

# LIQUID BASIC 2025

## User Guide

Version 3.14 – Released November 21, 2025

Written by Bryan Flick

© Copyright 2025 All Rights Reserved

---

### Introduction

---

BASIC, which stands for Beginner's All-purpose Symbolic Instruction Code, is a high-level language that was developed in the mid-1960s at Dartmouth College. It was designed to be an easy-to-learn language for beginners, hence the name.

BASIC is known for its simplicity and ease of use. It uses English-like commands, which makes it more accessible to non-programmers. It was one of the first programming languages to be widely used on personal computers.

Liquid BASIC is a modern BASIC dialect. It includes all the standard legacy BASIC commands, plus support for structured programming, user-defined types, arrays, matrices, functions, fibers, error handling, timers, files, databases, artificial intelligence, graphics, sound, and more.

### Getting Started

---

Liquid BASIC runs in a 1280x800 window. Once initialized, it will display a "Ready." prompt. Commands can be entered and executed immediately. This is known as "direct mode". If the line entered starts with a line number instead, then the current line is inserted into the program currently in memory based on the line number. Valid line numbers range from 0 to 65535.

Press the CTRL + Home keys together to reset the user interface (soft reset).

Press the CTRL + F12 keys together to load and run the included **Menu.bas** program:



This program has a simple menu interface to browse and run the demonstration programs. Use the up/down arrow keys to move the selection up or down. Press Enter to run the selected BASIC program (printed in white) or change to the selected directory (printed in green). Press Space to preview a file (if the file can be previewed it is printed in white, otherwise it is printed in gray). When previewing a text, BASIC, or XML file, use the up/down arrow keys to scroll through the file. Press Tab to toggle the thumbnail on or off (the thumbnail shows a small preview of the current highlighted file if a preview is available). Press ESC to exit the Menu program.

Press ESC to exit most of the demonstration programs or press the CTRL + C keys together to force a program to end.

## Legacy Mode vs. Modern Mode

Liquid BASIC has two unique modes: Legacy mode and Modern mode.

Legacy mode has several features to maintain compatibility with older BASICs from the 1970's and early 1980's:

- Line numbers are used for branching (i.e. GOTO and GOSUB).
- Line numbers are saved to disk.
- Numeric constants can start with a period '.' (for example, .123 is the same as 0.123).
- Arrays will automatically dimension to ten elements the first time they are referenced.
- When printing numbers, a leading space is printed if the number is positive, and a trailing space is printed after all numbers.
- When printing numbers, the leading 0 is not printed if the number is between -1 and 1.
- When printing a comma, the cursor will advance to the next "zone", which is 15 characters wide.

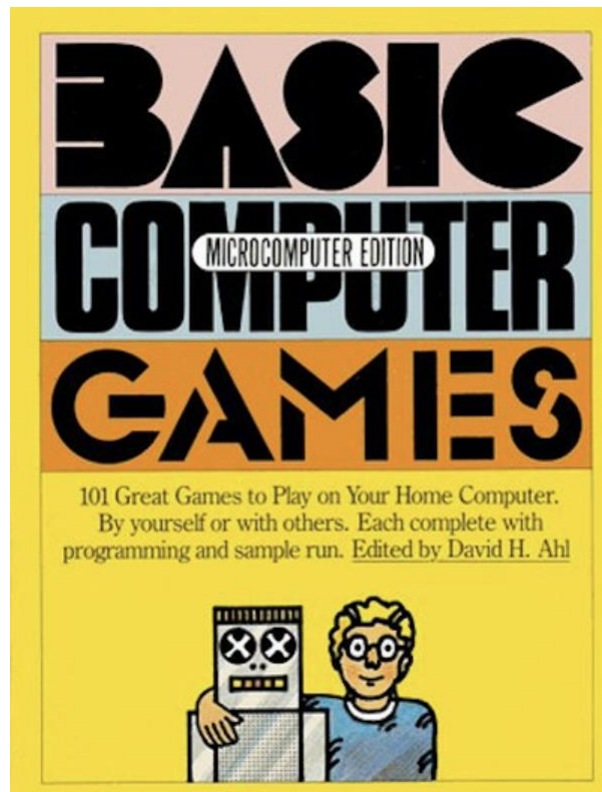
Modern mode has features that make programming easier and more relevant:

- Line numbers are not used for branching; they are simply used to sort the program while editing.
- Line numbers are not saved to disk.
- Numeric constants must use a leading 0 if the whole number portion is 0.
- Arrays must be explicitly dimensioned.
- When printing numbers, no leading or trailing spaces are printed.
- When printing numbers, the leading 0 is printed if the number is between -1 and 1.
- When printing a comma, the cursor will advance to the next “zone”, which is 16 characters wide.

By default, Liquid BASIC starts in Legacy mode. The LIST command will print the current mode. Use the MODERN command to switch to Modern mode. Use the LEGACY command to switch back to Legacy mode.

When a program is loaded, Liquid BASIC will automatically detect if the program has line numbers or not. If it does, then it will load in Legacy mode. Otherwise, it will load in Modern mode.

Liquid BASIC in Legacy mode has a high degree of compatibility with Microsoft 6502 BASIC (the same BASIC used as a foundation for Commodore BASIC, featured in the PET, VIC-20, and legendary Commodore 64 computers). It can run many of the programs unchanged from David H. Ahl’s classic 1973 book “BASIC Computer Games”, for example. These games are included in the “BasicComputerGames” folder. Be sure to leave CAPS LOCK on when playing, as many older computers operated with only uppercase letters and expect input to be capitalized.



---

## The Line Editor

The Liquid BASIC line editor is similar to the Commodore 64’s line editor. Lines are “physical” (a literal line of characters) and “logical” (multiple physical lines grouped together to read as one). Typing to the end of a physical line will insert a new physical line to create one long logical line.

The arrow keys can be used to move the cursor around.

Use the Home key to go to the beginning of a logical line. Use the End key to go to the end of a logical line.

Use the SHIFT + Home keys together to move the cursor to the “home” position (upper left corner of the screen).

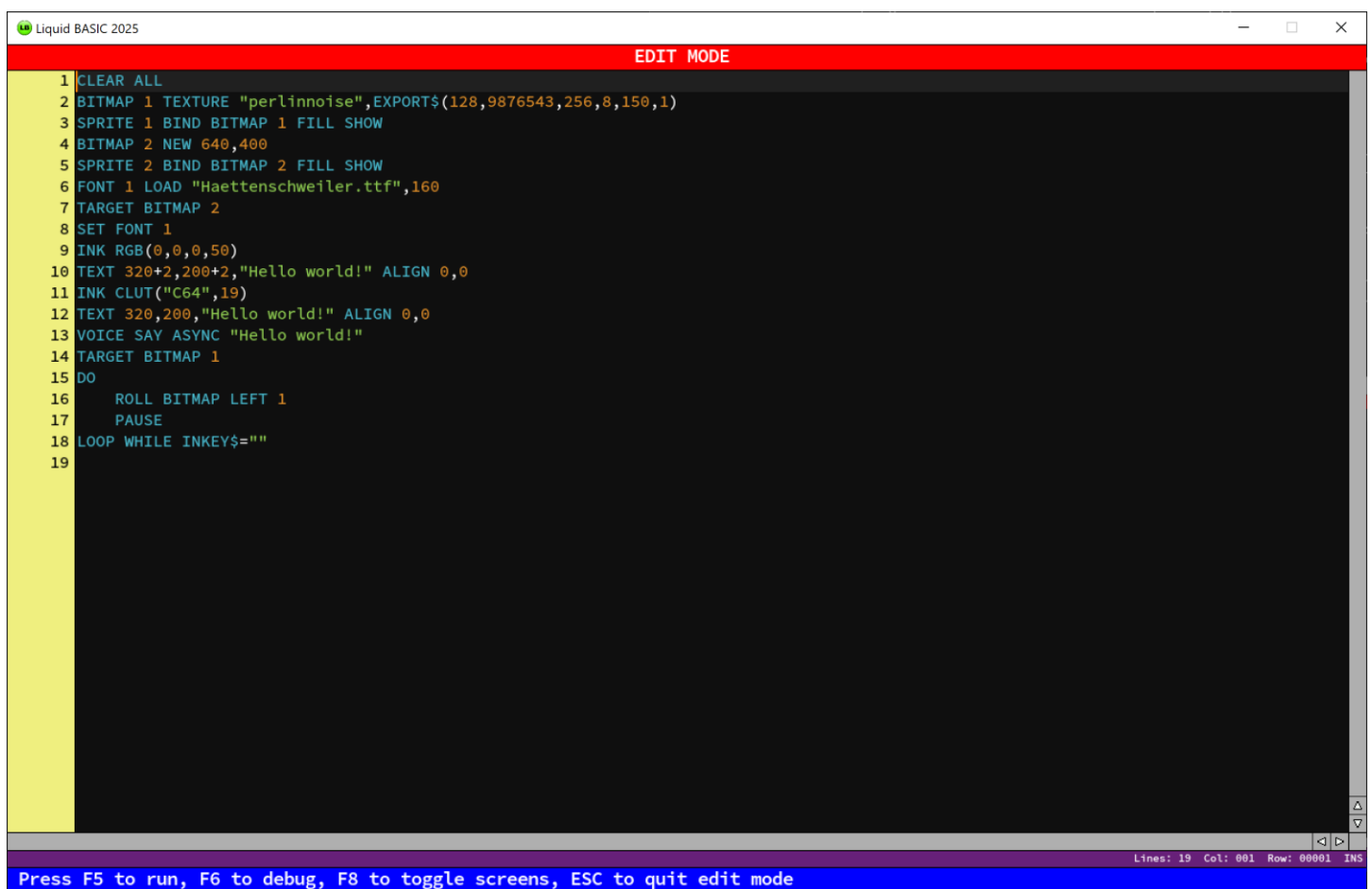
Use the Backspace key to delete the character to the left of the cursor on the current logical line. Use the Delete key to delete the character under the cursor on the current logical line. Use the Insert key to toggle between insert (half cursor) and overwrite (full cursor) mode.

## The Text Editor

Liquid BASIC also includes a modern way to edit BASIC programs using a full screen text editor. Enter EDIT to open the text editor. The EDIT command can only be used in Modern mode.

Text can be selected using the mouse or the SHIFT key. Use CTRL + C to copy the selected text, CTRL + X to cut, and CTRL + V to paste. Use CTRL + Z to undo an operation and CTRL + Y to redo it.

Press F5 to run the program. Press F6 to debug the program. Press F8 to toggle between the text editor and the Liquid BASIC screen. Press Esc to exit the full screen editor and place the editor contents back into program memory (renumbered starting at line number 10 with an increment of 10).



```
1 CLEAR ALL
2 BITMAP 1 TEXTURE "perlinnoise",EXPORT$(128,9876543,256,8,150,1)
3 SPRITE 1 BIND BITMAP 1 FILL SHOW
4 BITMAP 2 NEW 640,400
5 SPRITE 2 BIND BITMAP 2 FILL SHOW
6 FONT 1 LOAD "Haettenschweiler.ttf",160
7 TARGET BITMAP 2
8 SET FONT 1
9 INK RGB(0,0,0,50)
10 TEXT 320+2,200+2,"Hello world!" ALIGN 0,0
11 INK CLUT("C64",19)
12 TEXT 320,200,"Hello world!" ALIGN 0,0
13 VOICE SAY ASYNC "Hello world!"
14 TARGET BITMAP 1
15 DO
16     ROLL BITMAP LEFT 1
17     PAUSE
18 LOOP WHILE INKEY$=""
19
```

Lines: 19 Col: 001 Row: 00001 INS

Press F5 to run, F6 to debug, F8 to toggle screens, ESC to quit edit mode

---

## Direct Mode

---

### HELP

Print a quick reference to common commands.

### NEW

Clear (erase) the program in memory. Clear all user-defined constants, variables, and arrays. Switch Liquid BASIC to Legacy mode.

### LEGACY

Switch Liquid BASIC to Legacy mode.

### MODERN

Switch Liquid BASIC to Modern mode. Note that if a program is in memory, the line numbers will be lost when switching from Legacy to Modern mode. **It is recommended to save a backup of your program before switching to Modern mode.**

### RUN [<filename\$>]

Execute the BASIC program in memory or if <filename\$> is specified load the program from disk into memory and then execute it. All variables and arrays are cleared, and file and database handles are closed. Press the CTRL + C keys together to break execution. If the CTRL + D keys are pressed together, the Debugger is opened, the program is paused, and program execution goes to Single Step mode, allowing the program to be stepped through line by line.

### DEBUG

Open the Debugger and execute the BASIC program in memory.

### LIST [<first>][-<last>] [IF "filter\$"]

List the BASIC program currently in memory. Programs are listed from the first line of the program (unless the optional <first> value is specified) to the last line of the program (unless the optional <last> value is specified). The optional IF modifier will only list lines withing range that contain the text <filter\$>. The filter is not case sensitive. <filter\$> must be a string literal and not a string expression.

### LVAR

List all variables and arrays in memory and their values. Variables and arrays are listed in the order they were declared.

### DIR [-L] [<filter\$>]

Display the contents of the current directory, filtering the results to <filter\$>. If the "-L" switch is used a "long listing" is printed with the file date and file size for each file.

### LOAD <filename\$>

Load a program from disk into memory (the .BAS file extension is optional).

### SAVE <filename\$>

Store the program in memory to disk.

### AUTO [<inc>]

Start auto-incrementing when entering lines. Anytime a line that starts with a line number is entered the line editor will automatically print the next line number incremented by <inc>. Entering a line number by itself will end auto-increment mode.

#### **DELETE [<first>][-<last>]**

Delete the range of lines from <first> to <last>. The beginning of the program is assumed if <first> is omitted. The end of the program is assumed if <last> is omitted.

#### **RENUMBER [<start>][,<incr>]**

Renumber the program in memory, starting at line number <start> and incrementing by <incr> each line. In Legacy mode, the RENUMBER command will automatically update GOTO, GOSUB, IF/THEN, and other branching statements with the new renumbered line numbers. The program must not have any syntax errors for RENUMBER to work. Generally, if the program can RUN it can be renumbered. **It is recommended to save a backup of your program before renumbering.**

#### **FORMAT [<spaces>][,<Y|N>]**

Format the program in memory. Keywords, constants, and reserved variables and functions will be capitalized; proper indentation applied to each block; and spaces trimmed from the end of each line. Comments can also be stripped from the program. The optional <spaces> is the number of spaces to indent blocks and defaults to 4 if not specified. The optional <Y|N> switch determines whether to keep comments (Y – the default) or strip comments (N). The program must not have any syntax errors for FORMAT to work. Generally, if the program can RUN it can be formatted. **It is recommended to save a backup of your program before formatting.**

#### **EDIT [<filename\$>]**

Open the full screen text editor. If no <filename\$> is specified, then the program currently in memory is loaded into the editor. If an optional <filename\$> is specified, the editor will load <filename\$> when opened and automatically save the program back to disk when it is executed or the editor is closed. When the editor is closed, the program is renumbered and placed back into memory.

Only Modern programs can be edited using the full screen text editor.

#### **RESET**

Perform a soft reset. This is the same as pressing the CTRL + Home keys together.

#### **HARD RESET**

Perform a hard reset.

#### **QUIT**

Exit Liquid BASIC.

---

## **Variables and Expressions**

---

### **Variables**

---

Variables in Liquid BASIC are either numbers or strings.

There are four different types of numeric (Real) variables:

- Byte 8-bit whole number between 0 and 255
- Integer 32-bit whole number between -2,147,483,648 and +2,147,483,647
- Single 32-bit single precision floating point number with a range from  $\pm 1.5 \times 10^{-45}$  to  $\pm 3.4 \times 10^{38}$
- Double 64-bit double precision floating point number with a range from  $\pm 5.0 \times 10^{-324}$  to  $\pm 1.7 \times 10^{308}$

String variables are sequences of characters that can be any length.

Variable names are not case sensitive, must start with a letter, can include letters, numbers, and the underscore '\_' character, and can be any length.

A variable's type is determined by its suffix character at the end of the variable name:

- ? Byte
- % Integer
- ! Single
- # Double (default)
- \$ String

If no suffix character is appended, then the variable is assumed to be a Double (unless its type is explicitly declared using the DIM AS command). The suffix character not only determines the variable's type, it's also part of the variable's name. For example, B, B%, and B\$ are three different variables. This is a unique feature of the BASIC language.

## Literals

---

A literal is a constant number, like 3 or 5.123. A literal can start with a percent sign '%' to indicate the literal is in base 2 (binary – e.g. %1001001), or a dollar sign '\$' to indicate the literal is in base 16 (hexadecimal – e.g. \$FFD2). Otherwise, the literal is assumed to be in base 10 (decimal).

## Expressions and Operators

---

An expression is technically TRUE if it is non-zero, though -1 is the preferred value. An expression is FALSE if it is zero.

Each operator has a precedence level. Operators with a higher operator precedence level are executed before operators with a lower operator precedence level.

Supported binary operators (an operator with two subexpressions) and their precedence level from lowest to highest include:

TOKEN	OPERATOR	PRECEDENCE
<b>XOR</b>	Bitwise logical exclusive OR	1
<b>OR</b>	Bitwise logical OR	2
<b>AND</b>	Bitwise logical AND	3
<b>=</b>	Equal to	4
<b>&lt;&gt;</b>	Not equal to	4
<b>&lt;</b>	Less than	4
<b>&gt;</b>	Greater than	4
<b>&lt;=</b>	Less than or equal to	4
<b>&gt;=</b>	Greater than or equal to	4

<b>SHL</b>	Arithmetic shift left	5
<b>SHR</b>	Arithmetic shift right	5
<b>+</b>	Addition	6
<b>-</b>	Subtraction	6
<b>MOD</b>	Modulus	7
<b>DIV</b>	Integer division	8
<b>*</b>	Multiplication	9
<b>/</b>	Division	9
<b>^</b>	Exponentiation	11

There are also unary operators (an operator with one subexpression):

TOKEN	OPERATOR	PRECEDENCE
<b>+</b>	Ignored	N/A
<b>-</b>	Minus	10
<b>NOT</b>	Bitwise complement NOT	10
<b>( )</b>	Parentheses	12

Note that the Exponentiation binary operator takes precedence over the Minus and Binary (signed) NOT unary operators, so “-2 ^ 4” will result in -16 and not 16, because it is evaluated as “-(2 ^ 4)” and not “(-2) ^ 4”.

Parentheses take precedence over all other operators.

---



---

## Statements

---



---

### label:

An identifier followed by a colon ‘:’ is treated as a label and can be branched to when in Modern mode.

### REM

Use to comment your code. Anything on the line after REM is ignored. Comments do not affect the execution speed of your program. The apostrophe ‘’ can be substituted for the REM keyword.

### ASSERT <expr>

Use to debug your code. ASSERT is ignored if <expr> evaluates to true, otherwise program execution will stop with an “Assertion Failed” error.

### TRACE ON|OFF

Use to debug your code. When tracing is turned on with the ON modifier, Liquid BASIC will print the line number of each line as it runs. Use the OFF modifier to turn tracing off. The TRACE command is useful to trace your code’s execution over time.

### TRACK <var> ON|OFF



Use to debug your code. When tracking is turned on with the ON modifier, Liquid BASIC will print the variable <var> and its value anytime the variable is assigned. Use the OFF modifier to turn tracking off. The TRACK command is useful to track your variable's contents and changes over time.

## **BREAKPOINT**

Use to debug your code. When BREAKPOINT is encountered, the Debugger is opened, the program is paused, and program execution goes to Single Step mode, allowing the program to be stepped through line by line. This is the same as pressing the CTRL + D keys together while a program is running.

## **BREAK ON|OFF**

When breaking is turned on with the ON modifier, Liquid BASIC will force a program to end when the CTRL + C keys are pressed together, or when the STOP command is executed. Otherwise, if the OFF modifier is used to turn breaking off, CTRL + C and the STOP command are ignored. Breaking is turned on by default.

## **CLR**

Clear all user-defined constants, variables, and arrays.

## **CONST <const>=<value>[,<const>=<value>,...]**

Assign a constant to a specific value. Once set the constant cannot be changed. The constant type and value type must match (e.g., strings can only be assigned to String constants ending in a dollar sign '\$'). Constants are global and cannot be declared inside a FUNCTION.

## **[LET] <var>=<expr>**

Assign a variable to a specific value. The LET keyword is optional. The variable type and expression type must match (e.g., strings can only be assigned to String variables). Scalar (non-array) variables are declared the first time they are referenced. In Legacy mode, undeclared arrays are automatically declared to have ten elements the first time they are referenced. In Modern mode, all arrays must be explicitly declared.

Variables declared outside of a function are only visible outside of functions, unless the SHARE command or DIM GLOBAL command is used. Variables declared inside a function are only visible to that function.

## **GOTO <label>**

Jump to <label> (or line number if in Legacy mode). It is bad programming practice to jump into or out of control blocks using the GOTO statement as it may lead to unexpected results. You cannot jump into or out of FUNCTIONS.

## **GOSUB <label>**

Jump to <label> (or line number if in Legacy mode) and save the return address (the statement after the GOSUB statement) to be later RETURN'ed to. You cannot jump into or out of FUNCTIONS.

## **RETURN**

Return to the statement after the last GOSUB statement.

## **ON <expr> GOTO|GOSUB <label>[,<label>...] [ELSE <statements>]**

Jump to the <label> (or line number if in Legacy mode) in the list that <expr> indexes. <expr> must be a positive number. For example, if <expr> evaluates to 3, then jump to the third label in the list. If no label is specified for an index no jump is performed and the statements after the optional ELSE clause are run instead. In Legacy mode, <statements> can be replaced with a line number to jump to.

When GOSUB is used the return address (the statement after the ON ... GOSUB statement) is saved to be later RETURN'ed to.

#### **IF <expr> GOTO <line#>**

If <expr> evaluates to true, then jump to <line#>. This form of the IF statement can only be used in Legacy mode.

#### **IF <expr> THEN <statements> [ELSE <statements>]**

If <expr> evaluates to true, then run the statements after THEN. If <expr> evaluates to false and the optional ELSE clause is used, then run the statements after ELSE. In Legacy mode, <statements> can be replaced with a line number to jump to.

#### **IF <expr> THEN ... [ELSE IF <expr> ...][ELSE ...] END IF**

If <expr> evaluates to true, then run all the statements (even across multiple lines) after THEN. If <expr> evaluates to false and the optional ELSE clause is used, then run all the statements (even across multiple lines) after ELSE. END IF is required to signal the end of the IF/THEN block.

#### **EXIT IF**

Exit the current IF/THEN block.

#### **SELECT CASE <expr> CASE <testlist> ... [CASE <testlist> ...][CASE ELSE ...] END SELECT**

Evaluate <expr> and compare to each <testlist> expression. If there is a match, then run all the statements (even across multiple lines) after CASE. If there is no match, then run all the statements (even across multiple lines) after the optional CASE ELSE. END SELECT signals the end of the SELECT/CASE block.

A <testlist> can contain one or more simple expressions, separated by commas, to be compared to <expr>. All test lists must be of the same type as <expr>. A match occurs whenever any of the expressions in the test list evaluate to true. Tests can include equality, inequality, less than, less than or equal, greater than, greater than or equal, and range (from – to) testing. An equality test is assumed by default.

Example test lists include:

CASE 10	true if <expr> is equal to 10
CASE 2, 3, 4	true if <expr> is equal to 2, 3, or 4
CASE <50	true if <expr> is less than 50
CASE <=10, >100	true if <expr> is less than or equal to 10, or greater than 100
CASE 20 TO 30	true if <expr> is between 20 and 30
CASE "delete"	true if <expr> is a string and equal to "delete"
CASE "a" TO "e"	true if <expr> is a string and in the range of "a" to "e"

#### **EXIT SELECT**

Exit the current SELECT/CASE block.

#### **WHILE <expr> ... WEND**

Declare a loop. The loop will run while <expr> evaluates to true.

#### **CONTINUE WHILE**

Skip the remaining code in the current WHILE/WEND block and immediately proceed to the next iteration of the WHILE/WEND block (the loop condition is re-evaluated).

#### **EXIT WHILE**

Exit the current WHILE/WEND block.

#### **DO ... LOOP [WHILE|UNTIL <expr>]**

Declare a loop. If the WHILE and UNTIL clauses are not used the loop will run indefinitely. LOOP WHILE tests the <expr> at the end of the loop and repeats while <expr> is true. LOOP UNTIL tests the <expr> at the end of the loop and repeats until <expr> is true.

#### **DO [WHILE|UNTIL <expr>] ... LOOP**

Declare a loop. If the WHILE and UNTIL clauses are not used the loop will run indefinitely. DO WHILE tests the <expr> at the beginning of the loop and repeats while <expr> is true. DO UNTIL tests the <expr> at the beginning of the loop and repeats until <expr> is true.

#### **CONTINUE DO**

Skip the remaining code in the current DO/LOOP block and immediately proceed to the next iteration of the DO/LOOP block (the loop condition is re-evaluated).

#### **EXIT DO**

Exit the current DO/LOOP block.

#### **FOR <var>=<start> TO <end> [STEP <step>] ... NEXT [<var>]**

Assign <var> to <start>. If <var> is less than <end> then run the statements (even across multiple lines) before NEXT. If <var> is more than <end> at the beginning of the loop none of the loop is run. Once NEXT is encountered, increment <var> by <step> (<step> is 1 if not specified) and repeat the loop. <step> can also be negative to decrement <start> down to <end>. The <var> after NEXT is not required but must match the variable used in the FOR loop if specified.

#### **NEXT <var>,<var>[,<var>...]**

Multiple nested FOR blocks can be ended by specifying multiple variables separated by commas after NEXT. The <var> must match the variable used in the FOR loop being ended.

#### **CONTINUE FOR**

Skip the remaining code in the current FOR/NEXT block and immediately proceed to the next iteration of the FOR/NEXT block (the iteration variable is updated and the loop condition is re-evaluated).

#### **EXIT FOR**

Exit the current FOR/NEXT block.

#### **RESTORE [<label>]**

Set the <label> (or line number if in Legacy mode) where the next READ command should start looking for DATA. If no <label> is specified, then READ will look for DATA starting at the beginning of the program. The <label> cannot be inside a FUNCTION.

#### **DATA <value>[,<value>...]**

Define data to be read. <value> can be either a Real or String value. DATA statements are ignored when a running program encounters them. Use the READ command to read values from a DATA statement. The DATA statement cannot be inside a FUNCTION.

This command cannot be used in direct mode.

#### **READ <var>[,<var>...]**

Read the next item in a DATA statement and assign it to <var>. If <var> is a Real, then the item read must also be a Real. If <var> is a String, then the item read can be a Real or a String.

#### **SWAP <var1>,<var2>**

Swap the contents of <var1> and <var2>. The variables can be Real or String scalar variables, even if within a user-defined type. However, both variables must be of the same type.

#### **SORT [UCASE\$] <var>[,<var>...]**

Sort the variables. The variables can be Real or String variables, even if within a user-defined type. However, all variables must be of the same type. Real variables are sorted from smallest to largest. String variables are sorted alphabetically. When sorting string variables, the strings are case sensitive, unless the optional UCASE\$ modifier is used.

#### **BEEP**

Play the operating system's default Beep sound.

#### **PAUSE [<n>]**

Pause program execution for <n> milliseconds. If <n> is omitted (or PAUSE 0 is used) immediately yield the program to the host operating system until the next clock tick (the clock in Liquid BASIC ticks at 60 times a second, or 16.667 milliseconds per tick; each clock tick is known as a "jiffy"). Otherwise <n> must be between 15 and 86,400,000 milliseconds (24 hours).

#### **SLEEP [<n>]**

Put the program to sleep for <n> seconds. If <n> is omitted (or SLEEP 0 is used) the program will sleep until a key is pressed (SLEEP responds only to keystrokes that occur after it executes; it ignores characters in the keyboard buffer that were typed before it executed). Otherwise <n> must be between 1 and 86,400 seconds (24 hours).

#### **RANDOMIZE <seed>**

Set the seed of the random number generator to <seed>. The seed generates reproducible sequences. The same random numbers will always be generated by the RND function from the same seed. It is common to set the seed to TIMER at the beginning of a program as it will "randomize" the seed (depending on the time of day the program is run) and ensure decent random number generation.

#### **CHAIN [PRESERVE] <filename\$>**

Load a new program into memory and run it. If the optional PRESERVE modifier is used, all variables and arrays are preserved, allowing the current program to pass information to the chained program. File and database handles are also left open. If PRESERVE is omitted, all variables and arrays are cleared, and file and database handles are closed.

This command cannot be used in direct mode.

#### **NEST <filename\$>**

Load a new program into memory and run it. All variables and arrays are cleared, and file and database handles are closed. The current program reloads and restarts when the nested program ends. This is useful for menu-type programs (see **Menu.bas**).

This command cannot be used in direct mode.

## **STOP**

Immediately stop program execution. This is the same as pressing the CTRL + C keys together while a program is running. This command is ignored if breaking is turned off using the BREAK OFF command.

## **END**

Close all files and databases and end program execution.

---

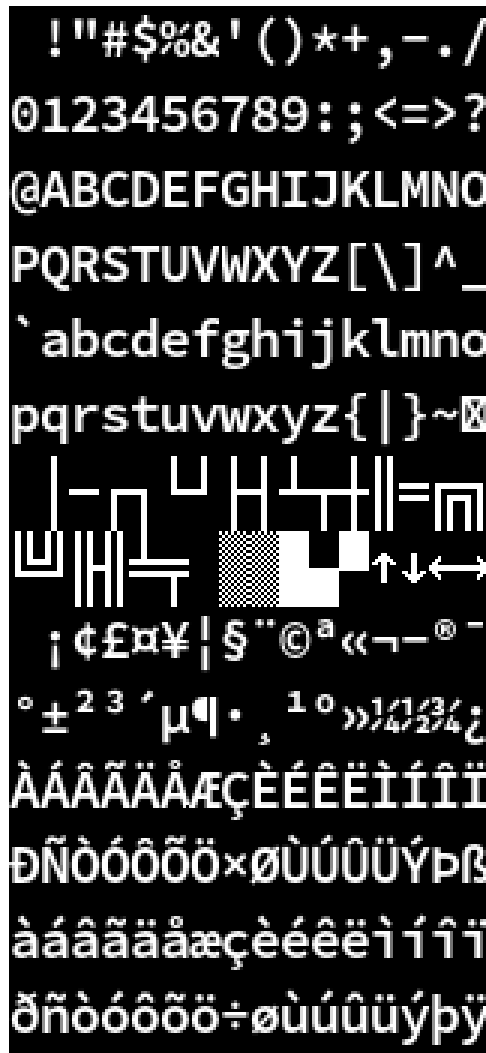
---

## **Text**

---

---

The Text Screen in Liquid BASIC is at the top of the graphics stack (see “Graphics” below). It is a simple 128x32 grid of cells, with each “cell” able to hold a single character of text from the high-resolution font (containing characters 32 to 255 from Windows code page 1252) below:



Individual cells contain a character, an independent foreground (ink) color, and an independent background (highlight) color. Each cell also has a set of text attributes: blink, bold, and underline. These attributes can be turned on and off and used in any combination.

## About Colors

---

Colors in Liquid BASIC are represented by a 32-bit unsigned integer, comprised of four 8-bit bytes. Each color has a byte for the red, green, blue, and alpha (transparency) levels of the color. Together this creates 16,777,216 color combinations with 256 levels of transparency.

The alpha level is used to determine transparency; it determines how to blend a color with the pixel beneath it when it is being written (0 is completely transparent, 255 is completely opaque, and any value in-between is a level of transparency).

The easiest way to get a proper 32-bit color value is to use the built-in HSV, RGB, and CLUT functions.

## Text Screen Commands

---

CLS

Clear the Text Screen.

#### **AUTO SCROLL <state>**

Turn auto-scrolling on (<state> 1) or off (<state> 0). When auto-scrolling is enabled, the screen will scroll up whenever a character is printed to the last character cell on-screen. Auto-scrolling is enabled by default.

#### **LOCATE [<x>][,<y>]**

Move the cursor to the <x>,<y> position. Both <x> and <y> start at 1, and each one is optional. If <x> is omitted, the column does not change. If <y> is omitted, the row does not change.

#### **GET <var>**

Return the next key enqueued in the keyboard buffer. If <var> is a Real variable, the next key is returned as a number if the key is between 0 and 9. Otherwise -1 is returned. If <var> is a String variable, the next key is returned as a one-character string, or "" is returned if the keyboard buffer is empty.

#### **GETKEY <var\$>**

Pause program execution until a key is pressed and store it in <var\$>. GETKEY ignores characters in the keyboard buffer that were typed before it executed.

#### **INPUT ["prompt\$";|,<var>[,<var>...]**

Input a line and assign it to <var>. If <var> is a Real then automatically try to convert the input to Real before assigning. If this fails a "Redo From Start" warning is printed, and the user must re-enter their input. Multiple values may be input by separating the input with commas. Each piece of input can be wrapped in quotes to preserve any commas in it. If more items are input than expected an "Extra Ignored" warning is printed and the extra items are discarded.

If <prompt\$> is omitted print a question mark '?' and a space by default. Otherwise, print the <prompt\$> before input (<prompt\$> must be a string literal and not a string expression). If the semicolon ';' separator is used after <prompt\$>, print a question mark '?' and a space following the prompt. Otherwise, if the comma ',' separator is used, do not print '?' and a space after the <prompt\$>.

This command cannot be used in direct mode.

#### **LINE INPUT ["prompt\$";<var\$> [PRESERVE]**

Input a line and assign it to <var\$>. <var\$> must be a String variable. No parsing is done on the input; the entire line is assigned to <var\$>. <prompt\$> is the optional prompt to print before input and must be a string literal and not a string expression. If the optional PRESERVE modifier is used, then the contents of <var\$> is used as the default input for the input field.

This command cannot be used in direct mode.

#### **PRINT <value>[<separator>]...**

Print text. <value> can be a Real or a String. When printing a Real, the following rules apply:

- In Legacy mode, a preceding space (if the Real is zero or positive) or minus sign (if the Real is negative) is printed before the number, and a space is printed after the number. Also, the leading 0 is not printed if the Real is between -1 and 1.
- In Modern mode, the Real is printed as is, with no leading or trailing spaces.

Multiple values can be printed, separated by a valid <separator>. Valid separators include:

- ; do not print a carriage return
- , print a tab
- | print a carriage return

Printing a tab advances the cursor to the next tab zone (by default, tab zones are 15 characters wide in Legacy mode, or 16 characters wide in Modern mode). If <separator> is omitted at the end of the PRINT statement a carriage return is printed by default.

A question mark '?' can be substituted for the PRINT keyword.

### **PRINT SPC(<x>)**

Print <x> number of spaces before printing.

### **PRINT TAB(<tab>)**

Move the cursor forward to <column> on the Text Screen, starting with the leftmost position of the current line. Columns start at 0. If <column> is greater than the Text Screen's width, TAB advances to the next line and the print position is set to (<column> MOD width). If the current print position is already beyond position <column>, the cursor is not moved.

### **PRINT TAB(<tabX>,<tabY>)**

Move the cursor to the column <tabX> and the row <tabY> on the Text Screen. Columns and rows start at 0.

### **PRINT LINE [LEFT|CENTER|RIGHT] [ON <row>,<value\$>][,<ink>][,<highlight>]**

Print the line <value\$>. If an optional alignment modifier (LEFT, CENTER, or RIGHT) is used print the <value\$> left justified, centered, or right justified, respectively, otherwise print at the current column. If the ON modifier is specified, print at line <row>, otherwise print on the current line. If the optional semicolon ';' is omitted, print a carriage return after printing <value\$>. If <ink> is specified, set the entire line's ink color. If <highlight> is specified, set the entire line's highlight color. This command can be useful for printing headers and footers.

### **PRINT CODE <value\$>[;]**

Print <value\$> using syntax highlighting followed by a carriage return (if the optional ';' is omitted). Syntax highlighting is a feature that colors and styles Liquid BASIC code to improve readability by visually distinguishing different elements like keywords, comments, and strings. It's a way to make code more understandable and easier to read. It does not affect how the code runs.

### **ZONE <n>**

Set the width of the tab zone. <n> can be between 1 and 100. By default, tab zones are 15 characters wide in Legacy mode, or 16 characters wide in Modern mode.

---

## **Color Commands**

### **BORDER [<color>][,<size>]**

Set the border color, where <color> is a 32-bit unsigned integer. The optional <size> also sets the size (in pixels) of the border. The border is drawn around the edge of the window. The border is not drawn if the <color> or <size> is 0.

### **BACKGROUND <color>**

Set the background color, where <color> is a 32-bit unsigned integer.



**INK <color>**

Set the ink color, where <color> is a 32-bit unsigned integer.

**HIGHLIGHT <color>**

Set the highlight color, where <color> is a 32-bit unsigned integer.

**Attribute Commands**

---

**BLINK <state>**

Turn blinking on (<state> 1) or off (<state> 0).

**BOLD <state>**

Turn bold on (<state> 1) or off (<state> 0).

**UNDERLINE <state>**

Turn underline on (<state> 1) or off (<state> 0).

**Cursor Commands**

---

**CURSOR MOVE <x>,<y>**

Move the cursor to the <x>, <y> position. Both <x> and <y> start at 0.

**CURSOR RESET**

Reset the cursor (put the cursor in “auto” mode, enable blinking, and tint yellow).

**CURSOR SHOW**

Turn on the cursor so it is always visible.

**CURSOR HIDE**

Turn off the cursor so it is never visible.

**CURSOR AUTO**

Put the cursor in “auto” mode, where the cursor is hidden while a program is running unless it is waiting on input. This is the default mode.

**CURSOR BLINK <state>**

If <state> is non-zero, the cursor will “blink” every 500 milliseconds (default). Otherwise, if <state> is zero, the cursor will appear solid and not blink.

**CURSOR TINT <color>**

Tint the cursor to <color>, where <color> is a 32-bit unsigned integer.

---

## User-Defined Types

---

A user-defined type is a data type that is defined by the programmer rather than being built into the programming language. This allows the programmer to create custom data structures that are tailored to the specific needs of their program.

User-defined types are an aggregate type made up of elementary BASIC types. They can encapsulate any scalar variable; however, they cannot include arrays. They can also encapsulate other user-defined types.

Liquid BASIC lets you define new data types using a TYPE block. The type's name cannot have a suffix character. The FIELD statement declares the encapsulated items within the type. Only FIELD statements can appear in a TYPE block.

```
TYPE <name>
    FIELD <var> [AS <type>]
    ...
END TYPE
```

Each FIELD statement declares one or more variables within the type. A field's type is declared either by the suffix character on its name (e.g. "%" for integers), or by explicitly using the AS clause. The following types are built-in: BYTE, INT, LONG, SINGLE, DOUBLE, and STRING. If the AS clause is used, the variable's name must not have a suffix character.

For example, the following TYPE statement defines a type, Employee:

```
TYPE Employee
    FIELD Name$
    FIELD Position$
    FIELD Salary
    FIELD StartDate$
END TYPE
```

The DIM...AS statement is used to declare a new variable as a user-defined type:

```
DIM SalesRep as Employee
```

Once a variable has been declared as a user-defined type, individual fields contained within the variable can be accessed using the period '.' operator. Fields can be read from and written to, just like any other variable:

```
SalesRep.Name$= "Joe Thomas"
PRINT SalesRep.StartDate$
```

A user-defined type can be encapsulated within another user-defined type, using the FIELD...AS statement:

```
TYPE Organization
    FIELD Name$
    FIELD Manager AS Employee
    FIELD Worker AS Employee
    FIELD EmployeeOfTheMonth AS Employee
```

## END TYPE

Once declared, any field and encapsulated user-defined type can be accessed:

```
DIM Business AS Organization  
Business.Name$="Software Company"  
Business.Worker.Name$="Beth Thompson"  
Business.EmployeeOfTheMonth=Business.Worker
```

---

## Arrays

---

An array is a data structure that stores a collection of elements of the same data type, in a contiguous block of memory. Each element in the array is accessed by its index, which is an integer value that represents its position in the array. Arrays are commonly used in programming to store and manipulate lists of data efficiently.

Before an array is used, it should be declared to set the size of the array, the type of array, and how many dimensions it has. Scalar (non-array) variables can also be declared for clarity, though they will be automatically declared the first time they are used. (In Legacy mode, undeclared arrays are automatically declared to have ten elements the first time they are referenced.)

### Variable and Array Declarations

---

**DIM [GLOBAL] <var>[ AS <type>][,<var>[ AS <type>],...]**

Declare a scalar variable. This is technically not required, as all variables are automatically declared the first time they are used. Multiple variables can be declared at once.

Variables declared inside a function are visible to that function only. This is known as local scope. When a variable is accessible anywhere in the program – inside or outside of any function – that is known as global scope. By default, variables declared outside a function are not global and not visible within any function. The optional GLOBAL modifier makes the variable global in scope and accessible anywhere.

A variable's type is defined either by the suffix character on its name (e.g. "%" for integers), or by explicitly using the AS clause. The following types are built-in: BYTE, INT, LONG, SINGLE, DOUBLE, and STRING. User-defined types can also be used. If the AS clause is used, the variable's name must not have a suffix character.

**DIM [GLOBAL] <array>(<dimension>)[ AS <type>]**

Declare a single dimensional array. An array is a collection of variables where each variable can be referenced by its index. Indexes start at 0. <dimension> is the number of elements the array should have. Arrays must have at least one element and no more than ten million elements. All elements are set to zero/null when initialized.

**DIM [GLOBAL] <array>(<dimension1>,<dimension2>[...,<dimension8>])[ AS <type>]**

Declare a multidimensional array. When two dimensions are provided, a matrix or grid array is allocated. When three dimensions are provided, a cube array is allocated. Arrays can have up to eight dimensions. Indexes start at 0. Arrays must have at least one element and no more than ten million elements total. All elements are set to zero/null when initialized.

## **REDIM <array>(<dimension1>,<dimension2>[...,<dimension8>])**

Redeclare an array. All elements are set to zero/null when initialized.

## **REDIM PRESERVE <array>(<dimension>)**

Redeclare an array but save the contents of the existing array. New elements are set to zero/null when initialized. REDIM PRESERVE can only be used on single dimensional arrays (vectors).

## **Array Commands**

---

### **ARRAY FILL <array>()[,<expr>] [SLICE <first>,<last>]**

Fill the array with <expr>. If <expr> is omitted a 0 (for Real arrays) or "" (for String arrays) is used to fill the array. If the optional SLICE modifier is used, then fill only the elements between <first> and <last>. The SLICE modifier can only be used on single dimension arrays.

ARRAY FILL cannot be used on arrays of user-defined types.

### **ARRAY COPY <array>(<index>) TO <dstArray>(<dstIndex>) [FOR <count>]**

Copy all or part of an array to another array. The items in <array> starting at <index> are copied to <dstArray> starting at <dstIndex>. If the optional FOR modifier is used, then <count> specifies the number of items to copy. Otherwise, the entire <array> is copied. ARRAY COPY can only be used on single dimensional arrays.

### **ARRAY FIND [UCASE\$] <array>()[.<fields>...],<test> [SLICE <first>,<last>] TO <var>**

Search an array to find the item that satisfies the <test> expression. If there is a match, then the index of the item that passed the test is assigned to <var>, otherwise -1 is assigned. If the optional SLICE modifier is used, then scan only the elements between <first> and <last>. When searching a string array, the strings are case sensitive, unless the optional UCASE\$ modifier is used. ARRAY FIND can only be used on single dimensional arrays.

When searching an array of user-defined types, the specific field to search must be specified after the array and its parentheses.

A <test> is a simple expression. The test must match the array's type. A match occurs if the test evaluates to true. Tests can include equality, inequality, less than, less than or equal, greater than, and greater than or equal. An equality test is assumed by default.

Example tests include:

10	true if array item is equal to 10
<50	true if array item is less than 50
>=100	true if array item is greater than or equal to 100
"insert"	true if array item is a string and equal to "insert"

### **ARRAY DELETE <array>(<index>)[,<expr>] [FOR <count>]**

Delete the item at <index> from the array. All elements above <index> are shifted down. A 0 (for Real arrays) or "" (for String arrays) will fill the top element, unless the optional <expr> is specified in which case it will fill the top element. If the optional FOR modifier is used, then <count> items are deleted. ARRAY DELETE can only be used on single dimensional arrays.

### **ARRAY INSERT <array>(<index>)[,<expr>] [FOR <count>]**

Insert the item <expr> into the array at <index>. All elements at <index> and above are shifted up and the top element is discarded. If <expr> is omitted a 0 (for Real arrays) or "" (for String arrays) is inserted. If the optional FOR modifier is used, then <count> items are inserted. ARRAY INSERT can only be used on single dimensional arrays.

### **ARRAY REVERSE <array>() [SLICE <first>,<last>]**

Reverse the elements in the array. If the optional SLICE modifier is used, then reverse only the elements between <first> and <last>. ARRAY REVERSE can only be used on single dimensional arrays.

### **ARRAY SHUFFLE <array>() [SLICE <first>,<last>]**

Shuffle the elements in the array. If the optional SLICE modifier is used, then shuffle only the elements between <first> and <last>. ARRAY SHUFFLE can only be used on single dimensional arrays.

### **ARRAY SORT [UCASE\$] <array>()[.<fields>...] [SLICE <first>,<last>]**

Sort the elements in the array. If the optional SLICE modifier is used, then sort only the elements between <first> and <last>. When sorting a string array, the strings are case sensitive, unless the optional UCASE\$ modifier is used. ARRAY SORT can only be used on single dimensional arrays.

When sorting an array of user-defined types, the specific field to sort by must be specified after the array and its parentheses.

### **ARRAY LOAD <path\$>,<array>()**

Read the file at <path\$> into an array. The array will be declared (or redeclared if it already exists). The file must be in the same format as generated by ARRAY SAVE.

ARRAY LOAD cannot be used on arrays of user-defined types.

### **ARRAY SAVE <path\$>,<array>()**

Write the data in an array to the file at <path\$>. If the file does not exist it will be created.

ARRAY SAVE cannot be used on arrays of user-defined types.

## **Array Operations**

---

There are several operations that can be performed on already dimensioned Real arrays. These operations provide a convenient way to quickly update many or all elements in an array. During an operation, the smallest array is always used when setting the items. For example, if <arrayA> has 9 elements, <arrayB> has 6 elements, and <arrayC> has 3 elements, then adding:

$$\text{ARRAY } a()=b()+c()$$

would only set the first three elements of a(), as setting any more would cause a bad subscript error on c().

### **ARRAY <array>()=(<expr>)**

Fill the array with <expr>. <expr> must be enclosed in parentheses to be parsed correctly.

### **ARRAY <array>()=[<expr>,<expr>...]**

Fill the array with the <expr>'s enclosed inside the square braces. If there are less items in the braces than in the array the remaining items in the array stay untouched.

e.g. **DIM n(10)**

**ARRAY n( )=[1,3,5,7,9] ' only n(0) thru n(4) are set**

**ARRAY <arrayA>()=<arrayB>()**

Copy <arrayB> to <arrayA>.

**ARRAY <arrayA>()=<arrayB>()+<arrayC>()**

Add <arrayB> to <arrayC> and assign to <arrayA>.

**ARRAY <arrayA>()=<arrayB>()+(<expr>)**

Add <expr> to every item in <arrayB> and assign to <arrayA>. <expr> must be enclosed in parentheses to be parsed correctly.

**ARRAY <arrayA>()=(<expr>)+<arrayB>()**

Add every item in <arrayB> to <expr> and assign to <arrayA>. <expr> must be enclosed in parentheses to be parsed correctly.

**ARRAY <arrayA>()=<arrayB>()-<arrayC>()**

Subtract <arrayC> from <arrayB> and assign to <arrayA>.

**ARRAY <arrayA>()=<arrayB>()-(<expr>)**

Subtract <expr> from every item in <arrayB> and assign to <arrayA>. <expr> must be enclosed in parentheses to be parsed correctly.

**ARRAY <arrayA>()=(<expr>)-<arrayB>()**

Subtract every item in <arrayB> from <expr> and assign to <arrayA>. <expr> must be enclosed in parentheses to be parsed correctly.

**ARRAY <arrayA>()=<arrayB>()\*<arrayC>()**

Multiply <arrayB> by <arrayC> and assign to <arrayA>.

**ARRAY <arrayA>()=<arrayB>()\*(<expr>)**

Multiply every item in <arrayB> by <expr> and assign to <arrayA>. <expr> must be enclosed in parentheses to be parsed correctly.

**ARRAY <arrayA>()=(<expr>)\*<arrayB>()**

Multiply <expr> by every item in <arrayB> and assign to <arrayA>. <expr> must be enclosed in parentheses to be parsed correctly.

**ARRAY <arrayA>()=<arrayB>()/<arrayC>()**

Divide <arrayB> by <arrayC> and assign to <arrayA>.

**ARRAY <arrayA>()=<arrayB>()/(<expr>)**

Divide every item in <arrayB> by <expr> and assign to <arrayA>. <expr> must be enclosed in parentheses to be parsed correctly.

## **ARRAY <arrayA>()=(<expr>)/<arrayB>()**

Divide <expr> by every item in <arrayB> and assign to <arrayA>. <expr> must be enclosed in parentheses to be parsed correctly.

---

---

## **Matrices**

---

---

A matrix is a two-dimensional array (declared using the DIM statement) arranged in rows and columns. Each element in a matrix is identified by its row and column position, and matrices can be added, subtracted, multiplied, and transformed. Matrices are commonly used to represent data in various applications such as image processing, machine learning, and computer graphics.

### **Matrix Commands**

---

#### **MAT <matrixA>()=ZER**

Set all elements of <matrixA> to zero.

#### **MAT <matrixA>()=CON**

Set all elements of <matrixA> to one.

#### **MAT <matrixA>()=IDN**

Establish <matrixA> as an identity matrix. An identity matrix is a square matrix in which all the elements of the main diagonal are one, and all other elements are zero. The identity matrix is a special type of matrix that acts like the number 1 in scalar multiplication, meaning that when it is multiplied by another matrix, the result is the original matrix.

#### **MAT <matrixA>()=INV(<matrixB>())**

Invert <matrixB> and assign to <matrixA>. Only a square matrix may be inverted. If a matrix is multiplied by its inverse matrix, the result is an identity matrix. <matrixB> must be either a single precision or double precision floating point array.

#### **MAT <matrixA>()=TRN(<matrixB>())**

Transpose <matrixB> and assign to <matrixA>. A transposed matrix is a matrix in which the rows and columns of the original matrix are swapped. In other words, the rows of the original matrix become the columns of the transposed matrix, and the columns of the original matrix become the rows of the transposed matrix.

#### **MAT <matrixA>()=<matrixB>()**

Assign <matrixB> to <matrixA>.

#### **MAT <matrixA>()=<matrixB>()+<matrixC>()**

Add <matrixB> to <matrixC> and assign to <matrixA>. <matrixB> and <matrixC> must be the same size.

#### **MAT <matrixA>()=<matrixB>()-<matrixC>()**

Subtract <matrixC> from <matrixB> and assign to <matrixA>. <matrixB> and <matrixC> must be the same size.

**MAT <matrixA>(<matrixB>)\*<matrixC>()**

Multiply <matrixB> by <matrixC> and assign to <matrixA>. <matrixB> and <matrixC> must be the same size. <matrixB> and <matrixC> must be either single precision or double precision floating point arrays.

**MAT <matrixA>(<expr>)\*<matrixB>()**

Multiply <matrixB> by the scalar value <expr>. The <expr> must be wrapped in parentheses to be parsed correctly. <matrixB> must be either a single precision or double precision floating point array.

**MAT PRINT <matrixA>(),<format\$>]**

Print the matrix <matrixA>. If the cursor is not at the beginning of the current line, a carriage return is printed before printing the matrix. Each row is printed, with a tab between each column. The optional <format\$> can be used to format the individual elements as they are printed (see the USING\$ function).

---

## Functions

---

A function is a block of code that performs a specific task or calculation. It can take input parameters, perform operations on them, and return a single result. Functions are used to organize code, make it more modular and reusable, and improve readability.

### Function Commands

---

**DEF FN<name>(<var>)=<expr>**

Define a single line function that returns a Real value. The <name> must start with FN, cannot be longer than three characters (including the FN), and cannot have a type suffix character (such as '%' or '\$'). A single <var> can be passed to the function, and the value passed to that variable when the function is called is substituted inside <expr>.

For example:

```
DEF FNA(Z)=30*EXP(-Z*Z/100)
```

```
? FNA(5)
```

```
23.3640234921421
```

```
? 30*EXP(-5*5/100)
```

```
23.3640234921421
```

This declares a single line function named "FNA". When called, a single value is passed to the temporary variable Z, which is then substituted wherever Z is used in the expression. The results are then returned to the caller.

**FUNCTION <name>([[REF ]<arg>[ AS <type>],[REF ]<arg>[ AS <type>]...]][ AS <type>] ... END FUNCTION**

Define a function. Functions are named subroutines (blocks of code) that can be CALL'ed from anywhere. Functions cannot be nested within one another. The FUNCTION block is skipped when encountered.

Functions can be "invoked" using the CALL command. When a function has ended it returns to the statement after the CALL command.



Arguments can be passed to the function. If the optional REF modifier is used, then the argument must be a variable and is passed by reference, and any changes made to the variable in the function will pass back to the caller when the function ends. Otherwise, each argument is evaluated and treated as a local variable within the function and freed when the function ends.

An argument's type is declared either by the suffix character on its name (e.g. "%" for integers), or by explicitly using the AS clause. The following types are built-in: BYTE, INT, LONG, SINGLE, DOUBLE, and STRING. User-defined types can also be used. If the AS clause is used, the variable's name must not have a suffix character.

User-defined types must be passed using the REF modifier. Arrays must be passed using the REF modifier and an empty set of parentheses. Arrays of user-defined types can also be passed using the REF modifier.

All variables and arrays inside the function are local to the function. This allows for recursion, where a function calls itself until a certain condition is met. Use the SHARE command or DIM GLOBAL command to access a variable or array outside the function.

Functions can optionally return a value by either assigning the value to return to the function's <name>, as if it were a variable, or using the RETURN command. This allows functions to be used in expressions. The function's suffix character determines its return type (Byte, Int, Single, Double, or String) if the optional AS clause is not used. The function's <name> must not have a suffix character if the AS clause is used.

Generally, the use of FUNCTION is preferred over DEF FN.

#### **SHARE <var>[,<var>...]**

By default, variables and arrays declared inside a function are local in scope, and variables and arrays declared outside a function are not accessible within the function. Sometimes this behavior can be unintended. The SHARE command is used inside a FUNCTION to declare that a variable or array outside the FUNCTION is shared and can be accessed. Global variables and arrays cannot be shared, as they are already accessible.

The SHARE command must be used immediately after the FUNCTION declaration and before any other commands are used.

#### **[CALL] <name>([[REF ]<arg>[, [REF ]<arg>...]]) [TO <var>]**

Pass the arguments <arg> to the function named <name> and call it, saving the return address (the statement after the CALL statement) to be later returned to when the function has finished executing. Arguments must be passed using the REF modifier if the function is expecting the argument to be passed by reference. If the optional TO modifier is used, the function's return value is assigned to the variable <var>. Otherwise, the function's return value is discarded. The CALL keyword is optional.

#### **RETURN <expr>**

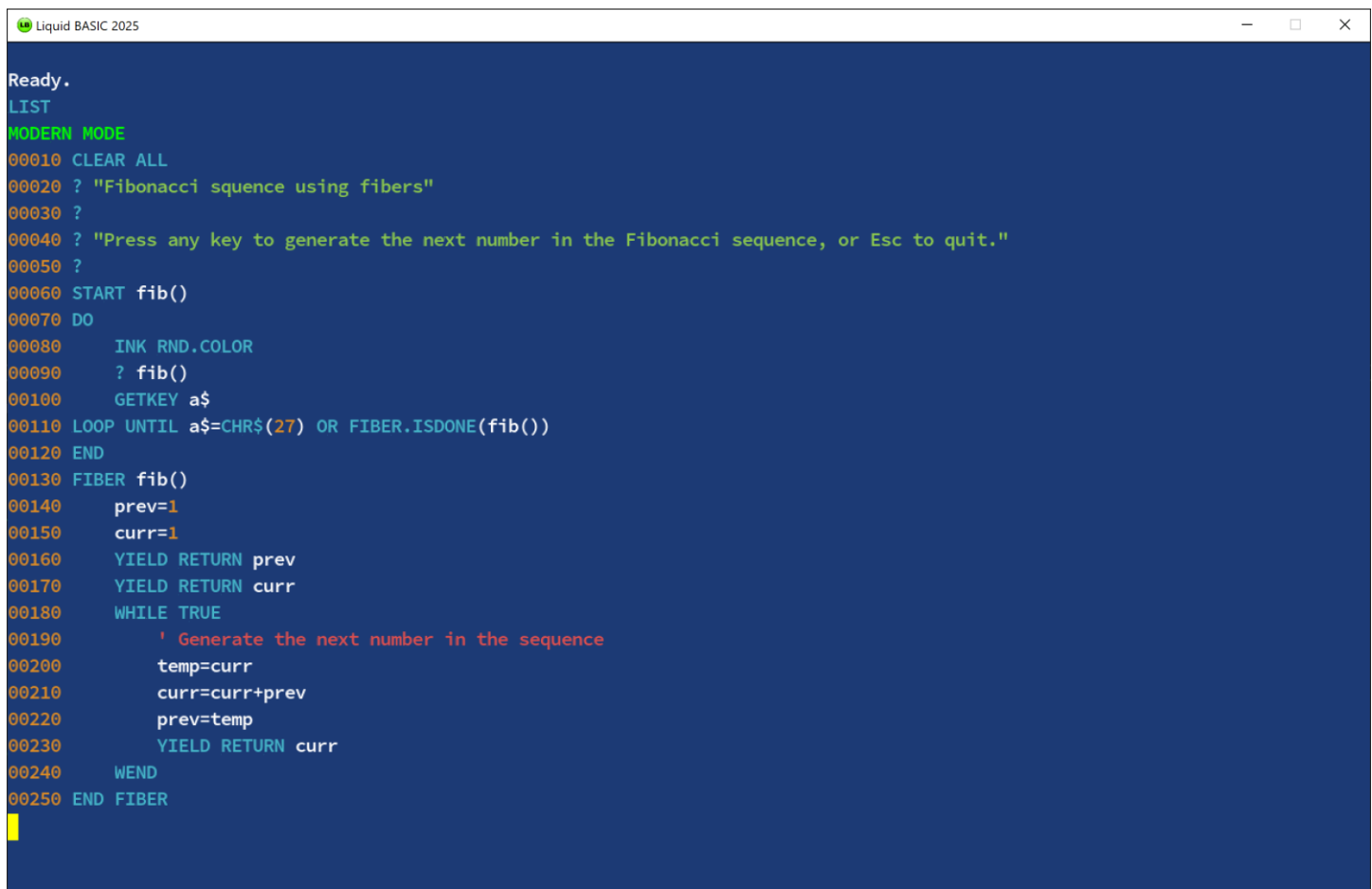
Return <expr> back to the caller and exit the current FUNCTION block.

#### **EXIT FUNCTION**

Exit the current FUNCTION block.

A fiber (also known as a coroutine) is a type of function that can suspend (yield) its execution and return control to the caller while maintaining its state. The fiber can later be resumed and will restore its state and continue its execution until it either yields again or has completed. This allows for more efficient cooperative multitasking and asynchronous programming, as fibers can be used to perform tasks such as waiting for input/output concurrently without the need for multiple threads and all the headaches that come along with threads.

Fibers can be in one of three states: Suspended, Running, or Finished. When a fiber is started, its state is initialized and is set to Running. If the fiber yields, its state changes to Suspended until it is resumed. Once the fiber is resumed, its state changes back to Running and execution resumes where the fiber yielded. Once a fiber has completed, its state is set to Finished. Attempting to resume a fiber that has already finished will result in an error, although it can be restarted to run again. The FIBER.ISDONE function can be used to determine if a fiber has finished or not.



```
Ready.
LIST
MODERN MODE
00010 CLEAR ALL
00020 ? "Fibonacci squence using fibers"
00030 ?
00040 ? "Press any key to generate the next number in the Fibonacci sequence, or Esc to quit."
00050 ?
00060 START fib()
00070 DO
00080     INK RND.COLOR
00090     ? fib()
00100     GETKEY a$
00110 LOOP UNTIL a$=CHR$(27) OR FIBER.ISDONE(fib())
00120 END
00130 FIBER fib()
00140     prev=1
00150     curr=1
00160     YIELD RETURN prev
00170     YIELD RETURN curr
00180     WHILE TRUE
00190         ' Generate the next number in the sequence
00200         temp=curr
00210         curr=curr+prev
00220         prev=temp
00230         YIELD RETURN curr
00240     WEND
00250 END FIBER
```

## Fiber Commands

**FIBER <name>([[REF ]<arg>[ AS <type>]],[REF ]<arg>[ AS <type>]...)] AS <type> ... END FIBER**

Define a fiber. Fibers are functions that can suspend themselves while maintaining their state, and resume at a later point with the state restored. Fibers cannot be nested within one another. The FIBER block is skipped when encountered.

Arguments can be passed to the fiber. If the optional REF modifier is used, then the argument must be a variable and is passed by reference, and any changes made to the variable in the fiber will pass back to the caller when

the fiber yields or ends. Otherwise, each argument is evaluated and treated as a local variable within the fiber and freed when the fiber ends.

An argument's type is declared either by the suffix character on its name (e.g. "%" for integers), or by explicitly using the AS clause. The following types are built-in: BYTE, INT, LONG, SINGLE, DOUBLE, and STRING. User-defined types can also be used. If the AS clause is used, the variable's name must not have a suffix character.

User-defined types must be passed using the REF modifier. Arrays must be passed using the REF modifier and an empty set of parentheses. Arrays of user-defined types can also be passed using the REF modifier.

All variables and arrays inside the fiber are local to the fiber. Fibers do not support recursion, however. Use the SHARE command or DIM GLOBAL command to access a variable or array outside the fiber.

Fibers can optionally return a value by assigning the value to return to the fiber's <name>, as if it were a variable, or using the RETURN or YIELD RETURN commands. This allows fibers to be used in expressions. The fiber's suffix character determines its return type (Byte, Int, Single, Double, or String) if the optional AS clause is not used. The fiber's <name> must not have a suffix character if the AS clause is used.

#### **SHARE <var>[,<var>...][,<array>()...]**

By default, variables and arrays declared inside a fiber are local in scope, and variables and arrays declared outside a fiber are not accessible within the fiber. Sometimes this behavior can be unintended. The SHARE command is used inside a FIBER to declare that a variable or array outside the FIBER is shared and can be accessed. Global variables and arrays cannot be shared, as they are already accessible.

The SHARE command must be used immediately after the FIBER declaration and before any other commands are used.

#### **START <name>([[REF ]<arg>[, [REF ]<arg>...]])**

Pass the arguments <arg> to the fiber named <name>, initialize the fiber's state, and start the fiber. Arguments must be passed using the REF modifier if the fiber is expecting the argument to be passed by reference.

#### **RESUME <name>()**

Restore the fiber's state and resume program execution where the fiber last yielded.

#### **YIELD**

Suspend the current fiber, save its state, and return program execution to the caller. Any arguments passed using the REF modifier during START are also updated.

#### **YIELD RETURN <expr>**

Suspend the current fiber, save its state, and return <expr> back to the caller. This is equivalent to assigning <expr> to the fiber's <name> and yielding. Any arguments passed using the REF modifier during START are also updated.

#### **RETURN <expr>**

Return <expr> back to the caller and exit the current FIBER block, ending the fiber.

#### **EXIT FIBER**

Exit the current FIBER block.

---

## Error Handling

---

Error handling is the process of anticipating, detecting, and resolving errors that may occur during the execution of a program. This involves implementing mechanisms to “trap” (handle) errors to prevent the program from crashing and to provide feedback to the user about the error. Error handling helps improve the reliability and robustness of a program by gracefully handling unexpected situations.

Error handlers are local to the function they reside in or are global if not inside a function. When a function is called, the current error handler is pushed on the call stack and the error handler is reset (same as calling TRAP OFF). When the function ends, the previous error handler is popped off the call stack and restored.

---

### Error Handling Commands

---

#### TRAP <label>

Enable the error handler at <label> (or line number if in Legacy mode). When a non-fatal error occurs record the error code and line number where the error occurred, disable further error handling, and jump to <label>. Use the reserved variables ERR and ERL to read the error code and line number where the error occurred, respectively. If TRAP is used inside a FUNCTION, the <label> must be inside the same FUNCTION.

#### TRAP RESUME NEXT

Enable the error handler. When a non-fatal error occurs record the error code and line number where the error occurred, and resume program execution at the statement following the one that caused the error. Use the reserved variables ERR and ERL to read the error code and line number where the error occurred, respectively.

#### TRAP OFF

Disable the error handler. When a non-fatal error occurs, halt program execution and print the error message and line number where the error occurred. This is the default behavior when a program is first run or a function is first called.

#### RESUME [NEXT] <label>

When a non-fatal error occurs and is handled by the error handler, re-enable error handling and resume program execution. With no parameters, RESUME attempts to re-execute the statement in which the error occurred. RESUME NEXT resumes execution at the statement immediately following the one containing the error. RESUME followed by a <label> (or line number if in Legacy mode) resumes execution at the <label>. If RESUME is used inside a FUNCTION, the <label> must be inside the same FUNCTION.

#### ERROR <code>

Simulate the occurrence of a BASIC error. If an error handler is available, control passes to the error handler. <code> specifies the error code and must be above zero. <code> does not have to be an internal Liquid BASIC error code; it can be any other number for user-defined error codes.

The ERROR statement and ERR reserved variable is often used together to raise an unhandled error in an error handler to halt program execution (e.g. ERROR ERR).

#### WHEN ERROR ... END WHEN

Define an error handler. When a non-fatal error occurs record the error code and line number where the error occurred, disable further error handling, and jump to the WHEN ERROR block. Use the reserved variables ERR and ERL to read the error code and line number where the error occurred, respectively. If a WHEN ERROR block

does not handle the error by the end of the block the error is printed, and program execution stops. Use RESUME or EXIT WHEN (equivalent to RESUME NEXT) before the block ends if that is not the desired effect. WHEN ERROR blocks cannot be nested or inside a FUNCTION.

---

## Timers

---

A timer is an event that causes an interruption to your program's execution at specified intervals. When an enabled timer occurs, the program is suspended while execution branches to a designated interrupt handler. Once the interrupt handler has finished, the program resumes execution. Timers that are not enabled are simply ignored. Timers are not enabled by default.

There can be up to eight independent timers. A WHEN TIMER block is used to define a timer. Whenever a WHEN TIMER block is encountered, it changes the current interrupt handler for the timer to the enclosed block. Timers can be enabled or disabled using the TIMER command.

Timers are global and cannot be defined inside a function. An enabled timer will still execute its interrupt handler when inside a function, however.

Further interrupts for the timer are disabled when inside the WHEN TIMER block to prevent recursive calls for the same timer.

Be careful with timers. They can introduce a lot of headaches that a non-interrupted tightly controlled main execution loop can avoid. It is generally best to have a clear execution path when running a program on a single CPU core, and timers can happen at any time, disrupting this execution path. This can cause issues if a program is not written properly to account for that.

### Timer Commands

---

#### **WHEN TIMER <n> ... END WHEN**

Define a timer's interrupt handler. When the timer is enabled, jump to the WHEN TIMER block at the specified interval and return at the end of the block. WHEN TIMER blocks cannot be nested or inside a FUNCTION.

#### **TIMER <n> ON [EVERY <interval> [FOR <count>]]**

Enable timer <n>. <n> must be between 1 and 8. The timer will reset and run as last configured. Use the optional EVERY modifier to configure the timer to execute every <interval> milliseconds. The <interval> must be between 50 milliseconds and 86,400,000 milliseconds (24 hours). If the optional FOR modifier is also used, the timer will run <count> number of times every <interval> milliseconds. Otherwise, the timer will run indefinitely until explicitly disabled.

#### **TIMER <n> OFF**

Disable timer <n>. <n> must be between 1 and 8.

---

## Files

---

A file is a collection of data or information that is stored on a computer. It can be a document, image, video, audio, program, or any other type of digital content. Files are typically organized and stored in a hierarchical structure within folders or directories on a computer's storage system. Each file is identified by a unique name and extension, which indicates the type of file it is.

## File Commands

---

### **OPEN <path\$> FOR <access> AS [#]<handle>**

Open a file using the specified file <access>:

<b>INPUT</b>	Used for input  If the file does not exist an error is raised
<b>STREAM</b>	Used for input/output  If the file does not exist it will be created, otherwise it will be opened
<b>OUTPUT</b>	Used for output  If the file does not exist it will be created, otherwise it will be overwritten
<b>APPEND</b>	Used for output at the end of a file  If the file does not exist it will be created, otherwise it will be opened  Seek to the end of an existing file

Up to 80 (numbered 1 to 80) files can be opened at once. <handle> is the handle to assign when opening a file. <path\$> is a valid path and filename.

### **SEEK [#]<handle>,<offset>[,<origin>]**

Seek within the file assigned to <handle>. <offset> is the number of bytes to jump and can be negative if jumping backwards in the file. The optional <origin> specifies where the jump should occur from:

< 0	Beginning of file (default)
0	Current position in file
> 0	End of file

### **GET #<handle>,<var>[,<var>...]**

Read a single byte from the file assigned to <handle> and assign it to <var>. If the end of the stream has been reached then -1 is assigned. Multiple bytes can be read. The file must have been opened with INPUT or STREAM access.

### **PUT #<handle>,<byte>[,<byte>...]**

Write a single byte to the file assigned to <handle>. Multiple bytes can be written. The file must have been opened with STREAM, OUTPUT, or APPEND access.

### **READ #<handle>,<var>[,<var>...]**

Read a variable from the file assigned to <handle> and assign it to <var>. Multiple variables can be read. User-defined type variables must be read one field at a time. The file must have been opened with INPUT or STREAM access.

#### **WRITE #<handle>,<value>[,<value>...]**

Write a value to the file assigned to <handle>. Real values are written as 8 bytes, and String values are written as a length-prefixed ASCII string (also known as a P-string). Multiple values can be written. The file must have been opened with STREAM, OUTPUT, or APPEND access.

#### **INPUT #<handle>,<var>[,<var>...]**

Input an ASCII string (up to and including the next carriage return in the file) from the file assigned to <handle> and assign it to <var>. If <var> is a Real, then automatically try to convert the string to Real before assigning. Multiple values can be inputted. The file must have been opened with INPUT or STREAM access.

#### **PRINT #<handle>,<value>[,|;<value>...]**

Print the <value> as an ASCII string to the file assigned to <handle>. <value> can be a Real or String. Multiple values can be printed. If a comma ',' is used between values, a carriage return is also printed. The file must have been opened with STREAM, OUTPUT, or APPEND access.

#### **ARRAY LOAD #<n>,<array>()[,<array>()]**

Read an array from the file assigned to <handle> and assign it to <array>. Multiple arrays can be loaded. ARRAY LOAD cannot be used on arrays of user-defined types. The file must have been opened with INPUT or STREAM access.

#### **ARRAY SAVE #<n>,<array>()[,<array>()]**

Write <array> to the file assigned to <handle>. Multiple arrays can be saved. ARRAY SAVE cannot be used on arrays of user-defined types. The file must have been opened with STREAM, OUTPUT, or APPEND access.

#### **CLOSE [#]<handle>|ALL**

Close the file assigned to <handle>. If the ALL modifier is used instead, close all files.

#### **FILE RENAME <sourcePath\$>,<destPath\$>**

Rename the file at <sourcePath\$> to <destPath\$>. An absolute or relative path can be used in both <sourcePath\$> and <destPath\$>. The paths must point to the same directory, and the filenames must not match.

#### **FILE MOVE <sourcePath\$>,<destPath\$>**

Move the file at <sourcePath\$> to <destPath\$>. An absolute or relative path can be used in both <sourcePath\$> and <destPath\$>. The paths must not point to the same directory, and the filenames must match. If no filename is specified in <destPath\$>, the filename from <sourcePath\$> is used.

#### **FILE COPY <sourcePath\$>,<destPath\$>**

Copy the file at <sourcePath\$> to <destPath\$>. If a file of the same name already exists in the destination directory, then it is replaced. An absolute or relative path can be used in both <sourcePath\$> and <destPath\$>.

#### **FILE DELETE <path\$>**

Delete the file at <path\$>. An absolute or relative path can be used in <path\$>.

#### **CHDIR <path\$>**

Change the current directory to <path\$>.

### **MKDIR <path\$>**

Create a new directory <path\$>.

### **RMDIR <path\$>**

Remove the directory <path\$>. The directory must be empty before it can be removed.

---

---

## **Databases**

---

---

Liquid BASIC uses the SQLite library to access databases. SQLite is public domain software and is the most used database engine in the world. It is included in all mobile phones and most modern operating systems. SQLite provides a proven, robust database engine used every day by billions of computers worldwide.

### **Database Commands**

---

#### **DB <handle> NEW**

Create a new SQLite database in memory. <handle> is the handle to assign the database. Up to 20 (numbered 1 to 20) databases can be opened at once.

#### **DB <handle> OPEN <path\$>**

Open a SQLite database. <handle> is the handle to assign the database. <path\$> is a valid path and filename to a SQLite database. If the file does not exist, it will be created. Up to 20 (numbered 1 to 20) databases can be opened at once.

#### **DB <handle> LOG <path\$>**

Enable logging in the database assigned to <handle> so every transaction is recorded to a text file. <path\$> is a valid path and filename. If the file already exists it is appended to, otherwise the file is created.

#### **DB <handle> LOG OFF**

Disable logging in the database assigned to <handle>.

#### **DB <handle> GET <sql\$> TO <var\$>**

Query the database assigned to <handle> and return the value in the first column in the first row from the result set. <sql\$> is a valid SQL statement. The results are assigned to the variable <var\$>.

#### **DB <handle> QUERY <sql\$>**

Query the database assigned to <handle>. <sql\$> is a valid SQL statement (e.g. SELECT). The results are stored in a result set and can be read with the DB READ NEXT command.

#### **DB <handle> READ NEXT**

Read the next row in a result set from the last executed query on the database assigned to <handle>. Use the DB functions to read the table names, column values, and data types of the row. Use the EOQ function to determine if at the end of a result set.



## DB <handle> WRITE <sql\$> [TO <var>]

Send a SQL command to the database assigned to <handle>. <sql\$> is a valid SQL statement (e.g. UPDATE, INSERT, or DELETE). If the optional TO modifier is used, the number of rows affected by the command is assigned to the variable <var>.

## DB <handle> CLOSE

Close the database assigned to <handle>.

---

---

# Artificial Intelligence

---

---

Liquid BASIC can also interface with OpenAI's APIs to support artificial intelligence. An OpenAI API key is required to make successful API calls. Sign up with OpenAI at [openai.com](https://openai.com) to create an account and generate your API key. Once you have your API key, paste it into a text file named **"OpenAI API Key.txt"** in the same directory as the Liquid BASIC executable (this is generally **"C:\Program Files\Liquid BASIC"** if using the default directory in the Windows installer). The next time Liquid BASIC starts up it will automatically load the OpenAI API key and validate it (it will print the word **"COPILOT"** next to the memory and BASIC lines available if successful). Use the COPILOT reserved variable to programmatically test whether the API key was successfully loaded.



```
Liquid BASIC 2025

**** Liquid BASIC 2025 ****

63.8GB RAM system 65536 BASIC lines available Copilot

Enter HELP for help or press CTRL and F12 keys together for Menu

Ready.
PRINT COPILOT
-1

CODE "Write a program to input a string and print it in reverse without using functions."

LIST
LEGACY MODE
00000 ' AI Generated: "Write a program to input a string and print it in reverse without using functions."
00010 INPUT "Enter a string: ", S$
00020 FOR I = LEN(S$) TO 1 STEP -1
00030 PRINT MID$(S$, I, 1);
00040 NEXT I
00050 END

RUN
Enter a string: Hello world!
!dlrow olleH
```

See <https://help.openai.com/> for more information.

## About Models

---

Liquid BASIC can use multiple OpenAI models. A model refers to a machine learning model. Models are designed to perform various natural language processing tasks, such as text generation, language translation, question answering, and more. These models are trained on large datasets using deep learning techniques, allowing them to understand and generate human-like text. OpenAI models, such as GPT-3.5 (Generative Pre-trained Transformer 3.5), can generate coherent and contextually relevant text across a wide range of applications. In fact, most of this paragraph was generated by GPT-3.5. 😊

For text data, such as chat, chatbots, and code generation, Liquid BASIC can use either the GPT-3.5 or GPT-4 models. The GPT-3.5 model is cheaper and faster than GPT-4. However, GPT-4 is more capable than GPT-3.5 and can do more complex tasks. These models also use “tokens” when dealing with text. You can think of tokens as pieces of words, where 1,000 tokens is about 750 words.

Liquid BASIC uses the DALL-E 3 model for image generation. DALL-E 3 generates good standard and high-resolution images.

## About Chat and Chatbots

---

Chat allows users to ask questions on a wide range of topics, whether general knowledge or specific subjects like history, math, science, and more. A chatbot is a computer program designed to simulate conversation with human users through text interactions. It uses artificial intelligence to understand and respond to user queries or commands in a conversational manner.

## Chat Commands

---

### CHAT <prompt\$> [TO <var\$>]

Submit a one-off <prompt\$> to ChatGPT. This instructs ChatGPT to answer without any instructions, use a sampling temperature of 0.1, generate a maximum of 4000 tokens, and use the GPT-4 model for best results.

If the optional TO modifier is used, the results are assigned to the variable <var\$>. Otherwise, the results are printed to the Text Screen.

### CHAT <prompt\$>,<instructions\$>,<temperature>,<tokens> [MODEL <model\$>] [TO <var\$>]

Submit <prompt\$> to ChatGPT. <instructions\$> are any instructions to give ChatGPT when formulating an answer (for example, “Answer as a haiku.”). <temperature> is the sampling temperature to use. Higher values mean the model will take more risks. Try 0.9 for more creative applications, and 0 for ones with a well-defined answer. <tokens> is the maximum number of tokens to return and must be between 16 and 4000. If the optional MODEL modifier is used, the default model is overridden by <model\$>.

If the optional TO modifier is used, the results are assigned to the variable <var\$>. Otherwise, the results are printed to the Text Screen.

### CHAT MODEL <model\$>

Set the default model to use for chat. <model\$> must be one of the following:

- gpt35 (default) Set the model to use for chat to GPT-3.5
- gpt4 Set the model to use for chat to GPT-4

## Chatbot Commands

---

### CHATBOT NEW

Start a new conversation with an interactive chatbot.

### CHATBOT INSTRUCT <instructions\$>

Submit <instruction\$> to the chatbot using the System role. Instructions help set the behavior of the chatbot.

### CHATBOT INPUT <prompt\$>

Submit <prompt\$> to the chatbot using the User role. Input is used to request a response from the chatbot. It can be provided by the end user or programmatically generated by the programmer.

### CHATBOT EXAMPLE OUTPUT <output\$>

Submit <prompt\$> to the chatbot using the Assistant role. Example output is used to give examples of desired behavior.

### CHATBOT RESPONSE TO <var\$>

Call the ChatGPT API to get a response and wait until the response is delivered. The response is assigned to <var\$>.

### CHATBOT MODEL <model\$>

Set the default model to use for chatbots. <model\$> must be one of the following:

- gpt35 (default) Set the model to use for chatbots to GPT-3.5
- gpt4 Set the model to use for chatbots to GPT-4

## About Code Generation

---

Code generation uses artificial intelligence to automatically generate computer code. Code generation can be used to automate repetitive or time-consuming coding tasks, such as generating boilerplate code, writing code snippets, or even creating entire programs:

```
Liquid BASIC 2025
CODE "Print the first 300 prime numbers."
LIST
LEGACY MODE
00000 ' AI Generated: "Print the first 300 prime numbers."
00010 DIM PRIME(300)
00020 PRIME(1) = 2
00030 N = 1
00040 FOR I = 3 TO 9999
00050     ISPRIME = 1
00060     FOR J = 1 TO N
00070         IF I MOD PRIME(J) = 0 THEN
00080             ISPRIME = 0
00090             EXIT FOR
00100         END IF
00110     NEXT J
00120     IF ISPRIME = 1 THEN
00130         N = N + 1
00140         PRIME(N) = I
00150         IF N = 300 THEN EXIT FOR
00160     END IF
00170 NEXT I
00180 FOR I = 1 TO 300
00190     PRINT PRIME(I)
00200 NEXT I
```

When generating code, a default set of rules is used to instruct the model how to write proper Liquid BASIC programs. Although the model is instructed to follow these rules pertaining to Liquid BASIC, some editing or revisions may be required for the generated code to run properly. Upon startup, Liquid BASIC loads the default rule set:

- Write a modern structured BASIC program.
- Only present the completed program, no explanation is needed.
- Programs should have line numbers.
- Blocks of code should be indented. Use four spaces to indent blocks.
- All string variables, string arrays, and string functions should end with a dollar sign.
- Individual string variables do not need to be dimensioned.
- Arrays always use parentheses for indexes, never brackets.
- Do not use "POS" as a variable name.
- Do not use "STR\$" as a variable name. Use "S\$" instead.
- The "RND" function takes one argument: 1 to generate a random number, 0 to repeat the last generated random number.
- Use the "DIV" keyword instead of the "\" operator for integer division.
- Use the "MOD" keyword instead of the "%" operator for modulus operations.
- Use the "ELSE IF" keywords instead of the "ELSEIF" keyword.
- Use the "END IF" keywords instead of the "ENDIF" keyword.
- Use the "WEND" keyword instead of the "ENDWHILE" keyword.
- Use the "PAUSE" keyword to pause for milliseconds.
- Use the "SLEEP" keyword to pause for seconds.
- Functions and subroutines should be in a "FUNCTION" block.

- A "FUNCTION" block returns a result by assigning a value to the name of the function.
- If a function returns a string, its name should end with a dollar sign.

The CODE.RULES\$ reserved variable returns the default rule set.

---

## Code Generation Commands

---

### CODE <prompt\$> [TO <var\$>()]

Submit <prompt\$> to ChatGPT to generate a BASIC program. This instructs ChatGPT to use the default rule set, use a sampling temperature of 0, generate a maximum of 4000 tokens, and use the GPT-4 model for best results.

If the optional TO modifier is not used, the results are put into program memory. It will raise an error if there is already a program currently in memory. Use NEW to clear that program first. If the optional TO modifier is used, the results are assigned to the array <var\$>.

This command cannot be used in direct mode if the optional TO modifier is not used.

### CODE <prompt\$>,<instructions\$>,<temperature>,<tokens> [MODEL <model\$>] [TO <var\$>()]

Submit <prompt\$> to ChatGPT to generate a BASIC program. <instructions\$> are any instructions to give ChatGPT when generating the program. <temperature> is the sampling temperature to use. Higher values mean the model will take more risks. Try 0.9 for more creative applications, and 0 for ones with a well-defined answer. <tokens> is the maximum number of tokens to return. <tokens> must be between 16 and 4000. If the optional MODEL modifier is used, the default model is overridden by <model\$>.

If the optional TO modifier is not used, the results are put into program memory. It will raise an error if there is already a program currently in memory. Use NEW to clear that program first. If the optional TO modifier is used, the results are assigned to the array <var\$>.

This command cannot be used in direct mode if the optional TO modifier is not used.

### CODE MODEL <model\$>

Set the default model to use for code generation. <model\$> must be one of the following:

- gpt35 (default) Set the model to use for code generation to GPT-3.5
- gpt4 Set the model to use for code generation to GPT-4

---



---

## Resources

---



---

A resource is a block of memory used for a specific task, such as a bitmap or sound. Types of resources include:

- Bitmap digital image
- Font typeface
- Tileset set of tiles used to construct Tilemaps
- Tilemap grid of tiles
- Shader OpenGL fragment shader
- Sound digital audio

Each type of resource is allocated 4096 instances. Each instance has a unique integer ID to identify it, numbered 1 to 4096. Use the FREE reserved variables (e.g. FREE.BITMAP or FREE.SOUND) to return the next unused ID for a given resource.

Supported file formats that can be loaded from and saved to include:

RESOURCE	LOAD FILE FORMATS	SAVE FILE FORMATS
<b>Bitmap</b>	BMP, GIF, ICO, IFF, JPG, LBM, PNG, TGA, WEBP, XML (Sprite sheets)	BMP, GIF (in Black & White), JPG, PNG, TGA
<b>Font</b>	OTF, TTF	-
<b>Tileset</b>	LTS, some TSX	LTS
<b>Tilemap</b>	LTM, some TMX	LTM
<b>Shader</b>	FS (Fragment Shader)	-
<b>Sound</b>	WAV, MP3, CDA, AAC, ADT, ADTS, FLAC	-

Resources allow memory to be abstracted, allowing almost unlimited approaches to graphics and sound. For example, Bitmaps can be rendered or AI generated off-screen, allowing for procedurally generated artwork. Tilesets can be quickly updated, giving Tilemaps “cheap” (performant) full screen animation. Tilemaps can be stacked for smooth parallax scrolling. The possibilities are endless.

## Chaining Modifiers

Commands that use modifiers can have the modifiers “chained” together on a single line for convenience. For example:

**SPRITE n SHOW MOVE 320,200 SCALE 2,2**

is the same as:

**SPRITE n SHOW**

**SPRITE n MOVE 320,200**

**SPRITE n SCALE 2,2**

---

---

## Bitmap Resources

---

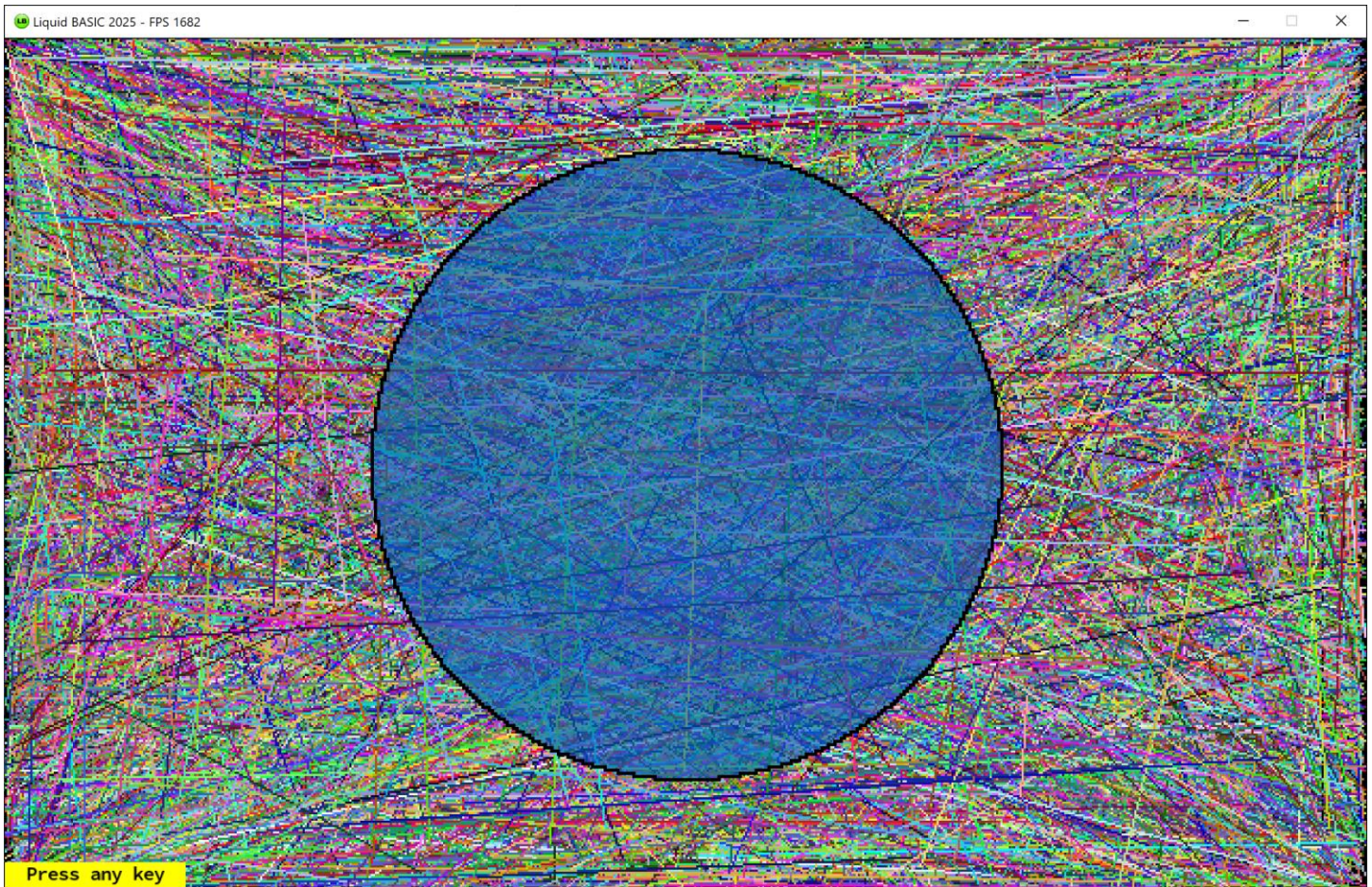
---

Bitmaps are 2D rectangular grids of pixels. They can be any size from 1x1 pixel up to the graphics card limit (typically at least 4096x4096 pixels on modern PCs). Each pixel is a color value and can be directly read from and written to. Anytime a pixel is written to it is first clipped to the current clipping region (pixels outside of the clipping region are not written) and the current pixel operator and pixel mask are applied (unless otherwise noted).

The pixel operator takes the “source” pixel (the pixel being written) and the “destination” pixel (the pixel presently in the Bitmap) and applies an operator to it. The simplest (and default) operator is the write operator: the source pixel simply overwrites the destination pixel. The blend operator is also useful: the source pixel is blended with the destination pixel to allow transparency effects. Other operators are used to fade pixels in or out, or average two pixels together.



The pixel mask is a 32-bit mask to apply to each pixel written. It defines which bits in a 32-bit color can pass through and be written. By default, all 32-bits are enabled, so all bits are written. With pixel masks it is possible to restrict pixel writes to the red, green, blue, or alpha bytes that comprise a color, or some combination of bits thereof.



## Bitmap Modifiers

---

### **BITMAP <id> NEW <width>,<height>**

Create a new Bitmap that is <width> pixels across and <height> pixels down and assign it to <id>. <width> and <height> must be between 1 and 4096.

### **BITMAP <id> TEXTURE <command\$>[,<arguments\$>]**

Create a new 256x256 Bitmap using the <command\$> texture generator (see below) and assign it to Bitmap <id>. The <command\$> is automatically converted to lowercase letters and all leading and trailing spaces removed. Optional <arguments\$> can be passed in a comma-separated list. Use the EXPORT\$ function to help with building a comma-separated list from one or more expressions.

### **BITMAP <id> COPY BITMAP <bitmapId>**

Copy the Bitmap <bitmapId> and assign it to the unique identifier <id>.

### **BITMAP <id> LOAD <path\$>**

Load a Bitmap resource and assign it to the unique identifier <id>. <path\$> is the valid path and filename of a file in a supported Bitmap file format.

#### **BITMAP <id> LOAD SHEET <path\$> [STRIP <extension\$>]**

Load a Sprite Sheet and assign it to the unique identifier <id>. <path\$> is the valid path and filename of a file in a supported Sprite Sheet file format. Sprite Sheets must be in XML format and define the full Bitmap to load in, as well as the names and coordinates for each subtexture in the Bitmap. The optional STRIP modifier will remove <extension\$> from the end of each name if it is present.

#### **BITMAP <id> SAVE <path\$>**

Save the Bitmap <id> to disk in a supported Bitmap file format.

#### **BITMAP <id> RESCALE <size>**

Rescale the Bitmap assigned to <id> to the size <size> while preserving the aspect ratio. Cannot be used on Sprite Sheets.

#### **BITMAP <id> RESCALE <width>,<height>**

Rescale the Bitmap assigned to <id> to the width <width> and <height> specified. Cannot be used on Sprite Sheets.

#### **BITMAP <id> FREE**

Free a Bitmap from memory.

### **Textures and Arguments**

---

#### **fill <color>**

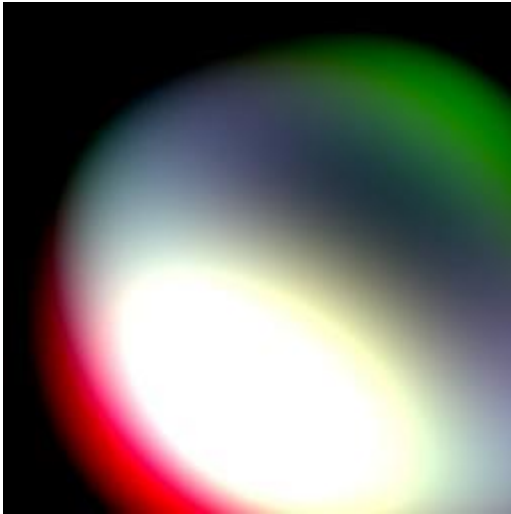
Create a new 256x256 Bitmap and fill it with <color>.



#### **blobs <seed>,<amount>,<rgbFlag>**

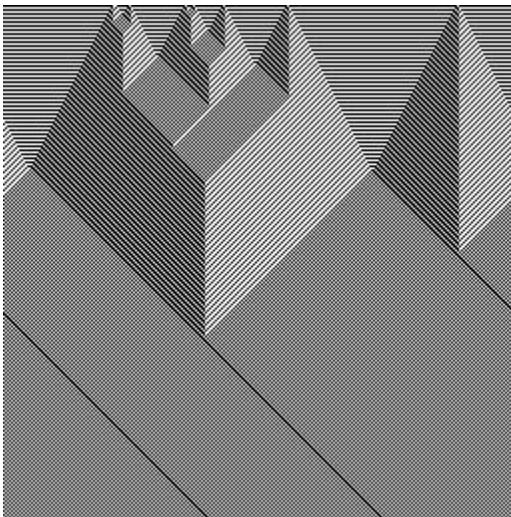
Create a new 256x256 Bitmap using Blobs. <rgb> is 1 for color, or 0 for greyscale.





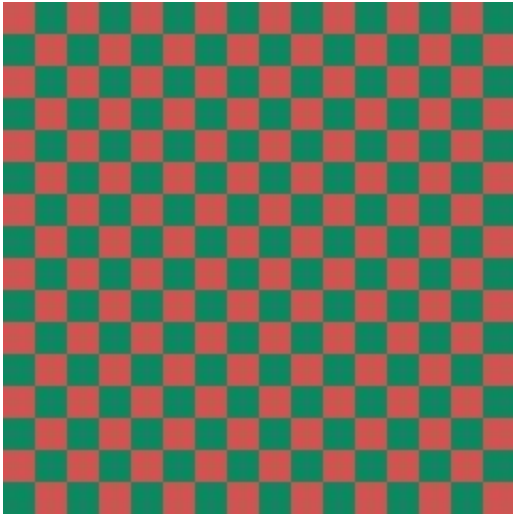
**cellmachine <seed>,<rule>**

Create a new 256x256 Bitmap using a Cell Machine.



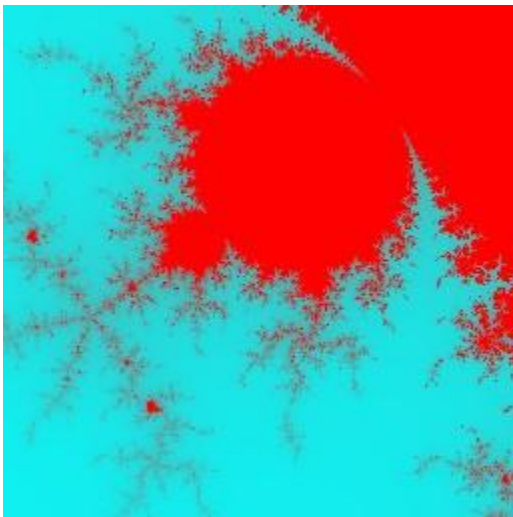
**checkerboard <dx>,<dy>,<color1>,<color2>**

Create a new 256x256 Bitmap using a Checkerboard pattern. <dx>,<dy> is the size (in pixels) of each block in the checkerboard. <color1> and <color2> are the colors to swap between.



**mandelbrot <p1>,<p2>,<p3>,<p4>**

Create a new 256x256 Bitmap using a Mandelbrot.



**particle <size>**

Create a new 256x256 Bitmap using a Particle. <size> is the size (in pixels) of the particle.



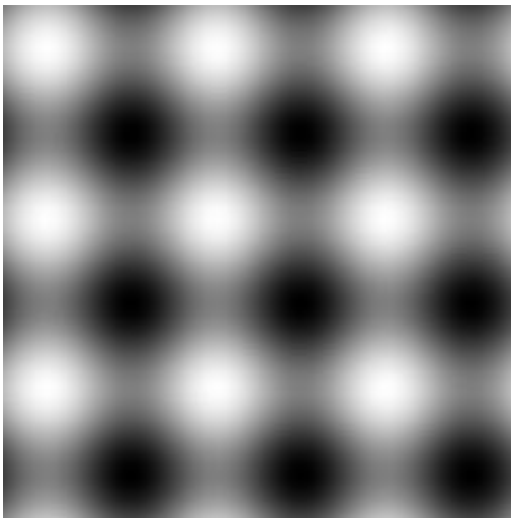
**perlinnoise <dist>,<seed>,<amplitude>,<octaves>,<persistence>,<rgb>**

Create a new 256x256 Bitmap using Perlin Noise. <rgb> is 1 for color, or 0 for greyscale.



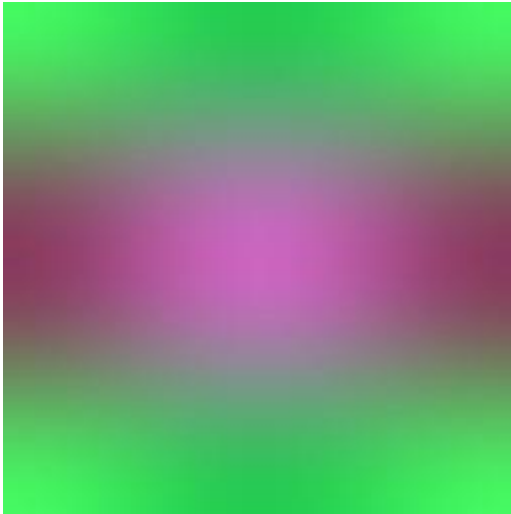
**sine <dx>,<dy>,<amplitude>**

Create a new 256x256 Bitmap using Sine.



**subplasma <dist>,<seed>,<amplitude>,<rgb>**

Create a new 256x256 Bitmap using Sub Plasma.



---

---

### Font Resources

---

---

Fonts are used when rendering text in a Bitmap. Liquid BASIC supports both TTF (TrueType Font) and OTF (OpenType Font) formats.

#### Font Modifiers

---

##### **FONT <id> LOAD <path\$>,<size>**

Load a Font from disk and assign it to the unique identifier <id>. <path\$> is the valid path and filename of a file in a supported Font file format. <size> is the point size of the Font. Point size refers to the height of the font. In digital publishing, 1 point = 1/72 of an inch.

##### **FONT <id> FREE**

Free a Font from memory.

---

---

### Tileset Resources

---

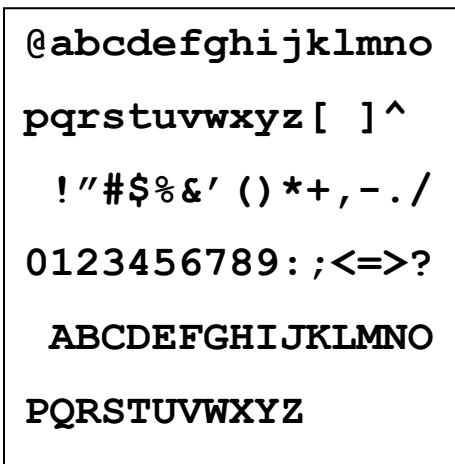
---

Tiles are small (usually 8x8 pixels or 16x16 pixels) pieces of a larger Bitmap that are collected into a Tileset. A Tileset can contain any number of tiles, limited only by the size of the Bitmap that it is built on. Individual tiles can be cleared, customized, and scrolled.

Liquid BASIC has a default (Commodore 64-based) Tileset. This Tileset has 512 8x8 pixel tiles, with a combination of uppercase, lowercase, and graphics characters (known as PETSCII):



For the default Tileset, tiles 256 to 351 are mapped to their respective ASCII character codes:



## Tileset Modifiers

---

### TILESET <id> NEW <width>,<height>

Create a new, blank Tileset with 256 tiles. Each tile is <width> by <height> pixels large. Tiles must be custom defined. No characters are mapped.

### TILESET <id> NEW <width>,<height>,<bitmapId>[,<spacing>]

Create a new Tileset using the Bitmap <bitmapId> for pixel data. Each tile is <width> by <height> pixels large. The optional <spacing> specifies the spacing in pixels between tiles and is 0 by default. The number of tiles created depends on how many tiles can fit in the size of the Bitmap. For example, a 512x256 Bitmap with 8x8 tiles and 0 spacing will have 2048 tiles:

- $512/8 = 16$ ,  $256/8 = 8 \rightarrow 16 \times 8 = 2048$  tiles

No characters are mapped.

### **TILESET <id> NEW <key\$>**

Create a new Tileset based on one of the five built-in:

KEY	TILE SIZE	CHARACTERS
BASIC	10x25	32 to 255 (Windows code page 1252)
C64	8x8	32 to 511 (PETSCII)
NES	8x8	32 to 127 (ASCII)
RETRO	8x16	32 to 127 (ASCII)
MSDOS	8x16	32 to 255 (extended ASCII; code page 437)

### **TILESET <id> COPY TILESET <tilesetId>**

Copy the Tileset <tilesetId> and assign it to the unique identifier <id>.

### **TILESET <id> LOAD <path\$>**

Load a Tileset resource and assign it to the unique identifier <id>. <path\$> is the valid path and filename of a file in a supported Tileset file format. Liquid BASIC does support limited loading of .TSX files, an XML file format used by the popular Tiled app. When using .TSX files the image data should be stored as a file and not embedded. Any embedded data should be embedded as CSV and not Base64 data.

### **TILESET <id> SAVE <path\$>**

Save the Tileset <id> to disk in a supported Tileset file format.

### **TILESET <id> MAP <ch>[,count] TO <tile>**

Map the ASCII character <ch> (and the number of <count> characters after) to <tile>. Whenever a string is put on-screen (using the PUT TILES command) or printed (using the TPRINT command) the ASCII characters in the string are looked up and mapped to their corresponding tiles. For example, ASCII character 65 "A" does not necessarily need to map to tile 65; it can map to any tile in the Tileset.

### **TILESET <id> MAP <text\$> TO <tile>**

Map the ASCII characters in <text\$> to the tiles starting at <tile>.

### **TILESET <id> MAP ALL**

Map all available ASCII characters starting at tile 0.

### **TILESET <id> FREE**

Free a Tileset from memory.



A Tilemap is a 2D rectangular grid of “cells”. They can be any size from 1x1 cell up to 256x256 cells. Individual cells contain a tile, an independent foreground (ink) color, and an independent background (highlight) color. Each cell also has a set of tile attributes: mirror, flip, and rotate. These attributes can be turned on and off and used in any combination. Tiles can be directly read from and written to memory.

Because Tilemaps use indexes to tiles, which in turn represent patterns of pixels in a larger Bitmap, entire sections of the screen can be quickly filled, manipulated, and scrolled by simply changing the indexes. Individual tiles can also be customized, and changes to a tile are reflected anywhere that tile is used in a Tilemap. This allows large portions of the screen to be animated or modified while changing a minimal amount of data. The smaller datasets that Tilemaps work with generally perform faster than the same operations in Bitmaps. However, Bitmaps allow more flexibility in that every single pixel can be independently manipulated.



## Tilemap Modifiers

---

### TILEMAP <id> NEW <columns>,<rows>

Create a new Tilemap using the default Commodore 64-based PETSCII tileset (featuring 512 8x8 tiles).

<columns> is the number of cells horizontally, <rows> is the number of cells vertically. Both must be between 1 and 256.

### TILEMAP <id> NEW <columns>,<rows>,<tilesetId>

Create a new TileMap using the Tilesset <tilesetId>. <columns> is the number of cells horizontally, <rows> is the number of cells vertically. Both must be between 1 and 256.

## **TILEMAP <id> SET TILESET <tilesetId>**

Change the Tilemap's Tileset to the Tileset <tilesetId>.

## **TILEMAP <id> COPY TILEMAP <tilemapId>**

Copy the Tilemap <tilemapId> and assign it to the unique identifier <id>.

## **TILEMAP <id> LOAD <path\$>[,<layer>]**

Load a Tilemap resource and assign it to the unique identifier <id>. <path\$> is the valid path and filename of a file in a supported Tilemap file format. Liquid BASIC does support limited loading of .TMX files, an XML file format used by the popular Tiled app. When using .TMX files the image data should be stored as a file and not embedded. Any embedded data should be embedded as CSV and not Base64 data. The optional <layer> specifies which "layer" of the Tilemap to load and is only used with .TMX files (a .TMX file can include more than one layer).

## **TILEMAP <id> SAVE <path\$>**

Save the Tilemap <id> to disk in a supported Tilemap file format.

## **TILEMAP <id> FREE**

Free a Tilemap from memory.

---

---

## **Shader Resources**

---

---

A fragment shader is a program that is executed on the GPU (Graphics Processing Unit) for each fragment (or pixel) of a Sprite. It is responsible for determining the final color of each fragment based on various factors such as lighting, textures, and other effects. Fragment shaders are written in the OpenGL Shading Language (GLSL) and are used to customize the appearance of Sprites.

### **Shader Modifiers**

---

#### **SHADER <id> LOAD <path\$>**

Load a Fragment Shader from disk and assign it to the unique identifier <id>. <path\$> is the valid path and filename of a file in a supported Shader file format.

#### **SHADER <id> FREE**

Free a Shader from memory.

---

---

## **Sound Resources**

---

---

### **Sound Modifiers**

---



### **SOUND <id> LOAD <path\$>**

Load a Sound resource and assign it to the unique identifier <id>. <path\$> is the valid path and filename of a file in a supported Sound file format.

### **SOUND <id> PLAY**

Play a Sound starting at the beginning.

### **SOUND <id> PAUSE**

Pause a Sound that is playing.

### **SOUND <id> RESUME**

Resume a Sound that was paused.

### **SOUND <id> STOP**

Stop a Sound that is playing.

### **SOUND <id> VOLUME <volume>**

Set the volume of the Sound. Valid <volume> ranges are between 0 (mute) and 100.

### **SOUND <id> FREE**

Free a Sound from memory.

---

---

## **Graphics**

---

---

Graphics in Liquid BASIC are comprised of a single Screen that can be Bitmap-based or Tilemap-based, and up to 256 independent Sprites.

Sprites are 2D moveable objects. They can be Bitmap-based or Tilemap-based and can be any size. Sprites have a definable priority to determine the order they are rendered and can be placed above or under the Screen. They can be shown, hidden, moved (translated), scaled, rotated, and tinted. They can also be “fit” to the window (center, stretch, fit, fill, or tile), and they can be mirrored horizontally or flipped vertically. Sprites also have simple collision detection to determine when two or more Sprites hit each other.

With the Screen and Sprites, it is possible to create impressive demos, games, visualizations, and other graphical effects. Sprites can fill the entire Liquid BASIC window. Multiple Sprites can be stacked for parallax scrolling, transparency, and other special effects. Sprites can also be used for individual characters, NPC’s (Non-Playable Characters), “missiles”, and other moving objects.

### **Targeting**

---

Both Bitmaps and Tilemaps can be “targeted”. Because the Screen can be a Bitmap or Tilemap itself, it can be targeted as well. When a Bitmap is targeted, all subsequent Bitmap commands are directed at the Bitmap. When a Tilemap is targeted, all subsequent Tilemap commands are directed at the Tilemap and all subsequent Tileset commands are directed at the Tilemap’s Tileset. Targeting allows Bitmaps and Tilemaps to be rendered off-screen. Only one resource can be targeted at a time. A resource must be targeted before it can be read from/written to.

## Clear Command

---

### CLEAR ALL

Perform the following actions:

- Set the border color and size to 0
- Set the background color to black
- Reset the ink and highlight colors
- Reset the attributes (blink, bold, and underline)
- Reset the pixel operator, pixel mask, and line stipple
- Reset the tile palette (used with CUSTOM TILE)
- Reset all resources
- Reset the speech synthesizer
- Clear the Screen, Sprites, and Text Screen

It is good practice to start your program with this command to ensure a clean slate at startup.

## Screen Commands

---

### SCREEN <mode>

Change the Screen to a predefined <mode>:

0	No Screen (same as calling SCREEN FREE)
1	40x25 Tilemap
2	80x50 Tilemap
3	160x100 Tilemap
4	320x200 Bitmap
5	640x400 Bitmap
6	1280x800 Bitmap

The Screen is stretched to fill the entire window.

SCREEN also redirects the “target” of Bitmap or Tilemap commands so all subsequent commands will render to the Screen, depending on whether the mode is a Bitmap or Tilemap.

### SCREEN BITMAP <width>,<height> [CENTER|STRETCH|FILL|FIT|TILE|XYSIZE <xs>,<ys>]

Create a custom-sized Bitmap screen that is <width> pixels across and <height> pxels down. <width> and <height> must be between 1 and 4096. The optional modifiers at the determine how the Bitmap should be displayed:

- CENTER            Center the screen
- STRETCH          Stretch the screen to fill the window while ignoring the aspect ratio (default)
- FILL              Scale the screen to fill the window while preserving the aspect ratio
- FIT                Scale the screen to fit within the window while preserving the aspect ratio
- TILE                Tile the screen to fill the entire window
- XYSIZE xs,ys     Scale the screen by <xs> across and <ys> down

SCREEN BITMAP also redirects the “target” of Bitmap commands, so all subsequent Bitmap commands are directed to the Bitmap.

## **SCREEN TILEMAP <columns>,<rows>[,<tilesetId>] [CENTER|STRETCH|FILL|FIT|TILE|XYSIZE <xs>,<ys>]**

Create a custom-sized Tilemap screen. <columns> is the number of cells horizontally, <rows> is the number of cells vertically. Both must be between 1 and 256. The default Commodore 64-based PETSCII tileset is used unless the optional <tilesetId> is specified. The optional modifiers at the end determine how the Tilemap should be displayed.

SCREEN TILEMAP also redirects the “target” of Tilemap commands, so all subsequent Tilemap and all subsequent Tileset commands are directed to the Tilemap and Tilemap’s Tileset.

## **SCREEN LOAD <path\$> [CENTER|STRETCH|FILL|FIT|TILE|XYSIZE <xs>,<ys>]**

Load a Tilemap or Bitmap and create a custom-sized screen for it. <path\$> is the valid path and filename of a file in a supported Bitmap or Tilemap file format. The optional modifiers at the end determine how the Tilemap or Bitmap should be displayed.

## **SCREEN SAVE <path\$>**

Save the current screen to disk in a supported Bitmap or Tilemap file format.

## **SCREEN FREE**

Free the Screen from memory.

## **Target Commands**

---

### **TARGET SCREEN**

Redirects the target of Bitmap or Tilemap commands so all subsequent commands will render to the Screen, depending on whether the Screen is a Bitmap or Tilemap.

### **TARGET BITMAP <n>**

Target the Bitmap <id>. All subsequent Bitmap commands are directed at the Bitmap.

### **TARGET TILEMAP <n>**

Target the Tilemap <id>. All subsequent Tilemap commands are directed at the Tilemap and all subsequent Tileset commands are directed at the Tilemap’s Tileset.

### **TARGET TILESET <n>**

Target the Tileset <id>. All subsequent Tileset commands are directed at the Tileset.

## **Buffering Commands**

---

### **SINGLE BUFFER**

Make the targeted Tilemap or Bitmap single buffered. Subsequent commands draw to the front buffer, so changes are immediately visible.

### **DOUBLE BUFFER**

Make the targeted Tilemap or Bitmap double buffered. Subsequent commands draw to the back buffer in memory, so changes are not visible. When the buffers are swapped with the SWAP BUFFERS command the back buffer becomes visible (front buffer) and the front buffer then becomes the back buffer. This is useful during

animation to prevent “screen tearing” and flickering, when the Tilemap or Bitmap is shown before updating has completed.

## SWAP BUFFERS [CLS]

Swap the back and front buffer in a double buffered Tilemap or Bitmap. If the optional CLS modifier is used the back buffer is cleared after the swap.

## Bitmap Commands

---

### CLEAR BITMAP

Clear the current targeted Bitmap.

### CLIP <x1>,<y1>,<x2>,<y2>

Set the clipping region. Pixels outside of the region are not written.

### PIXEL OPERATOR <op>

Set the current pixel operator. The pixel operator is applied anytime a pixel is written to a Bitmap. <op> must be between 1 and 16. Several constants are included to simplify pixel operators:

	CONSTANT	ACTION
1	PXO_WRITE (default)	overwrite the destination (d) pixel with the source (s) pixel
2	PXO_BLEND	blend source and destination pixels using the alpha channel
3	PXO_MIN	$d = \min(s, d)$
4	PXO_MAX	$d = \max(s, d)$
5	PXO_AND	$d = s \& d$
6	PXO_OR	$d = s \mid d$
7	PXO_XOR	$d = s \wedge d$
8	PXO_ADD	$d = s + d$
9	PXO_SUB	$d = s - d$
10	PXO_AVG	$d = (s + d) \gg 1$
11	PXO_MULT	$d = (s * d) / 255$
12	PXO_ZERO	only write if destination pixel has no RGB data
13	PXO_REPLACE	only write if destination pixel has RGB data
14	PXO_ALPHATEST	only write if source pixel has alpha data
15	PXO_MASK	$d = \text{ink}$ if source pixel has alpha data
16	PXO_COLLISION	$d = \text{ink}$ if source pixel has alpha data, record “collisions”

Pixel operators act on a pixel’s individual bytes, not the entire 32-bit pixel. For example, the add pixel operator will add the red byte, the green byte, the blue byte, and the alpha byte independently. Each will be clamped to 255 so it doesn’t exceed what can be stored in a single byte. Conversely, the subtract pixel operator will clamp each byte to 0 when subtracting.

When a pixel is written with the PXO\_COLLISION pixel operator the destination pixel being replaced is saved in a “collision list”. This collision list can then be queried using the PIXEL.COLLISION function to see if a particular pixel was overwritten. This is useful for pixel perfect collision between software sprites, such as Bitmaps and BOB’s. Create a hidden Bitmap the same size as the current targeted Bitmap, assign a solid color to each object you want to test for collisions, and render each object using PXO\_COLLISION on the hidden Bitmap and use the

PIXEL.COLLISION function to see if any colors overlap. The collision list is cleared whenever the Bitmap is cleared.

### **PIXEL MASK <mask>**

Set the current pixel mask. The pixel mask is applied anytime a pixel is written to a Bitmap. The pixel mask is referenced in ABGR format, where A is the alpha byte, B is the blue byte, G is the green byte, and R is the red byte. Each byte is 8 bits and can hold a value between 0 and 255.

Some useful <mask> values:

- \$FFFFFFFF write the entire pixel (default)
- \$FF0000FF only write the red-alpha bytes
- \$FF00FF00 only write the green-alpha bytes
- \$FFFF0000 only write the blue-alpha bytes
- \$00FFFFFF write the red-green-blue bytes (but not the alpha byte)

### **LINE STIPPLE <stipple>**

Define the 32-bit line stipple pattern (each bit represents which pixels to draw when drawing lines). This is useful for drawing dotted and dashed lines.

Some useful <stipple> values:

- \$FFFFFFFF solid line (default)
- \$55555555 dotted line (small dots)
- \$33333333 dotted line (big dots)
- \$0F0F0F0F dashed line

### **SET FONT <id>**

Set the font to use when printing text with the TEXT command to Font <id>.

### **SET TILESET <id>**

Set the tileset to use when blitting tiles with the BLIT TILE and BLIT TILES commands to Tileset <id>.

### **POKE <index>,<color>**

Poke (write directly to memory) the pixel <color> at <index>. Clipping, the pixel operator, and the pixel mask are ignored when poking a pixel. Use the PEEK function to read the pixel back.

### **SWAP PIXELS <oldColor>,<newColor> [WINDOW <x1>,<y1>,<x2>,<y2>]**

Swap <oldColor> with <newColor>. Use the optional WINDOW modifier to swap a section of the Bitmap and not the entire Bitmap. Clipping, the pixel operator, and the pixel mask are ignored when swapping pixels.

### **PLOT <x>,<y>[,<x>,<y>...]**

Plot a pixel at <x>,<y> using the current ink color. Multiple pixels can be plotted. Use the POINT and SAMPLE functions to read the pixel back.

### **HLINE <x1>,<x2>,<y>**

Draw a horizontal line from <x1>,<y> to <x2>,<y>.

### **VLINE <x>,<y1>,<y2>**

Draw a vertical line from <x>,<y1> to <x>,<y2>.

**LINE** <x1>,<y1>,<x2>,<y2> [TO <x3>,<y3>...]

Draw a line from <x1>,<y1> to <x2>,<y2>. Multiple lines can be drawn with the TO modifier.

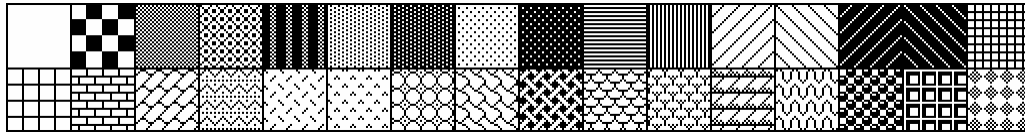
**SMOOTH LINE** <x1>,<y1>,<x2>,<y2> [TO <x3>,<y3>...]

Draw a smooth, anti-aliased line from <x1>,<y1> to <x2>,<y2>. Multiple lines can be drawn with the TO modifier.

**BRUSH** <brushId>

A brush is a Bitmap that is used when filling graphics primitives. Several Bitmap commands – like FILL, STRIP, RECT, AREA, CIRCLE, ELLIPSE, SLICE, and FLOOD FILL – allow the shape to be filled with a solid color or a repeating pattern.

The default brush is brush 1 (solid color). There are 32 reserved brushes. They provide a variety of different 32x32 bit patterns to use:



**BRUSH BITMAP** <id>

Select Bitmap <id> to use as a brush when filling graphics primitives.

**FILL** [<color>]

Fill the entire Bitmap with the specified <color> using the active brush. If <color> is omitted the current ink is used.

**STRIP [FILL]** <x1>,<y1>,<x2>,<y2>,<x3>,<y3>

Draw a triangle from <x1>,<y1> to <x2>,<y2> to <x3>,<y3>. Use FILL to fill the triangle using the active brush.

**RECT [FILL]** <x1>,<y1>,<x2>,<y2>

Draw a rectangle from <x1>,<y1> to <x2>,<y2>. Use FILL to fill the rectangle using the active brush.

**AREA [STEP]** <x>,<y>

Specify a single vertex in a convex polygon. A convex polygon is a polygon in which all its interior angles are less than 180 degrees, and all its vertices point outwards. In other words, no line segment connecting two points on the interior of the polygon will intersect with the exterior of the polygon. Up to twenty vertices can be submitted for a single polygon.

If STEP is included, <x> and <y> are offsets from the last vertex specified. Otherwise, <x> and <y> are absolute coordinates.

Once the vertices have been submitted, use the AREA FILL command to draw the polygon.

**AREA CLR**

Discard the vertices submitted by the AREA command.

**AREA FILL**

Fill in a convex polygon using the vertices submitted by the AREA command, using the active brush. The vertices are then discarded.

**CIRCLE [FILL]** <x>,<y>,<radius>

Draw a circle at <x>,<y> with the radius <radius>. Use FILL to fill the circle using the active brush.

**ELLIPSE [FILL] <x>,<y>,<xRadius>,<yRadius>**

Draw an ellipse at <x>,<y> with the radius <xRadius> and <yRadius>. Use FILL to fill the ellipse using the active brush.

**ARC <x>,<y>,<xRadius>,<yRadius>,<startAngle>,<endAngle>[,<interval>]**

Draw an arc at <x>,<y> with the radius <xRadius> and <yRadius>, starting at <startAngle> and ending at <endAngle>. Angles are specified as degrees (0 to 360). The <interval> specifies the plotting increment (in degrees) between each point on the arc. The higher the value the less calculations need to be performed, but at the expense of reduced accuracy. A default value of 1 (calculate every degree between <startAngle> and <endAngle>) is used if <interval> is omitted.

**RADIAL LINE <x>,<y>,<xRadius>,<yRadius>,<angle>**

Draw a radial line starting at <x>,<y> to the point on an ellipse with the radius <xRadius> and <yRadius> at <angle>. Angles are specified in degrees (0 to 360).

**SLICE [FILL] <x>,<y>,<xRadius>,<yRadius>,<startAngle>,<endAngle>**

Draw a slice of an ellipse at <x>,<y> with the radius <xRadius> and <yRadius>, starting at <startAngle> and ending at <endAngle>. Angles are specified in degrees (0 to 360). Use FILL to fill the slice using the active brush.

**BEZIER CURVE <x1>,<y1>,<x2>,<y2>,<x3>,<y3>,<x4>,<y4>**

Draw a Bezier curve within (x1,y1) and (x2,y2) and (x3,y3) and (x4, y4).

**FLOOD FILL <x>,<y>**

Flood fill the enclosed area at <x>,<y> using the active brush.

**TEXT <x>,<y>,<value> [ALIGN <alignX>,<alignY>]**

Print the text <value> at <x>,<y> using the current Font. The optional ALIGN modifier is used to align the text around <x>,<y>:

<alignX>:	-1 (left)	0 (center)	1 (right)
<alignY>:	-1 (top)	0 (center)	1 (bottom)

The default alignment is to the upper left corner of the text (-1, -1).

**PASTE <x>,<y>,<data\$> [ALIGN <alignX>,<alignY>]**

Paste the copied <data\$> (see the COPY\$ function) to <x>,<y>. The optional ALIGN modifier is used to align the pasted data around <x>,<y>.

**BLIT BITMAP <id> TO <x>,<y> [ALIGN <alignX>,<alignY>]**

Blit (copy) the Bitmap <id> to <x>,<y>. The optional ALIGN modifier is used to align the Bitmap around <x>,<y>.

**BLIT BITMAP <id>,<x1>,<y1>,<x2>,<y2> TO <x>,<y> [ALIGN <alignX>,<alignY>]**

Blit the rectangular Bitmap data from <x1>,<y1> to <x2>,<y2> in Bitmap <id> to <x>,<y>. The optional ALIGN modifier is used to align the Bitmap around <x>,<y>.

**BLIT BITMAP <id>,<name\$> TO <x>,<y> [ALIGN <alignX>,<alignY>]**

Blit a Bitmap's subtexture named <name\$> in the Sprite Sheet Bitmap <id> to <x>,<y>. The optional ALIGN modifier is used to align the Bitmap around <x>,<y>.

#### **BLIT TILE <tile> TO <x>,<y> [ALIGN <alignX>,<alignY>]**

Blit the tile <tile> to <x>,<y>. The optional ALIGN modifier is used to align the tile around <x>,<y>.

#### **BLIT TILES <text\$> TO <x>,<y> [ALIGN <alignX>,<alignY>]**

Blit the tiles mapped to the ASCII characters in <text\$> to <x>,<y>. The optional ALIGN modifier is used to align the tile around <x>,<y>.

#### **BLIT TILEMAP <id> TO <x>,<y> [ALIGN <alignX>,<alignY>]**

Blit the Tilemap <id> to <x>,<y>. The optional ALIGN modifier is used to align the tile around <x>,<y>.

#### **SHIFT BITMAP UP|DOWN|LEFT|RIGHT [<count>][,...]**

Smooth scroll the Bitmap at the pixel level in the specified direction <count> pixels, or one pixel if <count> is omitted. Subtextures cannot be shifted. Use the BITMAP.SCX and BITMAP.SCY functions to read the shift X and shift Y registers back.

#### **SHIFT BITMAP <dx>,<dy>**

Smooth scroll the Bitmap at the pixel level to the specified offset <dx>,<dy>. Subtextures cannot be shifted.

#### **ROLL BITMAP UP|DOWN|LEFT|RIGHT [<count>] [WINDOW <x1>,<y1>,<x2>,<y2>]**

Roll the Bitmap in the direction specified by <count> pixels, or one pixel if <count> is omitted. When rolling, the pixels that leave one side of the Bitmap appear on the other side. Use the optional WINDOW modifier to roll a section of the Bitmap and not the entire Bitmap.

#### **SCROLL BITMAP UP|DOWN|LEFT|RIGHT [<count>] [WINDOW <x1>,<y1>,<x2>,<y2>]**

Scroll the screen in the direction specified by <count> pixels, or one pixel if <count> is omitted. When scrolling, the pixels that leave one side of the Bitmap disappear and blank space appears on the other side. Use the optional WINDOW modifier to scroll a section of the Bitmap and not the entire Bitmap.

#### **SMOOTH BITMAP <state>**

Enable linear filtering if <state> is non-zero, otherwise disable linear filtering. Linear filtering is a method to interpolate texels (texture elements) when rendering Bitmaps. It helps smooth out the appearance of a Bitmap, especially when displayed at a different size or angle than its original resolution.

### **Tile Commands**

---

#### **CLEAR PALETTE**

Clear the color palette used when defining custom characters.

#### **PALETTE <ch>,<color>**

Assign a color to a character. This character then acts as a substitute for the color value when defining custom characters using the CUSTOM TILE command. <ch> is the ASCII code of the character (if a Real value) or a one-character string (if a String value) to assign <color> to.

#### **CLEAR TILE <tile>[,<color>]**



Clear the tile <tile> to <color>. If <color> is omitted, the tile is cleared to 0 (transparent).

#### **COPY TILE <tile>,<source>**

Copy the tile <source> to the tile <tile>.

#### **CUSTOM TILE <tile>,<scanline>,<data\$>**

Set the <scanline> in tile <tile> to <data\$>. <data\$> should be the same length as the width of the tile, and use characters defined in PALETTE to substitute color pixels.

#### **ROLL TILE <tile> UP|DOWN|LEFT|RIGHT [<count>]**

Roll the tile <tile> in the direction specified by <count> pixels, or one pixel if <count> is omitted. When rolling, the pixels that leave one side of the tile appear on the other side.

#### **SCROLL TILE <tile> UP|DOWN|LEFT|RIGHT [<count>]**

Scroll the tile <tile> in the direction specified by <count> pixels, or one pixel if <count> is omitted. When scrolling, the pixels that leave one side of the tile disappear and blank space appears on the other side.

### **Tilemap Commands**

---

#### **CLEAR TILEMAP**

Clear the current targeted Tilemap.

#### **SWAP TILES <oldTile>,<newTile> [WINDOW <x1>,<y1>,<x2>,<y2>]**

Swap <oldTile> with <newTile>. Use the optional WINDOW modifier to swap a section of the Tilemap and not the entire Tilemap.

#### **PUT TILE <x>,<y>,<tile>**

Draw the tile <tile> at <x>,<y> using the current ink and highlight color and attributes.

#### **PUT TILES <x>,<y>,<text\$>**

Map the ASCII characters in <text\$> to their corresponding tiles and draw at <x>,<y> using the current ink and highlight color and attributes.

#### **PUT HLINE <x1>,<x2>,<y>,<tile>**

Draw a horizontal line from <x1>,<y> to <x2>,<y> with <tile> using the current ink and highlight color and attributes.

#### **PUT VLINE <x>,<y1>,<y2>,<tile>**

Draw a vertical line from <x>,<y1> to <x>,<y2> with <tile> using the current ink and highlight color and attributes.

#### **PUT RECT [FILL] <x1>,<y1>,<x2>,<y2>,<tile>**

Draw a rectangle from <x1>,<y1> to <x2>,<y2> with <tile> using the current ink and highlight color and attributes. Use FILL to fill the rectangle.

#### **SHIFT TILEMAP UP|DOWN|LEFT|RIGHT [<count>][,...]**

Smooth scroll the Tilemap at the pixel level in the specified direction <count> pixels. Use the TILEMAP.SCX and TILEMAP.SCY functions to read the shift X and shift Y registers back.

#### **SHIFT TILEMAP <dx>,<dy>**

Smooth scroll the Tilemap at the pixel level to the specified offset <dx>,<dy>.

#### **ROLL TILEMAP UP|DOWN|LEFT|RIGHT [<count>] [WINDOW <x1>,<y1>,<x2>,<y2>]**

Roll the Tilemap in the direction specified by <count> tiles, or one tile if <count> is omitted. When rolling, the tiles that leave one side of the Tilemap appear on the other side. Use the optional WINDOW modifier to roll a section of the Tilemap and not the entire Tilemap.

#### **SCROLL TILEMAP UP|DOWN|LEFT|RIGHT [<count>] [WINDOW <x1>,<y1>,<x2>,<y2>]**

Scroll the Tilemap in the direction specified by <count> tiles, or one tile if <count> is omitted. When scrolling, the tiles that leave one side of the Tilemap disappear and blank space appears on the other side. Use the optional WINDOW modifier to scroll a section of the Tilemap and not the entire Tilemap.

### **More Tilemap Commands**

---

#### **TBACKGROUND <color>**

Set the background color of the targeted Tilemap, where <color> is a 32-bit unsigned integer.

#### **TMIRROR <state>**

Mirror tiles horizontally (<state> 1) or not (<state> 0) when writing new tiles to memory.

#### **TFLIP <state>**

Flip tiles vertically (<state> 1) or not (<state> 0) when writing new tiles to memory.

#### **TROTATE <state>**

Rotate tiles (<state> 1) or not (<state> 0) when writing new tiles to memory. Rotating a tile swaps the X/Y axis, flipping the bottom left and top right corners of the tile. Only square tiles can be rotated. This attribute is ignored if the tiles are not square.

#### **TCLS [<tile>]**

Clear the targeted Tilemap.

#### **TLOCATE [<x>][,<y>]**

Move the Tilemap's cursor to the <x>, <y> position. Both <x> and <y> start at 1 and are optional.

#### **TINPUT ["prompt\$",<default\$>];<var\$>**

Input a line and assign it to <var\$>. <var\$> must be a String variable. No parsing is done on the input; the entire line is assigned to <var\$>. <prompt\$> is the optional prompt to print before input. <default\$> is the optional default input for the input field. TINPUT is like LINE INPUT.

This command cannot be used in direct mode.

#### **TPRINT <value>[<separator>]...**

Print text. <value> can be a Real or a String. The tiles mapped to the ASCII characters in the Real or String are printed. When printing a Real, the following rules apply:

- In Modern mode, the Real is printed as is, with no leading or trailing spaces.
- In Legacy mode, a preceding space (if the Real is zero or positive) or minus sign (if the Real is negative) is printed before the number, and a space is printed after the number. Also, the leading 0 is not printed if the Real is between -1 and 1.

Multiple values can be printed, separated by a valid <separator>. Valid separators include:

- ;        do not print a carriage return
- ,        print a tab
- |        print a carriage return

Printing a tab advances the cursor to the next tab zone (by default, tab zones are 15 characters wide in Legacy mode, or 16 characters wide in Modern mode). If <separator> is omitted at the end of the PRINT statement a carriage return is printed.

### **TPRINT SPC(<x>)**

Print <x> number of spaces before printing.

### **TPRINT TAB(<tab>)**

Move the cursor forward to <column> on the Tilemap, starting with the leftmost position of the current line. Columns start at 0. If <column> is greater than the Tilemap's width, TAB advances to the next line and the print position is set to (<column> MOD width). If the current print position is already beyond position <column>, the cursor is not moved.

### **TPRINT TAB(<tabX>,<tabY>)**

Move the cursor to the column <tabX> and the row <tabY> on the Tilemap. Columns and rows start at 0.

### **TPRINT LINE [LEFT|CENTER|RIGHT] [ON <row>,<value\$>[:] [INK <ink>] [HIGHLIGHT <highlight>]**

Print the line <value\$>. If an optional alignment modifier (LEFT, CENTER, or RIGHT) is used print the <value\$> left justified, centered, or right justified, respectively, otherwise print at the current column. If the ON modifier is specified, print at line <row>, otherwise print on the current line. If the optional semicolon ';' is omitted, print a carriage return after printing <value\$>. If the INK modifier is specified, set the entire line's ink color. If the HIGHLIGHT modifier is specified, set the entire line's highlight color. This command can be useful for printing headers and footers.

### **TPRINT CODE <value>[:]**

Print <value> using syntax highlighting followed by a carriage return (if the optional ';' is omitted). Syntax highlighting is a feature that colors and styles Liquid BASIC code to improve readability by visually distinguishing different elements like keywords, comments, and strings. It's a way to make code more understandable and easier to read. It does not affect how the code runs.

### **TPOKE <index>,<tile>,<ink>,<highlight>,<attributes>]**

Poke (write directly to memory) the tile <tile>, the foreground color <ink>, the background color <highlight>, and the attributes <attributes>. The <tile>, <ink>, <highlight>, and <attributes> parameters are all optional. Use the TPEEK function to read the data back.

### **TPASTE <x>,<y>,<data\$> [MASK] [ALIGN <alignX>,<alignY>]**

Paste the copied <data\$> (see the TCOPY\$ function) to <x>,<y>. If the optional MASK modifier is used, cells that have an ink color of 0 (transparent) will be ignored when pasting. If omitted, all cells will be pasted, regardless of their ink color. The optional ALIGN modifier is used to align the pasted data around <x>,<y>.

---

## The Cel Engine

---

The Cel engine uses 3x3 matrices and matrix multiplication to apply affine transformations (such as translating, scaling, rotating, and shearing) to a Bitmap before rendering. This is useful to create “software sprites”, pseudo 3D graphics, and other bitmapped effects. The Cel engine is software-based but optimized and is much faster than running equivalent routines written in Liquid BASIC. It is not as fast however as hardware-based solutions, like Sprites.

The current ink color, clipping region, pixel operator, and pixel mask are all applied when the Cel engine writes pixels.

---

### Cel Commands

---

#### CEL RESET

Reset the Cel engine’s 3x3 matrices and clear its cache.

#### CEL TRANSLATE <x>,<y>

Translate (move) the current matrix <x>,<y> units.

#### CEL SCALE <sx>,<sy>

Scale the current matrix by <sx>,<sy> units.

#### CEL ROTATE <degrees>

Rotate the current matrix by <degrees> degrees.

#### CEL SHEAR <sx>,<sy>

Shear the current matrix by <sx>,<sy>.

#### CEL BLIT BITMAP <id>[,<x>,<y>]

Blit the Bitmap <id> using the current matrix for affine transformations. The Bitmap can be offset using the optional <x> and <y> values.

---

### View Frustum Commands

---

#### VIEW FRUSTUM BITMAP <id>,<worldX>,<worldY>,<worldA>,<near>,<far>,<fovHalf> [WINDOW <x1>,<y1>,<x2>,<y2>]

Project a view frustum onto Bitmap <id> and render to the current targeted Bitmap. This is useful for SNES Mode 7-style effects. <worldX>, <worldY>, and <worldA> are the coordinates of the camera. <near> is how near the camera the frustum should start, and <far> is how far from the camera the frustum should end. <fovHalf> is the field of view, divided in half (PI ‘ $\pi$ ’ divided by 4 is a good value to use). Use the optional WINDOW modifier to render to a section of the Bitmap and not the entire Bitmap.

A view frustum is a 3D geometric shape that represents the visible area of a scene that is rendered. It is defined by a near plane, a far plane, and four side planes that form a pyramid-like shape. The view frustum helps determine which objects in a scene are visible to the viewer and should be rendered.

---

## Filters

---

Filters are applied to the entire Bitmap and are used to create special effects and otherwise manipulate the Bitmap. Filters are software-based but optimized and are much faster than running equivalent routines written in Liquid BASIC.

### Filter Commands

---

#### **FILTER BITMAP <command\$>[,<arguments\$>]**

Run the <command\$> filter. The <command\$> is automatically converted to lowercase letters and all leading and trailing spaces removed. Optional <arguments\$> can be passed in a comma-separated list. Use the EXPORT\$ function to help with building a comma-separated list from one or more expressions.

### Original Image

---



### Filters and Arguments

---

#### **adjusthsv <h>,<s>,<v>**

Adjust the Bitmap's hue <h>, saturation <s>, and brightness <v>.



**adjustrgb <r>,<g>,<b>**

Adjust the Bitmap's red <r>, green <g>, and blue <b>.



**blur**

Blur the Bitmap (un-sharpen).



### **dilate**

Dilate the Bitmap (bright areas grow bigger).



### **edgehorizontal**

Find the edges of the Bitmap horizontally.



### **edgevertical**

Find the edges of the Bitmap vertically.



**emboss**

Emboss the Bitmap (faked 3D look).



**equalizefull**

Equalize the Bitmap (adjust histogram fully).





### **equalizestretch**

Equalize the Bitmap (adjust histogram to expose more details in the image).



### **erode**

Erode the Bitmap (dark areas grow bigger).



### **grayscale**

Gray scale the Bitmap.



**invert**

Invert the Bitmap.



**kaleidoscope <corner>**

Kaleidoscope the Bitmap (mirror and resize the bitmap 2x2).



## logpolar

Perform a log polar effect on the Bitmap (transform the picture like the human brain does).



## maketileable <strength>

Make the Bitmap tileable (prepare borders to make tile distort look better).



## mediancut

Median cut the Bitmap (similar to blur).



**motionblur <strength>**

Motion blur the Bitmap.



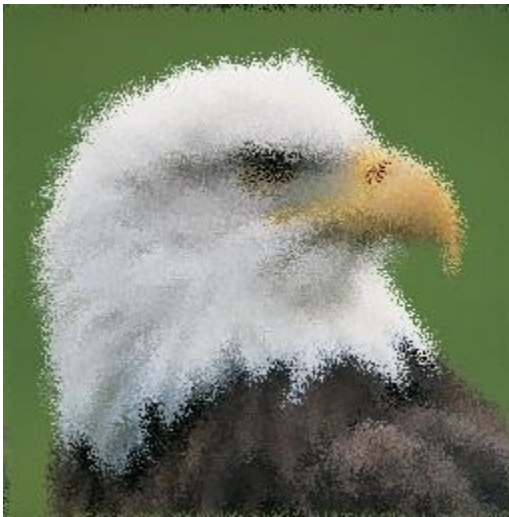
**move <dx>,<dy>**

Move distort the Bitmap to <dx>,<dy>.



**noise <seed>,<radius>**

Add noise to the Bitmap (noisy and natural look).



**pixels <color>,<count>**

Plot a number of random pixels in the Bitmap.



**scalehsv <h>,<s>,<v>**

Scale the Bitmap's hue <h>, saturation <s>, and brightness <v>.





**scalergb <r>,<g>,<b>**

Scale the Bitmap's red <r>, green <g>, and blue <b>.



**sculpture**

Sculpture the Bitmap (similar to a 3D chrome effect).



## **sepia**

Sepia the Bitmap (old-timey 1800's effect).



## **sharpen**

Sharpen the Bitmap (expose more details in the image).



## **sine <dx>,<depthx>,<dy>,<depthy>**

Sine distort the Bitmap.



**sinergb <r>,<g>,<b>**

Sine the Bitmap's red <r>, green <g>, and blue <b> colors.



**tile**

Tile the Bitmap (copy and resize the bitmap 2x2).





**tunnel <zoom>**

Tunnel the Bitmap.



**twirl <rot>,<scale>**

Twirl the Bitmap.



**wood**

Create woody/psychedelic colors in the Bitmap.

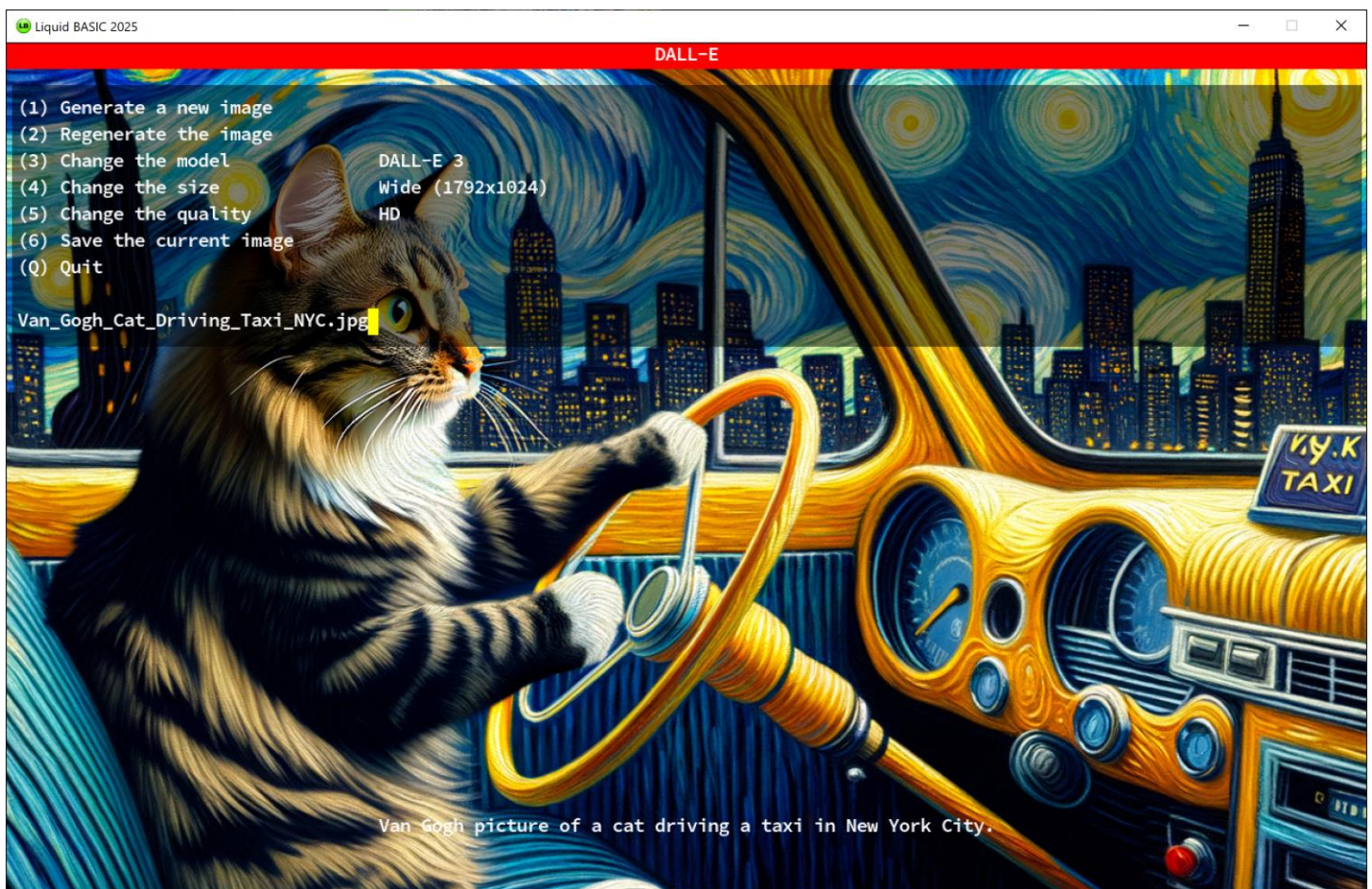


---

## Image Generation

---

Liquid BASIC uses DALL-E 3 to handle image generation. Using the Image Generator, complex and fascinating images can be generated from a single sentence, such as “Van Gogh picture of a cat driving a taxi in New York City”:



## Image Generation Commands

---

### DRAW <prompt\$>

Submit <prompt\$> to DALL-E to generate an image and fit it to the screen. The image is generated using the DALL-E 3 model in the “Wide” format (1792x1024 pixels) and “HD” quality for best results.

### DRAW BITMAP <id>,<prompt\$>,<size>,<quality\$>

Submit <prompt\$> to DALL-E to generate an image that is saved to Bitmap <id>. <size> is the size of the image and must be one of the constant values below:

	CONSTANT	SIZE
1	IG3_SQUARE	1024x1024 pixels
2	IG3_WIDE	1792x1024 pixels
3	IG3_TALL	1024x1792 pixels

<quality\$> is the quality of the generated image and can be either “Standard” or “HD”.

---

---

## Speech Synthesis

---

---

### Speech Synthesis Commands

---

#### VOICE SELECT <voice\$>

Select the voice <voice\$> to use when speaking. Use the VOICES\$ reserved variable to get a list of available voices.

#### VOICE SAY [ASYNC] <speech\$>

Say (speak) the text in <speech\$> using the current selected voice. Use the optional ASYNC modifier to continue program execution while the computer is talking.

#### VOICE VOLUME <volume>

Set the volume of the Voice. Valid <volume> ranges are between 0 (mute) and 100.

---

---

## Sprites

---

---

There are 256 independent Sprites, numbered from 1 to 256. Sprites are rendered from either a Tilemap or Bitmap resource. Each Sprite can be moved, scaled, rotated, and tinted.

In direct mode, holding down the CTRL key and clicking on a Sprite allows the Sprite to be dragged and dropped, and double clicking the Sprite brings it to the front of the render queue. Holding the CTRL key while pointing to a sprite and using the mouse wheel scales the Sprite up or down.

## Sprite Modifiers

---

### **SPRITE <id> BIND BITMAP <bitmapId>**

Bind the Bitmap <bitmapId> to the Sprite for rendering. Also reset the Sprite's fragment shader back to the default (texture mapping) shader. Only one Bitmap or Tilemap can be bound to a Sprite at a time.

### **SPRITE <id> BIND TILEMAP <tilemapId>**

Bind the Tilemap <tilemapId> to the Sprite for rendering. Also reset the Sprite's fragment shader back to the default (texture mapping) shader. Only one Bitmap or Tilemap can be bound to a Sprite at a time.

### **SPRITE <id> SET SHADER <shaderId>| OFF**

Use the Shader <shaderId> when rendering the Sprite, or OFF to use the default shader. Only one Shader can be attached to a Sprite at a time. A Sprite must have a Bitmap or Tilemap bound first before a Shader can be set.

### **SPRITE <id> SELECT <name\$>**

Use the Sprite Sheet's subtexture <name\$> to render the Sprite.

### **SPRITE <id> SELECT ALL**

Use the entire Bitmap to render the Sprite.

### **SPRITE <id> SHOW**

Turn the Sprite on (make visible).

### **SPRITE <id> HIDE**

Turn the Sprite off (make invisible). By default, the Sprite is turned off.

### **SPRITE <id> PRIORITY <priority>**

Assign the Sprite's priority to <priority>. <priority> must be between -32768 and 32767. The Sprite's priority determines the order the Sprites are drawn. Sprites with a higher priority appear on top of Sprites with a lower priority. Sprites with a positive priority appear above the Screen, while Sprites with a negative priority appear under the Screen. Sprites are always under the Text Screen.

### **SPRITE <id> TAG <tags\$>**

Tag the Sprite with the tags listed in the comma-separated list <tags\$>. Tags are useful for detecting collisions among a group of Sprites (e.g. "aliens" or "bullets") rather than detecting a collision against each individual Sprite. Use the collision functions (INSIDE, INTERSECT, TOUCH, and OVERLAP) to determine if a Sprite has collided with any other Sprite that has a matching tag.

### **SPRITE <id> CENTER**

Center the Sprite in the Liquid BASIC window. The aspect ratio is preserved, and the Sprite is not scaled or stretched.

### **SPRITE <id> STRETCH**

Stretch the Sprite to the Liquid BASIC window. The aspect ratio is ignored.

### **SPRITE <id> FILL**

Fill the Liquid BASIC window with the Sprite while maintaining the aspect ratio. This may cause the Sprite to be larger than the visible area of the window.

## **SPRITE <id> FIT**

Fit the Sprite to the Liquid BASIC window while maintaining the aspect ratio. This may cause the Sprite to not fill the entire visible area of the window.

## **SPRITE <id> TILE**

Tile the Sprite. The aspect ratio is preserved, and the Sprite is not scaled or stretched. It simply repeats to fill the entire Liquid BASIC window.

## **SPRITE <id> ANCHOR <anchorX>,<anchorY>**

Assign the Sprite's anchor point to <anchorX>,<anchorY>. The anchor points must be between 0 and 1. The anchor point determines the "center" of the Sprite. By default, a Sprite's anchor point is the center (0.5, 0.5) of the Tilemap or Bitmap the Sprite is using. When moving a Sprite to a coordinate this will center the Sprite on that coordinate. It also allows Sprites to be properly rotated around their center. Anchor points can be set to the upper left of the Sprite (0, 0) or the lower right (1, 1) or any point in-between.

## **SPRITE <id> XSTEP <xstep>,<ystep>**

Set the Sprite's XSTEP to <xstep>,<ystep>. The XSTEP is applied whenever a Sprite is moved. Sprites operate in 1280x800 mode. XSTEP is useful to fit a Sprite to a certain resolution (e.g. XSTEP 4,4 to fit a 320x200 Screen) and be automatically moved within that resolution.

## **SPRITE <id> XYSIZE <xsize>,<ysize>**

Set the Sprite's XYSIZE to <xsize>,<ysize>. The XYSIZE is applied whenever a Sprite is scaled. Sprites operate in 1280x800 mode. XYSIZE is useful to fit a Sprite to a certain resolution (e.g. XYSIZE 4,4 to fit a 320x200 Screen) and be automatically scaled within that resolution.

## **SPRITE <id> MOVE [RELATIVE] <x>,<y>**

Move a Sprite to <x>,<y>. The optional RELATIVE modifier moves the Sprite relative to its current position. The coordinates are multiplied by XSTEP and YSTEP.

## **SPRITE <id> MOVE [RELATIVE] TO <x>,<y>,<ms>**

Move a Sprite to <x>,<y> over time. The optional RELATIVE modifier moves the Sprite relative to its current position. The coordinates are multiplied by XSTEP and YSTEP. <ms> is the number of milliseconds to move the Sprite. Multiple MOVE TO commands can be queued, so when one finishes the next one begins.

## **SPRITE <id> SCALE <sx>,<sy>**

Scale the Sprite horizontally/vertically by <sx>,<sy>. The scale is multiplied by XSIZE and YSIZE.

## **SPRITE <id> SCALE TO <sx>,<sy>,<ms>**

Scale the Sprite horizontally/vertically by <sx>,<sy> over time. The scale is multiplied by XSIZE and YSIZE. <ms> is the number of milliseconds to scale the Sprite. Multiple SCALE commands can be queued, so when one finishes the next one begins.

## **SPRITE <id> ROTATE <degrees>**

Rotate the Sprite by <degrees> degrees.

## **SPRITE <id> ROTATE TO <degrees>,<ms>**

Rotate the Sprite by <degrees> degrees over time. <ms> is the number of milliseconds to rotate the Sprite. Multiple ROTATE commands can be queued, so when one finishes the next one begins.

**SPRITE <id> TINT <color>**

Tint the Sprite to <color>.

**SPRITE <id> TINT TO <color>,<ms>**

Tint the Sprite to <color> over time. <ms> is the number of milliseconds to tint the Sprite. Multiple TINT commands can be queued, so when one finishes the next one begins.

**SPRITE <id> MIRROR <state>**

Mirror the Sprite horizontally (<state> 1) or not (<state> 0).

**SPRITE <id> FLIP <state>**

Flip the Sprite vertically (<state> 1) or not (<state> 0).

**SPRITE <id> BORDER <color>**

Draw a border around the Sprite using <color>. If <color> is 0 (default) then no border is drawn. The <color> is not affected by the Sprite's tint. This command can be useful for debugging.

**SPRITE <id> CIRCLE [<radius>][,<color>]**

Set the Sprite's collision circle radius to the optional <radius>. The collision circle is the area of the Sprite that will trigger a collision when it "hits" another Sprite's collision circle. The default collision circle radius is either the width of the sprite divided in half, or the height of the sprite divided in half, whichever is greater. The optional <color> draws the collision circle around the Sprite when it is rendered. If <color> is 0 (default) then no collision circle is drawn. The <color> is not affected by the Sprite's tint. This command can be useful for debugging.

**SPRITE <id> FREE**

Free a Sprite from memory.

---

---

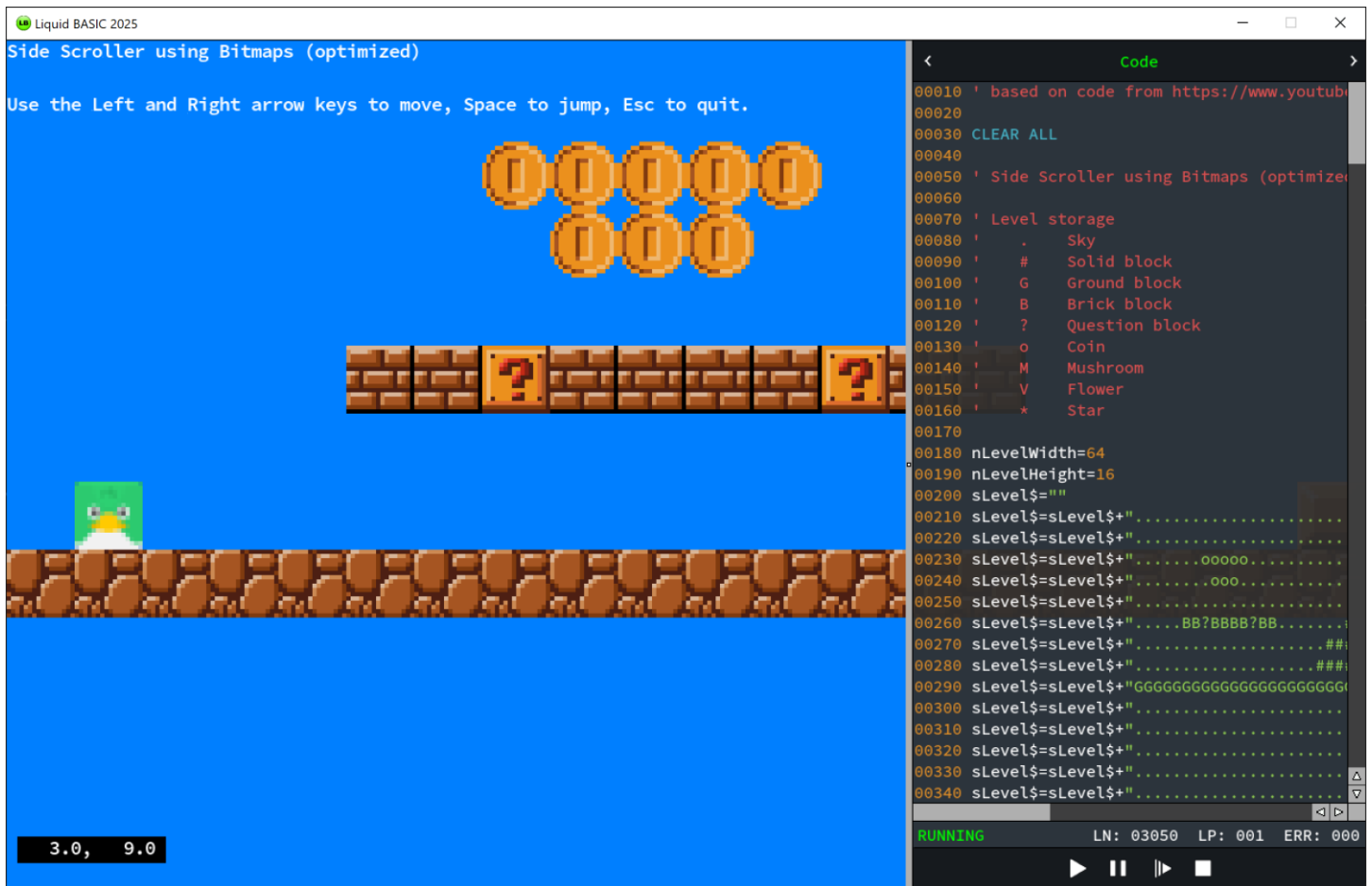
## The Debugger

---

---

The built-in Debugger can be used to single step through code as it runs and to examine variables and resources in real-time. Press CTRL + F1 to toggle the Debugger on or off or use the CTRL + D keys together while a program is running to pause the program and open the Debugger. The DEBUG command (or pressing F6 in the editor) can also be used to run a program and open the Debugger.





On the left of the Debugger is a vertical divider. Dragging the vertical divider will resize the Debugger. Double click the vertical divider to collapse the Debugger off-screen. Double click it again to bring the Debugger back.

There are five different tabs in the Debugger that can be cycled through by clicking the left and right arrows at the top of the Debugger:

- Code Shows the program being debugged, with the current line highlighted in yellow
- Variables Shows all variables and arrays currently allocated
- Resources Shows all resources currently allocated
- Render Stack Shows the Background, Screen, Sprites, and Text Screen
- Sprites Shows individual Sprites

When viewing Code, double click on a line to toggle a breakpoint at that location.

When viewing Resources, double clicking on a Tilemap will toggle between the Tilemap and the Tilesheet that comprises it. Double clicking on a Sound will play it; double clicking it again will stop playing it.

Near the bottom of the Debugger is the current state of the program (Running, Paused, Single Stepping, or Stopped), the current line number, the current line position, and the current error code.

There are also four buttons at the bottom of the Debugger:



The buttons run the program, pause the program, single step through the program, and stop the program.

---

## The Monitor

---

Liquid BASIC uses a two-pass Just-In-Time (JIT) compiler named **Blitz!** to convert BASIC commands and programs to a custom pseudo-code (pCode). This pCode in turn is interpreted by a stack-based Virtual Machine. This approach offers significant speed increases and better multitasking than using a simple BASIC interpreter.

The built-in Virtual Machine monitor can be used to examine how an expression or program is scanned into tokens and how these tokens are then converted into pCode.

There are 100 different pCode instructions to implement the BASIC language, and over 310 Kernal routines to implement input/output, graphics, and sound commands. (Kernal is deliberately misspelled as an homage to the Commodore line of computers, which also misspelled kernel.)

When the monitor is active, the up and down cursor keys are disabled, and the left and right cursor keys are restricted to the text entered on the current line. The monitor uses one letter commands to list, scan, and disassemble expressions and programs:

**?**

Print a quick reference to common commands in the Monitor.

**! <expr>**

Disassemble and evaluate an expression. This is useful to see how Liquid BASIC converts an expression into pCode.

**L [<first>][-<last>]**

List the BASIC program currently in memory. The optional <first> value specifies the first line to list, and the optional <last> value specifies the last line to list.

**S [<first>][-<last>]**

List the BASIC program's tokens. This is useful to see how Liquid BASIC converts a program into a token stream that can then be read and compiled.

**D [<first>][-<last>]**

Compile and disassemble the BASIC program in memory. This is useful to see how Liquid BASIC converts a program into pCode.

**X**

Execute the BASIC program in memory. Press the CTRL + C keys together to break execution and return to the Monitor.

**Q**

Quit the monitor.



---

---

# The Standard Library

---

---

The Standard Library is a collection of constants, reserved variables, and built-in functions that provide math, string, time, keyboard, mouse, and other useful routines.

Constants and reserved variables can be read but not assigned.

## Reserved Constants

---

### VERSION\$

Returns the version of Liquid BASIC in use.

### FALSE

Return 0.

### TRUE

Return -1.

### EMPTY\$

Return an empty ("" ) String.

### PI

Return 3.1415926535897931.

### FLICK

Return  $1.417233560090703^{-9}$ .

## Reserved Variables

---

### INT.MAX

Return the maximum value of a signed 32-bit integer (2,147,483,647).

### INT.MIN

Return the minimum value of a signed 32-bit integer (-2,147,483,648).

### TIMER

Return the number of seconds since midnight.

### TIME

Return the number of milliseconds since Liquid BASIC started.

#### **ELAPSED.TIME**

Return the number of milliseconds since ELAPSED.TIME was last read.

#### **TIME\$**

Return the current time in HH:MM:SS format.

#### **DATE\$**

Return the current date in MM/DD/YYYY format.

#### **GUID\$**

Return a unique GUID.

#### **ERR**

Return the last error code that is recorded.

#### **ERL**

Return the line number of the last error that is recorded.

#### **KBD**

Return the last key pressed. The keyboard buffer is not affected.

#### **INKEY\$**

Return the next key in the keyboard buffer as a one-character String value, or "" if the keyboard buffer is empty.

#### **FREEFILE**

Return the next valid unused file handle, or 0 if there are no free file handles available. This is useful when writing reusable code to only use free file handles that are not in use.

#### **CURDIR\$**

Return the current directory.

#### **ROOTDIR\$**

Return Liquid BASIC's root directory.

#### **FREE.DB**

Return the next valid unused database handle, or 0 if there are no free database handles available. This is useful when writing reusable code to only use free database handles that are not in use.

#### **KEY.SHIFT**

Return TRUE if the Shift key is pressed, otherwise return FALSE.

#### **KEY.CTRL**

Return TRUE if the Control key is pressed, otherwise return FALSE.

#### **KEY.ALT**

Return TRUE if the Alt key is pressed, otherwise return FALSE.

#### **MOUSE.X**

Return the current mouse X position.

#### **MOUSE.Y**

Return the current mouse Y position.

#### **MOUSE.WHEEL**

Return the next mouse wheel event in the mouse event buffer, or 0 if the mouse event buffer is empty.

#### **MOUSE.BUTTONS**

Return 1 if the left mouse button is down, 2 if the right mouse button is down, 3 if both buttons are down, or 0 if no mouse button is down.

#### **WINDOW.WIDTH**

Return the width of the Liquid BASIC window in pixels (1280).

#### **WINDOW.HEIGHT**

Return the height of the Liquid BASIC window in pixels (800).

#### **FPS**

Return the number of Frames Per Second being rendered.

#### **TEXT.COLUMNS**

Return the number of columns in the Text Screen.

#### **TEXT.ROWS**

Return the number of rows in the Text Screen.

#### **CURSOR.X**

Return the current cursor X position. Columns start at 0.

#### **CURSOR.Y**

Return the current cursor Y position. Rows start at 0.

#### **BORDER.COLOR**

Return the current border color.

#### **BACKGROUND.COLOR**

Return the current background color.

#### **INK.COLOR**

Return the current ink color.

#### **HIGHLIGHT.COLOR**

Return the current highlight color.

#### **RND.COLOR**

Return a random color.

#### **PLASMA.COLOR**

Return the constantly evolving plasma color.

#### **SCREEN.MODE**

Return 1 if the Screen is in Bitmap mode, or 2 if the Screen is in Tilemap mode. Otherwise, return 0.

#### **SCREEN.WIDTH**

Return the width of the Screen in pixels when the Screen is in Bitmap or Tilemap mode.

#### **SCREEN.HEIGHT**

Return the height of the Screen in pixels when the Screen is in Bitmap or Tilemap mode.

#### **SCREEN.COLUMNS**

Return the number of columns when the Screen is in Tilemap mode.

#### **SCREEN.ROWS**

Return the number of rows when the Screen is in Tilemap mode.

#### **SCREEN.SIZE**

Return the size of the Screen in pixels (width × height) when in Bitmap mode, or the number of cells in the Screen (columns x rows) when in Tilemap mode.

#### **SCREEN.SCX**

Return the current X shift of the Screen in pixels.

#### **SCREEN.SCY**

Return the current Y shift of the Screen in pixels.

## **PIXEL.OPERATOR**

Return the current pixel operator.

## **PIXEL.MASK**

Return the current pixel mask.

## **LINE.STIPPLE**

Return the current line stipple.

## **FREE.BITMAP**

Return the ID of the next available Bitmap instance, or 0 if none is available.

## **FREE.FONT**

Return the ID of the next available Font instance, or 0 if none is available.

## **FREE.TILESET**

Return the ID of the next available Tileset instance, or 0 if none is available.

## **FREE.TILEMAP**

Return the ID of the next available Tilemap instance, or 0 if none is available.

## **FREE.SHADER**

Return the ID of the next available Shader instance, or 0 if none is available.

## **FREE.SOUND**

Return the ID of the next available Sound instance, or 0 if none is available.

## **FREE.SPRITE**

Return the ID of the next available Sprite instance, or 0 if none is available.

## **VOICES\$**

Return a comma-separated list of the available voices for speech.

## **COPILOT**

Return TRUE if OpenAI is present. Otherwise return FALSE.

## **CHAT.ERROR\$**

Return the last error generated by a CHAT command. Otherwise return an empty string ("").

## **CHATBOT.ERROR\$**

Return the last error generated by a CHATBOT command. Otherwise return an empty string ("").

## **CODE.ERROR\$**

Return the last error generated by the CODE command. Otherwise return an empty string ("").

## **CODE.RULE\$**

Return the default rule set used for code generation.

## **Error Handling**

---

### **ERR\$(<code>)**

Convert an error code to a String.

## **Casting Functions**

---

### **CBYT(<value>)**

Return <value> cast as an 8-bit byte.

### **CINT(<value>)**

Return <value> cast as a 32-bit integer. The CINT function differs from the INT function in that CINT rounds the <value> to the nearest integer, whereas INT truncates the fractional portion of <value>.

### **CLNG(<value>)**

Return <value> cast as a 64-bit long integer.

### **CSNG(<value>)**

Return <value> cast as a 32-bit single precision floating point number.

### **CDBL(<value>)**

Return <value> cast as a 64-bit double precision floating point number. This function has no effect, as all Real calculations in Liquid BASIC are handled as 64-bit double precision floating point numbers.

## **Conversion Functions**

---

### **VAL(<d\$>)**

Convert a String to a Real and return it.

### **STR\$(<d>)**

Convert a Real to a String and return it. The following rules apply:

- In Modern mode, the Real is returned as is, with no leading or trailing spaces.
- In Legacy mode, the returned String is prefixed with a space (if the Real is zero or positive) or minus sign (if the Real is negative). Also, the leading 0 is not printed if the Real is between -1 and 1.

### **BIN\$(<d>)**

Convert a Real to an integer and return it as a String in base 2 (binary) format.

## **HEX\$(<d>)**

Convert a Real to an integer and return it as a String in base 16 (hexadecimal) format.

## **OCT\$(<d>)**

Convert a Real to an integer and return it as a String in base 8 (octal) format.

## **Bit Functions**

---

### **BIT.GET(<bits>,<index>)**

Return a one or zero, depending on whether the bit <index> is on or off in the integer <bits>. <index> must be between 0 and 31.

### **BIT.SET(<bits>,<index>)**

Return the integer <bits> with the bit <index> set.

### **BIT.RESET(<bits>,<index>)**

Return the integer <bits> with the bit <index> reset.

### **BIT.TOGGLE(<bits>,<index>)**

Return the integer <bits> with the bit <index> toggled.

### **BIT.CALC(<bits>,<index>,<value>)**

Return the integer <bits> with the bit set if <value> is not zero or reset if <value> is zero.

## **Byte and Word Functions**

---

### **LOBYTE(<value>)**

Return the low 8-bit byte of unsigned integer <value>.

### **HIBYTE(<value>)**

Return the high 8-bit byte of unsigned integer <value>.

### **LOWORD(<value>)**

Return the low 16-bit word of unsigned integer <value>.

### **HIWORD(<value>)**

Return the high 16-bit word of unsigned integer <value>.

## **Numeric Functions**

---

### **ABS(<value>)**

Return the absolute value of <value>.

### **ATN(<value>)**

Return the arctangent of <value>.

#### **CEIL(<value>)**

Round <value> up and return.

#### **CLAMP(<value>,<min>,<max>)**

Return <value> clamped to <min> and <max>.

#### **COS(<value>)**

Return the cosine of <value>.

#### **DEG(<value>)**

Return the radians <value> converted to degrees.

#### **EXP(<value>)**

Return the exponent of <value>.

#### **FLOOR(<value>)**

Round <value> down and return.

#### **INT(<value>)**

Return <value> cast as a 32-bit integer. The fractional portion of <value> is discarded.

#### **LERP(<value1>,<value2>,<amt>)**

Return the value between <value1> and <value2>, with <amt> specifying the linear interpolation (lerp) between the two values. <amt> is clamped to 0 and 1.

#### **LOG(<value>)**

Return the logarithm of <value>.

#### **MIN(<x1>,<x2>[,<x3>...])**

Return the minimum value out of the arguments passed.

#### **MAX(<x1>,<x2>[,<x3>...])**

Return the maximum value out of the arguments passed.

#### **NORM(<x>,<y>)**

Return the Euclidean norm (or magnitude) of a 2D vector:  $\text{SQR}(x * x + y * y)$

This is also known as the Pythagorean theorem when used to calculate the length of the hypotenuse of a right triangle. More formally, it's called the Euclidean distance from the origin to the point (x,y) in 2D space.

#### **RAD(<value>)**

Return the degrees <value> converted to radians.

#### **RANGE(<start>,<end>)**

Return a random integer number between <start> and <end>.

#### **RND([<limit>])**



Return a random number between 0 and 1 if <limit> is omitted. If <limit> is zero, then return the last random number generated by RND. If <limit> is less than 0, then always return the same number for <limit>. If <limit> is greater than 0, then return a random number between 0 and <limit>. The random number generated is based on the random number generator seed set with the RANDOMIZE statement.

### **ROUND(<value>[,<digits>])**

Round <value> to the specified number of <digits> and return. If <digits> is omitted, round to the nearest integer.

### **SGN(<value>)**

Return the sign of <value> (-1 if negative, 0 if zero, 1 if positive).

### **SIN(<value>)**

Return the sine of <value>.

### **SQR(<value>)**

Return the square root of <value>.

### **TAN(<value>)**

Return the tangent of <value>.

## **String Functions**

---

### **ASC(<string>[,<index>])**

Return the ASCII code of the character at the first position in string, or the character at the <index> position if specified. Character positions in Strings start at 1 in Liquid BASIC.

### **CHR\$(<ch>[,<ch>...])**

Return a string with the ASCII character <ch>. Multiple characters can be included.

### **CONCAT\$([<x1>[,<x2>]...])**

Take zero, one, or more arguments (Real or String) and concatenate them into a single String.

### **CSET\$(<text\$>,<width>)**

Return the string <text\$> centered and padded with leading and trailing spaces so the returned string is <width> characters long. If <text\$> is longer than <width>, it is centered and truncated to <width> characters.

### **EXPORT\$([<x1>[,<x2>]...])**

Take zero, one, or more arguments and combine them into a comma-separated list. Real values are converted to a String value, and String values are wrapped in quotes (to preserve commas). Complements the PARSE\$ and PARSECOUNT functions.

### **HASH(<text\$>)**

Return a 32-bit unsigned integer hashed from string <text\$>.

### **INSTR([<startindex>,<text\$>,<value\$>)**

Return the index where the string <value\$> is found in the string <text\$>, or 0 if <value\$> was not found. The optional <startIndex> is the index to start searching.

#### **LCASE\$(<text\$>)**

Return the string <text\$> converted to lowercase letters.

#### **LEFT\$(<text\$>,<length>)**

Return the left side of the string <text\$>, up to <length> characters.

#### **LEN(<text\$>)**

Return the length of the string <text\$>.

#### **LSET\$(<text\$>,<width>)**

Return the string <text\$> left justified and padded with trailing spaces so the returned string is <width> characters long. If <text\$> is longer than <width>, it is left justified and truncated to <width> characters.

#### **LTRIM\$(<text\$>[,<chars\$>])**

Return the string <text\$> with all leading <chars\$> removed (if <chars\$> is omitted then all leading spaces are removed).

#### **MID\$(<text\$>,<position>[,<length>])**

Return a substring from <text\$> starting at <position>. Up to <length> characters are returned if <length> is specified. Otherwise, the remainder of the string <text\$> is returned.

#### **PARSE\$(<text\$>,<index>)**

Return a single item from the comma-separated list in <text\$> at index <index>. Complements the EXPORT\$ function.

#### **PARSECOUNT(<text\$>)**

Return the number of items in the comma-separated list in <text\$>. Complements the EXPORT\$ function.

#### **REGEXPR([<startIndex>],<text\$>,<pattern\$>)**

Return the index where the regular expression <pattern\$> is found in the string <text\$>, or 0 if <pattern\$> was not found. The optional <startIndex> is the index to start searching.

#### **REGREPL\$(<text\$>,<pattern\$>,<replacement\$>)**

Return the string <text\$> but replace anywhere the regular expression <pattern\$> is found with <replacement\$>.

#### **RIGHT\$(<text\$>,<length>)**

Return the right side of the string <text\$>, up to <length> characters.

#### **RSET\$(<text\$>,<width>)**

Return the string <text\$> right justified and padded with leading spaces so the returned string is <width> characters long. If <text\$> is longer than <width>, it is right justified and truncated to <width> characters.

#### **RTRIM\$(<text\$>[,<chars\$>])**

Return the string <text\$> with all trailing <chars\$> removed (if <chars\$> is omitted then all trailing spaces are removed).

#### **SPACE\$(<count>)**

Return a string of spaces repeated <count> times.

#### **STRDELETE\$(<text\$>,<index>,<count>)**

Return the string <text\$> with <count> characters deleted starting at <index>.

#### **STRINSERT\$(<text\$>,<index>,<insert\$>)**

Return the string <text\$> with the string <insert\$> inserted at <index>.

#### **STRSUB\$(<text\$>,<index>,<char\$>)**

Return the string <text\$> with the character at <index> substituted with the character <char\$>.

#### **STRREPEAT\$(<text\$>,<count>)**

Return a string with <text\$> repeated <count> times.

#### **STRREPLACE\$(<text\$>,<value\$>,<replacement\$>)**

Return the string <text\$> but replace anywhere the string <value\$> is found with <replacement\$>.

#### **STRREVERSE\$(<text\$>)**

Return the string <text\$> reversed.

#### **TALLY(<text\$>,<value\$>)**

Return the number of occurrences of the string <value\$> in the string <text\$>.

#### **TRIM\$(<text\$>[,<char\$>])**

Return the string <text\$> with all leading and trailing <char\$> removed (if <char\$> is omitted then all leading and trailing spaces are removed).

#### **UCASE\$(<text\$>)**

Return the string <text\$> converted to uppercase letters.

#### **USING\$(<format\$>,<value>)**

Return <value> as a String formatted to the format list <format\$>. <value> can be a Real or a String.

### **FORMAT LIST**

For Real values, the pound sign '#' reserves room for a single character in the output field. If the formatted string contains more characters than there are pound signs in <format\$>, then the entire output field is filled with asterisks '\*'.  
For String values, the pound sign '#' reserves room for a single character in the output field. If the formatted string contains more characters than there are pound signs in <format\$>, then the entire output field is filled with asterisks '\*'.

- If <format\$> starts with a plus sign '+', then a plus sign will be printed at the beginning of the output field when the value is positive, and a minus sign will be printed at the beginning of the output field when the value is negative.
- If <format\$> starts with a minus sign '-', then a space will be printed at the beginning of the output field when the value is positive, and a minus sign will be printed at the beginning of the output field when the value is negative.

- If <format\$> does not start with either a plus sign or a minus sign, then a minus sign is printed before the first formatted character for negative values, taking up one character in the output field. Remember that if the number of characters (including the minus sign) exceeds the size of the output field the entire output field is replaced with asterisks.
- If <format\$> starts with a dollar sign '\$', then the output field will begin with a dollar sign and positive and negative values will be printed using the rule above.
- A period '.' in <format\$> is used to represent where the decimal point, if any, should appear. There cannot be more than one decimal point. If no decimal point is specified, the value is rounded to the nearest integer and printed without a decimal point.
- A comma in <format\$> is used to represent where commas, if any, should appear.

For String values, the pound sign '#' reserves room for a single character in the output field. If the formatted string contains more characters than there are pound signs in <format\$>, then the formatted string is left justified and truncated to fit the output field.

- If <format\$> starts with the less than sign '<', then the formatted output is left justified.
- If <format\$> starts with the equal sign '=', then the formatted output is centered within the output field.
- If <format\$> starts with the greater than sign '>', then the formatted output is right justified.

### **WORDWRAP\$(<text\$>[,<columns>])**

Return the string <text\$> word wrapped, where text is wrapped to the next line when it reaches the end of a line. <columns> specifies how many characters are on a single line. If the optional <columns> parameter is omitted, the current width of the Text Screen is used.

## **Memory Functions**

---

### **FORMAT.MEMORY\$(<bytes>)**

Return a string showing the number <bytes> formatted to the nearest KB, MB, or GB.

## **Compression Functions**

---

### **COMPRESS\$(<data\$>)**

Compress the data in <data\$> and return the compressed data as a string.

### **DECOMPRESS\$(<data\$>)**

Decompress the data in <data\$> and return the decompressed data as a string.

## **Text Screen Functions**

---

### **XPOS(<n>)**

Return the current cursor X position. <n> is ignored. Columns start at 1.

### **YPOS(<n>)**

Return the current cursor Y position. <n> is ignored. Rows start at 1.

### **TEXT.CELL(<x>,<y>)**

Return the character on the Text Screen at <x>,<y>.

## Keyboard Functions

---

### KEY.DOWN(<key>)

Return TRUE if the <key> is currently down. Otherwise return FALSE. Several constants are included to simplify the key codes:

	CONSTANT	KEY
1		(nothing)
2	VK_INSERT	Insert
3	VK_DELETE	Delete
4	VK_HOME	Home
5	VK_END	End
6	VK_PAGEUP	Page Up
7	VK_PAGEDOWN	Page Down
8	VK_BACKSPACE	Backspace
9	VK_TAB	Tab
10	VK_LINEFEED	Line Feed
11	VK_SHIFTTAB	SHIFT + Tab
12		(nothing)
13	VK_ENTER	Enter
14	VK_LEFT	Left
15	VK_RIGHT	Right
16	VK_UP	Up
17	VK_DOWN	Down
18	VK_F1	F1
19	VK_F2	F2
20	VK_F3	F3
21	VK_F4	F4
22	VK_F5	F5
23	VK_F6	F6
24	VK_F7	F7
25	VK_F8	F8
26	VK_F9	F9
27	VK_ESC	Esc
28	VK_F10	F10
29	VK_F11	F11
30	VK_F12	F12
31		(nothing)
32	VK_SPACE	Space

Values 33 to 126 for <key> map to their respective ASCII character.

### KEY.UP(<key>)

Return TRUE if the key <key> is currently up. Otherwise return FALSE.

### KEY.PRESSED(<key>)

Return TRUE if the key <key> was pressed. Otherwise return FALSE.

## GamePad Functions

---

### GAMEPAD.ISCONNECTED(<index>)

Return TRUE if a gamepad is connected. Otherwise return FALSE. <index> is the gamepad to read and must be between 0 and 3.

### GAMEPAD.DPAD(<index>)

Return a bit mask depending on which buttons on a gamepad's direction pad are currently pressed:

- 1      up
- 2      down
- 4      left
- 8      right

<index> is the gamepad to read and must be between 0 and 3.

### GAMEPAD.BUTTONS(<index>)

Return a bit mask depending on which buttons on a gamepad are currently pressed:

- 1      up
- 2      down
- 4      left
- 8      right

<index> is the gamepad to read and must be between 0 and 3.

### GAMEPAD.THUMBSTICK(<index>,<n>)

Return a Real value between -1 and 1, depending on the state of one of the thumb sticks. <n> is the thumb stick and axis to return:

- 1      left thumb stick X-axis
- 2      left thumb stick Y-axis
- 3      right thumb stick X-axis
- 4      right thumb stick Y-axis

<index> is the gamepad to read and must be between 0 and 3.

### GAMEPAD.TRIGGER(<index>,<n>)

Return a Real value between 0 and 1, depending on the state of a gamepad's trigger. <n> is the trigger to return:

- 1      left trigger
- 2      right trigger

<index> is the gamepad to read and must be between 0 and 3.

## Color Functions

---

### HSV(<h>,<s>,<v>[,<a>])

Return a color using the hue <h>, saturation <v>, brightness <v>, and optional <a> components given. If <a> is omitted the alpha value is assumed to be 255 (for a solid, opaque color). RGBA values are clamped between 0 and 255.

### **RGB(<r>,<g>,<b>[,<a>])**

Return a color using the red <r>, green <g>, blue <b>, and optional alpha <a> components given. If <a> is omitted the alpha value is assumed to be 255 (for a solid, opaque color). RGBA values are clamped between 0 and 255.

### **COLOR.RED(<color>)**

Return the red byte of the color <color>.

### **COLOR.GREEN(<color>)**

Return the green byte of the color <color>.

### **COLOR.BLUE(<color>)**

Return the blue byte of the color <color>.

### **COLOR.ALPHA(<color>)**

Return the alpha byte of the color <color>.

### **COLOR.PICKER(<color>)**

Open the host OS's Color Picker dialog and return the selected color (or <color> if no color was selected).

### **CLUT(<color\$>)**

Lookup a color from the default CLUT (Color LookUp Table). Available colors include BLACK, GRAY, SILVER, WHITE, RED, ORANGE, YELLOW, GREEN, CYAN, BLUE, PURPLE, BROWN, and PINK. <color\$> is the name of the color and is not case sensitive.

### **CLUT(<palette\$>,<index>)**

Lookup a color in one of the available CLUTs (Color LookUp Tables): C64 (32 colors), VGA (256 colors), or ANSI (256 colors). <palette\$> is the CLUT to use and is not case sensitive, and <index> is the index of the color to get (varies depending on the CLUT chosen).

### **DARKEN(<color>,<amt>)**

Return the color <color> darkened by <amt>. For example, if <amt> is 0.25, then the color returned will be the color darkened by 25%. <amt> is clamped to 0 and 16.

### **LIGHTEN(<color>,<amt>)**

Return the color <color> lightened by <amt>. For example, if <amt> is 0.25, then the color returned will be the color lightened by 25%. <amt> is clamped to 0 and 16.

### **GRADIENT(<color1>,<color2>,<amt>)**

Return the gradient color between <color1> and <color2>, with <amt> specifying the linear interpolation between the two colors. For example, if <amt> is 0.25, then the color returned will be a mix of 25% <color1> and 75% <color2>. <amt> is clamped to 0 and 1.

## Array Functions

---

### **LBOUND(REF <var>(),<dimension>)**

Return the lower bound of an array. The array must be passed to the function using REF and the name must be followed by an empty set of parentheses '()' to pass as an array and not a variable. The optional <dimension> must be between 1 and the number of dimensions in the array. If <dimension> is omitted, then 1 is assumed.

### **UBOUND(REF <var>(),<dimension>)**

Return the upper bound of an array. The array must be passed to the function using REF and the name must be followed by an empty set of parentheses '()' to pass as an array and not a variable. The optional <dimension> must be between 1 and the number of dimensions in the array. If <dimension> is omitted, then 1 is assumed.

### **ARRAY.MIN(REF <var>(),<first>,<last>)**

Return the minimum value in an array of Real variables. The optional <first> and <last> specify the first and last index of the elements to enumerate and can only be used on single dimensional arrays. If omitted, the entire array is enumerated.

### **ARRAY.MAX(REF <var>(),<first>,<last>)**

Return the maximum value in an array of Real variables. The optional <first> and <last> specify the first and last index of the elements to enumerate and can only be used on single dimensional arrays. If omitted, the entire array is enumerated.

### **ARRAY.SUM(REF <var>(),<first>,<last>)**

Return the sum of the elements in an array of Real variables. The optional <first> and <last> specify the first and last index of the elements to enumerate and can only be used on single dimensional arrays. If omitted, the entire array is enumerated.

### **ARRAY.PRODUCT(REF <var>(),<first>,<last>)**

Return the product of the elements in an array of Real variables. The optional <first> and <last> specify the first and last index of the elements to enumerate and can only be used on single dimensional arrays. If omitted, the entire array is enumerated.

### **ARRAY.AVERAGE(REF <var>(),<first>,<last>)**

Return the statistical mean (average) of the elements in an array of Real variables. The optional <first> and <last> specify the first and last index of the elements to enumerate and can only be used on single dimensional arrays. If omitted, the entire array is enumerated.

### **ARRAY.MEDIAN(REF <var>(),<first>,<last>)**

Return the statistical median of the elements in an array of Real variables. The optional <first> and <last> specify the first and last index of the elements to enumerate and can only be used on single dimensional arrays. If omitted, the entire array is enumerated.

### **ARRAY.VARIANCE(REF <var>(),<first>,<last>)**

Return the statistical variance of the elements in an array of Real variables. The optional <first> and <last> specify the first and last index of the elements to enumerate and can only be used on single dimensional arrays. If omitted, the entire array is enumerated.

### **ARRAY.SOFTMAX(REF <var>(),<temperature>)**



Perform a softmax function on an array. The softmax function is commonly used in machine learning and deep learning models to convert a vector of raw scores into a probability distribution. The output of the softmax function is a vector of probabilities representing the likelihood of each class or category. Each probability is in the range of [0..1], and the output vector sums up to 1, which is returned by the function.

To interpret the results of the softmax function, look at the probabilities assigned to each class or category. The class with the highest probability is considered the predicted class by the model. The probabilities of other classes show how confident the model is in its prediction.

The optional <temperature> controls the level of uncertainty in the output probabilities. The default temperature is 1.0 if omitted. A higher temperature (above 1.0) results in a more uniform distribution of probabilities, making the model more uncertain about its predictions. A lower temperature (below 1.0) sharpens the distribution, making the model more confident in its predictions. By adjusting the temperature, the softmax function can be used to control the trade-off between exploration and exploitation in reinforcement learning algorithms.

INPUT	TEMPERATURE	OUTPUT
[-1, -2.5, 3, 4.7]	1.0	[0.00282, 0.00063, 0.15393, 0.84262]
[-1, -2.5, 3, 4.7]	0.8	[0.00072, 0.00011, 0.10660, 0.89257]
[-1, -2.5, 3, 4.7]	1.2	[0.00690, 0.00198, 0.19345, 0.79767]

Note that you should ensure input values are within reasonable bounds for this function to work properly (due to potential underflow or overflow issues).

---

## Fiber Functions

### FIBER.ISDONE(<fiber>())

Return TRUE if <fiber> has finished. Otherwise return FALSE.

---

## File Functions

### DIR\$([<filter\$>[,<mode>]])

Return a comma-separated list depending on the <mode>:

- 1 Return a list of files in the current directory
- 2 Return a list of directories in the current directory
- 3 Return a list of both files and directories in the current directory (default)

The results are filtered to <filter\$>. Wildcard characters like the asterisk '\*' and question mark '?' can be used. The filter defaults to "\*. \*" if it is omitted.

### IS.FILE(<path\$>)

Return TRUE if <path\$> is a file. Otherwise return FALSE.

### IS.DIRECTORY(<path\$>)

Return TRUE if <path\$> is a directory. Otherwise return FALSE.

### EXISTS(<path\$>)

Return TRUE if the file or directory <path\$> exists. Otherwise return FALSE.

#### **EOF(<handle>)**

Return TRUE if at the end of the file assigned to <handle>. Otherwise return FALSE.

#### **LOC(<handle>)**

Return the position (in bytes) of the file assigned to <handle>.

#### **LOF(<handle>)**

Return the length (in bytes) of the file assigned to <handle>.

#### **FILE.NAME\$(<path\$>)**

Return the file name of <path\$>.

#### **FILE.EXTENSION\$(<path\$>)**

Return the file extension of <path\$>.

#### **FILE.PATH\$(<path\$>)**

Return the full path of <path\$>.

#### **FILE.DATE\$(<path\$>)**

Return the date that the file at <path\$> was last written to.

#### **FILE.SIZE(<path\$>)**

Return the size (in bytes) of the file at <path\$>.

#### **FILE.LINECOUNT(<path\$>)**

Return the number of lines in the text file at <path\$>.

#### **FILE.READTEXT\$(<path\$>)**

Return the contents of the entire text file at <path\$>.

#### **FILE.OPEN\$(<directory\$>,<extension\$>,<filter\$>)**

Open the host OS's File Open dialog and return the selected file (or "" if no file was selected). <directory\$> is the initial directory to start in (use "default" to restore to the last used directory). <extension\$> is the default file extension. <filter\$> is the filename filter string, which determines the choices that can be selected as a file type.

#### **FILE.SAVEAS\$(<directory\$>,<extension\$>,<filter\$>)**

Open the host OS's File Save As dialog and return the selected file (or "" if no file was selected). <directory\$> is the initial directory to start in (use "default" to restore to the last used directory). <extension\$> is the default file extension. <filter\$> is the filename filter string, which determines the choices that can be selected as a file type.

### **Database Functions**

---

#### **DB.ERROR\$(<handle>)**

Return the last error message on the database assigned to <handle> if there was an error, otherwise return an empty string ("").

**DB.ALLTABLES\$(<handle>)**

Return a comma-separated list of all tables in the database.

**DB.TABLECOLUMNS\$(<handle>,<table\$>)**

Return a comma-separated list of columns in <table\$>.

**DB.FIELD COUNT(<handle>)**

Return the number of fields (columns) in the current row read from the result set.

**DB.TABLE\$(<handle>,<index>)**

Return the table name of the column in the current row read from the result set.

**DB.TABLES\$(<handle>)**

Return a comma-separated list of the table names of the columns in the current row read from the result set.

**DB.COLUMN\$(<handle>,<index>)**

Return the column name in the current row read from the result set.

**DB.COLUMNS\$(<handle>)**

Return a comma-separated list of the column names in the current row read from the result set.

**DB.VALUE\$(<handle>,<index>)**

Return the value in column <index> in the current row read from the result set.

**DB.VALUES\$(<handle>)**

Return a comma-separated list of the values in the current row read from the result set.

**DB.DATATYPE\$(<handle>,<index>)**

Return the datatype of column <index> in the current row read from the result set.

**DB.DATATYPES\$(<handle>)**

Return a comma-separated list of the datatypes in the current row read from the result set.

**DB.STEPCOUNT(<handle>)**

Return the number of rows read so far in the result set. Use the DB READ NEXT command to read the next row; DB QUERY resets the counter.

**EOQ(<handle>)**

Return TRUE if the database assigned to <handle> has READ to the end of its result set. Otherwise return FALSE.

**Bitmap Functions**

---

**BITMAP.EXISTS(<id>)**

Return TRUE if the Bitmap <id> exists. Otherwise return FALSE.

**BITMAP.WIDTH(<id>)**

Return the width of Bitmap <id> in pixels.

#### **BITMAP.HEIGHT(<id>)**

Return the height of Bitmap <id> in pixels.

#### **BITMAP.SIZE(<id>)**

Return the size of Bitmap <id> in pixels (width × height).

#### **BITMAP.SCX(<id>)**

Return the current X shift of Bitmap <id> in pixels.

#### **BITMAP.SCY(<id>)**

Return the current Y shift of Bitmap <id> in pixels.

#### **BITMAP.SHEETNAMES\$(<id>)**

Return a comma-separated list of the subtexture names available in Bitmap <id>.

#### **PEEK(<index>)**

Peek (read directly from memory) the pixel at <index> and return. Complements the POKE command.

#### **POINT(<x>,<y>)**

Return the pixel color at <x>,<y>. The POINT function ignores the Bitmap's clipping region; it can be used to read pixels outside the clipping region. Complements the PLOT command.

#### **SAMPLE(<x>,<y>)**

Return the pixel color at <x>,<y>, where both <x> and <y> are normalized to  $0 \leq x \leq 1$ . For example, the coordinates 0.5, 0.5 would refer to the pixel at the center of the Bitmap.

#### **COPY\$(<x1>,<y1>,<x2>,<y2>)**

Copy the rectangular Bitmap data from <x1>,<y1> to <x2>,<y2> and return it as an encoded string. Complements the PASTE command.

#### **PIXEL.COLLISION(<src>,<dst>)**

Return if a collision occurred between the two colors <src> and <dst> when drawing to a Bitmap using the PXO\_COLLISION pixel operator.

### **Font Functions**

---

#### **FONT.EXISTS(<id>)**

Return TRUE if the Font <id> exists. Otherwise return FALSE.

#### **FONT.WIDTH(<id>,<text\$>)**

Return the width in pixels of the string <text\$> using Font <id>.

#### **FONT.HEIGHT(<id>)**

Return the height in pixels of Font <id>.

**FONT.COUNT(<id>)**

Return the number of characters in Font <id>.

**FONT.ISAVAILABLE(<id>,<ch>)**

Return TRUE if the character <ch> exists in Font <id>. Otherwise return FALSE.

**Tileset Functions**

---

**TILESET.EXISTS(<id>)**

Return TRUE if the Tileset <id> exists. Otherwise return FALSE.

**TILESET.WIDTH(<id>)**

Return the width in pixels of Tileset <id>.

**TILESET.HEIGHT(<id>)**

Return the height in pixels of Tileset <id>.

**TILESET.COUNT(<id>)**

Return the number of tiles in Tileset <id>.

**TILESET.ISAVAILABLE(<id>,<tile>)**

Return TRUE if the tile <tile> exists in Tileset <id>. Otherwise return FALSE.

**Tilemap Functions**

---

**TILEMAP.EXISTS(<id>)**

Return TRUE if the Tilemap <id> exists. Otherwise return FALSE.

**TILEMAP.WIDTH(<id>)**

Return the width of Tilemap <id> in pixels.

**TILEMAP.HEIGHT(<id>)**

Return the height of Tilemap <id> in pixels.

**TILEMAP.COLUMNS(<id>)**

Return the number of columns in Tilemap <id>.

**TILEMAP.ROWS(<id>)**

Return the number of rows in Tilemap <id>.

**TILEMAP.SIZE(<id>)**

Return the number of cells in Tilemap <id> (columns × rows).

**TILEMAP.SCX(<id>)**

Return the current X shift of Tilemap <id> in pixels.

### **TILEMAP.SCY(<id>)**

Return the current Y shift of Tilemap <id> in pixels.

### **GET.TILE(<x>,<y>)**

Return the tile at <x>,<y>. Complements the PUT TILE command.

### **TCHAR(<ch>)**

Return the tile mapped to the ASCII character <ch>. <ch> is the ASCII code of the character (if a Real value) or a one-character string (if a String value) to map.

### **TCHAR\$(<text\$>)**

Return the tiles mapped to the ASCII characters in <text\$>.

### **TPEEK(<index>,<mode>)**

Peek (read directly from memory) the data at <index> and return depending on <mode>:

- |   |                              |
|---|------------------------------|
| 0 | tile                         |
| 1 | ink (foreground color)       |
| 2 | highlight (background color) |
| 3 | attributes                   |

Complements the TPOKE command.

### **TCOPY\$(<x1>,<y1>,<x2>,<y2>)**

Copy the rectangular Tilemap data from <x1>,<y1> to <x2>,<y2> and return it as an encoded string.  
Complements the TPASTE command.

## **Shader Functions**

---

### **SHADER.EXISTS(<id>)**

Return TRUE if the Shader <id> exists. Otherwise return FALSE.

## **Sound Functions**

---

### **SOUND.EXISTS(<id>)**

Return TRUE if the Sound <id> exists. Otherwise return FALSE.

### **SOUND.ISPLAYING(<id>)**

Return TRUE if the Sound <id> is playing. Otherwise return FALSE.

### **SOUND.LENGTH(<id>)**

Return the length (in seconds) of the Sound <id>.

### **SOUND.POSITION(<id>)**

Return the current position (in seconds) of the Sound <id>.

## Sprite Functions

---

### **SPRITE.EXISTS(<id>)**

Return TRUE if the Sprite <id> exists. Otherwise return FALSE.

### **SPRITE.WIDTH(<id>)**

Return the width of Sprite <id> in pixels.

### **SPRITE.HEIGHT(<id>)**

Return the height of Sprite <id> in pixels.

### **SPRITE.ISVISIBLE(<id>)**

Return TRUE if the Sprite <id> is visible. Otherwise return FALSE.

### **SPRITE.PRIORITY(<id>)**

Return the priority of Sprite <id>.

### **SPRITE.TAGS\$(<id>)**

Return the tags (as a comma-separated list) of Sprite <id>.

### **SPRITE.X(<id>)**

Return the x-position of Sprite <id>.

### **SPRITE.Y(<id>)**

Return the y-position of Sprite <id>.

### **SPRITE.XSTEP(<id>)**

Return the x-step of Sprite <id>.

### **SPRITE.YSTEP(<id>)**

Return the y-step of Sprite <id>.

### **SPRITE.XSCALE(<id>)**

Return the x-scale of Sprite <id>.

### **SPRITE.YSCALE(<id>)**

Return the y-scale of Sprite <id>.

### **SPRITE.XSIZE(<id>)**

Return the x-size of Sprite <id>.

### **SPRITE.YSIZE(<id>)**

Return the y-size of Sprite <id>.

### **SPRITE.ROTATION(<id>)**

Return the rotation of Sprite <id>.

### **SPRITE.TINT(<id>)**

Return the tint of Sprite <id>.

#### **SPRITE.MOUSEOVER(<id>)**

Return TRUE if the mouse is positioned over Sprite <id>. Otherwise return FALSE.

#### **SPRITE.CLICKED(<id>)**

Return TRUE if the mouse is clicked on Sprite <id>. Otherwise return FALSE.

#### **SPRITE.DOUBLECLICKED(<id>)**

Return TRUE if the mouse is double clicked on Sprite <id>. Otherwise return FALSE.

#### **SPRITE.HIT(<id>)**

Return an integer that indicates Sprite <id>'s collision status with another visible Sprite. Individual bits are used to indicate the type of collision. Two types of collision detection are used: Separating Axis Theorem (SAT) for detecting collisions between convex shapes (this method is well-suited for handling rotated and scaled rectangles such as Sprites), and collision circles, which use a simple distance formula to tell if a Sprite's collision circle is touching, intersecting, or inside another visible Sprite's collision circle.

<b>BIT</b>	<b>MEANING</b>
0	Sprite <id> overlaps another visible Sprite (SAT)
1	Sprite <id>'s collision circle is touching another visible Sprite's collision circle
2	Sprite <id>'s collision circle is intersecting another visible Sprite's collision circle
3	Sprite <id>'s collision circle is inside another visible Sprite's collision circle

A returned value of 0 indicates the Sprite is not colliding with another visible Sprite.

#### **SPRITE.HIT(<id>,<tag\$>)**

Return an integer that indicates if there is a collision between Sprite <id> and another visible Sprite that is tagged <tag\$>.

#### **SPRITE.HIT(<id1>,<id2>)**

Return an integer that indicates if there is a collision between Sprite <id1> and Sprite <id2>.

---

## **Error Codes**

---

Liquid BASIC uses error codes to record an error. Error codes are stored in the ERR reserved variable. The ERR\$ function can be used to convert an error code to text.

1	Unknown error
2	error
3	Assertion Failed error
4	Input Statement Not Allowed error
5	Print Statement Not Allowed error
6	End Of Line Expected error
7	Unexpected End Of Program error



8	Closing Quote Expected error
9	Invalid Bin Data error
10	Invalid Hex Data error
11	Invalid Delimiter error
12	Invalid Operator error
13	Bad Value error
14	Syntax error
15	Valid Line Number Expected error
16	Valid Line Range Expected error
17	Valid Line Increment Expected error
18	Duplicate Label error
19	Label Not Found error
20	Cannot Edit error
21	Illegal Direct error
22	Illegal Quantity error
23	Duplicate Identifier error
24	Undeclared Variable error
25	Variable Is Constant error
26	Variable Is Array error
27	Variable Is Not Array error
28	Undeclared Array error
29	Array Dimensioned error
30	Array Not Dimensioned error
31	Array Not Vector error
32	Array Not Matrix error
33	Array Not Square Matrix error
34	Bad Subscript error
35	Bad Type error
36	Type Mismatch error
37	Division By Zero error
38	String Too Long error
39	Key Not Found error
40	Out Of Data error
41	Out Of Memory error
42	Bad EXIT error
43	Bad TRAP error
44	Stack Overflow error
45	RETURN Without GOSUB error
46	IF Without END IF error
47	EXIT IF Without IF error
48	SELECT Without END SELECT error
49	EXIT SELECT Without SELECT error
50	WHILE Without WEND error
51	EXIT WHILE Without WHILE error
52	DO Without LOOP error
53	EXIT DO Without DO error
54	FOR Without NEXT error
55	EXIT FOR Without FOR error
56	NEXT Without FOR error

57	Undeclared Type error
58	Cannot Nest TYPEs error
59	TYPE Without END TYPE error
60	END TYPE Without TYPE error
61	Undeclared Function error
62	Cannot Nest FUNCTIONs error
63	FUNCTION Without END FUNCTION error
64	EXIT FUNCTION Without FUNCTION error
65	END FUNCTION Without FUNCTION error
66	Invalid Number Of Arguments error
67	Must Pass REF error
68	Must Not Pass REF error
69	RETURN Without FUNCTION error
70	Undeclared Fiber error
71	No Fiber error
72	Bad Yield error
73	Fiber Suspended error
74	Fiber Running error
75	Fiber Finished error
76	Cannot Nest WHENs error
77	WHEN Without END WHEN error
78	EXIT WHEN Without WHEN error
79	END WHEN Without WHEN error
80	No Timer error
81	Valid Modifier Expected error
82	Valid Filename Expected error
83	File Corrupted error
84	File Not Found error
85	File Open error
86	File Not Open error
87	Access Denied error
88	Unable To Complete error
89	Directory Not Found error
90	Directory Already Exists error
91	Database Open error
92	Database Not Open error
93	No Task error
94	Bad Task State error
95	No Palette error
96	No Brush error
97	No Bitmap error
98	No Font error
99	No Tileset error
100	No Tilemap error
101	No Shader error
102	No Sound error
103	No Voice error
104	No Screen error
105	No Sprite error

106	Too Small error
107	Too Big error
108	Single Buffer error
109	Duplicate Subtexture error
110	Subtexture Not Found error
111	Modifier Not For Subtextures error
112	No Copilot error
113	Illegal Operation error
114	Valid Prompt Expected error
115	Unable To Generate Image error

---

## Inspiration

---

There they sit, the preschooler class encircling their mentor, the substitute teacher.

"Now class, today we will talk about what you want to be when you grow up. Isn't that fun?" The teacher looks around and spots the child, silent, apart from the others and deep in thought. "Jonny, why don't you start?" She encourages him.

Jonny looks around, confused, his train of thought disrupted. He collects himself, and stares at the teacher with a steady eye. "I want to code demos," he says, his words becoming stronger and more confident as he speaks. "I want to write something that will change peoples' perception of reality. I want them to walk away from the computer dazed, unsure of their footing and eyesight. I want to write something that will reach out of the screen and grab them, making heartbeats and breathing slow to almost a halt. I want to write something that, when it is finished, they are reluctant to leave, knowing that nothing they experience that day will be quite as real, as insightful, as good. I want to write demos."

Silence. The class and the teacher stare at Jonny, stunned. It is the teacher's turn to be confused. Jonny blushes, feeling that something more is required. "Either that or I want to be a fireman."

- Grant Smith (aka Denthor of Asphyxia)

1994

