Liberty BASIC Command Reference

Here is an alphabetical list of the Liberty BASIC commands and functions:

LOWER$(s$) s$ converted to all lowercase
more...

More Commands

LPRINT	print to hard copy
MENU	adds a pull-down menu to a window
MID$( )	return a substring from a string
NAME a$ AS b$	rename file named a$ to b$
NOMAINWIN	keep a program's main window from opening
NOTICE	open a notice dialog box
OPEN	open a file or window
Platform$	special variable containing platform name
PRINT	print to a file or window
PROMPT	open a prompter dialog box
PUT #h, n	puts a random access record n for #h
RADIOBUTTON	adds a radiobutton to a window
REM	adds a remark to a program
RETURN	return from a subroutine call
RIGHT$(s$, n)	n rightmost characters of s$
RND(n)	return pseudo-random seed
RUN s$, mode	run external program s$, with optional mode
SIN(n)	sine of n
SORT	sorts single and double dim'd arrays
STR$(n)	returns string equivalent of n
TAN(n)	tangent of n
TIME$( )	returns current time as string
TRACE n	sets debug trace level to n
TRIM$(s$)	returns s$ without leading/trailing spaces
UPPER$(s$)	s$ converted to all uppercase
USING( )	performs numeric formatting
VAL(s$)	returns numeric equivalent of s$
Version$	special variable containing LB version info
WHILE...WEND	performs looping action
WORD$(s$, n)	returns nth word from s$

ABS(n)

Description:

This function returns | n | (the absolute value of n).

Usage:

print abs(-5) produces: 5

print abs(6 - 13) produces: 7

ASC(s$)

Description:

This function returns the ASCII value of the first character of string s$.

Usage:

```
print asc( "A" )                    produces:   65

let name$ = "Tim"
firstLetter = asc(name$)
print firstLetter                   produces:   84

print asc( "" )                     produces:   0
```

ACS(n)

Description:

Returns the arc-cosine of the number n.

Usage:

```
.
.
for c = 1 to 45
   print "The arc-cosine of "; c; " is "; acs(c)
next c
.
.
```

Note: See also COS( )

ASN(n)

Description:

Returns the arcsine of the number n.

Usage:

```
.
.
for c = 1 to 45
   print "The arcsine of "; c; " is "; asn(c)
next c
.
.
```

Note: See also SIN( )

ATN(n)

Description:

Returns the arc-tangent of the number n.

Usage:

```
.
.
for c = 1 to 45
   print "The arctangent of "; c; " is "; atn(c)
next c
.
.
```

Note: See also TAN( )

BEEP

Description:

This command simply rings the system bell, as in CTRL-G

Usage:

```
   .
   .
[loop]
   input "Give me a number between 1 and 10?"; number
   if number < 1 or number > 10 then beep : print "Out of range!" : goto [loop]
   print "The square of "; number; " is "; number ^ 2
   .
   .
```

BMPBUTTON

BMPBUTTON #handle, filespec, return, corner, posx, posy

Description:

This statement lets you add bitmapped buttons to windows that you open. The main program window cannot have buttons added, but any window that you create via the OPEN command can have as many buttons as you want.

Usage:

Before you actually OPEN the window, each bitmapped button must be declared with a BMPBUTTON statement. Here is a brief description for each parameter as listed above:

#handle - You must use the same handle that will be used for the window that the button will belong to.

filespec - The full pathname of the *.bmp file containing the bitmap for the button you are creating. The button will be the same size as the bitmap.

return - Again, use only one word and do not bound it with quotes or use a string variable. If return is set to a valid branch label, then when the button is pressed, execution will restart there (just as with GOTO or GOSUB), but if return is not a valid branch label, then the value of return is used as input to a specified variable (as in input a$).

corner - UL, UR, LL, or LR specifies which corner of the window to anchor the button to. For example, if LR is used, then the button will appear in the lower right corner. UL = upper left, UR = upper right, LL = lower left, and LR = lower right

posx, posy - These parameters determine how to place the button relative to the corner it has been anchored to. For example if corner is LR, posx is 5, and posy is 5, then the button will be 5 pixels up and left of the lower right corner. Another way to use posx & posy is to use values less than one. For example, if corner is UL, posx is .9, and posy is .9, then the button will be positioned 9/10th of the distance of the window in both x and y from the upper left corner (and thus appear to be anchored to the lower right corner).

A collection of button *.bmp has been included with Liberty BASIC, including blanks. Windows Paint can be used to edit and make buttons for Liberty BASIC.

Program execution must be halted at an input statement in order for a button press to be read and acted upon.

See also: BUTTON, MENU

BUTTON

BUTTON #handle, label, return, corner, posx, posy

Description:

This statement lets you add buttons to windows that you open. The main program window cannot have buttons added, but any window that you create via the OPEN command can have as many buttons as you want.

Usage:

Before you actually OPEN the window, each button must be declared with a BUTTON statement. Here is a brief description for each parameter as listed above:

#handle - You must use the same handle that will be used for the window that the button will belong to.

label - Type the label desired for the button here. Do not bound the word with quotes, and do not use a string variable.

return - Again, use only one word and do not bound it with quotes or use a string variable. If return is set to a valid branch label, then when the button is pressed, execution will restart there (just as with GOTO or GOSUB), but if return is not a valid branch label, then the value of return is used as input to a specified variable (as in input a$).

corner - UL, UR, LL, or LR specifies which corner of the window to anchor the button to. For example, if LR is used, then the button will appear in the lower right corner. UL = upper left, UR = upper right, LL = lower left, and LR = lower right

posx, posy - These parameters determine how to place the button relative to the corner it has been anchored to. For example if corner is LR, posx is 5, and posy is 5, then the button will be 5 pixels up and left of the lower right corner. Another way to use posx & posy is to use values less than one. For example, if corner is UL, posx is .9, and posy is .9, then the button will be positioned 9/10th of the distance of the window in both x and y from the upper left corner (and thus appear to be anchored to the lower right corner).

Program execution must be halted at an input statement in order for a button press to be read and acted upon. See next page.

Here is a sample program:

```
' this button will be labeled Sample and will be located
' in the lower right corner.  When it is pressed, program
' execution will transfer to [test]

button #graph, Bell, [bell], LR, 5, 5

' this button will be labeled Example and will be located
```

```
   ' in the lower left corner.   When it is pressed, the string
   ' "Example" will be returned.

   button #graph, Quit, [quit], LL, 5, 5

   ' open a window for graphics
      open "Button Sample" for graphics as #graph

   ' print a message in the window
      print #graph, "\\This is a test"
      print #graph, "flush"

   ' get button input
[loop]
   input b$     ' stop and wait for a button to be pressed
   if b$ = "Example" then [example]
   goto [loop]

 ' the Sample button has been pressed, ring the terminal bell
 ' and close the window
[bell]
   beep
   close #graph
   end

 ' The Example button has been pressed, close the window
 ' without ringing the bell
[quit]
   close #graph
   end
```

Checkbox

CHECKBOX #handle.ext, "label", [set], [reset], xOrigin, yOrigin, width, height

Description:

Adds a checkbox control to the window referenced by #handle. Checkboxes have two states, set and reset. They are useful for getting input of on/off type information.

Here is a description of the parameters of the CHECKBOX statement:

"label" - This contains the visible text of the checkbox

[set] - This is the branch label to goto when the user sets the checkbox by clicking on it.

[reset] - This is the branch label to goto when the user resets the checkbox by clicking on it.

xOrigin - This is the x position of the checkbox relative to the upper left corner of the window it belongs to.

yOrigin - This is the y position of the checkbox relative to the upper left corner of the window it belongs to.

width - This is the width of the checkbox control

height - This is the height of the checkbox control

Usage: See the included program checkbox.bas for an example of how to use checkboxes

CHR$(n)

Description:

Returns a one character long string, consisting of the character represented on the ASCII table by the value n (0 - 255).

Usage:

```
' print each seperate word in text$ on its own line
text$ = "now is the time for all great men to rise"
for index = 1 to len(text$)
    c$ = mid$(text$, index, 1)
    ' if c$ is a space, change it to a carraige return
    if c$ = chr$(32) then c$ = chr$(13)
    print c$ ;
next index
```

Produces:

```
now
is
the
time
for
all
great
men
to
rise
```

CLOSE #handle

Description:

This command is used to close files and devices. This is the last step of a file read and/or write, or to close graphic, spreadsheet, or other windows when finished with them. If when execution of a program is complete there are any files or devices left open, Liberty BASIC will display a dialog informing you that it found it necessary to close the opened files or devices. This is designed as an aid for you so that you will be able to correct the problem. If on the other hand you choose to terminate the program early (this is done by closing the program's main window before the program finishes), then Liberty BASIC will close any open files or devices without posting a notice to that effect.

Usage:

```
open "Graphic" for graphics as #gWindow          ' open a graphics window
print #gWindow, "home"                           ' center the pen
print #gWindow, "down"                           ' put the pen down
for index = 1 to 100                             ' loop 100 times
   print #gWindow, "go "; index                  ' move the pen foreward
   print #gWindow, "turn 63"                     ' turn 63 degrees
next index
input "Press 'Return'."; r$                      ' this appears in main window
close #gWindow                                   ' close graphic window
```

CLS

Description:

Clears the main program window of text and sets the cursor back at the upper left hand corner. Useful for providing a break to seperate different sections of a program functionally. Additionally, since the main window doesn't actually discard past information on its own, the CLS command can be used to reclaim memory from your program by forcing the main window to dump old text.

Usage:

```
.
.
print "The total is: "; grandTotal
input "Press 'Return' to continue."; r$
cls
print "*** Enter Next Round of Figures ***"
.
.
```

CONFIRM

CONFIRM string; responseVar

Description:

This statement opens a dialog box displaying the contents of string and presenting two buttons marked 'Yes' and 'No'. When the selection is made, the string "yes" is returned if 'Yes' is pressed, and the string "no" is returned if 'No' is pressed. The result is placed in responseVar.

Usage:

[quit]

```
  ' bring up a confirmation box to be sure that
  ' the user wants to quit
  confirm "Are you sure you want to QUIT?"; answer$
  if answer$ = "no" then [mainLoop]
  end
```

COS(n)

Description:

Returns the cosine of the number n.

Usage:

```
.
.
for c = 1 to 45
   print "The cosine of "; c; " is "; cos(c)
next c
.
.
```

Note: See also SIN( ) and TAN( )

DATE$( )

Description:

Instead of adopting MBASIC's date$ variable, we decided to use a function instead, figuring that this might give us additional flexibility later. This function returns the current date in long format.

Usage:

```
print date$( )
```

Produces:

```
Feb 5, 1991
```

Or you can assign a variable the result:

```
d$ = date$( )
```

DIM

DIM array(size, size)

Description:

DIM sets the maximum size of an array. Any array can be dimensioned to have as many elements as memory allows. If an array is not DIMensioned explicitly, then the array will be limited to 10 elements, 0 to 9. Non DIMensioned double subscript arrays will be limited to 100 elements 0 to 9 by 0 to 9.

Usage:

```
print "Please enter 10 names."
for index = 0 to 9
   input names$ : name$(index) = name$
next index
```

The FOR . . . NEXT loop in this example is limited to a maximum value of 9 because the array names$( ) is not dimensioned, and therefore is limited to 10 elements. To remedy this problem, we can add a DIM statement, like so:

```
dim names$(20)
print "Please enter 20 names."
for index = 0 to 19
   input names$ : names$(index) = name$
next index
```

Double subscripted arrays can store information more flexibly, like so:

```
dim customerInfo$(10, 5)
print "Please enter information for 10 customers."
for index = 0 to 9
   input "Customer name >"; info$ : customerInfo$(index, 0) = info$
   input "Address >"; info$ : customerInfo$(index, 1) = info$
   input "City >"; info$ : customerInfo$(index, 2) = info$
   input "State >"; info$ : customerInfo$(index, 3) = info$
   input "Zip >"; info$ : customerInfo$(index, 4) = info$
next index
```

Drive$

Description:

Drives$ is a system variable. You can operate on it like any other variable. Use it in expressions, print it, perform functions on it, etc. It's special in that it contains the drive letters for all the drives installed on in the computer in use.

For example:

```
print Drives$
```

Would in many cases produce:

```
a: b: c:
```

Or you could use it to provide a way to select a drive like this:

```
'a simple example illustrating the use of the Drives$ variable
dim letters$(25)
index = 0
while word$(Drives$, index + 1) <> ""
      letters$(index) = word$(Drives$, index + 1)
      index = index + 1
wend

statictext #select, "Double-click to pick a drive:", 10, 10, 200, 20
listbox #select.list, letters$(, [selectionMade], 10, 35, 100, 150
open "Scan drive" for dialog as #select

input r$

[selectionMade]

close #select
end
```

DUMP

Description:

Forces anything that has been LPRINTed to be sent to the Print Manager.

Usage:

```
'sample program using LPRINT and DUMP
open "c:\autoexec.bat" for input as #source
while eof( #source ) = 0
   line input #source, text$          'print each line
   lprint text$
wend
close #source
dump            'force the print job
end
```

Note: see also LPRINT

EOF( )

Description:

Used to determine when reading from a sequential file whether the end of the file has been reached. If so, -1 is returned, otherwise 0 is returned.

Usage:

```
   open "testfile" for input as #1
   if eof(#1) < 0 then [skipIt]
[loop]
   input #1, text$
   print text$
   if eof(#1) = 0 then [loop]
[skipIt]
   close #1
```

END

Description:

Used to immediately terminate execution of a program. If any files or devices are still open (see CLOSE) when execution is terminated, then Liberty BASIC will close them for you and present you with a dialog expressing this fact. It is good programming practice to close files and devices before terminating execution.

Note: The STOP statement is functionally identical to END and is interchangable

Usage:

```
   .
   .
   print "Preliminary Tests Complete."
[askAgain]
   input "Would you like to continue (Y/N) ?"; yesOrNo$
   yesOrNo$ = left$(yesOrNo$, 1)
   if yesOrNo$ = "y" or yesOrNo$ = "Y" then [continueA]
   ifYesOrNo$ = 'n" or yesOrNo$ = "N" then end
   print "Please answer Y or N."
   goto [askAgain]
[continueA]
   .
   .
```

EXP(n)

Description:

This function returns e ^ n, e being 2.7182818 . . .

Usage:

print exp(5) produces: 148.41315

FIELD

FIELD #handle, # as varName, # as varName, . . .

Description:

FIELD is used with an OPEN "filename.ext" for random as #handle statement to specify the fields of data in each record of the opened file. For example in this program FIELD sets up 6 fields of data, each with an appropriate length, and associates each with a string variable that holds the data to be stored in that field:

```
   open "custdata.001" for random as #cust len = 70     ' open as random access
   field #cust, 20 as name$, 20 as street$, 15 as city$, 2 as state$, 10 as zip$, 3 as age

[inputLoop]
   input "Name >"; name$
   input "Street >"; street$
   input "City >"; city$
   input "State >"; state$
   input "Zip Code >"; zip$
   input "Age >"; age

   confirm "Is this entry correct?"; yesNo$     ' ask if the data is entered correctly
   if   yesNo$ = "no" then [inputLoop]

   recNumber = recNumber + 1     ' add 1 to the record # and put the record
   put #cust, recNumber

   confirm "Enter more records?"; yesNo$     ' ask whether to enter more records
   if yesNo$ = "yes" then [inputLoop]

   close #cust     ' end of program, close file
   end
```

Notice that Liberty BASIC permits the use of numeric variables in FIELD (eg. age), and it allows you to PUT and GET with both string and numeric variables, automatically, without needing LSET, RSET, MKI$, MKS$, MKD$, CVI, CVS, & CVD that are required with Microsoft BASICs.

Note: See also PUT and GET

FILEDIALOG

FILEDIALOG titleString, templateString, receiverVar$

Description:

This command opens a file dialog box. The titleString is used to label the dialog box. The templateString is used as a filter to list only files matching a wildcard, or to place a full suggested filename.

The box lets you navigate around the directory structure, looking at files that have a specific extension. You can then select one, and the resulting full path specification will be placed into receiverVar$, above.

The following example would produce a dialog box asking the user to select a text file to open:

```
filedialog "Open text file", "*.txt", fileName$
```

If then summary.txt were selected, and OK clicked, then program execution would resume after placing the string "c:\liberty\summary.txt" into fileName$.

If on the other hand Cancel were clicked, then an empty string would be placed into fileName$. Program execution would then resume.

Look at the program grapher1.bas for a practical application of this command.

FILES

Description:

The FILES statement collects file and directory information from any disk and or directory and fills a double-dimensioned array with the information.

Usage:

```
'you must predimension an array, even though FILES will redimension
'it to fit the information it provides
dim info$(10, 10)
.
.
.
files "c:\", info$
```

The above FILES statement will fill info$( ) in this fashion:

info$(0, 0) will contain a string specifying the qty of files found
info$(0, 1) will contain a string specifying the qty of subdirectories found
info$(0, 2) will contain the drive spec
info$(0, 3) will contain the directory path

Starting at info$(1, x) you will have file information like so:

info$(1, 0) will contain the file name
info$(1, 1) will contain the file size
info$(1, 2) will contain the file date/time stamp

Knowing from info$(0, 0) how many files we have (call it f), we know that our subdirectory information starts at f+1, so:

info$(f+1, 0) contains the complete path of a directory entry (\work\math)
info$(f+1, 1) contains just the name of the directory in specified (math)

See the dir.bas example included.

FOR...NEXT

Description:

The FOR . . . NEXT looping construct provides a way to execute code a specific amount of times. A starting and ending value are specified like so:

```
for var = 1 to 10
  {BASIC code}
next var
```

In this case, the {BASIC code} is executed 10 times, with var being 1 the first time, 2 the second, and on through 10 the tenth time. Optionally (and usually) var is used in some calculation(s) in the {BASIC code}. For example if the {BASIC code} is print var ^ 2, then a list of squares for var will be displayed upon execution.

The specified range could just as easily be 2 TO 20, instead of 1 TO 10, but since the loop always counts +1 at a time, the first number must be less than the second. The way around this limitation is to place STEP n at the end of for FOR statement like so:

```
for index = 20 to 2 step -1
  {BASIC code}
next index
```

This would loop 19 times returning values for index that start with 20 and end with 2. STEP can be used with both positive and and negative numbers and it is not limited to integer values. For example:

```
for x = 0 to 1 step .01
  print "The sine of "; x; " is "; sin(x)
next x
```

NOTE: It is not recommended to pass control of a program out of a FOR . . . NEXT loop using GOTO (GOSUB is acceptable). Liberty BASIC may behave unpredictably. For example:

```
for index = 1 to 10
  print "Enter Customer # "; index
  input customer$
  if customer$ = "" then [quitEntry]     ' <- don't cut out of a for ... next loop like this
  cust$(index) = customer$
next index
[quitEntry]
```

. . . is not allowed! Rather use while ... wend:

```
index = 1
while customer$ <> "" and index <= 10
  print "Enter Customer # "; index
  input customer$
  cust$(index) = customer$
  index = index + 1
wend
```

GET

GET #handle, recordNumber

Description:

GET is used after a random access file is opened to get a record of information (see FIELD) out of the file from a specified position.

Usage:

```
open "custdata.001" for random as #cust len = 70      ' open random access file
field #cust, 20 as name$, 20 as street$, 15 as city$, 2 as state$, 10 as zip$, 3 as age

' get the data from record 1
get #cust, 1

print name$
print street$
print city$
print state$
print zip$
print age

close #cust
end
```

Note: See also PUT, FIELD

GETTRIM

GETTRIM #handle, recordNumber

Description:

The GETTRIM command is exactly like the GET command, but when data is retrieved, all leading and trailing blank space is removed from all data fields before being committed to variables.

Note: see also GET

GOSUB label

Description:

GOSUB causes execution to proceed to the program code following the label if it exists, using the form 'GOSUB label'. The label can be either a traditional line number or a branch label in the format [???????] where the ?'s can be any upper/lowercase letter combination. Spaces and numbers are not allowed.

Here are some valid branch labels: [mainMenu] [enterLimits] [repeatHere]
Here are some invalid branch labels: [enter limits] mainMenu [1moreTime]

After execution is transferred to the point of the branch label, then each statement will be executed in normal fashion until a RETURN is encountered. When this happens, execution is transferred back to the statement immediately after the GOSUB. The section of code between a GOSUB and its RETURN is known as a 'subroutine.' One purpose of a subroutine is to save memory by having only one copy of code that is used many times throughout a program.

Usage:

```
   .
   .
   print "Do you want to continue?"
   gosub [yesOrNo]
   if answer$ = "N" then [quit]
   print "Would you like to repeat the last sequence?"
   gosub [yesOrNo]
   if answer$ = "Y" then [repeat]
   goto [generateNew]

[yesOrNo]
   input answer$
   answer$ = left$(answer$, 1)
   if answer$ = "y" then answer$ = "Y"
   if answer$ = "n" then answer$ = "N"
   if answer$ = "Y" or answer$ = "N" then return
   print "Please answer Y or N."
   goto [yesOrNo]
   .
   .
```

You can see how using GOSUB [yesOrNo] in this case saves many lines of code in this example. The subroutine [yesOrNo] could easily be used many other times in such a hypothetical program, saving memory and reducing typing time and effort. This reduces errors and increases productivity.

Note: see also GOTO

GOTO label

Description:

GOTO causes Liberty BASIC to proceed to the program code following the label if one exists, using the form 'GOTO label'. The label can be either a traditional line number or a branch label in the format [????????] where the ?'s can be any upper/lowercase letter combination. Spaces and digits are not allowed.

Here are some valid branch labels: [mainMenu] [enterLimits] [repeatHere]
Here are some invalid branch labels: [enter limits] mainMenu [1moreTime]

Usage:

```
   .
   .
[repeat]
   .
   .
[askAgain]
   print "Make your selection (m, r, x)."
   input selection$
   if selection$ = "M" then goto [menu]
   if selection$ = "R" then goto [repeat]
   if selection$ = "X" then goto [exit]
   goto [askAgain]
   .
   .
[menu]
   print "Here is the main menu."
   .
   .
[exit]
   print "Okay, bye."
   end
```

Notes:

In the lines containing IF . . . THEN GOTO, the GOTO is optional.

The expression IF . . . THEN [menu] is just as valid as
IF . . . THEN GOTO [menu]. But in the line GOTO [askAgain], the GOTO is required.

See also GOSUB

IF ... THEN

IF expression THEN expression(s)

Description:

The purpose of IF . . . THEN is to provide a mechanism for your computer software to make decisions based on the data available. A decision-making mechanism is used in very simple situations and can be used in combinations to engineer solutions to problems of great complexity.

The expression (see above) is a boolean expression (meaning that it evaluates to a true or false condition). In this expression we place the logic of our decision-making process. For example, if we are writing a inventory application, and we need to know when any item drops below a certain level in inventory, then our decision-making logic might look like this:

```
.
.
if level <= reorderLevel then expression(s)
next BASIC program line
.
.
```

The 'level <= reorderLevel' part of the above expression will evaluate to either true or false. If the result was true, then the expression(s) part of that line (consisting of a branch label or any valid BASIC statements) will be executed. Otherwise execution will immediately begin at the next BASIC program line.

The following are permitted:

```
if a < b then [lessThan]
```

This causes program execution to begin at branch label [lessThan] if a is less than b.

```
if sample < lowLimit or sample > highLimit then beep : print"Out of range!"
```

This causes the terminal bell to ring and the message Out of range! to be displayed if sample is less than lowLimit or greater then highLimit.

Note: see also IF...THEN...ELSE

IF...THEN...ELSE

IF expression THEN expression(s)1 ELSE expression(s)2

Description:

This extended form of IF . . . THEN adds expressiveness and simplifies coding of some logical decision-making software. Here is an example of its usefulness.

Consider:

```
[retry]
   input"Please choose mode, (N)ovice or e(X)pert?"; mode$
   if len(mode$) = 0 then print "Invalid entry! Retry" : goto [retry]
   mode$ = left$(mode$, 1)
   if instr("NnXx", mode$) = 0 then print "Invalid entry! Retry" : goto [retry]
   if instr("Nn", mode$) > 0 then print "Novice mode" : goto [main]
   print "eXpert mode"
[main]
   print "Main Selection Menu"
```

Look at the two lines before the [main] branch label. The first of these two lines is required to branch over the next line. These lines can be shortened to one line as follows:

```
   if instr("Nn",mode$)> 0 then print "Novice mode" else print "eXpert mode"
```

Some permitted forms are as follows:

```
   if a < b then statement else statement
   if a < b then [label] else statement
   if a < b then statement else [label]
   if a < b then statement : statement else statement
   if a < b then statement else statement : statement
   if a < b then statement : goto [label] else statement
   if a < b then gosub [label1] else gosub [label2]
```

Any number of variations on these formats are permissible. The a < b boolean expression is of course only a simple example chosen for convenience.

You must replace it with the correct expression to suit your problem.

Note: see also IF...THEN

INPUT

INPUT #handle "string expression"; variableName

Description:

This command has several possible forms:

  input var

    - stop and wait for user to enter data in the program's
    main window and press the 'Return' key, then assign
    the data entered to var.

  input "enter data"; var

    - display the string "enter data" and then stop
    and wait for user to enter data in the program's main window
    and press 'Return', then assign the data entered to var.

  input #name, var

    - Get the next data item from the open file or device using handle
    named #handle and assign the data to var. If no device or file exists
    that uses the handle named #handle, then return an error.

  input #name, var1, var2

    - The next two data items are fetched and assigned to var1 and var2.

  line input #name, var$

     - The line input statement will read from the file, ignoring commas in the
    input stream and completing the data item only at the next carraige
    return or at the end of file. This is useful for reading text with embedded
    commas

Usage:

```
  'Display a text file
  input "Please type a filename >";  filename$
  open filename$ for input as #text
[loop]
  if eof(#text) <> 0 then [quit]
  input #text, item$
  print item$
  goto [loop]
[quit]
  close #text
  print "Done."
  end
```

Note: In Liberty BASIC, INPUT cannot be used to input data directly into arrays, only into the simpler variables.

```
input a$(1)                          - is illegal
input string$ : a$(1) = string$      - use this instead
```

Most versions of Microsoft BASIC implement INPUT to automatically place a question mark on the display in front of the cursor when the user is prompted for information like so:

```
input "Please enter the upper limit"; limit
```

produces:

```
Please enter the upper limit ? |
```

Liberty BASIC permits you the luxury of deciding for yourself whether the question mark appears at all.

```
input "Please enter the upper limit :"; limit
```

produces:

```
Please enter the upper limit: |
```

and: input limit produces simply:

```
? |
```

In the simple form input limit, the question mark is inserted automatically, but if you do specify a prompt, as in the above example, only the contents of the prompt are displayed, and nothing more. If for some reason you wish to input without a prompt and without a question mark, then the following will achieve the desired effect:

```
input ""; limit
```

Additionally, in most Microsoft BASICs, if INPUT expects a numeric value and a non numeric or string value is entered, the user will be faced with a comment something like 'Redo From Start', and be expected to reenter. Liberty BASIC does not automatically do this, but converts the entry to a zero value and sets the variable accordingly. This is not considered a problem but rather a language feature, allowing you to decide for yourself how your program will respond to the situation.

One last note: In Liberty BASIC input prompt$; limit is also valid. Try:

```
prompt$ = "Please enter the upper limit:"
input prompt$; limit
```

INPUT$( )

INPUT$(#handle, items)

Description:

Permits the retrieval of a specified number of items from an open file or device using #handle. If #handle does not refer to an open file or device then an error will be reported.

Usage:

```
   'read and display a file one character at a time
   open "c:\autoexec.bat" for input as #1
[loop]
      if eof(#1) <> 0 then [quit]
      print input$(#1, 1);
      goto [loop]
[quit]
      close #1
      end
```

For most devices (unlike disk files), one item does not refer a single character, but INPUT$( ) may return items more than one character in length. In most cases, use of INPUT #handle, varName works just as well or better for reading devices.

INSTR( )

INSTR(string1, string2, starting)

Description:

This function returns the position of string2 within string1. If string2 occurs more than once in string 1, then only the position of the leftmost occurance will be returned. If starting is included, then the search for string2 will begin at the position specified by starting.

Usage:

```
print instr("hello there", "lo")
```

produces: 4

```
print instr("greetings and meetings", "eetin")
```

produces: 3

```
print instr("greetings and meetings", "eetin", 5)
```

produces: 16

If string2 is not found in string1, or if string2 is not found after starting, then INSTR( ) will return 0.

```
print instr("hello", "el", 3)
```

produces: 0

and so does:

```
print instr("hello", "bye")
```

INT(n)

Description:

This function removes the fractional part of number, leaving only the whole number part behind.

Usage:

```
[retry]
   input "Enter an integer number>"; i
   if i<>int(i) then bell: print i; " isn't an integer! Re-enter.": goto [retry]
```

KILL

KILL "filename.ext"

Description:

This command deletes the file specified by filename.ext. The filename can include a complete path specification.

LEFT$( )

LEFT$(string, number)

Description:

This function returns from string the specified number of characters starting from the left. So if string is "hello there", and number is 5, then "hello" would be the result.

Usage:

```
[retry]
   input "Please enter a sentence>"; sentence$
   if sentence$ = "" then [retry]
   for i = 1 to len(sentence$)
      print left$(sentence$, i)
   next i
```

Produces:

```
   Please enter a sentence>That's all folks!
   T
   Th
   Tha
   That
   That'
   That's
   That's_
   That's a
   That's al
   That's all
   That's all_
   That's all f
   That's all fo
   That's all fol
   That's all folk
   That's all folks
   That's all folks!
```

Note: If number is zero or less, then "" (an empty string) will be returned. If the number is greater than or equal to the number of characters in string, then string will be returned.

See also MID$( ) and RIGHT$( )

LEN( )

LEN(string)

Description:

This function returns the length in characters of string, which can be any valid string expression.

Usage:

```
prompt "What is your name?"; yourName$
print "Your name is "; len(yourName$); " letters long"
```

LET var = expr

Description:

LET is an optional prefix for any BASIC assignment expression. Most do leave the word out of their programs, but some prefer to use it.

Usage:

Either is acceptable:

```
let name$ = "John"
```
or
```
name$ = "John"
```

Or yet again:

```
let c = sqr(a^2 + b^2)
```
or
```
c = sqr(a^2 + b^2)
```

LINE INPUT

See INPUT

LOADBMP

LOADBMP "name", "filename.bmp"

Description:

This command loads standard Windows *.bmp bitmap file into Liberty BASIC. The "name" is a string you would choose to describe the bitmap you're loading, and the "filename.bmp" is the actual name of the bitmap file. Once loaded, the bitmap can then be displayed in a graphics window type using the drawbmp command (see help file GUI Programming/Window Types/View Graphics Window Commands).

Usage:

See the sample program ttt.bas for an example using LOADBMP.

LOF( )

LOF(#handle)

Description:

Returns the # of bytes contained in the file referenced by #handle.

Usage:

```
open "\autoexec.bat" for input as #1
qtyBytes = lof(#1)
for x = 1 to qtyBytes
      print input$(#1, 1) ;
next x
close #1
end
```

LOG(n)

Description:

This function returns the natural log of n.

Usage:

print log(7) produces: 1.9459101

LPRINT

LPRINT expr

Description:

This statement is used to send data to the default printer (as determined by the Windows Print Manager). A series of expressions can follow LPRINT (there does not need to be any expression at all), each seperated by a semicolon. Each expression is sent in sequence. When you are finished sending data to the printer, you should commit the print job by using the DUMP statement. Liberty BASIC will eventually send your print job, but DUMP forces the job to finish.

Usage:

```
lprint "hello world"            'This prints hello world
dump

lprint "hello ";                  'This also prints hello world
lprint "world"
dump

age = 23
lprint "Ed is "; age; " years old"        'This prints Ed is 23 years old
dump
```

Note: see also PRINT, DUMP

MENU

MENU #handle, title, text, branchLabel, text, branchLabel, | , . . .

Description:

Adds a pull down menu to the window at #handle. Title specifies the title of the menu, as seen on the menu bar of the window, and each text, branchLabel pair after the title adds a menu item to the menu, and tells Liberty BASIC where to branch to when the menu item is chosen. The | character can optionally be placed between menu items, to cause a seperating line to be added between the items with the menu is pulled down.

As an example, if you wanted to have a graphics window opened, and then be able to pull down some menus to control color and geometric objects, our opening code might look like this:

```
menu #geo, &Colors, &Red, [setRed], &Green, [setGreen], &Blue, [setBlue]
menu #geo, &Shapes, &Rectangle, [asRect], &Triangle, [asCircle], &Line, [asLine]
open "Geometric wite-board" for graphics_nsb as #geo
input r$  ' stop and wait for a menu item to be chosen
```

Notice that the MENU lines must go before the OPEN statement, and must use the same handle as the window that it will be associated with (#geo in this case). This is the same with the BUTTON statement (see BUTTON). See that execution must be stopped at an input statement for the menu choice to be acted upon. This is also the same as with BUTTON.

Also notice that the & character placed in the title and text items for the menu determines the accelerator placement for each menu. Try experimenting.

MID$( )

MID$(string, index, number)

Description:

Permits the extraction of a sequence of characters from string starting at index. If number is not specified, then all the characters from index to the end of the string are returned. If number is specified, then only as many characters as number specifies will be returned, starting from index.

Usage:

```
print mid$("greeting Earth creature", 10, 5)
```

Produces:

```
Earth
```

And:

```
string = "The quick brown fox jumped over the lazy dog"
for i = 1 to len(string$) step 5
   print mid$(string$, i, 5)
next i
```

Produces:

```
The_q
uick_
brown
_fox_
jumpe
d_ove
r_the
_lazy
_dog
```

Note:

See also LEFT$( ) and RIGHT$( )

NAME...AS

NAME stringExpr1 AS stringExpr2

Description:

This command renames the file specified in the string expression stringExpr1 to stringExpr2. StringExpr1 can represent any valid filename that is not a read-only file, and stringExpr2 can be any valid filename as long as it doesn't specify a file that already exists.

Usage:

```
'rename the old file as a backup
name rootFileName$ + ".fre" as rootFileName$ + ".bak"
'open a new file and write data
open rootFileName$ + ".fre" for output as #disk
.
.
```

NOMAINWIN

Description:

This command instructs Liberty BASIC not to open a main window for the program that includes this statement. Some simple programs which do not use seperate windows for graphics, spreadsheet, or text may use only the main window. Other programs may not need the main window to do their thing, and so simply including NOMAINWIN somewhere in your program source will prevent the window from opening.

If NOMAINWIN is used, when all other windows owned by that program are closed, then the program terminates execution automatically.

It is often better to place a NOMAINWIN statement in your program after it is completed and debugged, so that you can easily terminate an errant program just by closing its main window.

For examples of the usage of NOMAINWIN, examine the included Liberty BASIC programs (buttons1.bas for example).

NOTICE

Description:

This command pops up a dialog box which displays "string expression" and which has a button OK which the user presses after the message is read. Pressing Enter also closes the dialog box.

Two forms are allowed. If "string expression" has no Cr character (ASCII 13), then the title of the dialog box will be 'Notice' and "string expression" will be the message displayed inside the dialog box. If "string expression" does have a Cr character, then the part of "string expression" before Cr will be used as the title for the dialog box, and the part of "string expression" after Cr will be displayed as the message inside.

Usage:

```
notice "Super Stats is Copyright 1992, Mathware"
```

Or:

```
notice "Fatal Error!" + chr$(13) + "The entry buffer is full!"
```

OPEN

OPEN string FOR purpose AS #handle {LEN = #}

Description:

This statement has many functions. It can be used to open disk files, or to open windows of several kinds.

Using OPEN with Disk files:

A typical OPEN used in disk I/O looks like this:

```
OPEN "\autoexec.bat" for input as #read
```

This example illustrates how we would open the autoexec.bat file for reading. As you can see, string in this case is "\autoexec.bat", purpose is input, and #handle is read.

string - this must be a valid pathname. If the file does not exist, it will be created.

purpose - must be input, output, or random

#handle - use a unique descriptive word, but must start with a #. This special handle is used to identify the open file in later program statements

LEN = # - this is an optional addition for use only when opening a random access file. The # determines how many characters long each record in the file is. If this is not specified, the default length is 128 characters. See FIELD, GET, and PUT.

Using OPEN with windows:

A typical OPEN used in windows looks like this:

```
OPEN "Customer Statistics Chart" for graphics as #csc
```

This example illustrates how we would open a window for graphics. Once the window is open, there are a wide range of commands that can be given to it. As you can see, string in this case is "Customer Statistics Chart", which is used as the title of the window, purpose is graphics (open a window for graphics), and the #handle is #csc (derived from Customer Statistics Chart), which will be used as an identifier when sending commands to the window.

string - can be any valid BASIC string. used to label the window
purpose - there are a several of possibilities here:
graphics, spreadsheet, text
any of these can end in _fs, _nsbars (or other suffixes)
#handle - as above, must be a unique, descriptive word starting with #

Note: Any opened file or window must be closed before program execution is finished. See CLOSE

PRINT

PRINT #handle, pression ; expression(s) ;

Description:

This statement is used to send data to the main window, to a disk file, or to other windows. A series of expressions can follow PRINT (there does not need to be any expression at all), each seperated by a semicolon. Each expression is displayed in sequence. If the data is being sent to a disk file, or to a window, then #handle must be present.

PRINTing to a the main window:

When the expressions are displayed, then the cursor (that blinking vertical bar |) will move down to the next line, and the next time information is sent to the window, it will be placed on the next line down. If you do not want the cursor to move immediately to the next line, then add an additional semicolor to the end of the list of expressions. This prevents the cursor from being moved down a line when the expressions are displayed. The next time data is displayed, it will be added onto the end of the line of data displayed previously.

Usage:	Produces:
print "hello world"	hello world
print "hello "; print "world"	hello world
age = 23 print "Ed is "; age; " years old"	Ed is 23 years old

When sending to a disk file and in regard to the use of the semicolon at the end of the expression list, the rules are similar (only you don't see it happen on the screen). When printing to a window, the expressions sent are usually commands to the window (or requests for information from the window). For more information, see help file GUI Programming.

PROMPT

PROMPT string; responseVar

Description:

The PROMPT statement opens a dialog box, displays string, and waits for the user to type a response and press 'Return' (or press the OK or Cancel button). The entered information is placed in responseVar. If Cancel is pressed, then a string of zero length is returned. If responseVar is set to some string value before PROMPT is executed, then that value will become the 'default' or suggested response. This means that when the dialog is opened, the contents of responseVar will already be entered as a response for the user, who then has the option to either type over that 'default' response, or to press 'Return' and accept it.

Usage:

```
   .
   .
   response$ = "C:"
   prompt "Search on which Drive? A:, B:, or C:"; response$
[testResponse]
   if response$ = "" then [cancelSearch]
   if len(response$) = 2 and instr("A:B:C:", response$) > 0 then [search]
   prompt "Unacceptable response.  Please try again. A:, B:, or C:"; again$
   goto [testResponse]

[search]
   print "Starting search . . . "
   .
   .
```

PUT

PUT #handle, n

Description:

PUT is used after a random access file is opened to place a record of information (see FIELD) into the file #handle at record n. For example:

```
open "custdata.001" for random as #cust len = 70     ' open a random access file
field #cust, 20 as name$, 20 as street$, 15 as city$, 2 as state$, 10 as zip$, 3 as age

' enter data into customer variables
input name$
.
.
' put the data into record 1
put #cust, 1

close #cust
end
```

Note: See also GET, FIELD

RADIOBUTTON

RADIOBUTTON #handle.ext, "label", [set], [reset], xOrigin, yOrigin, width, height

Description:

Adds a radiobutton control to the window referenced by #handle. Radiobuttons have two states, set and reset. They are useful for getting input of on/off type information.

All radiobuttons on a given window are linked together, so that if you set one by clicking on it, all the others will be reset.

Here is a description of the parameters of the RADIOBUTTON statement:

"label"- This contains the visible text of the radiobutton

[set] - This is the branch label to goto when the user sets the radiobutton by clicking on it.

[reset] - This is the branch label to goto when the user resets the radiobutton by clicking on it. (this doesn't actually do anything because radiobuttons can't be reset by clicking on them).

xOrigin- This is the x position of the radiobutton relative to the upper left corner of the window it belongs to.

yOrigin- This is the y position of the radiobutton relative to the upper left corner of the window it belongs to.

width - This is the width of the radiobutton control

height - This is the height of the radiobutton control

Usage:

See the included program radiobtn.bas for an example of how to use radiobuttons.

Note: see also CHECKBOX

REM

REM comment

Description:

The REM statement is used to place comments inside of code to clearly explain the purpose of each section of code. This is useful to both the programmer who writes the code or to anyone who might later need to modify the program. Use REM statements liberally. There is a shorthand way of using REM, which is to use the ' (apostrophe) character in place of the word REM. This is cleaner to look at, but use whichever you prefer. Unlike other BASIC statements, with REM you cannot add another statement after it on the same line using a colon (:) to seperate the statements. The rest of the line becomes part of the REM statement.

Usage:

```
rem    let's pretend that this is a comment for the next line
print "The mean average is "; meanAverage
```

Or:

```
' let's pretend that this is a comment for the next line
print "The strength of the quake was "; magnitude
```

This doesn't work:

```
rem    thank the user : print "Thank you for using Super Stats!"
```

(even the print statement becomes part of the REM statement)

Note:

When using ' instead of REM at the end of a line, the statement seperator :
is not required to seperate the statement on that line from its comment.

For example:

```
print "Total dollar value: "; dollarValue : rem    print the dollar value
```

Can also be stated:

```
print "Total dollar value: "; dollarValue    ' print the dollar value
```

Notice that the : is not required in the second form.

RETURN

RETURN

See GOSUB

RIGHT$( )

RIGHT$(string, number)

Description:

Returns a sequence of characters from the right hand side of string using number to determine how many characters to return. If number is 0, then "" (an empty string) is returned. If number is greater than or equal to the number of characters in string, then string will itself be returned.

Usage:

```
print right$("I'm right handed", 12)
```

Produces:

```
right handed
```

And:

```
print right$("hello world", 50)
```

Produces:

```
hello world
```

Note: See also LEFT$( ) and MID$( )

RND( )

RND(number)

Description:

This function returns a pseudo random number between 0 and 1. This can be useful in writing games and some simulations. The particular formula used in this release might more accurately be called an arbitrary number generator (instead of random number generator), since if a distribution curve of the output of this function were plotted, the results would be quite uneven. Nevertheless, this function should prove more than adequate (especially for game play).

In MBASIC it makes a difference what the value of parameter number is, but in Liberty BASIC, it makes no difference. The function will always return an arbitrary number between 0 and 1.

Usage:

```
' print ten numbers between one and ten
for a = 1 to 10
      print int(rnd(1)*10) + 1
next a
```

RUN

RUN stringExpr1 [, mode]

Description:

This command runs external programs. StringExpr1 should represent the full path and filename of a Windows or DOS executable program, a Liberty BASIC *.TKN file, or a *.BAT file. This is not a SHELL command, so you must provide the name of a program or batch file, not a DOS command (like DIR, for example). Execution of an external program does not cause the calling Liberty BASIC program to cease executing.

Here are two examples:

```
RUN "QBASIC.EXE"       ' run Microsoft's QBASIC

RUN "WINFILE.EXE", SHOWMAXIMIZED    ' run the File Manager maximized
```

Notice in the second example we can include an additional parameter. This is because we are running a Windows program. Here is a list of the valid parameters we can include when running Windows programs:

HIDE
SHOWNORMAL (this is the default)
SHOWMINIMIZED
SHOWMAXIMIZED
SHOWNOACTIVE
SHOW
MINIMIZE
SHOWMINNOACTIVE
SHOWNA
RESTORE

SIN(n)

Description:

This function return the sine of n.

Usage:

```
.
.
for t = 1 to 45
   print "The sine of "; t; " is "; sin(t)
next t
.
.
```

Note: See also COS( ) and TAN( )

SORT

SORT arrayName(, start, end, [column]

Description:

This command sorts both doubleand single dimensioned arrays. Arrays can be sorted in part or in whole, and with double dimensioned arrays, the specific column to sort by can be declared. When this option is used, all the rows are sorted against each other according to the items in the specified column.

Usage:

Here is the syntax for the sort command:

```
sort arrayName(, i, j, [,n]
```

This sorts the array named arrayName(starting with element i, and ending with element j. If it is a double dimensioned array then the column parameter tells which nth element to use as a sort key. Each WHOLE row moves with its corresponding key as it moves during the sort. So let's say you have a double dimensioned array holding sales rep activity:

```
repActivity(x, y)
```

So you're holding data, one record per x position, and your record keys are in y. So for example:

```
repActivity(1,1) = "Tom Maloney" : repActivity(1,2) = "01-09-93"
repActivity(2,1) = "Mary Burns" : repActivity(2,2) = "01-10-93"
  .
  .
  .
repActivity(100,1) = "Ed Dole" : repActivity(100,2) = "01-08-93"
```

So you want to sort the whole 100 items by the date field. This is how the command would look:

```
sort repActivity(, 1, 100, 2
```

If you wanted to sort by name instead, then change the 2 to a 1, like this:

```
sort repActivity(, 1, 100, 1
```

STR$(n)

Description:

This function returns a string expressing the result of numericExpression. In MBASIC, this function would always return a string representation of the expression and it would add a space in front of that string. For example in MBASIC:

```
print len(str$(3.14))
```

Would produce the number 5 (a space followed by 3.14 for a total of 5 characters).

Liberty BASIC leaves it to you to decide whether you want that space or not. If you don't want it, then you need not do anything at all, and if you do want it, then this expression will produce the same result under Liberty BASIC:

```
print len(" " + str$(3.14))
```

Usage:

```
   .
   .
[kids]
   ' use str$( ) to validate entry
   input "How many children do you have?"; qtyKids
   qtyKids$ = str$(qtyKids)
   ' if the entry contains a decimal point, then the response is no good
   if instr(qtyKids$, ".") > 0 then print "Bad response. Reenter." : goto [kids]
   .
   .
```

TAN(n)

Description:

This function returns the tangent of n.

Usage:

```
.
.
for t = 1 to 45
   print "The tangent of "; t; " is "; tan(t)
next t
.
.
```

Note: See also SIN( ) and COS( )

TIME$( )

Description:

This function returns a string representing the current time of the system clock in 24 hour format. This function replaces the time$ variable used in MBASIC. See also DATE$( ).

Usage:

```
.
.
' display the opening screen
print "Main selection screen                Time now: "; time$( )
print
print "1. Add new record"
print "2. Modify existing record"
print "3. Delete record"
.
.
```

TRACE

TRACE number

Description:

This statement sets the trace level for its application program. This is only effective if the program is run using the Debug menu selection (instead of RUN). If Run is used, then any TRACE statements are ignored.

There are three trace levels: 0, 1, and 2. Here are the effects of these levels:

0 = single step mode or STEP
1 = animated trace or WALK
2 = full speed no trace or RUN

When any Liberty BASIC program first starts under Debug mode, the trace level is always initially 0. You can then click on any of the three buttons (STEP, WALK, RUN) to determine what mode to continue in. When a TRACE statement is encountered, the trace level is set accordingly, but you can recover from this new trace level by clicking again on the desired button.

If you are having trouble debugging code at a certain spot, then you can add a TRACE statement (usually level 0) just before that location, run in Debug mode and then click on RUN. When the TRACE statement is reached, then the debugger will kick in at that point.

Usage:

```
.
.
'Here is the trouble spot
trace 0    ' kick down to single step mode
for index = 1 to int(100*sin(index))
   print #graph, "go "; index ; " "; int(100*cos(index))
next index
.
.
```

TRIM$( )

TRIM$(stringExpression)

Description:

This function removes any spaces from the start and end of the string in stringExpression. This can be useful for cleaning up data entry among other things.

Usage:

```
sentence$ = "   Greetings   "
print len(trim$(sentence$))
```

Produces: 9

USING( )

USING(templateString, numericExpression)

Description:

This function formats numericExpression as a string using templateString. The rules for the format are like those in MBASIC's PRINT USING statement, but since USING( ) is a function, it can be used as part of a larger BASIC expression instead of being useful only for output directly.

Usage:

```
' print a column of ten justified numbers
for a = 1 to 10
      print using("####.##",   rnd(1)*1000)
next a
```

VAL( )

VAL(stringExpression)

Description:

This function returns a numeric value for stringExpression is stringExpression represents a valid numeric value or if it starts out as one. If not, then zero is returned. This function lets your program take string input from the user and carefully analyze it before turning it into a numeric value if and when appropriate.

Usage:

print 2 * val("3.14")	Produces:	6.28
print val("hello")	Produces:	0
print val("3 blind mice")	Produces:	3

WHILE...WEND

WHILE expression . . . WEND

Description:

These two statements comprise the start and end of a control loop. Between the WHILE and WEND statements place code (optionally) that will be executed repeatedly while expression evaluates to true. The code between any WHILE statement and its associated WEND statement will not execute even once if the WHILE expression initially evaluates to false. Once execution reaches the WEND statement, for as long as the WHILE expression evaluates to true, then execution will jump back to the WHILE statement. Expression can be a boolean, numeric, or string expression.

Usage:

```
' loop until midnight (go read a good book)
while time$ <> "00:00:00"
      ' some action performing code might be placed here
wend
```

Or:

```
' loop until a valid response is solicited
while val(age$) = 0
    input "How old are you?"; age$
    if val(age$) = 0 then print "Invalid response.  Try again."
wend
```

WORD$( )

WORD$(stringExpression, n)

Description:

This function returns the nth word in stringExpression. The leading and trailing spaces are stripped from stringExpression and then it is broken down into 'words' at the remaining spaces inside. If n is less than 1 or greater than the number of words in stringExpression, then "" is returned.

Usage:

```
print word$("The quick brown fox jumped over the lazy dog", 5)
```

Produces:

```
jumped
```

And:

```
' display each word of sentence$ on its own line
sentence$ = "and many miles to go before I sleep."
token$ = "?"
while token$ <> ""
      index = index + 1
      tokens$ = word$(sentence$, index)
      print token$
wend
```

Produces:

```
and
many
miles
to
go
before
I
sleep.
```

Platform$

Description:

This variable holds the string "Windows". When programming with Liberty BASIC for OS/2, the same variable holds "OS/2".

This is useful so that you can take advantage of whatever differences there are between the two platforms and between the versions of Liberty BASIC.

Note: see also Version$

Version$

Description:

This variable holds the version of Liberty BASIC, in this case "1.2".

This is useful so that you can take advantage of whatever differences there are between the different versions of Liberty BASIC.

Note: see also Platform$