaces and tabs
256	suppress all editing for characters within matching single or double quotes. If the matching quote is missing, suppress all editing up to the end of the string.

The codes can be added for combined editing operations.

environ$(*n%***)**
Return the *n%*th environment entry in the form *variable*=*value*, starting with 1. If *n%* is larger than the number of entries, an empty string is returned.

environ$(***variable$*)**
Return the value of the specified environment *variable$*. If there is no such variable, an empty string is returned.

eof(*channel%***)**
Return true if the end of the channel has been reached. This must be used to avoid that **input** tries to read past the end of a file.

erl Return the number of the line where the last exception was thrown.

err Return a numeric code for the last exception that was thrown. The use of this function is not portable.

exp(*x***)** Return the value of e raised to the power of *x*.

false Return 0.

find$(***pattern$*[,***nth%***])**
Return the first (or *nth%*, starting from 0, if specified) filename that matches the given pattern or the empty string, if no filename matches the pattern. This function is usually used to check for the existance of a file. The pattern may use the wildcards ***** to match an arbitrary number of characters and **?** to match a single character. Note: On some systems, the star does not match a dot inside a filename. In this implementation, the star matches everything and ***.*** only matches files with a dot in their name, not files without an extension. Some systems also encode file attributes in the eigth bit of the file name and programs strip that bit from the output of **find$**. It is recommended to use only 7-bit file names with applications using this function.

fix(*x***)** Return the integral part of a floating point value.

fp(*x***)** Return the fractional part of a floating point value.

frac(*x***)** Return the fractional part of a floating point value; same as **fp**.

freefile Return the first free file handle.

hex$(*n%***)**
Return a string containing the hexadecimal conversion of *n%*.

hex$(*n%***,***digits%***)**
Return a string containing the hexadecimal conversion of *n%* with the specified number of *digits%*.

inkey$[(*timeout%*[**,***channel*])]
Wait at most *timeout* hundredths of a second for a character to be read from the terminal. If a character could be read, return it, otherwise return the empty string. Omitting the *timeout%* will return immediatly if no character is available. Note: Some BASIC dialects wait until a character is available if no timeout is given instead of returning an empty string. Convert those programs by using **input$(1)** instead.

inp(*address***)**
Return the value of the I/O port *address*. Direct port access is not available in the portable version.

input$(*length*[**,***channel*])
Read a string of *length* characters from standard input or from the specified *channel*. The characters will not be echoed.

instr(*haystack$***,***needle$***)**
Return the position of *needle$* in *haystack$*. If *needle$* is not found, then 0 is returned.

instr(*start%***,***haystack$***,***needle$***)**
As above, but start searching at position *start%* (first position is 1).

instr(*haystack$***,***needle$***,***start%***)**
As above, but some BASIC dialects have this order of parameters.

instr(*haystack$***,***needle$***,***start%***,***length%***)**
As above, but only limit search to the first *length%* characters starting at position *start%*.

int(*x***)** Return the integral value nearest to *x* (rounded downwards).

int%(*x*)
Same as **int**, but return an integer.

ip(*x*) Return the integral part of a floating point value; same as **fix**.

**lcase$**(*string$*)
Return the string with all characters changed to lower case.

**lower$**(*string$*)
Same as **lcase**, some dialects call it this way.

**left$**(*string$*,*n%*)
Return the first *n%* characters of the *string*. If *n* is greater than the number of characters in the string, the whole string is returned.

len(*string$*)
Return the length (number of characters) of the *string*.

loc(*channel%*)
If used on random-access files, the number of the last accessed record is returned. For sequential files, the current read/write position is returned. Note: Some BASIC dialects return the record position in bytes and the read/write position in pseudo-records.

lof(*channel%*)
Return the size of the file that is attached to the channel (bytes for sequential or binary files, records for random-access files). This may not work correctly for files with sizes that exceed the range of integer numbers. Note: Some BASIC dialects return the number of bytes even for random-access files.

log(*x*) Return the natural logarithm of *x*.

log10(*x*)
Return the base-10 logarithm of *x*.

log2(*x*) Return the base-2 logarithm of *x*.

match(*needle$*,*haystack$*,*start%*)
Return the first position of *needle$* in *haystack$* that is greater than or equal *start%*. If the search fails or if *start%* exceeds the length of *haystack$*, 0 will be returned. The following characters in *needle$* have a special meaning: **!** matches any letter, **#** matches any digit, **?** matches any character and **** quotes the next character, e.g. **\?** matches a question mark.

max(*x*,*y*)
Return the maximum of *x* and *y*.

**ltrim$**(*string$*)
Return the string without leading spaces.

**mid$**(*string$*,*position%*[,*len%*])
Return the substring of *string* that begins at the given *position%* (the first character is at position 1). If *string* is too short for a substring of *len%* characters, fewer characters will be returned.

min(*x*,*y*)
Return the minimum of *x* and *y*.

mkd$(*x*)
Return a string whose characters contain the bytes of a C double precision number. The string length and byte encoding depends of the machine type and is not portable.

mks$(*x*)
Return a string whose characters contain the bytes of a C single precision number. The string length and byte encoding depends of the machine type and is not portable.

mki$(*x*)
: Return a string whose characters contain the bytes of a little endian integral value. The string length depends of the machine type, but the little endian encoding allows to store only e.g. the first two bytes if the value does not exceed the range of a signed 16 bit number.

oct$(*n%*)
: Return a string containing the octal conversion of *n%*.

peek(*address*)
: Return the value of the memory *address*. Direct memory access is not available in the portable version.

pi
: Return the constant pi.

pos(*dummy*)
: Return the current cursor position, starting with 1 as the leftmost position. The numeric *dummy* argument is needed, because old BASIC implementations did not allow functions without arguments.

pos(*haystack$*,*needle$*,*start%*)
: Same as **instr$**, some dialects use this function name.

rad(*degrees*)
: Convert degrees to radians.

**right$**(*string$*,*n%*)
: Return the last *n* characters of the *string*. If *n%* is greater than the number of characters in the string, the whole string is returned.

rnd([*x%*])
: Return a random integer number between 1 and *x%*. If *x%* is zero, one or missing, a real number between 0.0 and 1.0 is returned. If *x%* is negative, the random number generator will be seeded with **-***x%* and the functions returns a value as if **-***x%* had been passed to it.

rnd(*a%*,*b%*)
: Return a random integer number between *a%* and *b%*, inclusively. *a%* must be less than *b%*.

**rtrim$**(*string$*)
: Return the string without trailing spaces.

**seg$**(*string$*,*position%*,*len%*)
: Same as **mid$**, some dialects use this function name.

sgn(*x*)
: Return the sign *x*: −1 for negative numbers, 0 for 0 and 1 for positive numbers.

sin(*x_rad*)
: Return the sine value of *x_rad*.

space$(*length%*)
: Return a string containing *length%* spaces.

sqr(*x*)
: Return the square root of *x*.

str$(*x*)
: Return a string that contains the decimal represantation of *x*.

string$(*length*,*x*)
: Return a string of size *length* whose characters have the decimal code *x*.

string$(*length%*,*x$*)
: Return a string of size *length%* whose characters are the first character of *x$*.

strip$(*string*)
: Return the string with the eighth bit of each character cleared.

tan(*x_rad***)**
Return the tangent of *x_rad*.

time Return the current value of the centisecond counter.

time$ Return the time as a 8-character string in the form *hh−mm−ss*.

timer Return the number of seconds elapsed since midnight local time.

true Return −1.

ucase$(***string$*)**
Return the string with all characters changed to upper case.

upper$(***string$*)**
Same as **ucase$**, some dialects call it this way.

val(*string$***)**
If possible, then convert the *string$* into an integer or floating point value, ignoring trailing junk. Otherwise, return 0.0. Like anywhere else, hexadecimal values are specified by a leading **&h**.

OPTIONS

−b, **−−backslash−colon**
Convert backslashs to colons. By default, a backslash is the operator for integer division, but in some rare BASIC dialects it forms compound statements as the colon does.

−d, **−−do−repeat**
Convert **DO** to **REPEAT**. Some rare BASIC dialects do not have the ANSI **DO** loop, but **DO/UNTIL**. Although it would be possible to parse both versions, the resulting error message for faulty code would be very confusing. Programs for those dialects can be converted instead.

−l *file*, **−−lp** *file*
Write **LLIST** and **LPRINT** output to *file*. By default, that output will be written to **/dev/null**.

−r, **−−restricted**
Restricted operation which does not allow to fork a shell.

−u, **−−uppercase**
Output all tokens in uppercase. By default, they are lowercase, which is easier to read, but some BASIC dialects require uppercase. This option allows to save programs for those dialects.

−h, **−−help**
Output usage and exit.

−v, **−−version**
Display version information and exit.

AUTHOR

This program is © 1999–2022 Michael Haardt <michael@moria.de>; © 2023 LisiasT <projects@lisias.net>

THE SOFTWARE.

HISTORY

There has been a *bas*(1) command in UNIX v7, but its syntax was strongly influenced by C, unlike common classic BASIC dialects, and thus not compatible with this implementation.

NOTES

bas is meant to get old code running again. If you always missed a programming language as easy as BASIC used to be back then, then advance and learn Python instead. It brings back the same ease, fun and magic of programming, but with modern syntax and powerful libraries.

SEE ALSO

The Usenet group comp.lang.basic.misc discusses the classic BASIC dialect.