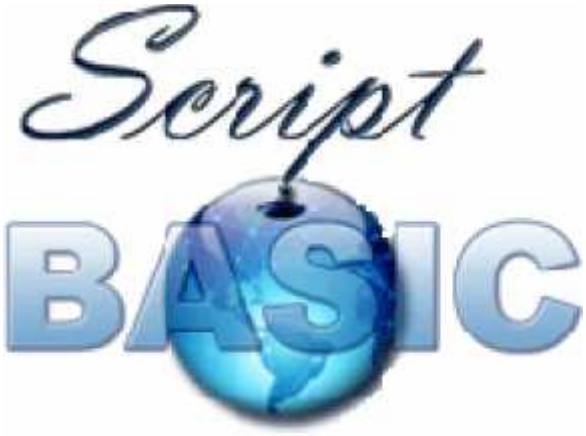

Script BASIC

Command and Function Reference

Peter Verhas

Version: November 27, 2014

This page intentionally left blank

ScriptBasic
Command and Function Reference

Contents

This page intentionally left blank

ScriptBasic
Command and Function Reference

List of Commands by Sections

array

ISARRAY()	LBOUND()	UBOUND()

error

ERROR	ON ERROR GOTO	ON ERROR RESUME
RESUME	STOP	

errord

ERRORD

file

BINMODE	CLOSE	CLOSE DIRECTORY
CRYPT()	DELETE	DELTREE
EOD()	EOF	FILEACCESSTIME()
FILECOPY	FILECREATETIME()	FILEEXISTS()
FILELEN()	FILEMODIFYTIME()	LOCK
FOWNER	FREEFILE	INPUT()
ISDIRECTORY()	ISFILE()	LINE INPUT
LOC()	LOCK	LOF()
MKDIR	NAME	NEXTFILE()
OPEN	PRINT	RESET
RESET DIRECTORY	REWIND	SEEK
SET FILE	TEXTMODE	TRUNCATE

loop

DO	DOUNTIL	DOWHILE
FOR	REPEAT	WHILE

math

ABS()
ACOS()
ASIN()
CINT()
COS()
EVEN()
EXP()
FALSE
FIX()
FRAC()
GCD()
INT()
LCM()
LOG()
LOG10()
MAXINT
MININT
ODD()
PI
POW()
RANDOMIZE
RND()
ROUND()
SIN()
SQR()
TRUE
VAL()

misc

ADDRESS()
CALL
CHDIR
CHDIR
CHDIR
COMMAND()
CRYPT()
CRYPT()
CRYPT()
CURDIR
CURDIR
CURDIR
DECLARE
COMMAND
DECLARE
SUB
END
ENVIRON()
EXIT
FUNCTION
EXIT
SUB
FUNCTION
GOSUB
GOTO
HOSTNAME()
HOSTNAME()
HOSTNAME()
ICALL
IF
LET
LETC
LETD
LETI
LETM
LETP
LETS
OPTION
OPTION()
PAUSE
POP
REF
RETURN
SLEEP()
SLEEP()
SLEEP()
STOP
SUB
SWAP
UNDEF

pattern

JOKER()
LIKE
SET [NO] JOKER
SET [NO] WILD

planned

ACOSECANT()
ACTAN()
ASECANT()
ATAN()
ATN()
BIN()
COSECANT()
COTAN()
COTAN2()
CVD()
CVI()
CVL()
CVS()
HCOS()
HCOSECANT()
HCTAN()
HSECANT()
HSIN()
HTAN()
IMAX()
IMIN()
MAX()
MIN()
MKD()
MKI()
MKL()
MKS()
SECANT()
TAN()
TAN2()

process

EXECUTE()
FORK()
KILL()
SYSTEM()
WAITPID()

string

ASC()
CHOMP()
CHR()
&
CONF()
FORMAT()
HEX()
INSTR()
INSTRREV()
ISSTRING()
JOIN()
JOKER()
LCASE()
LEFT()
LEN()
LIKE
LTRIM()
MID()
OCT()
PACK()
REPLACE()
RIGHT()
RTRIM()
SET [NO] JOKER
SET [NO] WILD
SPLIT
SPLITA
SPACE()
STR()
STRING()
SPLITAQ
TRIM()
UCASE()
STRREVERSE()
UNPACK

test

EVEN()
ISARRAY()
IsDefined()
ISEMPTY()
ISINTEGER()
ISNUMERIC()
ISREAL()
ISSTRING()
ISUNDEF()
ODD()
TYPE()

time

AddDay()
AddHour()
AddMinute()
AddMonth()
AddSecond()
AddWeek()
AddYear()
Day()
FORMATDATE()
GmTime()
GmToLocalTime()
Hour()
LocalToGmTime()
Minute()
Month()
Now()
Sec()
TimeValue()
WeekDay()
Year()
YearDay()

Commands

ABS

Returns the absolute value of the argument. If the argument is a string then it first converts it to integer or real value. The return value is integer or real value depending on the argument.

`ABS(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

ACOS

Calculates the arcus cosine of the argument, which is the inverse of the function COS(). If the argument is not between (-1.0,+1.0) the return value is `undef`.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`ACOS(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

ACOSECANT

This is a planned function to calculate the arcus cosecant of the argument.

ACTAN

This is a planned function to calculate the arcus cotangent of the argument.

ADDDAY

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the day by the second argument and returns the time value for the same hour and minute but some days later or sooner in case the second argument is negative.

This function is very simple from the arithmetic's point of view, because it simply adds 86400 times the second argument to the first argument and returns the result.

ADDHOUR

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the hours by the second argument and returns the time value for the same minute and seconds but some hours later or sooner in case the second argument is negative.

This function is very simple from the arithmetic's point of view, because it simply adds 3600 times the second argument to the first argument and returns the result.

ADDMINUTE

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the minutes by the second argument and returns the time value for the same seconds but some minutes later or sooner in case the second argument is negative.

This function is very simple from the arithmetic's point of view, because it simply adds 60 times the second argument to the first argument and returns the result.

ADDMONTH

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the month by the second argument and returns the time value for the same day, hour and minute but some months later or sooner in case the second argument is negative.

If the resulting value is on a day that does not exist on the result month then the day part of the result is decreased. For example:

```
print FormatTime("MONTH DAY, YEAR",AddMonth(TimeValue(2000,03,31),1))
```

will print

```
April 30, 2000
```

ADDRESS(myFunc())

Return the entry point of a function or subroutine. The returned value is to be used solely in a corresponding ICALL.

The returned value is an integer value that is the internal node number of the compiled code where the function starts. The different node numbers are in complex relation with each other and simple rules can not be applied. In other words playing around with the value returned by the function `ADDRESS` and then using it in an `ICALL` may result interpreter crash raising internal error.

Note that in the argument of the function `ADDRESS` the function name has to include the `()` characters. The function is NOT called by the code when the function `ADDRESS` is used. On the other hand forgetting the opening and closing parentheses will result erroneous value unusable by `ICALL`.

ADDSECOND

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the seconds by the second argument and returns the time value.

This function is the simplest from the arithmetic's point of view, because it simply adds the second argument to the first argument and returns the result.

ADDWEEK

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the week by the second argument and returns the time value for the same hour and minute but some weeks later or sooner in case the second argument is negative.

This function is very simple from the arithmetic's point of view, because it simply adds 604800 times the second argument to the first argument and returns the result.

ADDYEAR

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the year of the time value by the second argument and returns the time value for the same month, day, hour and minute but some years later or sooner in case the second argument is negative.

This is a bit more complex than just adding 365*24*60*60 to the value, because leap-years are longer and in case you add several years to the time value you should consider adding these longer years extra days. This is calculated correct in this function.

If the original time value is February 29 on a leap-year and the resulting value is in a year, which is not leap year the function will return February 28.

Note that because of this correction using the function in a loop is not the same as using it once. For example:

```
print AddYear(TimeValue(2000,02,29),4),"\n"
print AddYear(AddYear(TimeValue(2000,02,29),2),2),"\n"
```

will print two different values.

ASC(string)

Returns the ASCII code of the first character of the argument string.

ASECANT

This is a planned function to calculate the arcus secant of the argument.

ASIN

Calculates the arcus sine of the argument, which is the inverse of the function SIN(). If the argument is not between (-1.0,+1.0) the return value is `undef`.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`ASIN(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

ATAN

This is a planned function to calculate the arcus tangent of the argument.

ATN

This is a planned function to calculate the arcus tangent of the argument.

BIN

This is a planned function to convert the argument number to binary format. (aka. format as a binary number containing only 0 and 1 characters and return this string)

BINMODE [# fn] | input | output

Set an opened file handling to binary mode.

The argument is either a file number with which the file was opened or one of keywords `input` and `output`. In the latter case the standard input or output is set.

See also TEXTMODE

CALL subroutine

Use this command to call a subroutine. Subroutines can be called just writing the name of the already defined subroutine and the arguments. However in situation when the code calls a function that has not yet been defined the interpreter knows that the command is a subroutine call from the keyword `CALL`.

To be safe you can use the keyword before any subroutine call even if the subroutine is already defined.

Subroutines and functions can be called the same way. ScriptBasic does not make real distinction between subroutines and functions. However it is recommended that functions be used as functions using the return value and

code segments not returning any value are implemented and called as subroutine.

CHDIR directory

Change the current working directory (CWD). This command accepts one argument, the directory which has to be the CWD after the command is executed. If the CWD can not be changed to that directory then an error is raised.

Pay careful attention when you use this command in your code. Note that there is only one CWD for each process and not one for each thread. When an application embeds the BASIC interpreter in a multi-thread environment, like in the Eszter SB Application Engine this command may alter the CWD for all the threads.

For this reason the Eszter SB Application Engine switches off this command, raising error if ever a program executed in the engine calls this command whatever argument is given.

Thus usually BASIC programs should avoid calling this command unless the programmer is certain that the BASIC program will only be executed in a single thread environment (command line).

CHOMP()

Remove the trailing new line from the space. If the last character of the string is not new line then the original stringis returned. This function is useful to remove the trailing new line character when reading a line from a file using the command LINE INPUT

CHR(code)

Return a one character string containing a character of ASCII code `code`.

CINT

This function is the same as INT() and is present in ScriptBasic to be more compatible with other BASIC language variants.

CLOSE [#] fn

Close a previously successfully opened file. The argument of the command is the file number that was used in the command OPEN to open the file.

If the file number is not associated with a successfully opened file then error is raised.

```
REM open the file to read
open "test.bas" for input as 1
REM close the file
close#1

REM open two files for reading
open "test.bas" for input as 1
open "test.sb" for input as 2

REM close all files
close
```

You can also use the command without any argument. In this case all currently opened files and sockets are going to be closed. For those, who want to express this behaviour this command can be used with the keyword `CLOSEALL`. Note however that the keyword `CLOSEALL` is not a replacement for the keyword `CLOSE`. You can not close a single file or socket using the keyword `CLOSEALL`.

CLOSE DIRECTORY [#] dn

Close an opened directory and release all memory that was used by the file list.

See also OPEN DIRECTORY.

COMMAND()

This function returns the command line arguments of the program in a single string. This does not include the name of the interpreter and the name of the BASIC program, only the arguments that are to be passed to the BASIC program. For example the program started as

```
# scriba test_command.sb arg1 arg2  arg3
```

will see `"arg1 arg2 arg3"` string as the return value of the function `COMMAND()`.

details

Concatenate operator &

This operator concatenates two strings. The resulting string will contain the characters of the string standing on the left side of the operator followed by the characters of the string standing on the right hand side of the operator. The ScriptBasic interpreter automatically allocates the resulting string.

CONF("conf.key")

This function can be used to retrieve ScriptBasic configuration parameters. This is rarely needed by general programmers. This is needed only for scripts that maintain the ScriptBasic setup, for example install a new module copying the files to the appropriate location.

The argument `"conf.key"` should be the configuration key string. If this key is not on the top level then the levels should be separated using the dot chatacter, like `conf("preproc.internal.dbg")` to get the debugger DLL or SO file.

The return value of the function is the integer, real or string value of the configuration value. If the key is not defined or if the system manager set the key to be hidden (see later) then the function will raise an error

```
(0): error &H8:The argument passed to a module function is out of the
accepted range.
```

Some of the configuration values are not meant to be readable for the BASIC programs for security reasons. A typical example is the database connection password. The system manager can insert extra "dummy" configuration keys that will prevent the BASIC program to get the actual value of the configuration key. The extra configuration key has to have the same name as the key to be hidden with a `$` sign prepended to it.

For example the MySQL connection named `test` has the connection password under the key `mysql.connections.test.password`. If the key in the compiled configuration file `mysql.connections.test.$password` exists then the BASIC function `conf()` will result error. The value of this extra key is not taken into account.

The system manager can configure whole configuration branches to be hidden from the BASIC programs. For example the configuration key `mysql.connections.$test` defined with any value will prevent access of BASIC programs to any argument of the connection named `test`. Similarly the key `mysql.$connections` will prevent access to any configuration value of any MySQL connections if defined and finally the key `$mysql` will stop BASIC programs to discover any MySQL configuration information if defined.

The current implementation does not examine the actual value of the extra security key. However later implementations may alter the behaviour of this function based on the value of the key. To remain compatible with later versions it is recommended that the extra security key is configured to have the value `1`.

COS

Calculates the cosine of the argument.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`COS(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

COSECANT

This is a planned function to calculate the cosecant of the argument.

COTAN

This is a planned function to calculate the cotangent of the argument.

COTAN2

This is a planned function to calculate the cotangent of the ratio of the two arguments.

CRYPT(string,salt)

This function returns the encoded DES digest of the string using the salt as it is used to encrypt passwords under UNIX.

Note that only the first 8 characters of the string are taken into account.

CURDIR()

This function does not accept argument and returns the current working directory as a string.

CVD

This is a planned function to convert the argument string into a real number.

Converts a passed in string "str$" to a double-precision number. The passed string must be eight (8) bytes or longer. If less than 8 bytes long, an error is generated. If more than 8 bytes long, only the first 8 bytes are used.

CVI

This is a planned function to convert the argument string into an integer.

Converts a passed in string "str$" to an integer number. The passed string must be two (2) bytes or longer. If less than 2 bytes long, an error is generated. If more than 2 bytes long, only the first 2 bytes are used.

CVL

This is a planned function to convert the argument string into an integer.

Converts a passed in string "str$" to a long-integer number. The passed string must be four (4) bytes or longer. If less than 4 bytes long, an error is generated. If more than 4 bytes long, only the first 4 bytes are used.

CVS

This is a planned function to convert the argument string into an integer.

Converts a passed in string "str$" to a single precision number. The passed string must be four (4) bytes or longer. If less than 4 bytes long, an error is generated. If more than 4 bytes long, only the first 4 bytes are used.

DAY

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the day of the month (1 to 31) value of that time. If the argument is missing the function uses the actual local time to calculate the day of the month value. In other words it returns the day value of the actual date.

DECLARE COMMAND function ALIAS cfun LIB library

This command is used to declare a command implemented in an external ScriptBasic library. Do NOT use this command to invoke a function from an external DLL that was not specifically written for ScriptBasic. When you include module BASIC files that contain `DECLARE COMMAND` lines, you can call the functions declared this way as they were entirely written in BASIC. You use/write a `DECLARE COMMAND` command if you developed an external module for ScriptBasic programs in C.

details

DECLARE SUB function ALIAS cfun LIB library

This command is used to declare a function implemented in an external ScriptBasic library. Do NOT use this command to invoke a function from an external DLL that was not specifically written for ScriptBasic. When you include module BASIC files that contain `DECLARE SUB` lines, you can call the functions declared this way as they were entirely written in BASIC. You use/write a `DECLARE SUB` command if you developed an external module for ScriptBasic programs in C.

The difference between `DECLARE SUB` and `DECLARE COMMAND` is that the arguments passed to a function declared using `DECLARE SUB` evaluates its argument and passes the values to the C program implementing the function, whereas the functions declared using the command `DECLARE COMMAND` starts the function and evaluate the arguments one-by-one when and if the function implemented in C requests.

This difference is only important when you use expressions in the place of an argument that itself calls some other functions and has so called side effect.

Have a look at the following code:

```
import iff.bas

function side_effect()
  b = 1 + b
  side_effect = b
end function

b = 0
print iff(0,side_effect(),2)
print b
```

In the example above we use a hipotethical function implemented by a module and declared in the file `iff.bas`. This function evaluates the first argument and if that is true returns the second argument, otherwise it returns the third argument.

If the function `iff` was implemented as a command and declared accordingly using the command `DECLARE COMMAND` and if that module function evaluates only one of the second and third arguments then the global variable `b` remains unchanged.

If the function `iff` was implemented as a function and declared accordingly using the command `DECLARE SUB` and then the global variable `b` is increased.

details

DELETE file/directory_name

This command deletes a file or an **empty** directory. You can not delete a directory which contains files or subdirectories.

If the file or the directory can not be deleted an error is raised. This may happen for example if the program trying to delete the file or directory does not have enough permission.

See DELTREE for a more powerful and dangerous delete.

DELTREE file/directory_name

Delete a file or a directory. You can use this command to delete a file the same way as you do use the command DELETE. The difference between the two commands DLETE and `DELTREE` comes into place when the program deletes directories.

This command, `DELTREE` forcefully tries to delete a directory even if the directory is not empty. If the directory is not empty then the command tries to delete the files in the directory and the subdirectories recursively.

If the file or the directory cannot be deleted then the command raises error. However even in this case some of the files and subdirectories may already been deleted.

DO

This command is a looping construct that repeats commands so long as long the condition following the keyword `UNTIL` becomes `true` or the condition following the keyword `WHILE` becomes `false`.

The format of the command is

```
DO
 ...
 commands to repeat
 ...
LOOP WHILE expression
```

or

```
DO
 ...
 commands to repeat
 ...
LOOP UNTIL expression
```

The condition expression is evaluated every time after the loop commands were executed. This means that the loop body is executed at least once.

A DO/LOOP construct should be closed with a LOOP UNTIL or with a LOOP WHILE command but not with both.

The DO/LOOP UNTIL is practically equivalent to the REPEAT/UNTIL construct.

See also WHILE, DOUNTIL, DOWHILE, REPEAT, DO and FOR.

DO UNTIL condition

This command implements a looping construct that loops the code between the line DO UNTIL and LOOP util the expression following the keywords on the loop starting line becomes true.

```
DO UNTIL expression
 ...
 commands to repeat
 ...
LOOP
```

The expression is evaluated when the looping starts and each time the loop is restarted. It means that the code between the DO UNTIL and LOOP lines may be skipped totally if the expression evaluates to some TRUE value during the first evaluation before the execution starts the loop.

This command is practically equivalent to the construct

```
WHILE NOT expression
...
 commands to repeat
 ...
WEND
```

You can and you also should use the construct that creates more readable code.

See also WHILE, DOUNTIL, DOWHILE, REPEAT, DO and FOR.

DO WHILE condition

This command implements a looping construct that loops the code between the line `DO WHILE` and `LOOP` util the expression following the keywords on the loop starting line becomes `false`.

Practically this command is same as the command WHILE with a different syntax to be compatible with different BASIC implementations.

```
do while
 ...
loop
```

You can use the construct that creates more readable code.

See also WHILE, DOUNTIL, DOWHILE, REPEAT, DO and FOR.

END

End of the program. Stops program execution.

You should usually use this command to signal the end of a program. Although you can use STOP and `END` interchangeably this is the convention in BASIC programs to use the command `END` to note the end of the program and `STOP` to stop the program execution based on some extra condition inside the code.

ENVIRON("envsymbol") or ENVIRON(n)

This function returns the value of an environment variable. Environment variables are string values associated to names that are provided by the executing environment for the programs. The executing environment is usually the operating system, but it can also be the Web server in CGI programs that alters the environment variables provided by the surrounding operating system specifying extra values.

This function can be used to get the string of an environment variable in case the program knows the name of the variable or to list all the environment variables one by one.

If the environment variable name is known then the name as a string has to be passed to this function as argument. In this case the return value is the value of the environment variable. For example

```
MyPath = ENVIRON("PATH")
```

If the program wants to list all the environment variables the argument to the function `ENVIRON` should be an integer number `n`. In this case the function returns a string containing the name and the value of the `n`-th environment variable joined by a `=` sign. The numbering starts with `n=0`.

If the argument value is integer and is out of the range of the possible environment variable ordinal numbers (negative or larger or equal to the number of the available environment variables) then the function returns `undef`.

If the argument to the function is `undef` then the function also returns the `undef` value.

Note that ScriptBasic provides an easy way for the embedding applications to redefine the underlying function that returns the environment variable. Thus an embedding application can "fool" a BASIC program providing its own environment variable. For example the Eszter SB Application Engine provides an alternative environment variable reading function and thus BASIC applications can read the environment using the function `ENVIRON` as if the program was running in a real CGI environment.

details

EOD(dn)

Checks if there is still some file names in the directory opened for reading using the directory number `dn`.

See also NEXTFILE().

EOF(n)

This function accepts one parameter, an opened file number. The return value is `true` if and only if the reading has reached the end of the file.

ERROR() or ERROR n

The keyword `ERROR` can be used as a command as well as a built-in function. When used as a function it returns the error code that last happened. The error codes are defined in the header file `error.bas` that can be included with the command `import`. The error code is a vital value when an error happens and is captured by some code defined after the label referenced by the command `ON ERROR GOTO`. This helps the programmer to ensure that the error was really the one that the error handling code can handle and not something else.

If the keyword is used as a command then it has to be followed by some numeric value. The command does not ever return but generates an error of the code given by the argument.

Take care when composing the expression following the keyword `ERROR`. It may happen that the expression itself can not be evaluated due to some error conditions during the evaluation of the expression. The best practice is to use a constant expression using the symbolic constants defined in the include file `error.bas`.

Note that the codes defined in the include file are version dependant.

If an error is not captured by any `ON ERROR GOTO` specified error handler then the interpreter stops. The command line variation passes the error code to the executing environment as exit code. In other word you can exit from a BASIC program

ERROR$() or ERROR$(n)

Returns the English sentence describing the last error if the argument is not defined or returns the English sentence describing the error for the error code `n`.

If the error code `n` provided as argument is negative or is above all possible errors then the result of the function is `undef`.

EVEN

Return `true` if the argument is an even number. `EVEN(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

See also ODD().

EXECUTE("executable_program", time_out,pid_v)

This function should be used to start an external program and wait for it to finish.

The first argument of the function is the executable command line to start. The second argument is the number of seconds that the BASIC program should wait for the external program to finish. If the external program finishes during this period the function returns and the return value is the exit code of the external program. If the argument specifying how many seconds the BASIC program has to wait is `-1` then the BASIC program will wait infinitely.

If the program does not finish during the specified period then the function alters the third argument, which has to be a variable and raises error. In this case the argument `pid_v` will hold the PID of the external program. This value can be used in the error handling code to terminate the external program.

details details details

EXIT FUNCTION

This function stops the execution of a function and the execution gets back to the point from where the function was called. Executing this command has the same effect as if the execution has reached the end of a function.

EXIT SUB

This function stops the execution of a subroutine and the execution gets back to the point from where the subroutine was called. Executing this command has the same effect as if the execution has reached the end of a subroutine.

Same as EXIT FUNCTION

EXP

Calculates the x-th exponent of `e`. If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`EXP(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

FALSE

This built-in constant is implemented as an argument less function. Returns the value `false`.

FILEACCESSTIME(file_name)

Get the time the file was accessed last time.

This file time is measured in number of seconds since January 1, 1970 00:00. Note that the different file systems store the file time with different precision. Also FAT stores the file time in local time while NTFS for example stores the file time as GMT. This function returns the value rounded to whole seconds as returned by the operating system. Some of the file systems may not store all three file time types:

- the time when the file was created,
- last time the file was modified and
- last time the file was accessed

Trying to get a time not defined by the file system will result `undef`.

FILECOPY filename,filename

Copy a file. The first file is the existing one, the second is the name of the new file. If the destination file already exists then the command overwrites the file. If the destination file is to be created in a directory that does not exist yet then the directory is created automatically.

In the current version of the command you can not use wild characters to specify more than one file to copy, and you can not concatenate files using this command. You also have to specify the full file name as destination file and this is an error to specify only a directory where to copy the file.

Later versions of this command may implement these features.

If the program can not open the source file to read or the destination file can not be created then the command raises error.

FILECREATETIME(file_name)

Get the time the file was modified last time. See also the comments on the function FTACCESS. Get the time the file was modified last time. See also the comments on the function FTACCESS. Get the time the file was modified last time. See also the comments on the function FTACCESS.

FILEEXISTS(file_name)

Returns `true` if the named file exists. Returns `true` if the named file exists. Returns `true` if the named file exists.

FILELEN(file_name)

Get the length of a named file. If the length of the file can not be determined (for example the file does not exists, or the process running the code does not have permission to read the file) then the return value is `undef`.

This function can be used instead of LOC() when the file is not opened by the BASIC program.

FILEMODIFYTIME(file_name)

Get the time the file was modified last time. See also the comments on the function FTACCESS.

FIX

This function returns the integral part of the argument. The return value of the function is integer with the exception that `FIX(undef)` may return `undef`.

`FIX(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

The difference between `INT` and `FIX` is that `INT` truncates down while `FIX` truncates towards zero. The two functions are identical for positive numbers. In case of negative arguments `INT` will give a smaller number if the argument is not integer. For example:

```
int(-3.3) = -4
fix(-3.3) = -3
```

See INT().

LOCK # fn, mode

Lock a file or release a lock on a file. The `mode` parameter can be `read`, `write` or `release`.

When a file is locked to `read` no other program is allowed to write the file. This ensures that the program reading the file gets consistent data from the file. If a program locks a file to read using the lock value `read` other programs may also get the `read` lock, but no program can get the write lock. This means that any program trying to write the file and issuing the command `LOCK` with the parameter `write` will stop and wait until all read locks are released.

When a program write locks a file no other program can read the file or write the file.

Note that the different operating systems and therefore ScriptBasic running on different operating systems implement file lock in different ways. UNIX operating systems implement so called advisory locking, while Windows NT implements mandatory lock.

This means that a program under UNIX can write a file while another program has a read or write lock on the file if the other program is not good behaving and does not ask for a write lock. Therefore this command under UNIX does not guarantee that any other program is not accessing the file simultaneously.

Contrary Windows NT does lock the file in a hard way, and this means that no other process can access the file in prohibited way while the file is locked.

This different behavior usually does not make harm, but in some rare cases knowing it may help in debugging some problems. Generally you should not have a headache because of this.

You should use this command to synchronize the BASIC programs running parallel and accessing the same file.

You can also use the command `LOCK REGION` to lock a part of the file while leaving other parts of the file accessible to other programs.

If you heavily use record oriented files and file locks you may consider using some data base module to store the data in database instead of plain files.

FOR var=exp_start TO exp_stop [STEP exp_step]

Implements a FOR loop. The variable `var` gets the value of the start expression `exp_start`, and after each execution of the loop body it is incremented or decrement by the value `exp_step` until it reaches the stop value `exp_stop`.

```
FOR var= exp_start TO exp_stop [ STEP exp_step]
 ...
 commands to repeat
 ...
NEXT var
```

The `STEP` part of the command is optional. If this part is missing then the default value to increment the variable is 1.

If

- the expression `exp_start` is larger than the expression `exp_stop` and `exp_step` is positive or if
- the expression `exp_start` is smaller than the expression `exp_stop` and `exp_step` is negative

then the loop body is not executed even once and the variable retains its old value.

When the loop is executed at least once the variable gets the values one after the other and after the loop exists the loop variable holds the last value for which the loop already did not execute. Thus

```
for h= 1 to 3
next
print h
stop
```

prints `4`.

The expression `exp_start` is evaluated only once when the loop starts. The other two expressions `exp_stop` and `exp_step` are evaluated before each loop. Thus

```
j = 1
k = 10
for h= 1 to k step j
print h,"\n"
j += 1
k -= 1
next
print k," ",j,"\n"
stop
```

will print

```
1
3
6
7 4
```

To get into more details the following example loop

```
STEP_v = 5
for z= 1 to 10 step STEP_v
```

```
 print z,"\n"
 STEP_v -= 10
next z
```

executes only once. This is because the step value changes its sign during the evaluation and the new value being negative commands the loop to terminate as the loop variable altered value is smaller then the end value. In other words the comparison also depends on the actual value of the step expression.

These are not only the expressions that are evaluated before each loop, but the variable as well. If the loop variable is a simple variable then this has not too much effect. However if the loop variable is an array member then this really has to be taken into account. For example:

```
for j=1 to 9
  A[j] = 0
next

j = 1
for A[j]= 1 to 9

 for k=1 to 9
  print A[k]
 next k
 print

 j += 1
 if j > 9 then STOP

next
```

prints

```
100000000
110000000
111000000
111100000
111110000
111111000
111111100
111111110
111111111
```

so you can see that the loop takes, evaluates, compares and increments the actual array element as the variable `j` in the sample code above is incremented.

The loop variable or some other left value has to stand between the keyword `FOR` and the sign `=` on the start line of the loop but this is optional following the keyword `NEXT`. ScriptBasic optionally allow you to write the variable name after the keyword `NEXT` but the interpreter does not check if the symbol is a variable of the loop. The role of this symbol is merely documentation of the BASIC code. However, you can not write an array element following the keyword `NEXT`, only a simple variable name.

If the expression `exp_step` is zero then the loop variable is not altered and the loop is re-executed with the same loop variable value. This way you can easily get into infinite loop.

These are fine tuning details of the command `FOR` that you may need to be aware when you read some tricky code. On the other hand you should never create any code that depends on these features. The loop variable is recommended to be a simple variable and the expressions in the loop head should evaluate the same for each execution of the loop. If you need something more special that may depend on some of the features discussed above then you have to consider using some other looping construct to get more readable code.

FORK()

NOT IMPLEMENTED

This function is supposed to perform process forking just as the native UNIX function `fork` does. However this function is not implemented in ScriptBasic (yet). Until this function is implemented in ScriptBasic you can use the UX module fork function.

This function is supposed to perform process forking just as the native UNIX function `fork` does. However this function is not implemented in ScriptBasic (yet). Until this function is implemented in ScriptBasic you can use the UX module fork function.

This function is supposed to perform process forking just as the native UNIX function `fork` does. However this function is not implemented in ScriptBasic (yet). Until this function is implemented in ScriptBasic you can use the UX module fork function.

FORMAT()

The function format accepts variable number of arguments. The first argument is a format string and the rest of the arguments are used to create the result string according to the format string. This way the function `format` is like the C function `sprintf`.

The format string can contain normal characters and control substrings.

The control substring have the form `%[flags][width][.precision]type`. It follows the general `sprintf` format except that type prefixes are not required or allowed and type can only be "dioxXueEfgGsc". The `*` for width and precision is supported.

An alternate format BASIC-like for numbers has the form `%~format~` where `format` can be:

Digit or space

0 Digit or zero

^ Stores a number in exponential format. Unlike QB's USING format this is a place-holder like the #.

. The position of the decimal point.

, Separator.

- Stores minus if the number is negative.

+ Stores the sign of the number.

Acknowledgement: the function `format` was implemented by Paulo Soares

FORMATDATE

```
FormatDate("format",time)
```

Formats a time value (or date) according to the format string. The format string may contain placeholders. The first argument is the format string the second argument is the time value to convert. If the second argument is missing or it is `undef` then the local time is converted.

If the time value is presented it has to be the number of seconds elapsed since January 1, 1970. This is the usual time stamp value generally used under UNIX.

If the second argument is a negativevalue then this is treated relative to the current time point. For example

```
print FormatDate("YEAR MON DD HH:mm:ss", -60)
```

will print out the time that was one minute ago.

details

FILEOWNER(FileName)

This function returns the name of the owner of a file as a string. If the file does not exist or for some other reason the owner of the file can not be determined then the function returns `undef`.

FRAC

The function returns the fractional part of the argument. This function always returns a double except that `FRAC(undef)` may return `undef`. `FRAC(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

Negative arguments return negative value (or zero if the argument is a negative integer), positive arguments result positive values (or zero if the argument is integer).

FREEFILE()

This function returns a free file number, which is currently not associated with any opened file. If there is no such file number it returns `undef`.

The returned value can be used in a consecutive OPEN statement to specify a file number. Another way to get a free file number is to set a variable to hold the integer value zero and use the variable as file number in the statement OPEN. For more information on this method see the documentation of the statement OPEN

FUNCTION fun()

This command should be used to define a function. A function is a piece of code that can be called by the BASIC program from the main part or from a function or subroutine.

```
FUNCTION fun(a,b,c)
...
fun = returnvalue
...
END FUNCTION
```

The end of the function is defined by the line containing the keywords `END FUNCTION`.

details

GCD

This is a planned function that takes two or more integer argument and calculates the largest common divisor of them.

GMTIME

This function returns the GMT time expressed as seconds since January 1, 1970, 00:00am. The function does not accept any argument. This function is similar to the function Now() but returns the GMT time instead of the actual local time.

GMTOLOCALTIME

This function accepts one argument that has to be the number of seconds elapsed since January 1, 1970 0:00 am in GMT. The function returns the same number of seconds in local time. In other words the function converts a GMT time value to local time value.

GOSUB label

=H Gosub commands

This is the good old way implementation of the BASIC `GOSUB` command. The command `GOSUB` works similar to the command `GOTO` with the exception that the next return command will drive the interpreter to the line following the line with the `GOSUB`.

You can only call a code segment that is inside the actual code environment. In other words if the `GOSUB` is in a function or subroutine then the label referenced by the `GOSUB` should also be in the same function or subroutine. Similarly any `GOSUB` in the main code should reference a label, which is also in the main code.

To return from the code fragment called by the command `GOSUB` the command `RETURN` should be used. Note that this will not break the execution of a function or a subroutine. The execution will continue on the command line following the `GOSUB` line.

`GOSUB` commands can follow each other, ScriptBasic will build up a stack of `GOSUB` calls and will return to the appropriate command line following the matching `GOSUB` command.

When a subroutine or function contains `GOSUB` commands and the function or subroutine is finished so that one or more executed `GOSUB` command remains without executed `RETURN` then the `GOSUB`/`RATURN` stack is cleared. This is not an error.

See also RETURN.

GOTO label

Go to a label and continue program execution at that label. Labels are local within functions and subroutines. You can not jump into a subroutine or jump out of it.

Use of GOTO is usually discouraged and is against structural programming. Whenever you feel the need to use the `GOTO` statement (except `ON ERROR GOTO`) thin it twice whether there is a better solution without utilizing the statement `GOTO`.

Typical use of the `GOTO` statement to jump out of some error condition to the error handling code or jumping out of some loop on some condition.

HCOS

This is a planned function to calculate the cosinus hyperbolicus of the argument.

HCOSECANT

This is a planned function to calculate the cosecant hyperbolicus of the argument.

HCTAN

This is a planned function to calculate the cotangent hyperbolicus of the argument.

HEX(n)

Take the argument as a long value and convert it to a string that represents the value in hexadecimal form. The hexadecimal form will contain upper case alpha character if there is any alpha character in the hexadecimal representation of the number.

HOSTNAME()

This function accepts no argument and returns the host name of the machine executing the BASIC program.

This host name is the TCP/IP network host name of the machine.

HOUR

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the hour value of that time. If the argument is missing the function uses the actual local time.

HSECANT

This is a planned function to calculate the secant hyperbolicus of the argument.

HSIN

This is a planned function to calculate the sinus hyperbolicus of the argument.

HTAN

This is a planned function to calculate the tangent hyperbolicus of the argument.

ICALL n,v1,v2, ... ,vn

ICALL is implicit call. The first argument of an `ICALL` command or `ICALL` function should be the integer value returned by the function ADDRESS as the address of a user defined function.

The rest of the arguments are the arguments to be passed to the function to be called. The return value if the function `ICALL` is the value of the implicitly called function.

details

IF condition THEN

Conditional execution. There are two different ways to use this command: single line `IF` and multi line `IF`.

A single line `IF` has the form

```
IF condition THEN command
```

There is no way to specify any `ELSE` part for the command in the single line version. If you need `ELSE` command you have use multi line `IF`.

The multi line `IF` should not contain any command directly after the keyword `THEN`. It should have the format:

```
IF condition THEN
```

```
  commands
ELSE
  commands
END IF
```

The `ELSE` part of the command is optional, thus the command can have the format

```
IF condition THEN
  commands
END IF
```

as well. To be very precise the full syntax of the multi-line `IF` command is:

```
IF condition THEN
 commands
[ ELSE IF | ELSEIF | ELSIF | ELIF
  commands
  ... ]
[ ELSE
  commands ]
END IF | ENDIF
```

You can use as many `ELSE IF` branches as you like and at most one `ELSE` branch.

The keywords `ELSE IF`, `ELSEIF` and others are allowed for ease program porting from other BASIC dialect. There is no difference between the interpretation. The same is true for `END IF` in two words and written into a single keyword `ENDIF`.

IMAX

This is a planned function to select and return the index of the maximum of the arguments.

IMIN

This is a planned function to select and return the index of the minimum of the arguments.

INPUT(n,fn)

This function reads records from an opened file.

Arguments:

- `n` the first argument is the number of records to read. The size of the record in terms of bytes is defined as the `LEN` parameter when the file was opened. If this was missing the function reads `n` bytes from the file or socket.
- `fn` the second parameter is the file number associated to the opened file by the command OPEN. If this parameter is missing the function reads from the standard input.

The function tries but not necessarily reads `n` records from the file. To get the actual number of bytes (and not records!) read from the file you can use the function `LEN` on the result string.

Note that some Basic languages allow the form

```
a = INPUT(20,#1)
```

however this extra `#` is not allowed in ScriptBasic. The character `#` is an operator in ScriptBasic defined as no-operation and therefore you can use this form. On the other hand operators like `#` are reserved for the external modules and some external module may redefine this operator. Programs using such modules may end up in trouble when using the format above therefore it is recommended not to use the above format.

INSTR(base_string,search_string [,position])

This function can be used to search a sub-string in a string. The first argument is the string we are searching in. The second argument is the string that we actually want to find in the first argument. The third optional argument is the position where the search is to be started. If this argu-ment is missing the

search starts with the first character position of the string. The function returns the position where the sub-string can be found in the first string. If the searched sub-string is not found in the string then the return value is undef.

See INSTRREV()

INSTRREV(base_string,search_string [,position])

This function can be used to search a sub-string in a string in reverse order starting from the end of the string. The first argument is the string we are searching in. The second argument is the string that we actually want to find in the first argument. The third optional argument is the position where the search is to be started. If this argument is missing the search starts with the last character position of the string. The function returns the position where the sub-string can be found in the first string. If the searched sub-string is not found in the string then the return value is `undef`.

See INSTR()

INT

This function returns the integral part of the argument. `INT(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`. Other than this the function returns integer value.

The difference between `INT` and `FIX` is that `INT` truncates down while `FIX` truncates towards zero. The two functions are identical for positive numbers. In case of negative arguments `INT` will give a smaller number if the argument is not integer. For example:

```
int(-3.3) = -4
fix(-3.3) = -3
```

See FIX().

ISARRAY

This function can be used to determine whether a variable holds array value or ordinary value. If the variable passed as argument to the function is an array then the function returns `true`, otherwise the function returns `false`.

See also ISSTRING(), ISINTEGER(), ISREAL(), ISNUMERIC(), IsDefined(), ISUNDEF(), ISEMPTY(), TYPE().

ISDEFINED

This function can be used to determine whether an expression is defined or undefined (aka `undef`). If the argument is a defined value then the function returns `true`, otherwise the function returns `false`.

This function is the counter function of ISUNDEF().

See also ISARRAY(), ISSTRING(), ISINTEGER(), ISREAL(), ISNUMERIC(), ISUNDEF(), ISEMPTY(), TYPE().

ISDIRECTORY(file_name)

Returns `true` if the named file is a directory and `false` if the file is NOT a directory. Returns `true` if the named file is a directory and `false` if the file is NOT a directory. Returns `true` if the named file is a directory and `false` if the file is NOT a directory.

ISEMPTY

This function can be used to determine whether an expression holds an empty string. Because programmers tend to use the value `undef` where empty string would be more precise the function returns `true` if the argument is `undef`. Precisely:

The function returns true if the argument is `undef` or a string containing zero characters. Otherwise the function returns `false`.

See also ISARRAY(), ISSTRING(), ISINTEGER(), ISREAL(), ISNUMERIC(), IsDefined(), ISUNDEF(), TYPE().

ISINTEGER

This function can be used to determine whether an expression is integer or some other type of value. If the argument is an integer then the function returns `true`, otherwise the function returns `false`.

See also ISARRAY(), ISSTRING(), ISREAL(), ISNUMERIC(), IsDefined(), ISUNDEF(), ISEMPTY(), TYPE().

ISNUMERIC

This function can be used to determine whether an expression is numeric (real or integer) or some other type of value. If the argument is a real or an integer then the function returns `true`, otherwise the function returns `false`.

See also ISARRAY(), ISSTRING(), ISINTEGER(), ISREAL(), IsDefined(), ISUNDEF(), ISEMPTY(), TYPE().

ISREAL

This function can be used to determine whether an expression is real or some other type of value. If the argument is a real then the function returns `true`, otherwise the function returns `false`.

See also ISARRAY(), ISSTRING(), ISINTEGER(), ISNUMERIC(), IsDefined(), ISUNDEF(), ISEMPTY(), TYPE().

ISFILE(file_name)

Returns `true` if the named file is a regular file and `false` if it is a directory.
Returns `true` if the named file is a regular file and `false` if it is a directory.
Returns `true` if the named file is a regular file and `false` if it is a directory.

ISSTRING

This function can be used to determine whether an expression is string or some other type of value. If the argument is a string then the function returns `true`, otherwise the function returns `false`.

See also ISARRAY(), ISINTEGER(), ISREAL(), ISNUMERIC(), IsDefined(), ISUNDEF(), ISEMPTY(), TYPE().

ISUNDEF

This function can be used to determine whether an expression is defined or undefined (aka `undef`). If the argument is a defined value then the function returns `false`, otherwise the function returns `true`.

This function is the counter function of IsDefined().

See also ISARRAY(), ISSTRING(), ISINTEGER(), ISREAL(), ISNUMERIC(), IsDefined(), ISEMPTY(), TYPE().

JOIN(joiner,str1,str2,...)

Join the argument strings using the first argument as a joiner string. details

JOKER(n)

Return the actual match for the n-th joker character from the last executed LIKE operator. details

KILL(pid)

This function kills (terminates) a process given by the `pid` and returns true if the process was successfully killed. Otherwise it returns false.

Programs usually want to kill other processes that were started by themselves (by the program I mean) and do not stop. For example you can start an external program using the BASIC command `EXECUTE()` to run up to a certain time. If the program does not finish its work and does not stop during this time then that program that started it can assume that the external program failed and got into an infinite loop. To stop this external program the BASIC program should use the function `KILL`.

The BASIC program however can try to kill just any process that runs on the system not only those that were started by the program. It can be successful if the program has the certain permissions to kill the given process.

You can use this function along with the functions `SYSTEM()` and `EXECUTE`. You can list the processes currently running on an NT box using some of the functions of the module `NT`.

LBOUND

This function can be used to determine the lowest occupied index of an array. Note that arrays are increased in addressable indices automatically, thus it is not an error to use a lower index that the value returned by the function `LBOUND`. On the other hand all the element having index lower than the returned value are `undef`.

The argument of this function has to be an array. If the argument is an ordinary value, or a variable that is not an array the value returned by the function will be `undef`.

`LBOUND(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

See also UBOUND().

LCASE()

Lowercase a string.

LCM

This is a planned function that takes two or more integer argument and calculates the least common multiple of them.

LEFT(string,len)

Creates the left of a string. The first argument is the string. The second argument is the number of characters that should be put into the result. If this value is larger than the number of characters in the string then all the string is returned.

See also MID(), RIGHT()

details

LEN()

This function interprets its argument as a string and returns the length of the string. In ScriptBasic strings can hold any value thus the length of the string is the number of characters contained in the string containing any binary characters, even binary zero.

If the argument is not a string it is converted to string automatically and the length of the converted string is returned. The only exception is `undef` for which the result is also `undef`.

v = expression

Assign a value to a variable.

On the left side of the `=` a variable or some other ScriptBasic left value has to stand. On the right side an expression should be used. First the left value is evaluated and then the expression. Finally the left value's old value is replaced by the result of the expression.

The left value standing on the left side of the `=` can be a local or global variable, array element or associative array element.

v &= expression

Append a string to a variable.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v & expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

v /= expression

Divide a variable by an expression.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v / expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

v \= expression

Integer divide a variable by a value.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v \ expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

v -= expression

This command subtracts a value from a variable.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v - expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

v += expression

Add a value to a variable.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v + expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

v *= expression

Multiply a variable with a value.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v * expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

string LIKE pattern

Compare a string against a pattern.

```
string LIKE pattern
```

The pattern may contain joker characters and wild card characters. details

LINE INPUT

Read a line from a file or from the standard input.

The syntax of the command is

```
LINE INPUT [# i , ] variable
```

The parameter `i` is the file number used in the open statement. If this is not specified the standard input is read.

The `variable` will hold a single line from the file read containing the possible new line character terminating the line. If the last line of a file is not terminated by a new line character then the `variable` will not contain any new line character. Thus this command does return only the characters that are really in the file and does not append extra new line character at the end of the last line if that lacks it.

On the other hand you should not rely on the missing new line character from the end of the last line because it may and usually it happens to be there. Use rather the function EOF to determine if a file reading has reached the end of the file or not.

See also CHOMP()

You can also read from sockets using this command but you should be careful because data in a socket comes from programs generated on the fly. This means that the socket pipe may not contain the line terminating new line and not finished as well unlike a file. Therefore the command may start infinitely long when trying to read from a socket until the application on the other end of the line sends a new line character or closes the socket. When you read from a file this may not happen.

LOC()

Return current file pointer position of the opened file. The argument of the function is the file number that was used by the statement OPEN opening the file.

This function is the counter part of the statement SEEK that sets the file pointer position.

The file position is counted in record size. This means that the file pointer stands after the record returned by the function. This is not necessarily stands right after the record at the start of the next record actually. It may happen that the file pointer stands somewhere in the middle of the next record. Therefore the command

```
SEEK fn,LOC(fn)
```

may alter the actual file position and can be used to set the file pointer to a safe record boundary position.

If there was no record size defined when the file was opened the location is counted in bytes. In this case the returned value precisely defines where the file pointer is.

LOCATLTOGMTIME

This function accepts one argument that has to be the number of seconds elapsed since January 1, 1970 0:00 am in local time. The function returns the same number of seconds in GMT. In other words the function converts a local time value to GMT time value.

LOCK # fn, mode

Lock a file or release a lock on a file. The `mode` parameter can be `read`, `write` or `release`.

When a file is locked to `read` no other program is allowed to write the file. This ensures that the program reading the file gets consistent data from the file. If a program locks a file to read using the lock value `read` other programs may also get the `read` lock, but no program can get the write lock. This means that any program trying to write the file and issuing the command `LOCK` with the parameter `write` will stop and wait until all read locks are released.

When a program write locks a file no other program can read the file or write the file.

Note that the different operating systems and therefore ScriptBasic running on different operating systems implement file lock in different ways. UNIX operating systems implement so called advisory locking, while Windows NT implements mandatory lock.

This means that a program under UNIX can write a file while another program has a read or write lock on the file if the other program is not good behaving

and does not ask for a write lock. Therefore this command under UNIX does not guarantee that any other program is not accessing the file simultaneously.

Contrary Windows NT does lock the file in a hard way, and this means that no other process can access the file in prohibited way while the file is locked.

This different behavior usually does not make harm, but in some rare cases knowing it may help in debugging some problems. Generally you should not have a headache because of this.

You should use this command to synchronize the BASIC programs running parallel and accessing the same file.

You can also use the command `LOCK REGION` to lock a part of the file while leaving other parts of the file accessible to other programs.

If you heavily use record oriented files and file locks you may consider using some data base module to store the data in database instead of plain files.

LOCK REGION # fn FROM start TO end FOR mode

Lock a region of a file. The region starts with the record `start` and ends with the record `end` including both end positions. The length of a record in the file is given when the file is opened using the statement OPEN.

The mode can be `read`, `write` and `release`. The command works similar as whole file locking, thus it is recommended that you read the differences of the operating systems handling locking in the section of file locking for the command `LOCK`.

LOF()

This function returns the length of an opened file in number of records. The argument of the function has to be the file number that was used by the statement OPEN to open the file.

The actual number of records is calculated using the record size specified when the command OPEN was used. The returned number is the number of records that fit in the file. If the file is longer containing a fractional record at the end the fractional record is not counted.

If there was no record length specified when the file was opened the length of the file is returned in number of bytes. In this case fractional record has no meaning.

LOG

Calculates the natural log of the argument. If the argument is zero or less than zero the result is `undef`.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`LOG(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

LOG10

Calculates the log of the argument. If the argument is zero or less than zero the result is `undef`

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`LOG10(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

LTRIM()

Remove the space from the left of the string.

MAX

This is a planned function to select and return the maximum of the arguments.

MAXINT

This built-in constant is implemented as an argument less function. Returns the maximal number that can be stored as an integer value.

MID(string,start [,len])

Return a subpart of the string. The first argument is the string, the second argument is the start position. The third argument is the length of the sub-part in terms of characters. If this argument is missing then the subpart lasts to the last character of the argument `string`.

See also LEFT(), RIGHT().

details

MIN

This is a planned function to select and return the minimum of the arguments.

MININT

This built-in constant is implemented as an argument less function. Returns the minimal ("maximal negative") number that can be stored as an integer value.

MINUTE

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the minute value of that time. If the argument is missing it uses the actual local time.

MKD

This is a planned function to convert the argument real number to an 8 byte string.

Converts the double-precision number "n" into an 8-byte string so it can later be retrieved from a random-access file as a numeric value.

MKDIR directory_name

This command creates a new directory. If it is needed then the command attempts to create all directories automatically that are needed to create the final directory. For example if you want to create `public_html/cgi-bin` but the directory `public_html` does not exist then the command

```
MKDIR "public_html/cgi-bin"
```

will first create the directory `public_html` and then `cgi-bin` under that directory.

If the directory can not be created for some reason an error is raised.

This is not an error if the directory does already exist.

You need not call this function when you want to create a file using the command OPEN. The command OPEN automatically creates the needed directory when a file is opened to be written.

The created directory can be erased calling the command DELETE or calling the dangerous command DELTREE.

MKI

This is a planned function to convert the argument integer number to an 2 byte string.

Converts the integer number "n" into an 2-byte string so it can later be retrieved from a random-access file as a numeric value.

MKL

This is a planned function.

Converts the long-integer number "n" into an 4-byte string so it can later be retrieved from a random-access file as a numeric value.

MKS

This is a planned function.

Converts the single-precision number "n" into an 4-byte string so it can later be retrieved from a random-access file as a numeric value.

MONTH

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the month (1 to 12) value of that time. If the argument is missing it uses the actual local time. In other words it returns the actual month in this latter case. The months are numbered so that January is 1 and December is 12.

NAME filename,filename

Rename a file. The first file is the existing one, the second is the new name of the file. You can not move filed from one disk to another using this command.

This command merely renames a single file. Also you can not use wild characters in the source or destination file name.

If you can not rename a file for some reason, you can try to use the command FileCopy and then delete the old file. This is successful in some of the cases when `NAME` fails, but it is a slower method.

If the file can not be renamed then the command raises error.

NEXTFILE(dn)

Retrieve the next file name from an opened directory list. If there is no more file names it returns `undef`.

See also OPEN DIRECTORY and CLOSE DIRECTORY.

NOW

This function returns the local time expressed as seconds since January 1, 1970, 00:00am. The function does not accept any argument. This function is similar to the function GmTime() but returns the local time instead of the actual GMT.

OCT(n)

Take the argument as a long value and convert it to a string that represents the value in octal form.

ODD

Return `true` if the argument is an odd number. `ODD(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

See also EVEN()

ON ERROR GOTO [label | NULL]

Set the entry point of the error handling routine. If the argument is `NULL` then the error handling is switched off.

ON ERROR RESUME [label | next]

Setting `ON ERROR RESUME` will try to continue execution on the label or on the next statement when an error occurs without any error handling code.

See also ON ERROR GOTO, RESUME and ERROR.

OPEN file_name FOR mode AS [#] i [LEN=record_length]

Open or create and open a file. The syntax of the line is
`OPEN file_name FOR mode AS [ # ] i [ LEN=record_length ]`

The parameters:

- `file_name` if the name of the file to be opened. If the mode allows the file to be written the file is created if it did not existed before. If needed, directory is created for the file.
- `mode` is the mode the file is opened. It can be:
    - `input` open the file for reading. In this mode the file is opened in read only mode and can not be altered using the file number associated with the open file. Using any function or command that tries to write the file will result in error. In this mode the file has to exist already to open successfully. If the file to be opened for `input` does not exist the command `OPEN` raises an error.
    - `output` open the file for writing. If the file existed it's content is deleted first and a freshly opened empty file is ready to accept commands and functions to write into the file. When a file is opened this way no function or command trying to read from the file can be used using the file number associated with the file.

The file is opened in ASCII mode but the handling mode can be changed to binary any time.

- `append` open a possibly existing file and write after the current content. The same conditions apply as in the mode `output`, thus you can not read the file, only write. The file is opened in ASCII mode but the handling mode can be changed to binary any time.
- `random` open the file for reading and writing (textual mode). When you open a file using this mode the file can be written and the content of the existing file can be read. The file pointer can be moved back and forth any time using the command SEEK and thus quite complex file handling functions can be implemented. If the file did not exist it is created.
- `binary` open the file for reading and writing (binary mode). This mode is the same as `random` with the exception that the file is opened in binary mode.
- `socket` open a socket. In this case the file name is NOT a file name, but rather an Internet machine name and a port separated by colon, like `www.digital.com:80` You should not specify any method, like `http://` in front of the machine name, as this command opens a TCP socket to the machine's port and the protocol has to be implemented by the BASIC program.

- `#i` is the file number. After the file has been opened this number has to be used in later file handling functions and commands, like CLOSE to refer to the file. The `#` character is optional and is allowed for compatibility with other BASIC languages. The number can be between 1 and 512. This number is quite big for most of the applications and provides compatibility with VisualBasic.
- `record_length` is optional and specify the length of a record in the file. The default record length is 1 byte. File pointer setting commands usually work on records, thus SEEK, TRUNCATE and other commands and functions accept arguments or return values as number of records. The actual record length is not recorded anywhere thus the BASIC

program has to remember the actual length of a record in a file. This is not a BASIC program error to open a file with a different record size than it was created, although this may certainly be a programming error.

If the file number is specified as a variable and the variable value is set to integer zero then the command will automatically find a usable file number and set the variable to hold that value. Using any other expression of value integer zero is an error.

OPEN DIRECTORY dir_name PATTERN pattern OPTION option AS dn

Open a directory to retrieve the list of files.

- `dir_name` is the name of the directory.
- `pattern` is a wild card pattern to filter the file list.
- `option` is an integer value that can be composed AND-ing some of the following values
  - `SbCollectDirectories` Collect the directory names as well as file names into the file list.
  - `SbCollectDots` Collect the virtual . and .. directory names into the list.
  - `SbCollectRecursively` Collect the files from the directory and from all the directories below.
  - `SbCollectFullPath` The list will contain the full path to the file names. This means that the file names returned by the function NextFile will contain the directory path specified in the open directory statement and therefore can be used as argument to file handling commands and functions.
  - `SbCollectFiles` Collect the files. This is the default behavior.
  - `SbSortBySize` The files will be sorted by file size.
  - `SbSortByCreateTime` The files will be sorted by creation time.

  - `SbSortByAccessTime` The files will be sorted by access time.
  - `SbSortByModifyTime` The files will be sorted by modify time.
  - `SbSortByName` The files will be sorted by name. The name used for sorting is the bare file name without any path.
  - `SbSortByPath` The files will be sorted by name including the path. The path is the relative to the directory, which is currently opened. This sorting option is different from the value `sbSortByName` only when the value `sbCollectRecursively` is also used.
  - `SbSortAscending` Sort the file names in ascending order. This is the default behavior.
  - `SbSortDescending` Sort the file names in descending order.
  - `SbSortByNone` Do not sort. Specify this value if you do not need sorting. In this case directory opening can be much faster especially for large directories.
- `dn` is the directory number used in later references to the opened directory.

Note that this command can execute for a long time and consuming a lot of memory especially when directory listing is requested recursively. When the command is executed it collects the names of the files in the directory or directories as requested and builds up an internal list of the file names in the memory. The command NEXTFILE() uses the list to retrieve the next file name from the list.

This implies to facts:

- The function `NEXTFILE` will not ever return a file name that the file was created after, and did not exist when the command `OPEN DIRECTORY` was executed.
- Using CLOSE DIRECTORY after the list of the files is not needed as soon as possible is a good idea.

Using a directory number that was already used and not released calling CLOSE DIRECTORY raises an error.

If the list of the files in the directory can not be collected the command raises error.

See also CLOSE DIRECTORY and NEXTFILE().

OPTION symbol value

Set the integer value of an option. The option can be any string without the double quote. Option names are case insensitive in ScriptBasic.

This command has no other effect than storing the integer value in the option symbol table. The commands or extenal modules may access the values and may change their behavior accoring to the actual values associated with option symbols.

You can retrieve the actual value of an option symbol using the function OPTION()

OPTION("symbol")

Retrieve the actual value of an option symbol as an integer or `undef` if the option was not set. Unlike in the command OPTION the argument of this function should be double quoted.

pack("format",v1,v2,...,vn)

Pack list of arguments into a binary string.

The format strings can contain the packing control literals. Each of these characters optionally take the next argument and convert to the specific binary string format. The result is the concatenated sum of these strings.

Some control characters do not take argument, but result a constant string by their own.

- `SZ` the argument is stored as zero terminated string. If the argument already contains zchar that is taken as termnator and the rest of the string is ignored.
- `S1` the argument is stored as a string. One byte length and maximum 255 byte strings. If the argument longer than 255 bytes only the first 255 bytes are used, and the rest is ignored.
- `S2` same as `S1` but with two bytes for the length.
- `S3` same as `S1` but with three bytes for the length.
- `S4` same as `S1` but with four bytes for the length.
- `S5..8` the same as `S1` but with 5..8 bytes for the length.
- `Zn` one or more zero characters, does not take argument. `n` can be 1,2,3 ... positive numbers
- `In` integer number stored on n bytes. Low order byte first. If the number does not fit into n bytes the higher bytes are chopped. If the number is negative the high overflow bytes are filled with FF.
- `C` character (same as `I1`)
- `Un` same as `In` but for unsigned numbers.
- `An` store the argument as string on n bytes. If the argument is longer than n bytes only the first n bytes are stored. If the argument is shorter than n bytes the higher bytes are filled with space.
- `R` a real number.

See also UNPACK

PAUSE

This is a planned command.

```
PAUSE n
```

Suspend the execution of the interpreter (process or thread) for `n` milliseconds.

PI

This built-in constant is implemented as an argument less function. Returns the approximate value of the constant PI which is the ratio of the circumference of a circle to its diameter.

POP

Pop off one value from the GOSUB/RETURN stack. After this command a `RETURN` will return to one level higher and to the place where it was called from. For more information see the documentation of the command GOSUB and RETURN.

POW

Calculates the x-th exponent of 10. If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`POW(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

PRINT [# fn ,] print_list

This command prints the elements of the `print_list`. The argument `print_list` is a comma separated list of expressions. The expressions are evaluated one after the other and are printed to the standard output or to the file.

The command prints the `print_list` to an opened file given by the file number `fn`. If `fn` (along with the `#` character) is not specified the command prints to the standard output. The file has to be opened to some "output" mode otherwise

the command fails to print anything into the file. The command can also print into an opened socket (a file opened for mode socket). If the file is not opened then the expressions in the list `print_list` are not evaluated and the command actually does nothing. If the file is opened, but not for a kind of "output" mode then the expressions in the `print_list` are evaluated but the printing does not happen. Neither printing to a non-opened file number nor printing to a file opened for some read-only mode generates error.

If there is no `print_list` specified the command prints a new line. In other words if the keyword `PRINT` stands on the command with the optional `#` and the file number but without anything to print then the command will print a new line character.

Note that unlike other BASIC implementations the command `PRINT` does not accept print list formatters, like `AT` or semicolons and does not tabify the output. The arguments are printed to the file or to the standard output one after the other without any intrinsic space or tab added. Also the print statements does not print a new line at the end of the print list unless the new line character is explicitly defined or if there is no print list at all following the command.

RANDOMIZE

Seed the random number generator. If the command is presented without argument the random number generator is seed with the actual time. If argument is provided the random number generator is seed with the argument following the keyword `RANDOMIZE`.

ref v1 = v2

Assign a variable to reference another variable. Following this command altering one of the variables alters both variables. In other words this command can be used to define a kind of alias to a variable. The mechanism

is the same as local variable of a function is an alias of a variable passed to the function as actual argument. The difference is that this reference is not automatically released when some function returns, but rather it is alive so long as long the referencing variable is not undefined saying `undef variable` in a command.

To have an alias to a variable is not something of a great value though. It becomes a real player when the 'variable' is not just an ordinary 'named' variable but rather part of an array (or associative array). Using this mechanisms the programmer can build up arbitrary complex memory structures without caring such complex things as pointers for example in C. This is a simple BASIC way of building up complex memory structures.

REPEAT

This command implements a loop which is repeated so long as long the expression standing after the loop closing line `UNTIL` becomes `true`. The loop starts with a line containing the keyword `REPEAT` and finishes with the line `UNTIL expression`.

```
repeat
  ...
  commands to repeat
  ...
until expression
```

The expression is evaluated each time after the loop is executed. This means that the commands inside the loop are executed at least once.

This kind of loop syntax is not usual in BASIC dialects but can be found in languages like PASCAL. Implementing this loop in ScriptBasic helps those programmers, who have PASCAL experience.

See also WHILE, DOUNTIL, DOWHILE, REPEAT, DO and FOR.

REPLACE(string, string, string [,number] [,position])

```
REPLACE(base_string,search_string,replace_string [,number_of_replaces]
[,position])
```

This function replaces one or more occurrences of a sub-string in a string. `REPLACE(a,b,c)` searches the string `a` seeking for occurrences of sub-string `b` and replaces each of them with the string `c`.

The fourth and fifth arguments are optional. The fourth argument specifies the number of replaces to be performed. If this is missing or is `undef` then all occurrences of string `b` will be replaced. The fifth argument may specify the start position of the operation. For example the function call

```
REPLACE("alabama mama", "a","x",3,5)
```

will replace only three occurrences of string `"a"` starting at position 5. The result is `"alabxmx mxma"`.

RESET

This command closes all files opened by the current BASIC program. This command usually exists in most BASIC implementation. There is no need to close a file before a BASIC program finishes, because the interpreter automatically closes all files that were opened by the program.

RESET DIRECTORY [#] dn

Reset the directory file name list and start from the first file name when the next call to NEXTFILE() is performed.

See also OPEN DIRECTORY, CLOSE DIRECTORY, NEXTFILE(), EOD().

RESUME [label | next]

Resume the program execution after handling the error. `RESUME` without argument tries to execute the same line again that caused the error. `RESUME`

`NEXT` tries to continue execution after the line that caused the error. `RESUME label` tries to continue execution at the specified label.

See also ON ERROR GOTO, ON ERROR RESUME and ERROR.

RETURN

Return from a subroutine started with GOSUB. For more information see the documentation of the command GOSUB.

REWIND [#]fn

Positions the file cursor to the start of the file. This is the same as `SEEK fn,0` or `SEEK #fn,0`

The argument to the statement is the file number used in the OPEN statement to open the file. The character `#` is optional and can only be used for compatibility reasons.

RIGHT(string,len)

Creates the right of a string. The first argument is the string. The second argument is the number of characters that should be put into the result. If this value is larger than the number of characters in the string then all the string is returned.

See also MID(), LEFT().

details

RND

Returns a random number as generated by the C function `rand()`. Note that this random number generator usually provided by the libraries implemented for the C compiler or the operating system is not the best quality ones. If you

need really good random number generator then you have to use some other libraries that implement reliable RND functions.

ROUND

This function rounds its argument. The first argument is the number to round, and the optional second argument specifies the number of fractional digits to round to.

The function rounds to integer value if the second argument is missing.

The return value is long if the number of decimal places to keep is zero, otherwise the return value is double.

Negative value for the number of decimal places results rounding to integer value.

`ROUND(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

Also `ROUND(val,undef)` is equivalent to `ROUND(value)`.

See also INT(), FRAC() and FIX()

RTRIM()

Remove the space from the right of the string.

SEC

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the seconds value of that time. If the argument is missing the function uses the actual local time.

SECANT

This is a planned function to calculate the secant of the argument.

SEEK fn,position

Go to a specified position in an open file. You can use this command to position the file pointer to a specific position. The next read or write operation performed on the file will be performed on that very position that was set using the command `SEEK`. The first argument is the file number that was used in the statement `OPEN` to open the file. The second argument is the position where the file pointer is to be set.

The position is counted from the start of the file counting records. The actual file pointer will be set **after** the record `position`. This means that if for example you want to set the file pointer To the start of the file then you have to `SEEK fn,0`. This will set the File pointer before the first record.

If there was no record length specified when the file was opened the counting takes bytes. There is no special "record" structure of a file as it is usually under UNIX or Windows NT. The record is merely the number of bytes treated as a single unit specified during file opening.

SET FILE filename parameter=value

Set some of the parameters of a file. The parameter can be:

- `owner` set the owner of the file. This operation requires `root` permission on UNIX or `Administrator` privileges on Windows NT. The value should be the string representation of the UNIX user or the Windows NT domain user.
- `createtime`
- `modifytime`
- `accesstime`

- Set the time of the file. The value should be the file time in seconds since January 1,1970. 00:00GMT.

If the command can not be executed an error is raised. Note that setting the file owner also depends on the file system. For example FAT file system does not store the owner of a file and thus can not be set.

Also setting the file time on some file system may be unsuccessful for values that are successful under other file systems. This is because different file systems store the file times using different possible start and end dates and resolution. For example you can set a file to hold the creation time to be January 1, 1970 0:00 under NTFS, but not under FAT.

The different file systems store the file times with different precision. Thus the actual time set will be the closest time not later than the specified in the command argument. For this reason the values returned by the functions `File***Time` may not be the same that was specified in the `SET FILE` command argument.

SET JOKER "c" TO "abcdefgh..."

Set a joker character to match certain characters when using the LIKE operator. The joker character `"c"` can be one of the following characters

```
*  #  $  @  ?  &  %  !  +  /  |  <  >
```

The string after the keyword `TO` should contain all the characters that the joker character should match. To have the character to match only itself to be a normal character say

```
SET NO JOKER "c"
```

See also SET [NO] WILD, LIKE (details), JOKER()

SET WILD "c" TO "abcdefgh..."

Set a wild character to match certain characters when using the LIKE operator. The wild character `"c"` can be one of the following characters

```
*  #  $  @  ?  &  %  !  +  /  |  <  >
```

The string after the keyword `TO` should contain all the characters that the wild card character should match. To have the character to match only itself to be a normal character say

```
SET NO WILD "c"
```

See also SET [NO] JOKER, LIKE (details), JOKER()

SIN

Calculates the sine of the argument. If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`SIN(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

SLEEP(n)

Suspend the execution of the interpreter (process or thread) for `n` seconds.

Whenever the program has to wait for a few seconds it is a good idea to call this function. Older BASIC programs originally designed for old personal computers like Atari, Amiga, ZX Spectrum intend to use empty loop to wait time to elapse. On modern computers this is a bad idea and should not be done.

If you execute an empty loop to wait you consume CPU. Because the program does not access any resource to wait for it actually consumes all the

CPU time slots that are available. This means that the computer slows down, does not respond to user actions timely.

Different computers run with different speed and an empty loop consuming 20sec on one machine may run 2 minutes on the other or just 10 millisec. You can not reliably tell how much time there will be during the empty loop runs.

When you call `SLEEP(n)` the operating system is called telling it that the code does not need the CPU for `n` seconds. During this time the program is suspended and the operating system executes other programs as needed. The code is guaranteed to return from the function `SLEEP` not sooner than `n` seconds, but usually it does return before the second `n+1` starts.

SPACE(n)

Return a string of length `n` containing spaces.

SPLIT string BY string TO var_1,var_2,var_3,...,var_n

Takes the string and splits into the variables using the second string as delimiter.

SPLITA string BY string TO array

Split a string into an array using the second string as delimiter. If the string has zero length the array becomes undefined. When the delimiter is zero length string each array element will contain a single character of the string.

See also SPLIT

SPLITAQ string BY string QUOTE string TO array

Split a string into an array using the second string as delimiter. The delimited fields may optionally be quoted with the third string. If the string to be split has

zero length the array becomes undefined. When the delimiter is a zero length string each array element will contain a single character of the string.

Leading and trailing delimiters are accepted and return an empty element in the array. For example :-

```
SPLITAQ ",'A,B',C," BY "," QUOTE "'" TO Result
```

will generate

```
Result[0] = ""
Result[1] = "A,B"
Result[2] = "C"
Result[3] = ""
```

Note that this kind of handling of trailing and leading empty elements is different from the handling of the same by the command SPLIT and SPLITA which do ignore those empty elements. This command is useful to handle lines exported as CSV from Excel or similar application.

The QUOTE string is really a string and need not be a single character. If there is an unmatched quote string in the string to be split then the rest of the string until its end is considered quoted.

If there is an unmatched

See also SPLITA

This command was suggested and implemented by Andrew Kingwell (Andrew.Kingwell@idstelecom.co.uk)

SQR

Calculates the square root of the argument.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`SQR(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

If the argument is a negative number the result of the function is `undef` or the function raises error if the option `RaiseMathError` has the bit `sbMathErrDiv` set.

If the square root of the argument is an integer number then the function returns an integer number. In other cases the returned value is real even if the argument itself is integer.

Note that this function has the opposite meaning in the language PASCAL, namely the square of the number. This may cause some problem if you are experienced in PASCAL programming. In that language `SQRT` notes the square *root* of a number.

STOP

This command stops program execution. There is no possibility to restart program execution after this command was executed.

See also END.

STR(n)

Converts a number to string. This function is rarely needed, because conversion is done automatically. details

STRING(n,code)

Create a string of length `n` containing characters `code`. If `code` is a string then the first character of the string is used to fill the result. Otherwise `code` is converted to long and the ASCII code is used.

STRREVERSE(string)

Return the reversed string (aka. all the characters in the string in reverse order).

SUB fun()

This command should be used to define a subroutine. A subroutine is a piece of code that can be called by the BASIC program from the main part or from a function or subroutine.

```
SUB sub(a,b,c)
...
END SUB
```

The end of the subroutine is defined by the line containing the keywords `END SUB`.

Note that functions and subroutines are not really different in ScriptBasic. ScriptBasic allows you to return a value from a subroutine and to call a function using the command `CALL`. It is just a convention to have separately `SUB` and `FUNCTION` declarations.

For detailed information please read the documentation of the command FUNCTION

swap a,b

Planned command.

This command swaps two variables.

SYSTEM(executable_program)

This function should be used to start an external program in a separate process in asynchronous mode. In other words you can start a process and

let it run by itself and not wait for the process to finish. After starting the new process the BASIC program goes on parallel with the started external program.

The return value of the function is the PID of the newly created process.

If the program specified by the argument can not be started then the return value is zero. Under UNIX the program may return a valid PID even in this case. This is because UNIX first makes a copy of the process that wants to start another and then replaces the new process image with the program image to be started. In this case the new process is created and the command `SYSTEM` has no information on the fact that the new process was not able to replace the executable image of itself. In this case, however, the child process has a very short life.

TAN

This is a planned function to calculate the tangent of the argument.

TAN2

This is a planned function to calculate the tangent of the ratio of the two arguments.

TEXTMODE [# fn] | input | output

Set an opened file handling to text mode.

The argument is either a file number with which the file was opened or one of keywords `input` and `output`. In the latter case the standard input or output is set.

See also BINMODE Set an opened file handling to text mode.

TIMEVALUE

This function gets zero or more, at most six arguments and interprets them as year, month, day, hour, minute and seconds and calculates the number of seconds elapsed since January 1, 1970 till the time specified. If some arguments are missing or `undef` the default values are the following:

- year = 1970
- month = January
- day = 1st
- hours = 0
- minutes = 0
- seconds = 0

TRIM()

Remove the space from both ends of the string.

TRUE

This built-in constant is implemented as an argument less function. Returns the value `true`.

TRUNCATE fn,new_length

Truncate an opened file to the specified size. The first argument Has to be the file number used in the OPEN statement opening the file. The second argument is the number of records to be in the file after it is truncated.

The size of a record has to be specified when the file is opened. If the size Of a record is not specified in number of bytes then the command `TRUNCATE` Does truncate the file to the number of specified bytes instead of records. (In other words the record length is one byte.)

When the file is actually shorter than the length specified by the command argument the command `TRUNCATE` automatically extends the file padding with bytes containing the value 0.

TYPE

This function can be used to determine the type of an expression. The function returns a numeric value that describes the type of the argument. Although the numeric values are guaranteed to be the one defined here it is recommended that you use the predefined symbolic constant values to compare the return value of the function against. The function return value is the following

- `SbTypeUndef` 0 if the argument is `undef`.
- `SbTypeString` 1 if the argument is string.
- `SbTypeReal` 2 if the argument is real.
- `SbTypeInteger` 3 if the argument is integer.
- `SbTypeArray` 4 if the argument is an array.

See also ISARRAY(), ISSTRING(), ISINTEGER(), ISREAL(), ISNUMERIC(), IsDefined(), ISUNDEF(), ISEMPTY().

UBOUND

This function can be used to determine the highest occupied index of an array. Note that arrays are increased in addressable indices automatically, thus it is not an error to use a higher index that the value returned by the function `UBOUND`. On the other hand all the element having index larger than the returned value are `undef`.

The argument of this function has to be an array. If the argument is an ordinary value, or a variable that is not an array the value returned by the function will be `undef`.

`UBOUND(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

See also LBOUND().

UCASE()

Uppercase a string.

UNDEF variable

Sets the value of a variable (or some other ScriptBasic left value) to be undefined. This command can also be used to release the memory that was occupied by an array when the variable holding the array is set to `undef`.

When this command is used as a function (with or without, but usually without parentheses), it simply returns the value `undef`.

details

UNPACK string BY format TO v1,v2,...,vn

Unpack the binary string `string` using the format string into the variables. The format string should have the same format as the format string the in the function PACK().

VAL

Converts a string to numeric value. If the string is integer it returns an integer value. If the string contains a number presentation which is a float number the returned value is real. In case the argument is already numeric no conversion is done.

`VAL(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

WAITPID(PID,ExitCode)

This function should be used to test for the existence of a process.

The return value of the function is 0 if the process is still running. If the process has exited (or failed in some way) the return value is `1` and the exit code of the process is stored in `ExitCode`.

WEEKDAY

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the week day value of that time. If the argument is missing the function uses the actual local time. In other words it returns what day it is at the moment.

WHILE condition

Implements the 'while' loop as it is usually done in most basic implementations. The loop starts with the command `while` and finished with the line containing the keyword `wend`. The keyword `while` is followed by an expression and the loop is executes so long as long the expression is evaluated `true`.

```
while expression
  ...
  commands to repeat
  ...
wend
```

The expression is evaluated when the looping starts and each time the loop is restarted. It means that the code between the `while` and `wend` lines may be skipped totally if the expression evaluates to some `false` value during the first evaluation before the execution starts the loop.

In case some condition makes it necessary to exit the loop from its middle then the command GOTO can be used.

ScriptBasic implements several looping constructs to be compatible with different BASIC language dialects. Some constructs are even totally interchangeable to let programmers with different BASIC experience use the one that fit they the best. See also WHILE, DOUNTIL, DOWHILE, REPEAT, DO and FOR.

YEAR

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the year value of that time. If the argument is missing it uses the actual local time to calculate the year value. In other words it returns the actual year.

YEARDAY

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the year-day value of that time. This is actually the number of the day inside the year so that January 1st is #1 and December 31 is #365 (or 366 in leap years). If the argument is missing the function uses the actual local time.

This page intentionally left blank

Reserved Words

ABS	ACOS	ACOSECANT
ACTAN	ADDDAY	ADDHOUR
ADDMINUTE	ADDMONTH	ADDRESS
ADDSECOND	ADDWEEK	ADDYEAR
ALIAS	AND	AS
ASC	ASECANT	ASIN
ATAN	ATN	
BIN	BINMODE	BY
BYVAL		
CALL	CHDIR	CHOMP
CHR	CHR$	CINT
CLOSE	CLOSEALL	COMMAND
CONF	CONST	COS
COSECANT	COTAN	COTAN2
CRYPT	CURDIR	CVD
CVI	CVL	CVS
DAY	DECLARE	DELETE
DELTREE	DIRECTORY	DO
ELIF	ELSE	ELSEIF
ELSIF	END	ENDIF
ENVIRON	ENVIRON$	EOD
EOF	ERROR	ERROR$
EVEN	EXECUTE	EXIT
EXP		

FALSE	FILE	FILEACCESSTIME
FILECOPY	FILECREATETIME	FILEEXISTS
FILELEN	FILEMODIFYTIME	FILEOWNER
FIX	FOR	FORK
FORMAT	FORMATDATE	FORMATTIME
FRAC	FREEFILE	FROM
FUNCTION		

GCD	GLOBAL	GMTIME
GMTIMETOLOCALTIME	GO	GOSUB
GOTO		

HCOS	HCOSECANT	HCTAN
HEX	HEX$	HOSTNAME
HOUR	HSECANT	HSIN
HTAN		

ICALL IF IMAX
IMIN INPUT INSTR
INSTRREV INT ISARRAY
ISDEFINED ISDIRECTORY ISEMPTY
ISFILE ISINTEGER ISNUMERIC
ISREAL ISSTRING ISUNDEF

JOIN JOKER

KILL

LBOUND LCASE LCASE$
LCM LEFT LEFT$
LEN LET LIB
LIKE LINE LOC
LOCAL LOCALTIMETOGMTIME LOCK
LOF LOG LOG10
LOOP LOWER LOWER$
LTRIM LTRIM$

MAX MAXINT MID
MID$ MIN MININT
MINUTE MKD MKD$
MKDIR MKI MKI$
MKL MKL$ MKS
MKS$ MODULE MONTH
NAME NEXT NEXTFILE

NO NOT NOW
NULL

OCT	OCT$	ODD
ON	OPEN	OPTION
OR	OUTPUT	
PACK	PATTERN	PAUSE
PI	POP	POW
PRINT	PRINTNL	
QUOTE		
RANDOMIZE	REF	REGION
REPEAT	REPLACE	RESET
RESUME	RETURN	REWIND
RIGHT	RIGHT$	RND
ROUND	RTRIM	RTRIM$

SEC	SECANT	SEEK
SET	SGN	SIN
SLEEP	SPACE	SPACE$
SPLIT	SPLITA	SPLITAQ
SQR	STEP	STOP
STR	STR$	STRING
STRING$	STRREVERSE	STRREVERSE$
SUB	SWAP	SYSTEM
TAN	TAN2	TEXTMODE
THEN	TIME	TIMEVALUE
TO	TRIM	TRIM$
TRUE	TRUNCATE	TYPE
UBOUND	UCASE	UCASE$
UNDEF	UNPACK	UNTIL
UPPER	UPPER$	
VAL	VAR	
WAITPID	WEEKDAY	WEND
WHILE	WILD	
XOR		
YEAR	YEARDAY	

This page intentionally left blank

Reserved Words - not yet implemented

BIN	This is a planned function to convert the argument number to binary format. (aka. format as a binary number containing only 0 and 1 characters and return this string)
GCD	Mathematical function has become a reserved word, but are not implemented.
LCM	Mathematical function has become a reserved word, but are not implemented.
ATN	This is a planned function to calculate the arcus tangent of the argument.
ATAN	This is a planned function to calculate the arcus tangent of the argument.
TAN	This is a planned function to calculate the tangent of the argument.
TAN2	This is a planned function to calculate the tangent of the ratio of the two arguments.
COTAN	This is a planned function to calculate the cotangent of the argument.
COTAN2	This is a planned function to calculate the cotangent of the ratio of the two arguments.
ACTAN	This is a planned function to calculate the arcus cotangent of the argument.
SECANT	This is a planned function to calculate the secant of the argument.
COSECANT	This is a planned function to calculate the cosecant of

	the argument.
ASECANT	This is a planned function to calculate the arcus secant of the argument.
ACOSECANT	This is a planned function to calculate the arcus cosecant of the argument.
HSIN	This is a planned function to calculate the sinus hyperbolicus of the argument.
HCOS	This is a planned function to calculate the cosinus hyperbolicus of the argument.
HTAN	This is a planned function to calculate the tangent hyperbolicus of the argument.
HCTAN	This is a planned function to calculate the cotangent hyperbolicus of the argument.
HSECANT	This is a planned function to calculate the secant hyperbolicus of the argument.
HCOSECANT	This is a planned function to calculate the cosecant hyperbolicus of the argument.
MAX	This is a planned function to select and return the maximum of the arguments.
MIN	This is a planned function to select and return the minimum of the arguments.
IMAX	This is a planned function to select and return the index of the maximum of the arguments.
IMIN	This is a planned function to select and return the index of the minimum of the arguments.
CVD	This is a planned function to convert the argument string into a real number. (8 byte)

CVI	This is a planned function to convert the argument string into an integer. (2 bytes)
CVL	This is a planned function to convert the argument string into an long integer. (4 bytes)
CVS	This is a planned function to convert the argument string into an integer. (4 byte)
MKD	This is a planned function to convert the argument real number to an 8 byte string. (8 byte)
MKI	This is a planned function to convert the argument integer number to a string. (2 byte)
MKS	This is a planned function to converts a single-precision number "n" into a string so it can later be retrieved from a random-access file as a numeric value. (4 byte)
MKL	This is a planned function converts a long-integer number "n" into a string so it can later be retrieved from a random-access file as a numeric value. (4-byte)

This page intentionally left blank

Appendix A: ASCII table

Control Codes

DEC	HEX	BIN	Symbol	HTML Number	HTML Name	Description
0	00	00000000	NUL	&#000;		Null char
1	01	00000001	SOH	&#001;		Start of Heading
2	02	00000010	STX	&#002;		Start of Text
3	03	00000011	ETX	&#003;		End of Text
4	04	00000100	EOT	&#004;		End of Transmission
5	05	00000101	ENQ	&#005;		Enquiry
6	06	00000110	ACK	&#006;		Acknowledgment
7	07	00000111	BEL	&#007;		Bell
8	08	00001000	BS	&#008;		Back Space
9	09	00001001	HT			Horizontal Tab
10	0A	00001010	LF
	Line Feed
11	0B	00001011	VT	&#011;		Vertical Tab
12	0C	00001100	FF			Form Feed
13	0D	00001101	CR
	Carriage Return
14	0E	00001110	SO	&#014;		Shift Out / X-On
15	0F	00001111	SI	&#015;		Shift In / X-Off
16	10	00010000	DLE	&#016;		Data Line Escape
17	11	00010001	DC1	&#017;		Device Control 1 (oft. XON)
18	12	00010010	DC2	&#018;		Device Control 2
19	13	00010011	DC3	&#019;		Device Control 3 (oft. XOFF)
20	14	00010100	DC4	&#020;		Device Control 4
21	15	00010101	NAK	&#021;		Negative Acknowledgement
22	16	00010110	SYN	&#022;		Synchronous Idle
23	17	00010111	ETB	&#023;		End of Transmit Block
24	18	00011000	CAN	&#024;		Cancel
25	19	00011001	EM	&#025;		End of Medium
26	1A	00011010	SUB	&#026;		Substitute
27	1B	00011011	ESC	&#027;		Escape
28	1C	00011100	FS	&#028;		File Separator
29	1D	00011101	GS	&#029;		Group Separator
30	1E	00011110	RS	&#030;		Record Separator
31	1F	00011111	US	&#031;		Unit Separator

Standard (7-bit) Character Set

DEC	HEX	BIN	Symbol	HTML Number	HTML Name	Description
32	20	00100000		&#32;		Space
33	21	00100001	!	&#33;		Exclamation mark
34	22	00100010	"	&#34;	&quot;	Double quotes
35	23	00100011	#	&#35;		Number
36	24	00100100	$	&#36;		Dollar
37	25	00100101	%	&#37;		Percentage sign
38	26	00100110	&	&#38;	&amp;	Ampersand
39	27	00100111	'	&#39;		Single quote
40	28	00101000	(	&#40;		Open parenthesis
41	29	00101001	)	&#41;		Close parenthesis
42	2A	00101010	*	&#42;		Asterisk
43	2B	00101011	+	&#43;		Plus
44	2C	00101100	,	&#44;		Comma
45	2D	00101101	-	&#45;		Hyphen
46	2E	00101110	.	&#46;		Period or full stop
47	2F	00101111	/	&#47;		Slash or divide
48	30	00110000	0	&#48;		Zero
49	31	00110001	1	&#49;		One
50	32	00110010	2	&#50;		Two
51	33	00110011	3	&#51;		Three
52	34	00110100	4	&#52;		Four
53	35	00110101	5	&#53;		Five
54	36	00110110	6	&#54;		Six
55	37	00110111	7	&#55;		Seven
56	38	00111000	8	&#56;		Eight
57	39	00111001	9	&#57;		Nine
58	3A	00111010	:	&#58;		Colon
59	3B	00111011	;	&#59;		Semicolon
60	3C	00111100	<	&#60;	&lt;	Less
61	3D	00111101	=	&#61;		Equals
62	3E	00111110	>	&#62;	&gt;	Greater than
63	3F	00111111	?	&#63;		Question mark
64	40	01000000	@	&#64;		At symbol
65	41	01000001	A	&#65;		Uppercase A

DEC	HEX	BIN	Symbol	HTML Number	HTML Name	Description
66	42	01000010	B	&#66;		Uppercase B
67	43	01000011	C	&#67;		Uppercase C
68	44	01000100	D	&#68;		Uppercase D
69	45	01000101	E	&#69;		Uppercase E
70	46	01000110	F	&#70;		Uppercase F
71	47	01000111	G	&#71;		Uppercase G
72	48	01001000	H	&#72;		Uppercase H
73	49	01001001	I	&#73;		Uppercase I
74	4A	01001010	J	&#74;		Uppercase J
75	4B	01001011	K	&#75;		Uppercase K
76	4C	01001100	L	&#76;		Uppercase L
77	4D	01001101	M	&#77;		Uppercase M
78	4E	01001110	N	&#78;		Uppercase N
79	4F	01001111	O	&#79;		Uppercase O
80	50	01010000	P	&#80;		Uppercase P
81	51	01010001	Q	&#81;		Uppercase Q
82	52	01010010	R	&#82;		Uppercase R
83	53	01010011	S	&#83;		Uppercase S
84	54	01010100	T	&#84;		Uppercase T
85	55	01010101	U	&#85;		Uppercase U
86	56	01010110	V	&#86;		Uppercase V
87	57	01010111	W	&#87;		Uppercase W
88	58	01011000	X	&#88;		Uppercase X
89	59	01011001	Y	&#89;		Uppercase Y
90	5A	01011010	Z	&#90;		Uppercase Z
91	5B	01011011	[	&#91;		Opening bracket
92	5C	01011100	\	&#92;		Backslash
93	5D	01011101	]	&#93;		Closing bracket
94	5E	01011110	^	&#94;		Caret - circumflex
95	5F	01011111	_	&#95;		Underscore
96	60	01100000	`	&#96;		Grave accent
97	61	01100001	a	&#97;		Lowercase a
98	62	01100010	b	&#98;		Lowercase b
99	63	01100011	c	&#99;		Lowercase c
100	64	01100100	d	&#100;		Lowercase d
101	65	01100101	e	&#101;		Lowercase e
102	66	01100110	f	&#102;		Lowercase f
103	67	01100111	g	&#103;		Lowercase g
104	68	01101000	h	&#104;		Lowercase h

DEC	HEX	BIN	Symbol	HTML Number	HTML Name	Description
105	69	01101001	i	&#105;		Lowercase i
106	6A	01101010	j	&#106;		Lowercase j
107	6B	01101011	k	&#107;		Lowercase k
108	6C	01101100	l	&#108;		Lowercase l
109	6D	01101101	m	&#109;		Lowercase m
110	6E	01101110	n	&#110;		Lowercase n
111	6F	01101111	o	&#111;		Lowercase o
112	70	01110000	p	&#112;		Lowercase p
113	71	01110001	q	&#113;		Lowercase q
114	72	01110010	r	&#114;		Lowercase r
115	73	01110011	s	&#115;		Lowercase s
116	74	01110100	t	&#116;		Lowercase t
117	75	01110101	u	&#117;		Lowercase u
118	76	01110110	v	&#118;		Lowercase v
119	77	01110111	w	&#119;		Lowercase w
120	78	01111000	x	&#120;		Lowercase x
121	79	01111001	y	&#121;		Lowercase y
122	7A	01111010	z	&#122;		Lowercase z
123	7B	01111011	{	&#123;		Opening brace
124	7C	01111100	|	&#124;		Vertical bar
125	7D	01111101	}	&#125;		Closing brace
126	7E	01111110	~	&#126;		Tilde
127	7F	01111111		&#127;		Delete

Extended (8-bit) Character Set

DEC	HEX	BIN	Symbol	HTML Number	HTML Name	Description
128	80	10000000	€	€	€	Euro sign
129	81	10000001				
130	82	10000010	‚	‚	‚	Single low-9 quotation mark
131	83	10000011	ƒ	ƒ	ƒ	Latin small letter f with hook
132	84	10000100	„	„	„	Double low-9 quotation mark
133	85	10000101	…	…	…	Horizontal ellipsis
134	86	10000110	†	†	†	Dagger
135	87	10000111	‡	‡	‡	Double dagger
136	88	10001000	ˆ	ˆ	ˆ	Modifier letter circumflex accent
137	89	10001001	‰	‰	‰	Per mille sign
138	8A	10001010	Š	Š	Š	Latin capital letter S with caron
139	8B	10001011	‹	‹	‹	Single left-pointing angle quotation
140	8C	10001100	Œ	Œ	Œ	Latin capital ligature OE
141	8D	10001101				
142	8E	10001110	Ž	Ž		Latin captial letter Z with caron
143	8F	10001111				
144	90	10010000				
145	91	10010001	‘	‘	‘	Left single quotation mark
146	92	10010010	’	’	’	Right single quotation mark
147	93	10010011	“	“	“	Left double quotation mark
148	94	10010100	”	”	”	Right double quotation mark
149	95	10010101	•	•	•	Bullet
150	96	10010110	–	–	–	En dash
151	97	10010111	—	—	—	Em dash
152	98	10011000	˜	˜	˜	Small tilde
153	99	10011001	™	™	™	Trade mark sign
154	9A	10011010	š	š	š	Latin small letter S with caron
155	9B	10011011	›	›	›	Single right-pointing angle quotation mark
156	9C	10011100	œ	œ	œ	Latin small ligature oe
157	9D	10011101				
158	9E	10011110	ž	ž		Latin small letter z with caron
159	9F	10011111	Ÿ	Ÿ	ÿ	Latin capital letter Y with diaeresis
160	A0	10100000				Non-breaking space

DEC	HEX	BIN	Symbol	HTML Number	HTML Name	Description
161	A1	10100001	¡	¡	¡	Inverted exclamation mark
162	A2	10100010	¢	¢	¢	Cent sign
163	A3	10100011	£	£	£	Pound sign
164	A4	10100100	¤	¤	¤	Currency sign
165	A5	10100101	¥	¥	¥	Yen sign
166	A6	10100110	¦	¦	¦	Pipe, Broken vertical bar
167	A7	10100111	§	§	§	Section sign
168	A8	10101000	¨	¨	¨	Spacing diaeresis - umlaut
169	A9	10101001	©	©	©	Copyright sign
170	AA	10101010	ª	ª	ª	Feminine ordinal indicator
171	AB	10101011	«	«	«	Left double angle quotes
172	AC	10101100	¬	¬	¬	Not sign
173	AD	10101101	-			Soft hyphen
174	AE	10101110	®	®	®	Registered trade mark sign
175	AF	10101111	¯	¯	¯	Spacing macron - overline
176	B0	10110000	°	°	°	Degree sign
177	B1	10110001	±	±	±	Plus-or-minus sign
178	B2	10110010	²	²	²	Superscript two - squared
179	B3	10110011	³	³	³	Superscript three - cubed
180	B4	10110100	´	´	´	Acute accent - spacing acute
181	B5	10110101	µ	µ	µ	Micro sign
182	B6	10110110	¶	¶	¶	Pilcrow sign - paragraph sign
183	B7	10110111	·	·	·	Middle dot - Georgian comma
184	B8	10111000	¸	¸	¸	Spacing cedilla
185	B9	10111001	¹	¹	¹	Superscript one
186	BA	10111010	º	º	º	Masculine ordinal indicator
187	BB	10111011	»	»	»	Right double angle quotes
188	BC	10111100	¼	¼	¼	Fraction one quarter
189	BD	10111101	½	½	½	Fraction one half
190	BE	10111110	¾	¾	¾	Fraction three quarters
191	BF	10111111	¿	¿	¿	Inverted question mark
192	C0	11000000	À	À	À	Latin capital letter A with grave
193	C1	11000001	Á	Á	Á	Latin capital letter A with acute
194	C2	11000010	Â	Â	Â	Latin capital letter A with circumflex
195	C3	11000011	Ã	Ã	Ã	Latin capital letter A with tilde
196	C4	11000100	Ä	Ä	Ä	Latin capital letter A with diaeresis
197	C5	11000101	Å	Å	Å	Latin capital letter A with ring above
198	C6	11000110	Æ	Æ	Æ	Latin capital letter AE
199	C7	11000111	Ç	Ç	Ç	Latin capital letter C with cedilla

DEC	HEX	BIN	Symbol	HTML Number	HTML Name	Description
200	C8	11001000	È	È	È	Latin capital letter E with grave
201	C9	11001001	É	É	É	Latin capital letter E with acute
202	CA	11001010	Ê	Ê	Ê	Latin capital letter E with circumflex
203	CB	11001011	Ë	Ë	Ë	Latin capital letter E with diaeresis
204	CC	11001100	Ì	Ì	Ì	Latin capital letter I with grave
205	CD	11001101	Í	Í	Í	Latin capital letter I with acute
206	CE	11001110	Î	Î	Î	Latin capital letter I with circumflex
207	CF	11001111	Ï	Ï	Ï	Latin capital letter I with diaeresis
208	D0	11010000	Ð	Ð	Ð	Latin capital letter ETH
209	D1	11010001	Ñ	Ñ	Ñ	Latin capital letter N with tilde
210	D2	11010010	Ò	Ò	Ò	Latin capital letter O with grave
211	D3	11010011	Ó	Ó	Ó	Latin capital letter O with acute
212	D4	11010100	Ô	Ô	Ô	Latin capital letter O with circumflex
213	D5	11010101	Õ	Õ	Õ	Latin capital letter O with tilde
214	D6	11010110	Ö	Ö	Ö	Latin capital letter O with diaeresis
215	D7	11010111	×	×	×	Multiplication sign
216	D8	11011000	Ø	Ø	Ø	Latin capital letter O with slash
217	D9	11011001	Ù	Ù	Ù	Latin capital letter U with grave
218	DA	11011010	Ú	Ú	Ú	Latin capital letter U with acute
219	DB	11011011	Û	Û	Û	Latin capital letter U with circumflex
220	DC	11011100	Ü	Ü	Ü	Latin capital letter U with diaeresis
221	DD	11011101	Ý	Ý	Ý	Latin capital letter Y with acute
222	DE	11011110	Þ	Þ	Þ	Latin capital letter THORN
223	DF	11011111	ß	ß	ß	Latin small letter sharp s - ess-zed
224	E0	11100000	à	à	à	Latin small letter a with grave
225	E1	11100001	á	á	á	Latin small letter a with acute
226	E2	11100010	â	â	â	Latin small letter a with circumflex
227	E3	11100011	ã	ã	ã	Latin small letter a with tilde
228	E4	11100100	ä	ä	ä	Latin small letter a with diaeresis
229	E5	11100101	å	å	å	Latin small letter a with ring above
230	E6	11100110	æ	æ	æ	Latin small letter ae
231	E7	11100111	ç	ç	ç	Latin small letter c with cedilla
232	E8	11101000	è	è	è	Latin small letter e with grave
233	E9	11101001	é	é	é	Latin small letter e with acute
234	EA	11101010	ê	ê	ê	Latin small letter e with circumflex
235	EB	11101011	ë	ë	ë	Latin small letter e with diaeresis
236	EC	11101100	ì	ì	ì	Latin small letter i with grave
237	ED	11101101	í	í	í	Latin small letter i with acute
238	EE	11101110	î	î	î	Latin small letter i with circumflex

DEC	HEX	BIN	Symbol	HTML Number	HTML Name	Description
239	EF	11101111	ï	ï	ï	Latin small letter i with diaeresis
240	F0	11110000	ð	ð	ð	Latin small letter eth
241	F1	11110001	ñ	ñ	ñ	Latin small letter n with tilde
242	F2	11110010	ò	ò	ò	Latin small letter o with grave
243	F3	11110011	ó	ó	ó	Latin small letter o with acute
244	F4	11110100	ô	ô	ô	Latin small letter o with circumflex
245	F5	11110101	õ	õ	õ	Latin small letter o with tilde
246	F6	11110110	ö	ö	ö	Latin small letter o with diaeresis
247	F7	11110111	÷	÷	÷	Division sign
248	F8	11111000	ø	ø	ø	Latin small letter o with slash
249	F9	11111001	ù	ù	ù	Latin small letter u with grave
250	FA	11111010	ú	ú	ú	Latin small letter u with acute
251	FB	11111011	û	û	û	Latin small letter u with circumflex
252	FC	11111100	ü	ü	ü	Latin small letter u with diaeresis
253	FD	11111101	ý	ý	ý	Latin small letter y with acute
254	FE	11111110	þ	þ	þ	Latin small letter thorn
255	FF	11111111	ÿ	ÿ	ÿ	Latin small letter y with diaeresis