

# **LIQUID STUDIO**

## **REFERENCE GUIDE**

© Copyright 2008 Global Heavy Industries. All Rights Reserved.

Liquid Player and Liquid Studio are trademarks of Global Heavy Industries.  
Other brand and product names are trademarks or registered trademarks of  
their respective holders.

Printed in the U.S.A.  
10 9 8 7 6 5 4 3 2 1

# Table of Contents

Introduction.....	1
Integrated Development Environment.....	2
Echoes.....	6
Programs.....	7
Comments.....	7
Multiple Lines.....	7
Data Types.....	7
Variables.....	8
Constants.....	9
Operators.....	10
Assignment Operators.....	11
Control Structures.....	12
Functions.....	13
Modular Programming.....	16
Objects.....	16
Classes.....	17
User Defined Types.....	19
Inheritance.....	19
Casting.....	20
Cloning.....	20
Operator Overloading.....	21
Exceptions.....	21
WITH statement.....	23
Garbage Collection.....	24
Collections.....	24
FOR EACH statement.....	25
Entities.....	26
Render Component.....	26
Update Component.....	27
Behavior Component.....	27
Consoles.....	27
Text Mode.....	28
Console Manipulation.....	29
Colors.....	30
The Cursor.....	31
Custom Characters.....	31
Bitmap Mode.....	33
Graphics Cursor.....	34
Brushes.....	34
Clipping.....	35
Graphics Primitives.....	35
Turtle Graphics.....	37
Images.....	37
Scrolling.....	39
Multiscreen Mode.....	39
Smooth Scrolling.....	39
Shapes.....	40
Sprites.....	41
Text.....	42
The Event Queue.....	43
Interrupts.....	44
Applications.....	44
Screens.....	45
Graphics Methods.....	45

2D Graphics.....	46
3D Graphics.....	49
Messages.....	51
The Message Queue.....	52
Evolution.....	53
API Reference.....	53
Bits.....	53
Blobs.....	54
Bytes.....	55
Colors.....	55
Compression.....	56
Conditionals.....	57
Desktop.....	57
Fields.....	57
Fonts.....	59
Icons.....	59
Internet.....	59
Keyboard.....	60
Math.....	61
Mouse.....	67
Parsing.....	70
Random Numbers.....	70
Sound.....	70
Strings.....	71
Timers.....	76
Words.....	78
Class Reference.....	78
App.....	78
Array.....	97
Audio.....	101
BinarySearchTree.....	103
BinaryTree.....	104
BitArray.....	105
BitmapLayer.....	108
CharSet.....	119
Collection.....	122
Console.....	122
Databank.....	145
Database.....	147
Deque.....	148
DoubleLinkedList.....	149
Entity.....	153
Entity3D.....	161
Exception.....	167
File.....	169
Fog.....	171
Font.....	174
Font3D.....	176
FTP.....	177
HashTable.....	179
Layer.....	180
Light.....	181
List.....	183
Matrix.....	187
Message.....	191
Object.....	192

PriorityQueue.....192  
PriorityQueueEx.....193  
Process.....195  
Queue.....201  
RecordSet.....202  
RedBlackTree.....205  
RNG.....205  
Shape.....207  
SingleLinkedList.....208  
SkipList.....212  
SplayTree.....215  
Sprite.....215  
Stack.....217  
Task.....218  
TCP.....219  
Text.....221  
Text3D.....224  
TextLayer.....227  
TextMap.....233  
Texture.....239  
Vector.....265  
XML.....268



# Introduction

Liquid Studio allows you to create programs for the Liquid Virtual Machine (LVM). A virtual machine is a program that runs on top of your operating system and hardware to create an abstract computer. Programs written for this abstract computer will run on any platform, regardless of the operating system or hardware, as long as the platform supports the virtual machine.

Liquid Player is a separate Windows application that implements a stable "sandbox" environment to run programs for the Liquid Virtual Machine. Liquid Player is included with Liquid Studio (and is available as a free download to end users from [www.globalheavyindustries.com](http://www.globalheavyindustries.com)).

At the heart of the Liquid Virtual Machine is the *object*. Each object has its own function and purpose, and is independent of other objects. Liquid Studio includes a number of hard-coded objects for real time 2D and 3D graphics, audio, SQL databases, XML, Internet access, and much more. You can also choose to create your own objects. Objects are pieced together to build complete software applications that can then be redistributed and run on any platform that supports the Liquid Virtual Machine.

## Credits

---

Liquid Studio was conceived, engineered, and developed by Global Heavy Industries.

This software uses the UPX open source executable packer. See <http://upx.sourceforge.net> for details.

This software uses the FreeImage open source image library. See <http://freeimage.sourceforge.net> for details. FreeImage is used under the FIPL, version 1.0.

This software uses the FreeType open source font engine. See <http://freetype.sourceforge.net> for details. FreeType is used under the FTL, version 1.0.

This software uses the public domain SQLite database engine. See <http://www.sqlite.org> for details.

This software uses the zLib compression library. See <http://www.zlib.net> for details.

## System Requirements

---

The minimum system requirements for Liquid Studio are as follows:

- Windows XP
- 500MHz CPU
- 64 MB RAM
- 16 MB video card with hardware accelerated OpenGL 1.2

Liquid Studio may run on slower machines with less memory, although its performance will suffer as a consequence. It is important to have a graphics card that supports hardware accelerated OpenGL 1.2 graphics to use Liquid Studio effectively.

## Installation

---

Run the installer to install Liquid Studio to a directory of your choice. Liquid Studio does not modify the Windows registry in any way, and all DLL files used by Liquid Studio are contained in its own directory.

Once Liquid Studio is installed it can be run by double clicking the "Liquid Studio" shortcut on the Windows' Desktop (if one was created), or by double clicking the "LiquidStudio.exe" executable in the folder where Liquid Studio was installed.

# Integrated Development Environment

Liquid Studio includes an Integrated Development Environment (IDE). The IDE allows you to enter and edit Liquid Studio programs, as well as edit simple text and HTML files.

## Registration

---

Click on the Liquid Studio icon to start the IDE.

If Liquid Studio is not registered, the following message box appears:



Click the "Use the Trial version" button to use a feature limited trial version of Liquid Studio. Click the "Register" button to register this copy of Liquid Studio:

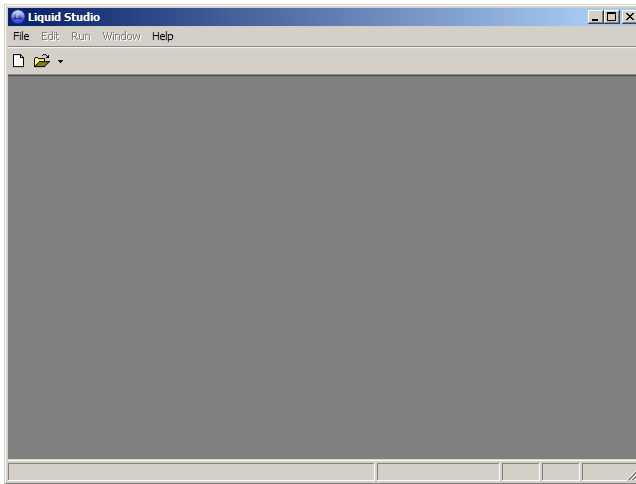


Enter your name and registration key exactly as it appeared in the e-mail sent to you when you purchased your registration key. If done correctly, Liquid Studio will save the registration key and unlock any limitations in the trial version.

## Getting Started

---

The Liquid Studio IDE starts up in the following window:



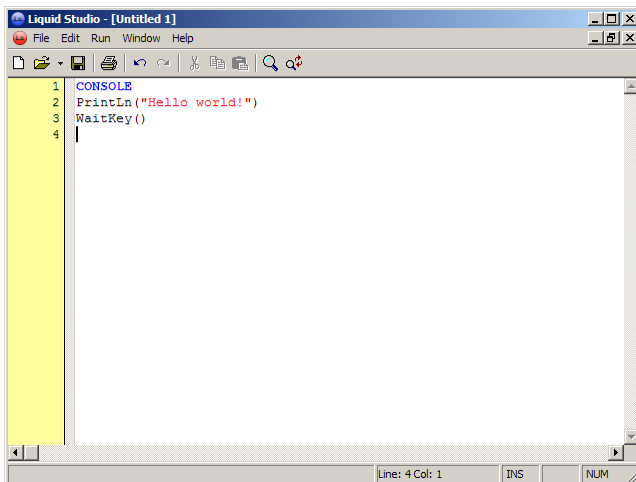
Click on the "New File" icon () to create a new file. Click on the "Open File" icon () to open an existing file.

Liquid Studio includes several sample programs in the Samples\ directory.

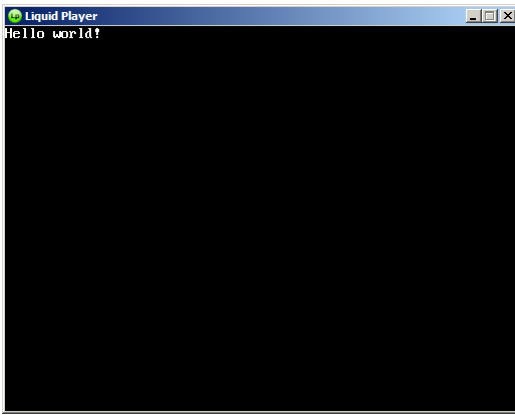
## Your First Program

---

Click on the "New File" icon and enter the following program:



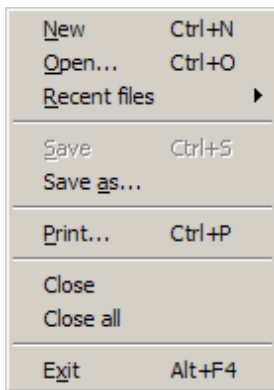
Save the file to disk with the filename Hello.lds and hit F5 to compile and run the program:



Congratulations! You have just written, compiled, and run your first Liquid Studio program.

## The File menu

---



The New menu option is used to create a new program file. Multiple program files can be open at once.

"Open..." is used to open an existing file on disk.

Recent files maintains a list of the last eight opened files.

Save is used to save the current file to disk.

"Save as..." is used to name and save the current file to disk.

"Print..." is used to print the current file.

Close is used to close the current file.

Close all is used to close all open files.

Exit is used to exit Liquid Studio.

## The Edit menu

---

Nothing to undo	Ctrl+Z
Nothing to redo	Ctrl+Y
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Delete	Delete
Select all	Ctrl+A
Block selection	▶
Find...	Ctrl+F
Find next	F3
Find previous	Shift+F3
Replace...	Ctrl+R
Go to line...	Ctrl+G
Bookmarks	▶

The Undo menu option is used to undo the last change in the editor.

Redo is used to redo the last undo.

Cut is used to cut the selected text to the clipboard.

Copy is used to copy the selected text to the clipboard.

Paste is used to paste the clipboard contents into the current file.

Delete is used to delete the selected text.

Select all is used to select the entire contents of the current file.

Block selection is used to indent, outdent, comment, or uncomment the selected text.

"Find..." is used to find a specific string in the current file.

Find next is used to find the next occurrence of the string in the current file.

Find previous is used to find the previous occurrence of the string in the current file.

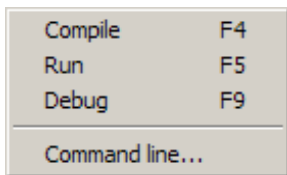
"Replace..." is used to find a specific string in the current file and replace it.

"Go to line..." is used to jump to a line in the current file.

Bookmarks is used to toggle, go to next, go to previous, and clear all bookmarks.

## **The Run menu**

---



The Compile menu option is used to compile the current file.

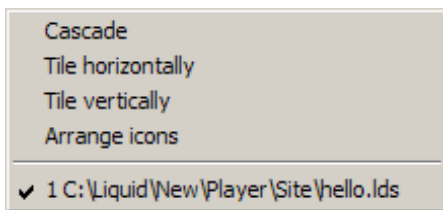
Run is used to compile and run the current file.

Debug is used to compile and run the current file in debug mode.

"Command line..." is used to specify a command line to pass to the program when run.

## The Window menu

---



The Cascade menu option is used to cascade the open windows.

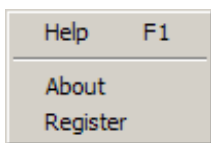
Tile horizontally is used to tile the open windows horizontally.

Tile vertically is used to tile the open windows vertically.

Arrange icons is used to arrange the icons of any minimized windows.

## The Help menu

---



The Help menu option launches Liquid Studio's online help.

About is used to show information about Liquid Studio.

Register is used to register your copy of Liquid Studio. The unregistered version of Liquid Studio can compile up to 1,000 lines of source code at a time. The registered version can compile any size source code into a Liquid Executable, which can be distributed royalty free.

## Echoes

Programs for the Liquid Virtual Machine are written in Echoes, a programming language with many modern features including classes, single inheritance, objects, collections, etc. Programs are stored as simple 7-bit ASCII text files. A program is compiled into an efficient bytecode, which in turn is run inside the Liquid Virtual Machine.

## Getting Started

---

An IDE is included to write and edit programs and run the compiler. However any text editor (such as Notepad) can be used to edit programs.

## Programs

The first line of a program determines what type of program you are compiling. Liquid Studio presently has two types of programs: CONSOLE and APP (short for application). Each type of program is explained in more detail later. For now use a console, it is the easiest to get started with.

Following the program type is the optional name of the program, which is enclosed in quotes. The program name is used in a window's caption, and in the task bar when the window is minimized. If no program name is given the program is named "Liquid Player".

```
CONSOLE                                the program is a console, named "Liquid Player"  
APP "Looking Glass"                   the program is an app, named "Looking Glass"
```

## Comments

The Liquid Studio compiler ignores everything to the right of a semicolon. Comments are not included in a compiled program (and therefore have no impact on performance or size), so use comments liberally to document your program, what it does, and how it works.

## Multiple Lines

Multiple lines can be placed on a single line by separating them with a colon.

## Data Types

The most basic piece of data used in Liquid Studio is called a *data type*.

## Boolean

---

Boolean is the simplest data type. A Boolean can be in one of two states: TRUE or FALSE.

## Integers

---

An integer data type is a whole number (no fraction). There are four distinct integer data types: BYTE, SHORT INT, INT, and LONG INT.

Integer Data Type	Memory Required	Range	Signed?
BYTE	1 byte	0 to 255	No
SHORT INT	2 bytes	-32,768 to 32,767	Yes
INT	4 bytes	-2,147,483,648 to 2,147,483,647	Yes

LONG INT	8 bytes	-9.22x10 <sup>18</sup> to 9.22x10 <sup>18</sup>	Yes
----------	---------	---	-----

The integer data types can be *cast* (converted) to a string.

## Floating Point

---

A floating point number can contain a decimal point. There are two distinct floating point data types:

Floating Point Data Type	Memory Required	Range	Precision
FLOAT	4 bytes	8.43x10 <sup>-37</sup> to 3.40x10 <sup>38</sup>	6 digits of precision
DOUBLE	8 bytes	4.19x10 <sup>-307</sup> to 1.79x10 <sup>308</sup>	16 digits of precision

The floating point data types can be cast to a string.

## String

---

A string is a collection of ASCII characters. Strings can be any length. A string can be cast to an integer or a floating point.

## Variables

Programs use *variables*, or tagged pieces of allocated memory, to store data.

### Declaration

---

Variables are "tagged" in Liquid Studio. Tags must start with a letter, and can be followed by any mix of numbers, letters, and the underscore character. Tags are not case sensitive and can be any length.

A variable must be declared before it can be used in Liquid Studio. Variables are declared by using the name of the data type, followed by the variable's tag. Multiple variables of the same data type can be declared at once by separating their tags with commas.

```
INT x                declares an int variable tagged "x"
STRING b, c, d       declares three string variables, tagged "b", "c", and "d"
```

A variable's scope is determined by where the variable is declared. If a variable is declared outside of a class, method, or function, the variable is global. Otherwise it is local to the class, method, or function it is encapsulated in.

Variable declarations can be anywhere inside a method or function, as long as it is before the variable is used.

### Assignment

---

Variables are assigned using the "=" operator:

```
var=expression      assigns variable var to expression
```

### Casting

---

Expressions can be converted from one data type to another by *casting* it. To cast an expression, use the name of the data type to cast to, followed by the expression to cast in parentheses. Not all data types can be cast. A compiler error is generated if trying to cast a data type that cannot be cast.

STRING (3.141)  
INT ("3.141")

converts the float 3.141 to the string "3.141"  
converts the string "3.141" to the int 3

## Implicit Casting

---

Data types are implicitly cast by the compiler so accuracy is not lost (or minimized) when handling two different data types. Below is the order the compiler attempts to evaluate an expression:

- BOOLEAN
- BYTE
- SHORT INT
- INT
- LONG INT
- FLOAT
- DOUBLE
- STRING

All integer math automatically casts BYTE and SHORT INT to an INT. All long integer math automatically casts BYTE, SHORT INT, and INT to a LONG INT. All double precision floating point math automatically casts FLOAT to a DOUBLE.

## Operations

---

There are three commands that operate directly on variables:

DECR var	decrement an int or float variable by one
INCR var	increment an int or float variable by one
SWAP foo, bar	swap two variables (must be of the same data type)

## Constants

Unlike a variable, which can change throughout the execution of a program, a constant is a value that does not change (hence, it is constant).

Examples of constants include:

123	decimal integer
3.141	decimal float
50%	decimal percentage (same as float divided by 100, in this case 0.5)
0xFFFFFFFF	hexadecimal integer
'x'	ASCII character (treated as an integer)
'\n'	ASCII escape character (treated as an integer)
"world"	string
'"Hello\nWorld\n"'	string with escape characters

Valid escape characters include:

\n	carriage return
\t	tab
\"	quote
\\	backslash

There are several reserved constants used in Liquid Studio:

TRUE	Boolean state
FALSE	Boolean state
FAIL	same as FALSE
SUCCESS	same as TRUE
PI	float: 3.141593
NULL	a Null (empty) object

There are other reserved constants that are used by Liquid Studio's classes. They are explained in the Class Reference.

## Tagged Constants

---

You can declare your own tagged constants in programs using the `CONST` keyword. `CONST` is followed by the data type, the tag of the constant, an equal sign, and the defined constant value. Multiple constants can be declared by separating them with commas.

```
CONST INT WIDTH = 640, HEIGHT = 480
CONST STRING APPNAME = "My App"
CONST INT LTBLUE = 0xFFB55E6C, BLUE = 0xFF792835
```

Using tagged constants allows you to declare a constant once and refer to it many times. For example, we declared `WIDTH` as an integer equal to 640, and `HEIGHT` as an integer equal to 480. We can use `WIDTH` and `HEIGHT` freely within the code, and if we ever decide to change the width and height later, we only have to change it in one spot, rather than wherever it might be used in the program.

It is good programming practice (but not required) to capitalize tagged constants. Doing so reminds you that it is a constant and not a variable.

## Unique Constants

---

The data type can be omitted to declare a unique integer constant.

```
CONST MSG_TURNLEFT, MSG_MOVEON
```

`MSG_TURNLEFT` and `MSG_MOVEON` are set to a unique integer constant. The actual value of the constant is not important. What is important is that a unique tag has been declared and can be referenced throughout the program. This is useful for declaring message body constants that can be passed around to different objects within a program. (More on messages later.)

## Operators

Binary operators use integer operands only. Comparison and logical operators evaluate to a Boolean (`TRUE` or `FALSE`).

Operators are listed in the order of their precedence.

### Power

---

`x^y` raises `x` to the `yth` power

### Binary NOT

---

`!x` performs a binary NOT on `x`

## Multiply, divide

---

$x * y$	multiplies x and y
$x / y$	divides x by y

## Modulus

---

$x \text{ MOD } y$	returns the remainder of x divided by y
--------------------	---

## Addition, subtraction

---

$x + y$	adds x and y
$x - y$	subtracts y from x

## Binary shift left, binary shift right

---

$x \ll y$	shifts x left y bits
$x \gg y$	shifts x right y bits

## Binary AND

---

$x \& y$	performs a binary AND between x and y
----------	---------------------------------------

## Binary OR, binary XOR

---

$x   y$	performs a binary OR between x and y
$x \# y$	performs a binary XOR between x and y

## Comparison

---

$x = y$	returns TRUE if x is equal to y
$x <> y$	returns TRUE if x is not equal to y
$x < y$	returns TRUE if x is less than y
$x \leq y$	returns TRUE if x is less than or equal to y
$x > y$	returns TRUE if x is greater than y
$x \geq y$	returns TRUE if x is greater than or equal to y

## Logical NOT

---

<code>NOT x</code>	returns TRUE if x is FALSE, FALSE if x is TRUE
--------------------	--

## Logical AND

---

$x \text{ AND } y$	returns TRUE if x is TRUE and y is TRUE
--------------------	---

## Logical OR, logical XOR

---

$x \text{ OR } y$	returns TRUE if x is TRUE or y is TRUE
$x \text{ XOR } y$	returns TRUE if both x and y are TRUE or x and y are FALSE

## Assignment Operators

Certain operators can be combined with the assignment operator to perform an operation on a single variable:

$\wedge =$	can be used with ints and floats
$! =$	can be used with ints
$* =$	can be used with ints and floats

/=	can be used with ints and floats
MOD=	can be used with ints and floats
+=	can be used with ints, floats, and strings
-=	can be used with ints and floats
<<=	can be used with ints
>>=	can be used with ints
&=	can be used with ints
=	can be used with ints
#=	can be used with ints

Assignment operators offer a convenient shortcut when working with a single variable. For example, `x = x + 5` and `x += 5` are equivalent.

## Control Structures

A control structure is a block of code that is only executed under certain conditions. They allow the programmer to control the flow of execution.

### IF statement

---

```
IF boolean_expression THEN statements

IF boolean_expression THEN
; statements
[ELSEIF boolean_expression THEN
; statements
[ELSE
; statements
END IF
```

### WHILE statement

---

```
WHILE boolean_expression
; statements
WEND
```

### SELECT statement

---

The SELECT statement can only be used with int, float, and string data types.

```
SELECT CASE variable
CASE boolean_expression
; statements
[CASE boolean_expression
; statements]
[CASE ELSE
; statements]
END SELECT
```

### DO statement

---

```
DO int_expression TIMES
; statements
```

LOOP

```
DO [WHILE boolean_expression | UNTIL boolean_expression]
; statements
LOOP [WHILE boolean_expression | UNTIL boolean_expression]
```

---

## FOR statement

The FOR statement can only be used with int or float data types.

```
FOR variable = begin TO end [STEP inc]
; statements
END FOR
```

```
FOR variable = end DOWNTO begin [STEP dec]
; statements
END FOR
```

The NEXT statement can be substituted for END FOR.

---

## ITERATE statement

The ITERATE statement is used to jump back to the beginning of a control structure.

```
ITERATE WHILE                iterate current while loop
ITERATE DO (or ITERATE LOOP) iterate current do..loop
ITERATE FOR                  iterate current for loop
```

---

## EXIT statement

The EXIT statement is used to jump out of a control structure.

```
EXIT IF                      exit current if statement
EXIT WHILE                   exit current while loop
EXIT SELECT                  exit current select statement
EXIT DO (or EXIT LOOP)      exit current do..loop
EXIT FOR                     exit current for loop
```

---

## Functions

Code that is often used can be wrapped in a function. Functions can be passed variables by value (meaning a copy of the variable is passed) or by reference (meaning a pointer to the variable is passed). A function can optionally return an object back to the caller.

---

## Declaration

A function must be declared before it can be used:

```
DECLARE FUNCTION tag ( [arguments] ) [AS return_class]
```

---

## Arguments

The arguments portion is optional. It is comprised of a list of variables (also called *parameters*) that get passed to the

function. Only the variable's class needs listed in a declaration; the variable's tag is not needed and is ignored if given. By default a variable is passed by value. Precede the class name with BYREF to pass the variable by reference. Multiple variables are separated with a comma.

## Implementation

---

Once a function is declared, it is implemented:

```
FUNCTION tag ( [arguments] ) [AS return_type]
  ; local variable declarations
  ; statements
END FUNCTION
```

The arguments portion (and return class) must match the declaration used in the DECLARE statement. Valid tags must be given for each variable passed as an argument. Variables declared inside a function are local to that function and are discarded upon exiting the function.

## EXIT Statement

---

The EXIT statement is used to exit a function. If the function has a return value, FALSE is returned for Boolean functions, 0 is returned for int and float functions, and "" is returned for string functions. Use RETURN in a function to return a set value.

```
EXIT FUNCTION                exit current function
```

## RETURN Statement

---

The RETURN statement is used to return a value back to the caller and can only be used inside a function with a return value. The value returned must match the data type declared after AS. Use casting if necessary.

```
RETURN expr                  return expr back to the caller
```

## Calling a Function

---

When a function is called it is said to be *activated*. A function can be activated using the following format:

```
tag ( [arguments] )
```

The arguments to pass to the function are evaluated and pushed onto the program's stack, and are treated as local variables within the function. Control then passes to the function.

Functions with a return value can be called in an expression, where the returned value from the function is substituted as an operand for evaluation. When calling a function outside of an expression, the results from the function (if any) are discarded.

```
AddTwo (5)                  the results from this function call are discarded
x=AddTwo (10)                the results from this function call are stored in x
```

## Passing by Reference

---

An argument passed by value (the default) is passed as a copy to the function. As such, any changes made to this variable are local to the function. Arguments can also be passed BYREF. With BYREF, a variable passed as an argument gets passed as a pointer to the variable, so any changes made inside the function will affect the variable passed. If, instead of a variable

being passed, an expression is passed BYREF the expression is evaluated, a temporary block of memory is allocated, and a pointer to the temporary block of memory is passed. Any changes to the variable in the function technically change the temporary block of memory, but this temporary block of memory is discarded when the function exits.

Arguments should be passed by value because it is faster and less prone to logic errors. Only use BYREF if you need to return more than one item of data back to a caller.

## Recursion

---

Local variables are unique to each activation of the function. As such a function allows *recursion*, meaning a function can call itself.

## Static variables

---

Variables can be declared *static* using the STATIC keyword. A static variable is a variable who is only visible inside a function. However it is not discarded when the function ends. Instead it retains its value, so the next time the function is activated the static variables are preserved.

## PREPARE statement

---

Sometimes it is useful to initialize some data the first time a function is activated. The PREPARE statement facilitates this:

```
PREPARE
; statements
END PREPARE
```

Code inside a PREPARE block is only run the first time the function is called. Otherwise it is ignored.

## Overloading

---

Functions can also be *overloaded*, meaning that the name of a function can be reused as long as its arguments are different.

The collective arguments of a function are called its *signature*. The BYREF option affects the signature, so a parameter list of one int passed by value has a different signature than a parameter list of one int passed by reference. Each overloaded function must have a unique signature.

When a function is overloaded, the compiler uses the following rules when evaluating the parameters to determine the proper function to call:

- find an exact match
- widen the data types to find a match
- narrow the data types to find a match
- generate a compiler error if no match

If an exact match is not found, the compiler will first try to widen the data types (e.g. from a BYTE to an INT). If that fails, then the compiler will try to narrow the data types (e.g. from an INT to a BYTE). If that fails, a compiler error is generated.

Explicit casting can be used to force the compiler to call a particular function.

## MAIN Function

---

Programs require a MAIN function to start. The MAIN function must be declared, can be located anywhere, takes no parameters, and must return an int. The int returned by the MAIN function is the program's "errorlevel", which is returned to

the operating system upon the program's exit.

Consoles have an exception: if no functions or classes are declared or implemented in the console, the first line of code is considered to be the start point. Also, all declared variables are assumed to be global (which they are, since there can be no functions or classes). The following is a valid console program:

```
CONSOLE
STRING Name
Print("What is your name? ")
Name=Input()
Print("Hello " + Name + "!")
WaitKey()
```

This is intended to allow quick and simple consoles to be written. If the console is of any complexity it is best to use a MAIN function to designate the start point.

## Modular Programming

Functions and classes that are used often can be broken out and saved in their own file. The INCLUDE keyword is used to insert another file into the current file:

```
INCLUDE "filename"          insert a file into the current file
```

This is very useful for reusing code. For example, Liquid Studio has a complete GUI that can be brought in when needed:

```
INCLUDE "gui.ldc"
```

This will insert the file gui.ldc into the current file, allowing the current file to use all the constants, functions, and classes defined in gui.ldc.

## Objects

Objects bundle code (logic) and data together into a single, cohesive unit. An object is built from a *class*. A class defines the data that makes up an object and defines the *methods* and *functions* that can be called on that data. Using the class as a blueprint, an object is *instantiated* (created). Each object is an *instance* of the class, and is independent, separate from other objects. Programmers can create their own classes, and even *inherit* data and code from a parent class (more on this later).

Once a class is written it can be reused again and again, anytime an object of that class is required. Objects are then pieced together, making it possible to create very intricate software with minimal effort.

## Classes

Classes spell out the exact requirements necessary to build an object, including how much memory the object will need and "actions" that the object can take and that can be taken on the object.

The Liquid Virtual Machine includes about 60 built-in classes from which to build objects. These built-in classes provide a simple interface to the operating system and various data structures and hardware.

## Fields

Each class can encapsulate smaller, more defined objects to build itself. These encapsulated objects are called *fields*, and can include data types, built-in classes, or any class already declared elsewhere in the program.

## Methods

---

A *method* is a block of code that can access an instance of a class. The instance is implicitly passed to the method, which can then reference the instance using the THIS keyword. Additional information, called *parameters*, can be passed to a method (for instance, a "paint" method needs to know what color to paint something). Methods can optionally return a single object back to the caller.

## Properties

---

A *property* filters an object's access through a defined GET and SET parameter. The parameter can be a field of the same class as the property, in which case the property simply maps onto the field. Alternatively, the parameter can be a method, using the same class as the property (returning the class for the GET parameter, or accepting one instance of the class for the SET parameter). Using a method allows logic to be assigned to an object's access (a GET method might calculate a return value, a SET method might perform some error checking before setting the property). Using properties also allows fields to be designated as read only (by using a GET parameter and not a SET parameter) or write only (by using a SET parameter but not a GET parameter).

## Functions

---

Unlike a method, which acts on an instance of a class, a *function* encapsulated inside a class uses no instance of the class. Instead, information is passed to the function via parameters. Functions are often called *static methods* in other programming languages. Functions can optionally return a single object back to the caller.

## Members

---

A *member* of a class refers to something that is encapsulated in (belongs to) a class. Members include all of the class's fields, methods, properties, and functions.

## Scope

---

Fields, methods, and properties have three different scopes:

- `private`                                    only visible to the class that they are encapsulated in
- `protected`                                visible to the class they are encapsulated in and any class(es) that inherit from it
- `interface`                                visible inside and outside the class

Functions must be declared inside the class interface. They can not be private or protected.

## Declaration

---

The class declaration declares the fields, methods, properties, and functions of a class. The declaration also defines the class's *interface*, or how the object can be interacted with.

Classes are declared using the following structure:

```
[ABSTRACT | FINAL] CLASS tag [EXTENDS class]
  [PRIVATE
    class tag [, tag ...]
    DECLARE METHOD tag ( [arguments] ) [AS return_type]
    PROPERTY class tag [GET field | method] [SET field | method]
    ...]
  [PROTECTED
```

```

class tag [, tag ...]
DECLARE [VIRTUAL] METHOD tag ( [arguments] ) [AS return_type]
PROPERTY class tag [GET field | method] [SET field | method]
...]
[INTERFACE
DECLARE [VIRTUAL] METHOD tag ( [arguments] ) [AS return_type]
PROPERTY class tag [GET field | method] [SET field | method]
DECLARE FUNCTION tag ( [arguments] ) [AS return_type]
...]
END CLASS

```

## Implementation

---

The class implementation defines the code (logic) portion that implements the class. Classes are implemented using the following structure:

```

IMPLEMENT tag
  [OVERRIDE] METHOD tag ( [arguments] ) [AS return_type]
    ; local variable declarations
    ; statements
  END METHOD
  FUNCTION tag ( [arguments] ) [AS return_type]
    ; local variable declarations
    ; statements
  END FUNCTION
...
END IMPLEMENT

```

## Interface

---

One of the golden rules of object oriented programming is to always program to a class's interface, and never to a class's implementation. Following this rule avoids *coupling* of objects, where object implementations become dependent on each other. By always programming to the interface, the implementation of the class can change freely without affecting the object, or (more importantly) any objects that interact with the object. For example, a class might encapsulate an array. If other classes accessed this array directly, and it was later decided to change the array to a linked list, all of the classes that access the array have to be rewritten to use the new linked list. Far better is to make the array private or protected, and use a method or property to wrap the array. The array can then be accessed using the method or property. Then, if it is decided to use a linked list (or skip list or binary tree or whatever) instead of the array, only the class itself needs rewritten. All other classes continue to use the appropriate method or property to access the data they need.

To enforce *decoupling*, fields must be private or protected in a class. They can not be declared inside the class interface. Instead a method or property can be used to wrap a field and expose it to the outside world via the interface.

## Constructor

---

A class can have an optional *constructor* that is run when the object is instantiated, or created. The NEW keyword is used to instantiate an object. When an object is instantiated, memory for the object's fields are allocated. If no constructor is specified in the class, all fields are set to zero (or Null) by default. The constructor method is used to initialize the object. Constructors can have an argument list. It is common practice to overload constructors so that an object can be instantiated in different ways (for example, the Texture object can instantiate a blank texture, load an image from disk, or load a thumbnail, depending on the parameters passed to its constructor).

## Destructor

---

Classes can also have an optional *destructor* method that runs when the object is freed. By default all fields of an object are automatically freed when the object itself is freed. The destructor method can be used to perform additional cleanup work before Liquid Studio automatically frees the object's fields. Destructors do not have an argument list. It is not possible to determine when a destructor is actually run, as the garbage collector (see below) will free objects at indeterminate times.

## User Defined Types

A user defined type is a special type of class that is comprised solely of public fields, which can be accessed from outside the type. A user defined type is declared using the following structure:

```
TYPE tag
  class tag [, tag ...]
  ...
END TYPE
```

User defined types do not have an implement block. Like classes, the NEW keyword is used to instantiate a TYPE. Unlike other programming languages, user defined types can encapsulate strings and objects. The garbage collector automatically cleans up after a user defined type when it is no longer in use.

## Inheritance

A class can *inherit* all the PROTECTED and INTERFACE members from a base class using the EXTENDS keyword. Inheritance allows you to build upon existing classes. Generally it is used to refine an existing abstract class to make it more concrete. For instance, you might have a Pet class that is just that – a pet. A class such as Cat or Dog would inherit from Pet, because a Cat or Dog is in fact a pet (usually, anyway), but is also a unique kind of pet. Since both Cat and Dog are derived from Pet, it is possible to assign a Cat or Dog to a Pet variable (but not vice versa: a cat is a pet but a pet is not necessarily a cat), or pass a Cat or Dog as a Pet parameter. This allows you to use objects in either an abstract fashion (Pet) or a specific fashion (Cat or Dog).

## Virtual methods

---

A *virtual method* is a method that is resolved at runtime, as opposed to compile time. This allows inherited objects to respond differently to the same method call. Virtual methods must be protected or in the class interface, as it is not possible to override a private method (since it is not inherited).

For example, suppose the Pet class has a MakeNoise virtual method. The Cat class, which extends the Pet class, overrides the default MakeNoise method with a method that makes a meow sound. The Dog class, which also extends the Pet class, overrides the default MakeNoise method with a method that makes a bark sound. Either a Cat or Dog object can be assigned to a Pet variable. When the Pet variable has its MakeNoise method invoked, it is not known at compile time if the actual variable is a Pet, a Cat, or a Dog (or any other class that extends the Pet class). However since MakeNoise is a virtual method, the actual method to call is resolved at runtime. The Pet object will do nothing, the Cat object will meow, and the Dog object will bark. This allows for *polymorphism* (Greek: "having many faces") and is a key feature of object-oriented languages.

## The Object class

---

Every class in Liquid Studio derives from the Object class. If no EXTENDS is specified in a CLASS block it is assumed that the class extends Object. The Object class is the "root" of the class hierarchy in Liquid Studio. As such it provides a generic way to address objects.

## ABSTRACT

---

The ABSTRACT keyword is used to declare a class as an abstract class. An abstract class is a class that can not be instantiated. Typically abstract classes define a generic type of object and the interface to that object. It is up to classes that extend the abstract class to actually implement the interface. Once an implementation is defined the class can be instantiated.

## FINAL

---

The FINAL keyword is used to declare a class as a final class. A final class is a class that can not be extended further.

## OVERRIDE

---

The OVERRIDE keyword is used to override an existing method in an inherited class. This is used to change the behavior of a method for a derived class.

## INHERITED

---

The INHERITED keyword is used inside an overridden method to call the original method in the base class.

## Casting

An object can be cast to a different type of class using the CAST method:

```
METHOD CAST() AS class
```

The return object can be any valid data type or class. Multiple cast methods can be used to cast an object to different types.

For example, the Font object can be cast to a string, in which case it returns the name of the font:

```
CONSOLE "Cast"  
FONT f  
f = NEW FONT("Arial", 10, FALSE, FALSE, FALSE)  
PrintLn (STRING (f))  
WaitKey()
```

This sample code will cast the font f to a string ("Arial") and print it.

## Cloning

An object can make a clone of itself using the CLONE method:

```
METHOD CLONE() AS class
```

The return object must match the class of the object to clone.

## Operator Overloading

A class can overload operators to provide a specific implementation for the operator. The following operators can be overloaded:

```
^, !, *, /, MOD, +, -, <<, >>, &, |, #
```

To overload an operator, use a function with the keyword OPERATOR:

```
FUNCTION OPERATOR + (class lhs, class rhs) AS class
```

Place the operator to overload after the keyword OPERATOR. Unary operators (! and -) must have one argument, which must match the class the function is in. Binary operators must have two arguments, one of which must match the class the function is in. The function can return any type of object.

Operator overloading is a powerful feature and should be used only when needed. For example, consider a complex number class, which encapsulates a complex number (an ordered pair of real numbers). The class could benefit from having the mathematical operators +, -, \*, and / overloaded. This would allow complex numbers to be used in expressions, the same way as data types can be used.

Overloading an operator does not affect its precedence. You can not overload operators for the internal data types (such as ints, floats, and strings).

Overloaded operators are not commutative (that is, independent of order; a+b and b+a are the same, hence they are commutative, whereas a-b is not the same as b-a). To make an operator commutative a second function is required:

```
FUNCTION OPERATOR * (ComplexNumber t, FLOAT s) AS ComplexNumber
; code
END FUNCTION

; make the * operator commutative
FUNCTION OPERATOR * (FLOAT s, ComplexNumber t) AS ComplexNumber
; wrap the first function
RETURN t + s
END FUNCTION
```

Classes can also overload relational operators using the COMPARE keyword:

```
FUNCTION COMPARE(class lhs, class rhs) AS INT
```

The two parameters must match the class the function is in. An int must be returned from the function. The function should return -1 if lhs is less than rhs, 1 if lhs is greater than rhs, or 0 if lhs is equal to rhs. If an object has a compare function defined then it will use it when evaluating the =, <, <=, >, >=, and <> operators. The compare function is also used in collections, such as arrays and vectors, when sorting objects.

## Exceptions

An *exception* is an object that indicates an error (such as divide by zero or file not found) occurred during runtime. Error trapping is the ability to "catch" exceptions. Error trapping is turned off (disabled) by default, but can be enabled using the Trap method:

```
trap(state)
```

If state is TRUE, then error trapping is enabled. If state is FALSE, then error trapping is disabled.

When an exception occurs, and error trapping is disabled, the program automatically unloads and reports the exception to the user via a Windows message box. If error trapping is enabled, the exception is recorded in the program's *trap*, and execution of the program resumes.

There are two ways to read the current trap while error trapping is enabled to see if an exception occurred:

```
exception:checktrap()
exception:emptytrap()
```

The first method, CheckTrap, returns the current exception. The second method, EmptyTrap, returns the current exception and "resets the trap" (sets the trap to Null, or no error).

## Exception Constants

---

Listed below are the error codes in Liquid Studio (bolded error codes are errors that can not be trapped, even with error trapping enabled):

VM_NONE	no reported exception
<b>VM_INTERNAL_ERROR</b>	an internal error has occurred
VM_USER	a user generated exception has occurred
<b>VM_OUT_OF_MEMORY</b>	Liquid Studio has run out of memory
VM_DENIED	the requested operation was denied
VM_ILLEGAL_QUANTITY	an illegal quantity was used
VM_NOT_DIMENSIONED	a collection was used before it was properly dimensioned
VM_BAD_SUBSCRIPT	a bad subscript in a collection was used
VM_KEY_NOT_FOUND	the specified key was not found in a collection
VM_DUPLICATE_KEY	the specified key already exists in a collection
VM_NOT_SORTED	a collection is not sorted
VM_BAD_ITEM	a bad item was used in a collection
VM_ITEM_NOT_FOUND	the specified item was not found in a collection
VM_DUPLICATE_ITEM	the specified item already exists in a collection
<b>VM_TIMEOUT</b>	Liquid Studio has timed out (stopped responding)
<b>VM_STACK_OVERFLOW</b>	the stack has overflowed (ran out of space)
VM_OVERFLOW	an int or float overflowed
VM_DIVISION_BY_ZERO	an int or float was divided by zero
VM_NULL_OBJECT	an attempt to use a Null (empty) object was made
VM_BAD_HANDLE	a bad handle was used
VM_BAD_FILE_MODE	a bad file mode was used
VM_FILE_NOT_FOUND	the specified file was not found
VM_FILE_OPEN	a file is already open
VM_FILE_NOT_OPEN	a file is not open
VM_NOT_BINARY_FILE	a binary operation was made on a file that is not a binary file
VM_NOT_INPUT_FILE	an input operation was made on a file that is not an input file
VM_NOT_OUTPUT_FILE	an output operation was made on a file that is not an output file
VM_BAD_FILE_FORMAT	a file is in a bad (unsupported) file format
VM_TCP_TIMEOUT	a TCP/IP operation timed out
VM_TCP_OPEN	a TCP/IP socket is already open
VM_TCP_NOT_OPEN	a TCP/IP socket is not open
VM_FTP_ERROR	an FTP specific error occurred, the exception data holds the error message
VM_DATABASE_ERROR	a database specific error occurred, the exception data holds the error message
VM_XML_ERROR	an XML specific error occurred, the exception data holds the error message

## User exceptions

---

The THROW statement is used to generate a user exception:

```
THROW "string"
```

The exception VM\_USER is thrown, with the exception's data holding the string. Enable error trapping to catch any user generated exceptions.

## WITH statement

The WITH statement makes it convenient to repeatedly reference an object's members:

```
WITH object
; statements
END WITH
```

Any valid object can be used. Statements within the WITH block can reference the object's members using the period (.) prefix without the object name.

The following two code snippets are the same:

```
SimpleAnimationObject.Show()
SimpleAnimationObject.Center()
SimpleAnimationObject.Scale(200%, 200%)
```

```
WITH SimpleAnimationObject
    .Show()
    .Center()
    .Scale(200%, 200%)
END WITH
```

The second snippet not only saves typing but allows for easier change down the road if (when) you decide to change it. If SimpleAnimationObject needs to change to a different object, there is only one place to change it instead of three.

WITH blocks can be nested. The period prefix always references the object in the current WITH block.

## Example

```
CONSOLE

TEXTURE t
SPRITE sp

BitmapMode(256, 256)

t = NEW TEXTURE(256, 256, TX_LINEAR)

sp = NEW SPRITE(t)
sp.Show()
sp.Center()

WITH t
; methods prefixed with . reference the texture object t
    .Particle(100)
    .Tile()
    .MoveDistort(64, 64)
    .Twirl(500, 5000)
    .SineDistort(50, 30, 60, 20)
    .EqualizeFull()
    .Refresh()
END WITH

WaitKey()
```

## Garbage Collection

Liquid Studio has automatic memory management. This means that memory is automatically allocated and deallocated as needed for objects. As such you do not need to worry about freeing an object. Liquid Studio can automatically detect when an object is no longer in use and will free it for you.

To accomplish this Liquid Studio uses *garbage collection*. Garbage collection is a process to handle memory deallocation to reclaim lost memory that is no longer in use. Liquid Studio tracks the number of references to each object. When all of the references to an object have disappeared, the object is marked as no longer in use. At intermittent times a *sweeper* runs, going through all the marked objects and safely deleting them.

Garbage collection is an automatic process. The time that an object is actually deleted is indeterminate. As long as the object is marked as no longer in use it can be deleted at anytime by the virtual machine. Because of this the destructor of an object should **never** be assumed to run at a given time.

This might seem counter-intuitive at first. Why bother with garbage collection and the extra overhead of marking and sweeping unused objects? Garbage collection offers two key benefits. First, because memory is automatically deallocated, it prevents many common programming mistakes, such as using a pointer to an object that no longer exists or forgetting to deallocate memory, resulting in memory leaks. Second, if an object was freed immediately when no longer used, it could stall the program. For example, consider having an array of 20,000 objects. If the array is cleared, the program would stall for a moment as 20,000 objects were immediately freed. Instead, by marking the 20,000 objects for deletion and doing the deletes in small increments over time when the program is idle, it allows the deletions to be amortized. This results in faster, more responsive code.

## Collections

A *collection* is a special type of object that is used to hold zero, one, or more objects of the same class. The most widely known collection in BASIC and C is the array, a single block of objects that can be referenced using an index. Liquid Studio supports many types of collections:

<i>Collection</i>	<i>Description</i>
bitarray	a collection of bits
array	a collection of objects of the same class
matrix	a 2D array
vector	a collection of objects that can dynamically grow and shrink
hashtable	an array that is indexed using a unique string, instead of an int (also known as an associative array)
list	a generic linked list implementation
stack	objects can be pushed to and popped from the stack's tail (FILO)
queue	objects are enqueued to the queue's tail and dequeued from the queue's head (FIFO)
deque	objects can be pushed to and popped from the deque's head or tail
singlelinklist	a single linked list implementation
doublelinklist	a double linked list implementation
skiplist	an ordered linked list with multiple forward links
binarytree	a binary tree, where each node can have zero, one, or two children
binarysearchtree	an ordered binary tree

splaytree	a splay tree
redblacktree	a red-black tree (automatically balances itself)
priorityqueue	a sorted queue where each object has its own defined priority
priorityqueueex	a sorted queue where each object has its own defined priority; objects can be removed or change priority

For more information on each type of collection see the Class Reference section.

## FOR EACH statement

The For Each statement is used to iterate through a collection and perform an action on each individual item:

```
FOR EACH var IN collection
    ; statements
END FOR
```

The first iteration populates var with the first item in collection (var must match the class of the collection). Each successive iteration the next item in the collection populates var and the loop repeats. If the collection is empty the FOR EACH block is skipped.

The FOR EACH block can be used on the array, vector, and list collections. It is good practice to use, as it ensures that each and every item in the collection is touched, avoiding a common mistake in programming.

## Example

```
CONSOLE
ARRAY INT i
INT Inx, c
i = NEW ARRAY()
i.Dim(20)
FOR Inx = 0 TO 20
    i[Inx] = Range(1, 1000)
END FOR
FOR EACH c IN I
    PrintLn(STRING(c))
END FOR
WaitKey()
```

## Entities

The Entity class extends the Object class and adds three additional *components*. The Render component defines how the entity should appear on-screen. The Update component defines how the entity should act. The Behavior component defines how the entity should react to messages.

A console can not extend the Entity class or any child of the Entity class. A console can, however, use three predefined entities: shape, sprite, and text.

Only an app can extend the Entity class with a defined Render, Update, and/or Behavior component.

## Render Component

The Render method appears in the class interface and is used to define the Render component of an entity:

```
METHOD RENDER ()  
    ; code  
END METHOD
```

If the entity is visible the Render method is called automatically in consoles, and during the gxRender method in apps.

Each entity can be independently moved, tinted, blended, scaled, and rotated. A child object (one encapsulated in a parent object) assumes the traits of its parent. For example, consider a GiantRobot object that encapsulates the RobotArms and RobotLegs objects. The RobotArms and RobotLegs objects can be rotated individually. Rotating the GiantRobot object, however, also rotates any objects encapsulated in GiantRobot, **including** the RobotArms and RobotLegs objects.

The following methods are used on an entity's Render component:

boolean:GotFocus ()	return TRUE if the entity has the keyboard focus
SetFocus ()	set the entity to have the keyboard focus
boolean:IsMouseOver ()	return TRUE if the mouse is over the entity
boolean:IsMouseOverNode (n)	return TRUE if the mouse is over node n in the entity
int:GetMouseOverNode ()	return the current node the mouse is over in the entity
boolean:IsClicked ()	return TRUE if the mouse is clicked on the entity
boolean:IsNodeClicked (n)	return TRUE if the mouse is clicked on node n in the entity
int:GetNodeClicked ()	return the current node the mouse is clicked on in the entity
boolean:IsVisible ()	return TRUE if the entity is visible
Show ()	show the entity
Hide ()	hide the entity
int:GetDepth ()	return the entity's depth
SetDepth (n)	set the entity's depth (entities are sorted by depth)
int:GetTint ()	return the entity's tint
Tint (color)	tint the entity
float:GetBlend ()	return the entity's blend
Blend (n)	blend the entity
int:GetXPos ()	return the entity's X coordinate
int:GetYPos ()	return the entity's Y coordinate
SetXPos (x)	set the entity's X coordinate
SetYPos (y)	set the entity's Y coordinate
Center ()	center the entity on the screen
Move (x, y)	move the entity to (x, y)
MoveRel (x, y)	move the entity relative to its current position
MoveDir (angle, rad)	move the entity in the direction angle by rad (radius) pixels
float:GetXScale ()	return the entity's X scale
float:GetYScale ()	return the entity's Y scale
SetXScale (xs)	set the entity's X scale
SetYScale (ys)	set the entity's Y scale
Scale (s)	scale the entity
Scale (xs, ys)	scale the entity using the X and Y scale
float:GetRotate ()	return the entity's rotation (in degrees)
SetRotate (d)	rotate the entity
Rotate (d)	rotate the entity
string:GetToolTip ()	return the entity's ToolTip
SetToolTip (tt)	set the entity's ToolTip (shows if the mouse hovers over the entity for a second)

The `IsMouseOver`, `IsMouseOverNode`, and `GetMouseOverNode` methods only work if the mouse button is down or mouse feedback is enabled (see the `MouseFeedback` method in the API).

## Update Component

The `Update` method appears in the class interface and is used to define the `Update` component of an entity:

```
METHOD UPDATE ()  
    ; code  
END METHOD
```

If the entity is running the `Update` method is called automatically in consoles, and during the `Evolve` method in apps.

The following methods are used on an entity's `Update` component:

<code>boolean: IsRunning ()</code>	return TRUE if the entity is running
<code>Start ()</code>	start the entity
<code>Stop ()</code>	stop the entity

## Behavior Component

The `Behavior` component is found in the class interface and is used to define the `Behavior` component of an entity:

```
METHOD BEHAVIOR (MESSAGE m) AS BOOLEAN  
    ; code  
END METHOD
```

If the entity is enabled the `Behavior` method is called when a message is dispatched to the entity. The dispatched message is passed as a parameter to `Behavior`. The return `Boolean` indicates if the `Behavior` processed the message (`TRUE`) or not (`FALSE`). If the message was not processed, it is passed up to the entity's parent (the entity encapsulating this entity) for processing.

The following methods are used on an entity's `Behavior` component:

<code>boolean: IsEnabled ()</code>	return TRUE if the entity is enabled
<code>Enable ()</code>	enable the entity
<code>Disable ()</code>	disable the entity

## Consoles

A console is similar to an MS-DOS program or Windows console. Consoles use a linear programming model. This frees the programmer from the Windows event-driven model, where a program must constantly run a "message pump" to handle interactions with other windows and the operating system. Events, such as keyboard and mouse events, are still sent to a console and enqueued for later retrieval, but the console is not dependent on the message pump to stay responsive. Instead, the virtual machine uses *interrupts* to automatically dispatch messages, render the screen, perform garbage collection, etc.

Consoles have many different *methods* (commands) that can be used to manipulate the console, print text, draw graphics, and more.

## Text Mode

Text mode is used to input and output text to a console:



<code>PrintAt(x, y, string)</code>	print a string at (x, y)
<code>CenterText(string)</code>	center a string
<code>float:InputNum()</code>	input a number from the user
<code>string:Input()</code>	input a string from the user
<code>ClearKey()</code>	clear the keyboard buffer
<code>int:GetKey()</code>	return the next key in the event queue
<code>WaitKey()</code>	wait until a key is pressed

The `Print`, `PrintLn`, `PrintAt`, and `CenterText` methods are all overloaded, so they can accept a string, int, or float as a parameter. For example, `PrintLn(3.141)` will print the float 3.141, followed by a line feed.

Every time a key is pressed in a console it gets stored in the console's event queue. The event queue is used to hold *events* sent to the console in a first in first out (FIFO) fashion. The `GetKey` method gets the next keydown event in the event queue and returns the character pressed as an int. `WaitKey` discards all the events currently in the event queue, and pauses the console until a key is pressed (the event is not stored in the event queue).

With these methods you can write simple consoles that input and output data to the console, allowing you to experiment with Echoes.

### Example: "Hello world!"

```
CONSOLE
PrintLn("Hello world!")
WaitKey()
```

### Reading and Writing Characters

The `Poke` and `Peek` methods can be used to quickly read and write characters to the text screen:

<code>Poke(addr, ch)</code>	set the character at addr to ch
<code>int:Peek(addr)</code>	return the character at addr

`addr` is the "address" to poke or peek to. Address 0 is the upper left corner of the screen. Address 1 is the position of the next character to the right of that, and so on down the row. Address 63 is the right-most position of the first row. The next address (64) following the last character on a row is the first character on the next row down. Address 1535 is the lower right corner of the screen (64 columns \* 23 rows + 63<sup>rd</sup> column = 1535).

### Console Manipulation

The console window can be renamed, moved, and resized as needed:

<code>string:GetConsoleName()</code>	return the console's name
<code>RenameConsole(caption)</code>	rename the console (window caption)
<code>int:GetConsoleX()</code>	return the console's X position
<code>int:GetConsoleY()</code>	return the console's Y position
<code>MoveConsole(x, y)</code>	move the console to (x, y)
<code>int:GetConsoleWidth()</code>	return the console's width in pixels
<code>int:GetConsoleHeight()</code>	return the console's height in pixels
<code>ResizeConsole(w, h)</code>	resize the console to width, height

### Colors

Colors in Liquid Studio are represented with a 32-bit integer. The int can be broken down into four separate components:

red, green, blue, and alpha. Each component is 8-bits (0 to 255), and specifies the intensity of the component (for example, if the red component is 255, then the color has a maximum intensity of red in it; if the red component is 0, then the color has no red in it). The alpha component indicates how opaque a color is. An alpha value of 255 indicates the color is opaque, a value of 0 indicates the color is transparent, and a value between 1 and 254 indicates a varying degree of how translucent the color is. Colors are made by mixing these various components together.

Hexadecimal numbers can be used to define a color. Keep in mind that the hexadecimal number is in the format ABGR, or alpha-blue-green-red. For example, 0xFF008000 would define a solid, medium green (alpha is 0xFF, green is 0x80).

There are also several API functions that are used to define a color:

<code>int:RGB(r, g, b)</code>	return an integer with the specified RGB components (A=255)
<code>int:RGBA(r, g, b, a)</code>	return an integer with the specified RGBA components
<code>int:GetRandomColor()</code>	return an integer representing an opaque random color (A=255)
<code>int:GetPlasmaColor()</code>	return an integer representing an opaque plasma color (A=255)

The RGB and RGBA functions accept ints which are clamped from 0 (minimum) to 255 (maximum). These functions are also overloaded; they can be called with floats instead of ints, in which case the floats are clamped from 0 (minimum) to 1 (maximum). For example, `RGB(100%, 50%, 0%)` returns `0xFF0080FF`, the 32-bit integer that corresponds to a color with 100% red, 50% green, and 0% blue, or the color orange.

`GetRandomColor` returns an opaque random color, and `GetPlasmaColor` returns an opaque plasma color (the plasma color is maintained internally, and is a color that constantly cycles through all the colors).

Liquid Studio reserves the following color (int) constants: BLACK, WHITE, GRAY, SILVER, RED, ORANGE, YELLOW, GREEN, CYAN, BLUE, PURPLE, PINK, and BROWN.

## Using Colors

---

Consoles have methods to change the colors used in a console:

<code>int:GetPaper()</code>	return the current paper color as an int
<code>Paper(color)</code>	set the current paper color
<code>int:GetInk()</code>	return the current ink color as an int
<code>Ink(color)</code>	set the current ink color
<code>int:GetHighlight()</code>	return the current highlight color as an int
<code>Highlight(color)</code>	set the current highlight color

Each character cell in a console can have its own independent foreground (ink) and background color (highlight). If no highlight color is specified for an individual character cell, the console's paper color is used as the background color.

When text is printed using one of the Print methods the current ink and highlight color are applied to the printed text.

### Example: Counter

---

```
CONSOLE "Counter"
INT Inx
TextMode(16, 24)
DO
    Ink(GetRandomColor())
    INCR Inx
    PrintLn(Inx)
LOOP UNTIL GetKey()
```

## The Cursor

Liquid Studio uses a cursor in text mode to indicate where the next text to be printed appears. The cursor appears on-screen as a flashing yellow block. By default the cursor is hidden when a console is running; it only appears during the Input method when it is time to get input from the user. Once the user enters his or her input and presses Enter, the cursor is hidden again.

<code>boolean:IsCursorVisible()</code>	return TRUE if the cursor is visible
<code>ShowCursor()</code>	show the cursor all the time
<code>HideCursor()</code>	only show the cursor during input (default)
<code>int:GetCursor()</code>	return the cursor's color
<code>Cursor(color)</code>	set the cursor's color
<code>int:GetCursorX()</code>	return the cursor's x coordinate
<code>int:GetCursorY()</code>	return the cursor's y coordinate
<code>Locate(x, y)</code>	move the cursor to the specified (x, y) coordinates

## AutoScroll

If the cursor moves past the bottom of the screen the screen is automatically scrolled up to make room for a new line. This behavior can be changed by using the AutoScroll method:

`AutoScroll(state)` enable (TRUE) or disable (FALSE) auto scrolling

If state is TRUE the screen will automatically scroll up as needed (default). If state is FALSE the screen does not scroll, and the cursor is simply moved to the left hand side on the last line.

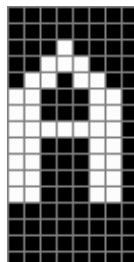
## Custom Characters

Liquid Studio uses a *character set* to render text to the text screen. By default the character set used is the classic DOS character set stored in your computer's BIOS. However these characters are pulled down into memory, and as such can be redefined on the fly, allowing custom characters on a text screen. Custom characters can be used to add simple graphics and animation to text screens.

Two methods are used to retrieve and set the current character set:

`charset:GetCharSet()` return the current character set  
`SetCharSet(cs)` set the current character set

Each character is comprised of individual pixels. For example, characters in the default font are 8 pixels across and 16 pixels down. The letter A looks like this:



This letter is comprised of 16 "scan lines", one scan line for each pixel row, numbered from 0 (top) to 15 (bottom). Each scan line can be independently written using the Palette and Write methods of the CharSet object:

`Palette(ch, c)` assign an ASCII character (ch) to color c

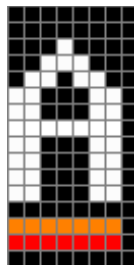
Write(ch, y, string)                      replace scanline y in character ch with the pixel row in string

Each CharSet has its own palette of 256 colors. The Palette method is used to assign an ASCII character to a color. The Write method is then used to "plot" a new pixel row in a character. The pixel row is a string, where each ASCII character is replaced by the corresponding color in the CharSet's palette.

For example:

```
CharSet cs
cs = GetCharSet()
cs.Palette('-', ORANGE)
cs.Palette('=', RED)
cs.Write('A', 13, "-----")
cs.Write('A', 14, "=====")
```

This code snippet would replace scan lines 13 and 14 (remember scan lines start at 0) in the character A with orange and red pixels:



Now the character A has scan lines 13 and 14 updated with the new pixels. This change only occurs in system memory, however. The last step is to update video memory with the new changes, using the Refresh method in CharSet.

Refresh()                                  refresh the character set in video memory

The Refresh method is fairly significant in terms of processing time, so it is best to update all the characters in the CharSet that need updated, and then Refresh once, rather than Refresh after every single change.

The following console program gives a simple demonstration of custom characters:

```
CONSOLE
CHARSET cs
TEXT t
; Get the default console character set
cs = GetCharSet()
; Underline the letter A with an orange and red stripe
cs.Palette('-', ORANGE)
cs.Palette('=', RED)
cs.Write('A', 13, "-----")
cs.Write('A', 14, "=====")
cs.Refresh()
; Use a text object to show the letter A
t = NEW TEXT(cs, "A")
t.Show()
t.Center()
t.Scale(500%)
; Wait for a keypress
WaitKey()
```

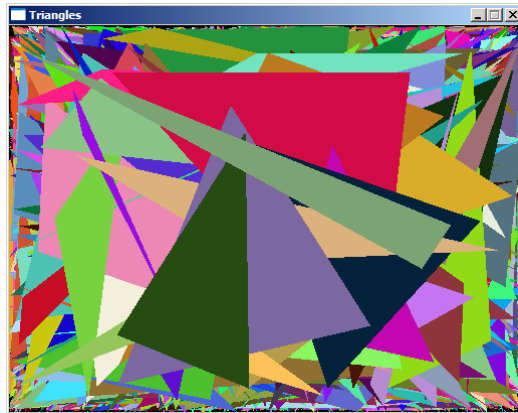
The other method that needs mentioned in CharSet is the Scroll method. The Scroll method in CharSet can scroll a single character in any direction:

```
Scroll(ch, dir, count, wrap)
```

ch is the character in the character set to scroll. dir is the direction (1-up, 2-down, 3-left, 4-right). count is the number of pixels to scroll. wrap indicates whether pixels that scroll off one side reappear on the other side (TRUE) or simply disappear (FALSE). As with the Write method, the character set needs to be refreshed after scroll to see the changes.

## Bitmap Mode

Unlike traditional Windows consoles, that are restricted to text mode only, Liquid Studio allows consoles to have bitmap graphics. Using bitmap graphics you can set (plot) each individual pixel to a specific color, allowing for high resolution graphics and animation not possible using a simple text screen:



## Changing Modes

---

The BitmapMode command is used to switch to bitmap mode:

```
BitmapMode(w, h)           switch to bitmap mode with width, height  
BitmapMode(w, h, dx, dy)  switch to bitmap mode with width, height scaled to dx, dy
```

## Coordinates

---

Similar to text mode, bitmap mode arranges the screen as a grid of pixels (picture elements). The pixel in the upper left corner of the screen is (0, 0). The x coordinate indicates the horizontal position of the pixel, and starts at 0 on the left and increases as it moves to the right. The y coordinate indicates the vertical position of the pixel, and starts at 0 on the top and increases as it moves to the bottom.

## Buffers

---

By default bitmap mode starts using a "single buffer". This means any pixels plotted display immediately. While this is useful, it does not allow for smooth animation, as the end user can see the screen being drawn each frame. For animation, the bitmap mode can be set to use a "double buffer". Here any pixels plotted are written to a buffer in memory that is not visible to the end user, called the workscreen. When the frame is complete, the workscreen is swapped to the visible screen. This allows bitmap mode to have smooth animation.

```
SingleBuffer()           use a single buffer  
DoubleBuffer()          use a double buffer
```

Flip()

when using a double buffer, flip the workscreen to the visible screen

## Reading and Writing Pixels

---

Similar to text mode, the Poke and Peek methods can be used to quickly read and write pixels to the bitmap screen:

Poke(addr, color)                    set the pixel at addr to color  
int:Peek(addr)                        return the color at addr

addr is the "address" to poke or peek to. Address 0 is the upper left corner of the screen. Address 1 is the position of the next pixel to the right of that, and so on down the row. Assuming the bitmap screen is 512 pixels across by 384 pixels down, address 511 is the right-most position of the first row. The next address (512) following the last pixel on a row is the first pixel on the next row down. Address 196607 is the lower right corner of the screen (512 columns \* 383 rows + 511<sup>th</sup> column = 196607).

## Graphics Cursor

As in text mode, the bitmap screen keeps its own graphics cursor. The graphics cursor remembers the last point drawn.

int:GetCursorX()                    return the cursor's x coordinate  
int:GetCursorY()                    return the cursor's y coordinate

## Brushes

Liquid Studio has over a dozen different "brushes" that can be applied when plotting pixels:

TXB_WRITE	the plotted pixel overwrites the pixel in memory
TXB_BLEND	the plotted pixel blends with the pixel in memory using the alpha component
TXB_MIN	write the minimum RGBA of the plotted pixel vs. the pixel already in memory
TXB_MAX	write the maximum RGBA of the plotted pixel vs. the pixel already in memory
TXB_AND	binary AND the pixel in memory with the plotted pixel
TXB_OR	binary OR the pixel in memory with the plotted pixel
TXB_XOR	binary XOR the pixel in memory with the plotted pixel
TXB_ADD	add the plotted pixel to the pixel in memory (clamp components to 255)
TXB_SUB	subtract the plotted pixel from the pixel in memory (clamp components to 0)
TXB_AVG	average the plotted pixel and pixel in memory
TXB_ZERO	only plot the pixel if the pixel in memory's RGB components are zero
TXB_REPLACE	only plot the pixel if the pixel in memory's RGB components are non-zero
TXB_ALPHATEST	only plot the pixel if the pixel's alpha component is non-zero

Whenever a pixel is plotted, it is applied in the current context of the brush. The brush can be used to apply special effects to the pixel, like adding (useful for fading to white, for example) or subtracting (useful for fading to black, for example).

The Brush method is used to set the current brush:

Brush(b)

where b is a valid TXB\_ constant listed above.

A brush can also be used to define what components of a plotted pixel are actually written to memory. Before each pixel is written to memory, it can be binary ANDed with a 32-bit mask:

Brush(b, mask)

The mask stores the 8-bit components of a color in the order ABGR (alpha – blue – green – red). Using the mask you can define that certain components should not be written to memory; the contents of the pixel already in memory remain unchanged. For example:

`Brush (b, 0x00FFFFFF)`                      only the RGB components of a pixel are plotted  
`Brush (b, 0xFF0000FF)`                      only the red and alpha components of a pixel are plotted

The AND mask is a fairly advanced operation, but allows for some impressive special effects.

## Clipping

Pixels can also be "clipped" to a rectangular region on the bitmap screen:

`Clip (x1, y1, x2, y2)`                      define the clipping region as from (x1, y1) to (x2, y2)

Only pixels inside the clipping region are written to memory. Pixels outside of the clipping region are ignored.

## Graphics Primitives

There are several different graphic primitives for drawing to the bitmap screen. All of the graphic primitives use the current ink color (the ink color can be set using the same Ink method used in text mode; likewise, the Paper method can be used to set the bitmap screen's background color), brush, and clipping region when drawing.

### Dots

The Plot and Point methods are used to write a pixel to memory and read it back:

`Plot (x, y)`                                      plot a pixel at (x, y)  
`int:Point (x, y)`                                return the pixel at (x, y)

Plot will plot a pixel at (x, y) and move the graphics cursor to (x, y). Point reads the pixel back (the graphics cursor is not affected).

### Lines

`Line (x1, y1, x2, y2)`

Draw a line from (x1, y1) to (x2, y2). Move the graphics cursor to (x2, y2).

`Line (angle, rad)`

Draw a line from the current graphics cursor to a point, rad (radius) pixels away from the graphics cursor at the angle ang. This form of Line does not affect the graphics cursor. Along with the Circle method, this is useful for drawing pie charts.

`DrawTo (x, y)`

Draw a line from the graphics cursor to (x, y). Move the graphics cursor to (x, y).

`hLine (x1, x2, y)`

Draw a horizontal line from (x1, y) to (x2, y). Move the graphics cursor to (x2, y).

`vLine (x, y1, y2)`

Draw a vertical line from (x, y1) to (x, y2). Move the graphics cursor to (x, y2).

## Triangles

---

Triangle(x1, y1, x2, y2, x3, y3, f)

Draw a triangle with the three vertices (x1, y1), (x2, y2), and (x3, y3). If f is not zero then fill the triangle. Triangle does not affect the graphics cursor.

## Rectangles

---

Rect(x1, y1, x2, y2, f)

Draw a rectangle from (x1, y1) to (x2, y2). If f is not zero then fill the rectangle. Rectangle does not affect the graphics cursor.

## Circles

---

Circle(x, y, rad, f)

Draw a circle at (x, y) with the radius rad. If f is not zero then fill the circle. Move the graphics cursor to (x, y).

## Ellipses

---

Ellipse(x, y, xr, yr, f)

Draw an ellipse at (x, y) with the radius (xr, yr). If f is not zero then fill the ellipse. Move the graphics cursor to (x, y).

## Bezier Curves

---

BezierCurve(x1, y1, x2, y2, x3, y3, x4, y4)

Draw a Bezier curve using the four control points specified: (x1, y1), (x2, y2), (x3, y3), and (x4, y4). BezierCurve does not affect the graphics cursor.

## Flood Fill

---

FloodFill(x, y)

Fill an enclosed area. FloodFill does not affect the graphics cursor.

## Turtle Graphics

Bitmap mode also supports "turtle graphics", a simple and fun way to draw graphics. Turtle graphics were first used in the popular LOGO programming language. With turtle graphics, an imaginary "turtle" exists on the bitmap screen, and can move forward, backward, and turn left or right. As the turtle moves, it draws a line behind it (imagine the turtle with a pen tied to its tail). The pen can be picked up or put down to stop/start drawing as needed. Turtle graphics use the current ink color, brush, and clipping region when drawing.

Home ()

center the turtle

PenDown ()

put the turtle's pen down, so when it moves it leaves a trail

PenUp ()

lift the turtle's pen up, so it does not leave a trail when it moves

GoForward(n)	move the turtle forward n pixels
GoBackward(n)	move the turtle backward n pixels
TurnLeft(d)	rotate the turtle counter clockwise d degrees
TurnRight(d)	rotate the turtle clockwise d degrees
float:GetHeading()	return the current heading, in degrees, of the turtle
SetHeading(n)	set the current turtle heading
float:GetTurtleX()	return the turtle's X coordinate
float:GetTurtleY()	return the turtle's Y coordinate
SetPosition(x, y)	move the turtle to (x, y)

## Images

The LoadImage method can be used to load an image file:

```
LoadImage(filename)
```

The image is automatically scaled to fit the bitmap screen. The filename extension determines what type of image file to load. A wide variety of formats are supported using the FreeImage library:

File Extension	Image Format
BMP	Windows or OS/2 Bitmap
CUT	Dr. Halo
DDS	DirectX Surface
EXR	ILM OpenEXR
G3	Raw fax format CCITT G.3
GIF	Graphics Interchange Format
HDR	High Dynamic Range
ICO	Windows Icon
IFF, LBM	IFF Interleaved Bitmap
J2K, J2C	JPEG-2000 codestream
JNG	JPEG Network Graphics
JP2	JPEG-2000 File Format
JPG, JIF, JPEG, JPE	Independent JPEG Group
KOA	C64 Koala Graphics
MNG	Multiple Network Graphics
PBM	Portable Bitmap (ASCII or binary)
PCD	Kodak PhotoCD
PCX	Zsoft Paintbrush
PGM	Portable Greymap (ASCII or binary)
PNG	Portable Network Graphics
PPM	Portable Pixelmap (ASCII or binary)
PSD	Adobe Photoshop
RAS	Sun Raster Image

SGI	Silicon Graphics SGI image format
TGA, TARGA	Truevision Targa
TIF, TIFF	Tagged Image File Format
WAP, WBMP, WBM	Wireless Bitmap
XBM	X11 Bitmap Format
XPM	X11 Pixmap Format

Likewise, the `SaveImage` method is used to save the bitmap screen to a file:

```
SaveImage(filename)
```

The following formats are supported when saving an image:

File Extension	Image Format
BMP	Windows or OS/2 Bitmap
EXR	ILM OpenEXR
GIF	Graphics Interchange Format
HDR	High Dynamic Range
ICO	Windows Icon
J2K, J2C	JPEG-2000 codestream
JP2	JPEG-2000 File Format
JPG, JIF, JPEG, JPE	Independent JPEG Group
PBM	Portable Bitmap (ASCII or binary)
PGM	Portable Greymap (ASCII or binary)
PNG	Portable Network Graphics
PPM	Portable Pixmap (ASCII or binary)
TGA, TARGA	Truevision Targa
TIF, TIFF	Tagged Image File Format
WAP, WBMP, WBM	Wireless Bitmap
XPM	X11 Pixmap Format

## Scrolling

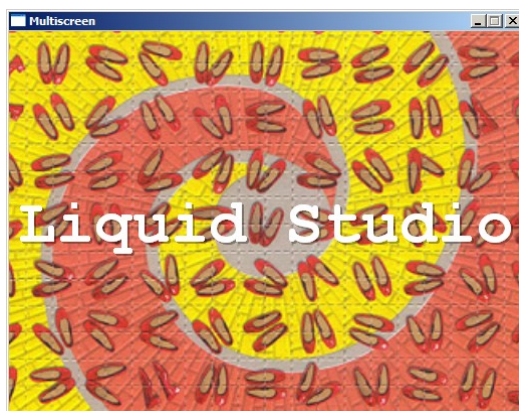
The `Scroll` method allows you to scroll all or part of the text or bitmap screen.

```
Scroll(dir, count, wrap)
Scroll(dir, count, x1, y1, x2, y2, wrap)
```

The first form of `Scroll` is used to scroll the entire screen, the second form is used to scroll a rectangular region of the screen. `dir` is the direction (1-up, 2-down, 3-left, 4-right), `count` is the number of character cells (in text mode) or pixels (in bitmap mode) to scroll, and `wrap` indicates whether character cells (in text mode) or pixels (in bitmap mode) that scroll off one side reappear on the other side (TRUE) or if they simply disappear (FALSE). When using the second form of `Scroll`, only the rectangular region specified by `(x1, y1)` to `(x2, y2)` will be scrolled.

## Multiscreen Mode

Multiscreen mode allows multiple text and bitmap *layers* to be stacked, much like traditional cel animation:



The same text mode methods can be used on a text layer, and the same bitmap mode methods can be used on a bitmap layer. Layers are entities, and as such can be independently moved, tinted, blended, scaled, and rotated.

## Smooth Scrolling

Layers can be moved about in one pixel increments. This, in combination with the Scroll method, can be used to have smooth scrolling. Smooth scrolling is often used to simulate huge playfields that do not fit in the console's visible area.

The steps to produce a smooth scrolling text layer are straightforward:

1. Create a text layer that is two characters larger horizontally and two characters larger vertically than the screen. This will allow you to print characters outside the visible area, and smoothly scroll them into view.
2. Use the Move or MoveRel method to designate the pixel increments to shift the text layer. Cycle through the entire width and/or height of a character to smoothly scroll characters outside the visible area into view.
3. Use the Scroll method (or the LineFeed method, if at the bottom of the text layer to scroll the text layer up) to scroll the entire text layer a single character.
4. Place new characters to scroll outside the visible area, and goto step 2.

It is possible to have smooth scrolling bitmap layers using the same technique shown above for text layer.

### Example: Smooth Scrolling

```
CONSOLE "Smooth scrolling"

TEXTLAYER t1
INT Inx

; switch to multiscreen mode
MultiscreenMode(512, 384)

; create a new text layer, slightly larger than our 64x24 screen
t1 = NEW TEXTLAYER(66, 26)
t1.Show()
t1.Center()

; main loop
DO
```

```

; position for first scroll
t1.Center()
t1.MoveRel(0, 15)

; print text at bottom of text layer
t1.Locate(21, 25)
t1.Ink(GetRandomColor())
t1.Print("*** Liquid Dreams ***")
Flip()

; move text layer up by 1 pixel increments
FOR Inx = 14 DOWNT0 0
    t1.Center()
    t1.MoveRel(0, Inx)
    Flip()
END FOR

; scroll text layer
t1.LineFeed()
LOOP UNTIL GetKey()

```

## Shapes

The Shape entity is used to display basic geometric shapes on the screen. Like any entity, these can be independently moved, tinted, blended, scaled, and rotated using the methods in the Entity class. It uses the following constructor:

Shape(t, x1, y1, x2, y2)      create a new shape

The parameter t is one of the following constants:

SHP_LINE	line
SHP_RECT	rectangle
SHP_RRECT	rounded rectangle
SHP_ELLIPSE	ellipse

Shapes are defined by two sets of 2D coordinates, (x1,y1) and (x2, y2). The line shape draws a line from (x1, y1) to (x2, y2). The rectangle shape draws a rectangle from (x1, y1) to (x2, y2). The rounded rectangle shape draws a rounded rectangle from (x1, y1) to (x2, y2). The ellipse shape draws an ellipse, centered within the rectangle from (x1, y1) to (x2, y2).

Each shape has a fill color and an outline color (the fill color is ignored in the line shape). By default the fill color is 0, or transparent, so only an outline of the shape is drawn. The outline can also have a variable thickness (by default the thickness is one):

SetCoords(x1, y1, x2, y2)	set the (x1, y1) and (x2, y2) coordinates of the shape
int:GetLineWidth()	return the outline width (default is one)
SetLineWidth(n)	set the outline width
int:GetLineColor()	return the outline color
SetLineColor(color)	set the outline color
int:GetFillColor()	return the fill color
SetFillColor(color)	set the fill color

## Sprites

The Sprite entity is a movable 2D bitmap. Like any entity, these can be independently moved, tinted, blended, scaled, and rotated using the methods in the Entity class. Sprites have built-in collision detection, so it is possible to determine when two sprites collide (overlap).

Sprites are created from the Texture object. The Texture object is a two dimensional bitmap. All of the methods that can be used in bitmap mode can also be used on a texture. Texture objects have several different constructors:

<code>Texture(w, h, f)</code>	create a texture with width, height
<code>Texture(filename, f)</code>	create a texture from an image file
<code>Texture(filename, w, h, f)</code>	create a texture from an image file, scale to width, height
<code>Texture(filename, s, f)</code>	create a texture from an image file, scale to size s (preserve aspect ratio)

The `f` parameter in the texture constructor is the texture's filter:

<code>TX_NEAREST</code>	use nearest filter
<code>TX_LINEAR</code>	use bilinear filter (smooth edges)
<code>TX_MIPMAP</code>	use mipmap filter (auto create smaller textures for performance)

Sprites have the following constructors:

<code>Sprite(texture)</code>	create a sprite using texture
<code>Sprite(texture, w, h)</code>	create a sprite using texture, scale to width, height

It should be noted that when a sprite is scaled, as in the second form of the constructor, the texture itself is not modified. Instead the texture is "stretched" to the specified width, height during the sprite's Render component.

It is possible to have many sprites on-screen at once, all built from the same texture.

Sprites have some additional methods:

<code>texture: GetTexture()</code>	return the sprite's texture
<code>SetTexture(t)</code>	set the sprite's texture
<code>int: GetWidth()</code>	return the width of the sprite
<code>int: GetHeight()</code>	return the height of the sprite
<code>Resize(w, h)</code>	resize the sprite to width, height
<code>MoveSpr(angle, speed)</code>	move the sprite during Update in angle at speed
<code>boolean: Hit(sprite)</code>	return TRUE if the sprite is colliding with another sprite (bounding box)
<code>boolean: HitEx(sprite)</code>	return TRUE if the sprite is colliding with another sprite (pixel perfect)

The `MoveSpr` method moves the sprite during the sprite's Update component. As such the sprite must be running (using the `Start` method) so the Update component runs. When a sprite moves off the side of the screen in `MoveSpr` it will automatically reappear on the other side of the screen.

Below is an example of creating a simple texture (a single red filled circle), and then using the texture as a sprite that is displayed, centered, on the console. The sprite then moves at a 90 degree angle (right) at 5 pixels per update:

```
CONSOLE "Sprite"

TEXTURE t
SPRITE sp

t = NEW TEXTURE(256, 256, TX_NEAREST)
t.Ink(RED)
```

```

t.Circle(128, 128, 64, 1)
t.Mask(BLACK, 0)
t.Refresh()

sp = NEW SPRITE(t)
sp.Show()
sp.Center()
sp.MoveSpr(90, 5)
sp.Start()

WaitKey()

```

## Text

The Text entity is used to display text on the console. Like any entity, these can be independently moved, tinted, blended, scaled, and rotated using the methods in the Entity class. Text can be built from any font (not just fixed width fonts or character sets), and moved in one pixel increments (they are not limited to character cells like the console's text mode).

Text is built from a Font object. The Font object has two constructors:

Font(name, ps, b, i, u)	build a font
Font(name, ps)	build a font using the FreeType library

The first form of the constructor uses Windows to build the font. name is the name of the font to use (for example, "Arial"), and must be a valid Windows TrueType font name. ps is the point size. b determines whether the font is bolded (TRUE) or not (FALSE). i determines whether the font is italicized (TRUE) or not (FALSE). u determines whether the font is underlined (TRUE) or not (FALSE).

Alternatively, you can skip Windows and build a font directly from a font file (with the extension TTF) using the FreeType library. Fonts built using the FreeType library are anti-aliased (meaning jagged pixel edges are smoothed out), and generally look "cleaner" than their Windows counterparts. FreeType fonts can not be bolded, italicized, or underlined, however.

Text can be instantiated using the following constructors:

Text(string)	create a text object of string using the default system font
Text(font, string)	create a text object of string using font

If the font is omitted, then the default system font (Arial 10 pt., loaded using FreeType) is used.

## The Event Queue

The console uses an *event queue* to store events in a FIFO (first in first out) fashion. An *event* signifies that something happened. Events include keyboard events, mouse events, and timer events.

The following methods are used to access the event queue:

ClearEventQueue()	discard all the events in the event queue
int:GetEvent()	return the next event in the event queue
int:GetEventType(int)	return the event type of an event
int:GetEventData(int)	return the event data of an event
int:WaitEvent()	wait for an event

The GetEvent method returns the next event in the event queue, or 0 if the queue is empty. The WaitEvent method will pause the console until an event is enqueued, and then returns the event.

GetEventType and GetEventData are passed the event returned from GetEvent or WaitEvent. An event consists of an event type and event data. The following event types are declared as constants:

EVT_KEYDOWN	a key is down
EVT_MOUSEMOVE	the mouse has moved
EVT_MOUSEDOWN	the mouse button is down
EVT_MOUSEUP	the mouse button (that was down) is now up
EVT_CLICKED	the mouse has been clicked (down and up)
EVT_DBLCLICKED	the mouse has been double clicked
EVT_MOUSEWHEEL	the mouse wheel has changed
EVT_TIMER	a timer fired

The KEYDOWN event has the ASCII character of the key as data. The MOUSEDOWN, MOUSEUP, CLICKED, and DBLCLICKED events have the mouse button (1-left, 2-right, 4-middle) as data. The TIMER event has the timer number (0 to 255) as data.

## Keyboard Events

---

There are two additional methods that simplify KEYDOWN events:

int:GetKey()	return the ASCII code of the next KEYDOWN event
int:WaitKey()	wait for a KEYDOWN event and return the ASCII code

The GetKey method will consume (ignore) all events until it finds a KEYDOWN event. The ASCII code the KEYDOWN event is returned, or 0 if there was no KEYDOWN event in the event queue. The WaitKey method clears the event queue and waits until a KEYDOWN event happens, which then has its ASCII code returned.

## Timers

---

A single console can have up to 256 independent timers. A *timer* is used to enqueue a TIMER event at set intervals, ms milliseconds apart:

AddTimer(timer, ms)	add timer with a delay of ms milliseconds
RemoveTimer(timer)	remove timer

The AddTimer method accepts a timer, which must be between 0 and 255, and ms, the number of milliseconds in-between TIMER events. Once the timer is added a TIMER event with the timer # as the event data will get enqueued in the event queue every ms milliseconds. The RemoveTimer method will stop the given timer.

## Example

---

```
CONSOLE
INT e
AddTimer(1, 500)
DO
  e = WaitEvent()
  IF GetEventType(e) = EVT_TIMER AND GetEventData(e) = 1 THEN
    PrintLn("Half a second")
  ELSEIF GetEventType(e) = EVT_KEYDOWN THEN
    EXIT LOOP
  END IF
LOOP
```

## Interrupts

The console has a built-in *interrupt* that occurs every 15 milliseconds (roughly 60 frames per second). Each interrupt the console automatically takes the following actions in order:

- update all running entities
- dispatch all enqueued messages in the message queue
- run the router (delivers any pending delayed and pulsed messages)
- run the sweeper (frees any unused objects)
- run the Windows message pump
- flip the workscreen to the visible screen
- clear the workscreen
- update the plasma color
- update the mouse
- render the console to the workscreen

The Flip method serves two purposes: copy the workscreen to the visible screen (if in double buffered bitmap mode), and wait until the next interrupt before proceeding. While a console waits for the next interrupt control is turned over to the host operating system (Windows), so it can run other processes, etc. (in other words, it lets Windows do whatever it is Windows does outside of the Liquid Virtual Machine). The Flip method is a convenient way to pause the console so the screen can be refreshed to the user.

## Applications

An app (application) uses an event-driven programming model, similar to Windows. In Windows, everything revolves around a *message pump*. The message pump is responsible for processing any messages sent to the window (such as mouse clicks and key presses) and responding to the different messages as needed (redrawing a window, adding text to a field, etc.). Message pumps must constantly run in order for a window to remain responsive.

Applications provide direct access to hardware accelerated 2D and/or 3D graphics, and use double buffered graphics for smooth animation. Each frame the application renders the screen to a workscreen that is not visible. Once the frame is rendered, the workscreen is "flipped", or copied to the visible screen.

Applications also feature a unique "evolution" model. A virtual world is created, and populated with *entities*. Each entity has its own code on how it should appear on-screen, and act or react to its environment. When an application "evolves" all the entities it contains are automatically updated. Additionally, an application can monitor the evolution, and react to changes in the environment as needed.

## Screens

Applications output to a logical *screen*. A screen can be either windowed or fullscreen.

<code>Screen ()</code>	open a new fullscreen the size of the Windows Desktop
<code>Screen (w, h)</code>	open a new fullscreen the size of width, height
<code>Screen (w, h, m)</code>	open a new screen in mode (1=fullscreen, 2=window)
<code>int:GetScreenWidth ()</code>	return the screen width
<code>int:GetScreenHeight ()</code>	return the screen height
<code>CloseScreen ()</code>	close the screen

A screen can not switch between windowed and fullscreen without first closing. When a screen is closed all the fonts and textures are lost and must be reloaded in a new screen.

## Graphics Methods

An application has many graphics methods (commands) similar to a console, except the methods in an application are prefixed with the `gx` extension. The `gx` extension indicates the command is sent directly to the video card, as all graphics in an application are hardware accelerated (whereas graphics in a console are rendered in software).

## Modes

---

A screen has two different modes: 2D and 3D. Each mode has its own set of methods to render graphics; both modes also share a common set of methods that can be used in either.

<code>OrthoMode()</code>	switch to ortho (2D) mode
<code>PerspectiveMode()</code>	switch to perspective (3D) mode

## Vertical Synchronization

---

When vertical synchronization is enabled, the screen can "flip" as fast as the monitor refresh rate, but no faster. When it is disabled, the screen flips as fast as the code can run. While this might seem optimal, it can cause "tearing" in animation, as the screen is updated faster than the monitor can reflect the changes. By default, vertical synchronization is enabled.

<code>gxvSync(boolean)</code>	enable (TRUE) or disable (FALSE) vertical synchronization
-------------------------------	---

## Colors

---

Like consoles, an application has an ink (foreground) color and a paper (background) color.

<code>int:gxGetPaper()</code>	return the paper color
<code>gxPaper(color)</code>	set the paper color
<code>int:gxGetInk()</code>	return the ink color
<code>gxInk(color)</code>	set the ink color
<code>gxInk(r, g, b)</code>	set the ink color to red, green, blue (all floats, alpha = 1)
<code>gxInk(r, g, b, a)</code>	set the ink color to red, green, blue, alpha (all floats)

## Point Size

---

<code>float:gxGetPointSize()</code>	return the point size (default is 1)
<code>gxPointSize(n)</code>	set the point size

## Line Styles

---

<code>float:gxGetLineWidth()</code>	return the line width (default is 1)
<code>gxLineWidth(n)</code>	set the line width
<code>gxLineStipple(f, p)</code>	set the line stipple (f = factor, p = bit pattern)
<code>gxNoLineStipple()</code>	disable the line stipple (default)

## Alpha Testing

---

<code>boolean:gxIsAlphaTest()</code>	return TRUE if alpha testing is enabled
<code>gxAlphaTest(b)</code>	enable (b=TRUE) or disable (b=FALSE) alpha testing
<code>gxAlphaFunc(f, r)</code>	set the alpha function (f = function, ref = reference)

## Blending

---

<code>boolean:gxIsBlend()</code>	return TRUE if blending is enabled
<code>gxBlend(b)</code>	enable (b=TRUE) or disable (b=FALSE) blending
<code>gxBlendFunc(sf, df)</code>	set the blending (sf = source factor, df = destination factor)

## Depth Testing

---

<code>boolean:gxIsDepthTest()</code>	return TRUE if depth testing is enabled
<code>gxDepthTest(b)</code>	enable (b=TRUE) or disable (b=FALSE) depth testing
<code>gxDepthFunc(f)</code>	set the depth function (f = function)

## The Matrix

---

Both 2D and 3D mode use a matrix to calculate points in virtual space:

<code>gxLoadIdentity()</code>	reset the matrix
<code>gxPushMatrix()</code>	push the matrix onto the stack
<code>gxPopMatrix()</code>	pop the matrix from the stack

## Display Lists

---

A display list stores graphics instructions (those that start with `gx*`) in video memory to increase render speed.

<code>int:gxNewList()</code>	begin a new display list
<code>gxEndList()</code>	end the display list
<code>gxCallList(dl)</code>	render a display list

The `gxNewList` method stores all subsequent `gx*` commands in a list in video memory. Any graphics drawn after the `gxNewList` method (but before the `gxEndList` method) are automatically stored in the display list. Once the display list has been defined, it can be rendered by calling the `gxCallList` method with the unique identifier returned by `gxNewList`.

## 2D Graphics

### The Matrix

---

`gxTranslate(x, y)`

Translate the matrix by (x, y).

`gxScale(x, y)`

Scale the matrix by (x, y).

`gxRotate(d)`

Rotate the matrix by d (degrees).

### Nodes

---

`gxNode(n)`

Set the current node to n. When 2D graphics are drawn the current "node" is saved for each pixel. The mouse can then read this node back, allowing the mouse to select 2D objects with pixel perfect precision.

## Points

---

`gxPlot(x, y)`

Plot a dot at (x, y) in the current ink color.

## Lines

---

`gxLine(x1, y1, x2, y2)`

Draw a line from (x1, y1) to (x2, y2).

`gxGouraudLine(x1, y1, x2, y2, c1, c2)`

Draw a Gouraud-shaded line from (x1, y1) to (x2, y2). The point (x1, y1) is rendered in color c1, and the color is gradually blended until it reaches point (x2, y2), which is rendered in color c2.

## Triangles

---

`gxTriangle(x1, y1, x2, y2, x3, y3, fillmode)`

Draw a triangle from (x1, y1) to (x2, y2) to (x3, y3). fillmode is the fill mode, and is 0 for empty or 1 for filled.

## Rectangles

---

`gxRect(x1, y1, x2, y2, fillmode)`

Draw a rectangle from (x1, y1) to (x2, y2). fillmode is the fill mode, and is 0 for empty or 1 for filled.

`gxGouraudRect(x1, y1, x2, y2, c1, c2)`

Draw a Gouraud-shaded rectangle from (x1, y1) to (x2, y2). The point (x1, y1) is rendered in color c1, and the color is gradually blended until it reaches point (x2, y2), which is rendered in color c2.

## Frames

---

`gxFrame(x1, y1, x2, y2, c1, c2, thickness)`

Draw a frame from (x1, y1) to (x2, y2). Color c1 is the color of the left and top of the frame. Color c2 is the color of the right and bottom of the frame. thickness is the thickness of the frame, in pixels.

## Circles

---

`gxCircle(x, y, rad, fillmode)`

Draw a circle at (x, y), with the radius rad. fillmode is the fill mode, and is 0 for empty or 1 for filled.

## Ellipses

---

`gxEllipse(x, y, xr, yr, fillmode)`

Draw an ellipse at (x, y), with a horizontal radius of xr and a vertical radius of yr. fillmode is the fill mode, and is 0 for empty or 1 for filled.

## Bezier Curves

---

`gxBezierCurve(x1, y1, x2, y2, x3, y3, x4, y4)`

Draw a Bezier curve using the four control points specified: (x1, y1), (x2, y2), (x3, y3), and (x4, y4).

## Textures

---

`gxStamp(t, x, y)`

The `gxStamp` method "stamps" texture t at (x, y).

`gxTexRect(t, x1, y1, x2, y2)`

Texture map the rectangle from (x1, y1) to (x2, y2) with the texture t.

`gxTexQuad(t, x1, y1, x2, y2, x3, y3, x4, y4)`

Texture map the quad from (x1, y1) to (x2, y2) to (x3, y3) to (x4, y4) with the texture t.

## Text

---

`gxPrintChar(f, x, y, ch)`

Print the ASCII character ch at (x, y), using the font f.

`gxPrint(f, x, y, s)`

Print the string s at (x, y), using the font f.

`gxAlignPrint(f, x, y, ha, va, s)`

Print the string s at (x, y), using the font f and the specified horizontal and vertical alignment. ha is a constant and specifies the horizontal alignment (HA\_LEFT, HA\_CENTER, or HA\_RIGHT). va is a constant and specifies the vertical alignment (VA\_LEFT, VA\_CENTER, or VA\_RIGHT).

## Clipping

---

`gxClipRect(x1, y1, x2, y2, style, bg)`

Define a clipping region from (x1, y1) to (x2, y2) and push it on the *clipping stack*. Only pixels inside this region are rendered. Multiple regions can be defined, each encapsulated inside the one before it.

There are several different styles that can be applied: 0-none, 1-outline the clipping region with a single black pixel border, 2-outline the clipping region with a thick frame, with white on the left and top and gray on the right and bottom, 3-outline the clipping region with a thick frame, with gray on the left and top and white on the right and bottom. bg is the color to fill the clipping region (use zero if the clipping region is not to be filled).

`gxNoClipRect ()`

Pop the last clipping region off the clipping stack.

## 3D Graphics

### The Matrix

---

`gxTranslate (x, y, z)`

Translate the matrix by (x, y, z).

`gxScale (x, y, z)`

Scale the matrix by (x, y, z).

`gxRotate (d, x, y, z)`

Rotate the matrix by d (degrees) in the x, y, and z planes.

### Lighting

---

`boolean:gxIsLighting ()`

return TRUE if lighting is enabled

`gxLighting (b)`

enable (b=TRUE) or disable (b=FALSE) lighting

### Primitives

---

`gxBegin (m)`

Begin a submission to the graphics card. m is the mode:

0	<code>GX_POINTS</code>	draw submitted vertexes as points
1	<code>GX_LINES</code>	draw submitted vertexes as lines
2	<code>GX_LINE_LOOP</code>	draw submitted vertexes as a line loop
3	<code>GX_LINE_STRIP</code>	draw submitted vertexes as a line strip
4	<code>GX_TRIANGLES</code>	draw submitted vertexes as triangles
5	<code>GX_TRIANGLES_STRIP</code>	draw submitted vertexes as a triangle strip
6	<code>GX_TRIANGLE_FAN</code>	draw submitted vertexes as a triangle fan
7	<code>GX_QUADS</code>	draw submitted vertexes as quads
8	<code>GX_QUAD_STRIP</code>	draw submitted vertexes as a quad strip
9	<code>GX_POLYGON</code>	draw submitted vertexes as a polygon

`gxVertex (x, y, z)`

Submit vertex (x, y, z) to the graphics card.

`gxNormal (x, y, z)`

Submit normal (x, y, z) to the graphics card. Normals are required to light 3D objects correctly.

`gxEdgeFlag (state)`

Enable the edge flag if state is TRUE, disable the edge flag if state is FALSE.

`gxEnd()`

End the current submission to the graphics card.

## Texture Mapping

---

`gxActiveTexture(n)`

Set the active texture to *n*. Liquid Studio supports "multitexturing", a technique of combining multiple textures in a single pass on the graphics card. Most graphics cards support at least 2 textures (meaning that 2 textures can be combined at once), many support 4, 8, or more textures.

`gxTexMap(t)`

Set the active texture map to the texture *t*.

`gxTexCoord(u, v)`

Submit texture coordinate (*u*, *v*) to the graphics card.

`gxMultiTexCoord(n, u, v)`

Submit multitexture coordinate (*u*, *v*) for texture *n* to the graphics card.

## Quadrics

---

`gxCylinder(base, top, height, slice, stacks)`

Draw a 3D cylinder. *base* specifies the radius of the cylinder at *z*=0. *top* specifies the radius of the cylinder at *z*=height. *height* specifies the height of the cylinder. *slices* specifies the number of subdivisions around the *z* axis. *stacks* specifies the number of subdivisions along the *z* axis.

`gxDisk(inner, outer, slices, loops)`

Draw a 3D disk. *inner* specifies the inner radius of the disk (may be 0). *outer* specifies the outer radius of the disk. *slices* specifies the number of subdivisions around the *z* axis. *loops* specifies the number of concentric rings around the origin into which the disk is subdivided.

`gxPartialDisk(inner, outer, slices, loops, start, sweep)`

Draw a 3D arc of a disk. *inner* specifies the inner radius of the partial disk (can be 0). *outer* specifies the outer radius of the partial disk. *slices* specifies the number of subdivisions around the *z* axis. *loops* specifies the number of concentric rings around the origin into which the partial disk is subdivided. *start* specifies the starting angle, in degrees, of the disk portion. *sweep* specifies the sweep angle, in degrees, of the disk portion.

`gxBSphere(radius, slices, stacks)`

Draw a 3D sphere. *radius* specifies the radius of the sphere. *slices* specifies the number of subdivisions around the *z* axis (similar to lines of longitude). *stacks* specifies the number of subdivisions along the *z* axis (similar to lines of latitude).

## Messages

A *message* is an object that encapsulates some sort of communication between objects. Messages can be sent from one object to another. A message is comprised of an int (body) and a string (data).

The following methods are used to parse a message:

```
boolean: IsFrom (object)      return TRUE if the message is from object
boolean: IsTo (object)       return TRUE if the message is to object
int: GetBody ()              return the message body
string: GetData ()           return the message data
int: GetTimeStamp ()         return the message timestamp
```

Who the message is from depends on where the message was sent. If the message was sent from within an entity, the message is considered from the entity. Otherwise, the message is considered from the application.

Several message body constants are automatically declared:

Message Body	Meaning	Message Data
MSG_EXIT	the user has ended the application	none
MSG_KEYDOWN	a key is down	ASCII code of key that is down
MSG_KEYUP	a key (that was down) is now up	ASCII code of key that is up
MSG_KEYPRESSED	a key has been pressed (down and up)	ASCII code of key that is pressed
MSG_MOUSEMOVE	the mouse has moved	none
MSG_MOUSEOVER	the mouse is over an entity (only sent if the mouse button is down or mouse feedback is enabled; see the MouseFeedback method in the API)	node of the entity the mouse pointer is over
MSG_MOUSEDOWN	the mouse button is down	node of the entity the mouse pointer is down
MSG_MOUSEUP	the mouse button (that was down) is now up	node of the entity the mouse pointer is up
MSG_CLICKED	the mouse has been clicked (down and up)	node of the entity the mouse pointer is clicked
MSG_DBLCLICKED	the mouse has been double clicked	node of the entity the mouse pointer is double clicked
MSG_MOUSEWHEEL	the mouse wheel has moved	negative # if mousewheel moved up, positive # if mousewheel moved down

## Unique Message Bodies

You can use the CONST keyword to declare a unique integer constant. This is useful for declaring new message body constants. For example, a checkbox entity might send a message with a CB\_CLICKED message body whenever the checkbox is clicked. If this constant is defined to an absolute value (for example, 10000), this may interfere with other classes that also use the value 10000 to represent a message body. CONST allows you to declare integer constants that are guaranteed to be unique, so messages from different classes don't become "crossed".

## The Message Queue

Each application has its own FIFO (first in first out) *message queue*. Whenever a message is sent, it is enqueued in the application's message queue. The application can check the message queue for a pending message and can either parse it or ignore it. Messages can also be *dispatched*. Dispatched messages are sent to an entity, where it is parsed by executing the entity's Behavior component. If an entity's Behavior component returns TRUE the message is considered processed. If FALSE is returned, the message is immediately dispatched to the entity's parent, up the chain until the message is processed.

Applications can explicitly dequeue and dispatch messages between frames.

<code>ClearMessageQueue ()</code>	discard all the messages in the message queue
<code>boolean:PeekMessageQueue ()</code>	return TRUE if a message is waiting in the queue
<code>message:DequeueMessage ()</code>	dequeue the next message in the queue
<code>DispatchMessage (m)</code>	dispatch message m to the recipient entity's Behavior component

Any messages leftover in the message queue when APP calls the Flip method are automatically dequeued and dispatched.

## Sending Messages

---

The `SendMessage` method is used to send a message to an entity:

```
SendMessage (d, body, data)
```

d is the object to send the message to. body is the message's body and is an int. data is the message's data and is a string.

Messages can be delayed, to be delivered at a later time:

```
int:DelayMessage (d, body, data, delay)
```

d is the entity to send the message to. body is the message's body and is an int. data is the message's data and is a string. delay is the number of milliseconds to wait before sending the message. A handle to the router transaction responsible for delivering the delayed message is returned.

Finally, messages can be pulsed, or delivered over and over, with a set delay between deliveries:

```
int:PulseMessage (d, body, data, delay, count)
```

d is the entity to send the message to. body is the message's body and is an int. data is the message's data and is a string. delay is the number of milliseconds to wait in between pulses. count is the number of times to pulse the message (a zero pulses the message indefinitely). A handle to the router transaction responsible for delivering the pulsed message is returned.

To remove a pending router transaction (for a delayed or pulsed message), use the `RemoveMessage` method:

```
RemoveMessage (int h)
```

h is the handle returned by `DelayMessage` or `PulseMessage`.

## Evolution

An application can *evolve* all the running entities in it by calling the `Evolve` method. The `Evolve` method runs the `Update` component for each running entity. It is typical to evolve right before calling the `Flip` method.

The `Flip` method takes the following actions in order:

- dispatch all enqueued messages in the message queue
- run the router (delivers any pending delayed and pulsed messages)
- run the sweeper (frees any unused objects)
- run the Windows message pump
- flip the workscreen to the visible screen
- clear the workscreen
- update the plasma color
- update the mouse

All graphics in an application are double buffered, meaning the graphics are drawn to a workscreen in memory. When the frame is completed, the entire workscreen is copied over to the visible screen. This creates smooth, flicker-free animation. While the application is "flipping" control is turned over to the host operating system (Windows), so it can run other processes, etc. (in other words, it lets Windows do whatever it is Windows does outside of the Liquid Virtual Machine).

## API Reference

Liquid Studio has an Application Program Interface, or API. The API is a collection of functions that can be called from any program. Functions for timers, math, strings, colors, keyboard, mouse, and the Internet are included.

### Bits

---

#### BitCalc

---

##### Syntax

```
int:bitcalc(int bits, int pos, boolean b)
```

##### Usage

Set the bit at position pos in bits if b is TRUE, otherwise reset the bit at position pos in bits. Return bits.

#### BitGet

---

##### Syntax

```
boolean:bitget(int bits, int pos)
```

##### Usage

Return TRUE if the bit at position pos in bits is on.

##### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if pos is less than 0 or greater than 31.

#### BitReset

---

##### Syntax

```
int:bitreset(int bits, int pos)
```

##### Usage

Reset the bit at position pos in bits, and return bits.

##### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if pos is less than 0 or greater than 31.

#### BitSet

---

##### Syntax

```
int:bitset(int bits, int pos)
```

##### Usage

Set the bit at position pos in bits, and return bits.

##### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if pos is less than 0 or greater than 31.

---

## BitToggle

### Syntax

```
int:bittoggle(int bits, int pos)
```

### Usage

Toggle the bit at position pos in bits, and return bits.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if pos is less than 0 or greater than 31.

---

## Blobs

---

### BinToBlob

#### Syntax

```
string:bintoblob(string d)
```

#### Usage

Convert the binary data in string d to a "blob" and return it. Blobs have the following format:

```
X'####'
```

where #### is a hexadecimal representation of the individual bytes in string d. Blobs are used to represent binary data as a simple ASCII text string, which can then be easily inserted into a database, saved to disk, or transmitted over a network.

---

### BlobToBin

#### Syntax

```
string:blobtobin(string d)
```

#### Usage

Convert the "blob" in string d to binary data and return it. Blobs are used to represent binary data as a simple ASCII text string, which can then be easily retrieved from a database, loaded from disk, or transmitted over a network.

---

## Bytes

---

### HiByte

#### Syntax

```
int:hibyte(int n)
```

#### Usage

Return the high byte (upper 8 bits) of the lowest word in the 32-bit integer n.

---

### LoByte

#### Syntax

```
int:lobyte(int n)
```

#### Usage

Return the low byte (lower 8 bits) of the lowest word in the 32-bit integer `n`.

## Colors

---

### Darken

---

#### Syntax

```
int:darken(int c, float n)
```

#### Usage

Return color `c`, darkened by `n`. `n` should be a float between 0 and 1. For example, `Darken(RED, 50%)` would return the constant `RED` with its intensity cut in half (cut by 50%).

### GetPlasmaColor

---

#### Syntax

```
int:getplasmacolor()
```

#### Usage

Return a plasma color. Liquid Studio has its own evolving plasma color, that constantly changes as a program runs. The color is always opaque (the alpha component is maxed).

### GetRandomColor

---

#### Syntax

```
int:getrandomcolor()
```

#### Usage

Return a random color. The color is always opaque (the alpha component is maxed).

### Lighten

---

#### Syntax

```
int:lighten(int c, float n)
```

#### Usage

Return color `c`, lightened by `n`. `n` should be a float between 0 and 1. For example, `Lighten(RED, 25%)` would return the constant `RED` with its intensity raised 25%.

## RGB

---

#### Syntax

```
int:rgb(int r, int g, int b)
```

#### Usage

Return color with specified red, green, and blue components. Each component should be between 0 (nothing) and 255 (max). Components are clamped to the range  $0 \leq x \leq 255$  if needed.

#### Syntax

```
int:rgb(float r, float g, float b)
```

#### Usage

Return color with specified red, green, and blue components. Each component should be between 0 (nothing) and 1 (max).

Components are clamped to the range  $0 \leq x \leq 1$  if needed.

## RGBA

---

### Syntax

```
int:rgba(int r, int g, int b, int a)
```

### Usage

Return color with specified red, green, blue, and alpha components. Each component should be between 0 (nothing) and 255 (max). Components are clamped to the range  $0 \leq x \leq 255$  if needed.

### Syntax

```
int:rgba(float r, float g, float b, float a)
```

### Usage

Return color with specified red, green, blue, and alpha components. Each component should be between 0 (nothing) and 1 (max). Components are clamped to the range  $0 \leq x \leq 1$  if needed.

## Compression

---

### Compress

---

#### Syntax

```
string:compress(string z)
```

#### Usage

Return string z compressed.

### Decompress

---

#### Syntax

```
string:decompress(string z, int length)
```

#### Usage

Return string z decompressed. Length is the length of the original string before it was compressed.

## Conditionals

---

### IIf

---

#### Syntax

```
int:iif(boolean b, int c, int d)
```

#### Usage

If boolean b is TRUE then the integer c is returned, otherwise the integer d is returned.

#### Syntax

```
float:iif(boolean b, float c, float d)
```

#### Usage

If boolean b is TRUE then the float c is returned, otherwise the float d is returned.

#### Syntax

```
string:iif(boolean b, string c, string d)
```

**Usage**

If boolean b is TRUE then the string c is returned, otherwise the string d is returned.

## Desktop

### GetDesktopHeight

---

**Syntax**

```
int:getdesktopheight()
```

**Usage**

Return the height (in pixels) of the Windows Desktop.

### GetDesktopWidth

---

**Syntax**

```
int:getdesktopwidth()
```

**Usage**

Return the width (in pixels) of the Windows Desktop.

## Fields

### DeleteFields

---

**Syntax**

```
string:deletefields(string z, int pos, int n)
```

**Usage**

Return string with n fields deleted at position pos.

### GetDelimiter

---

**Syntax**

```
int:getdelimiter()
```

**Usage**

Return the current ASCII delimiter. By default the delimiter is 0.

### GetField

---

**Syntax**

```
string:getfield(string z, int pos)
```

**Usage**

Return field in string z at position pos.

### GetFieldCount

---

**Syntax**

```
int:getfieldcount(string z)
```

**Usage**

Return the number of fields in string z.

**GetFields**

---

**Syntax**

```
string:getfields(string z, int pos, int n)
```

**Usage**

Return n fields in string z at position pos.

**InsertEmptyFields**

---

**Syntax**

```
string:insertemptyfields(string z, int pos, int n)
```

**Usage**

Return string z with n empty fields inserted at position pos.

**SetDelimiter**

---

**Syntax**

```
setdelimiter(int delim)
```

**Usage**

Set the ASCII delimiter to delim.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if delim is less than 0 or greater than 255.

**SetField**

---

**Syntax**

```
string:setfield(string z, int pos, string new)
```

**Usage**

Return string z with field at position pos replaced with string new.

**SetFields**

---

**Syntax**

```
string:setfields(string z, string new, int pos)
```

**Usage**

Return string z with fields at position pos replaced with string new.

**Fonts**

---

**GetConsoleCharSet**

---

**Syntax**

`charset:getconsolecharset()`

**Usage**

Return the built-in console character set.

---

**GetSystemFont**

**Syntax**

`font:getssystemfont()`

**Usage**

Return the built-in system font.

---

**GetSystemBoldFont**

**Syntax**

`font:getssystemboldfont()`

**Usage**

Return the built-in system bold font.

---

**Icons**

---

**GetSystemIcons**

**Syntax**

`charset:getssystemicons()`

**Usage**

Return the built-in system icons.

---

**Internet**

---

**GetInternetConnection**

**Syntax**

`int:getinternetconnection()`

**Usage**

Return the Internet connection available:

0	NET_NONE	no Internet connection is available
1	NET_DIALUP	a dial up Internet connection is available
2	NET_BROADBAND	a broadband Internet connect is available
3	NET_PROXY	proxy access to an Internet connection is available

---

**HostAddr**

**Syntax**

`string:hostaddr(string z)`

**Usage**

Return host name *z* translated to its IP address.

---

## HostName

---

### Syntax

```
string:hostname(string z)
```

### Usage

Return IP address *z* translated to its host name.

---

## Keyboard

---

### KeyDown

---

#### Syntax

```
boolean:keydown(int ch)
```

#### Usage

Return TRUE if the key *ch* was pressed since the last time `KeyDown` was called.

#### Exceptions

Throws `VM_ILLEGAL_QUANTITY` if *ch* is less than 0 or greater than 255.

---

### KeyState

---

#### Syntax

```
boolean:keystate(int ch)
```

#### Usage

Return TRUE if the key *ch* is currently pressed, otherwise return FALSE if the key *ch* is currently not pressed.

#### Exceptions

Throws `VM_ILLEGAL_QUANTITY` if *ch* is less than 0 or greater than 255.

---

### KeyUp

---

#### Syntax

```
boolean:keyup(int ch)
```

#### Usage

Return TRUE if the key *ch* was released since the last time `KeyUp` was called.

#### Exceptions

Throws `VM_ILLEGAL_QUANTITY` if *ch* is less than 0 or greater than 255.

---

## Math

---

### Abs

---

#### Syntax

```
int:abs(int n)
```

#### Usage

Return the absolute value of n.

**Syntax**

`float:abs(float n)`

**Usage**

Return the absolute value of n.

---

**ArcCos**

**Syntax**

`float:arccos(float n)`

**Usage**

Return arccosine of float n.

---

**ArcCosH**

**Syntax**

`float:arccosh(float n)`

**Usage**

Return hyperbolic arccosine of float n.

---

**ArcSin**

**Syntax**

`float:arcsin(float n)`

**Usage**

Return arcsine of float n.

---

**ArcSinH**

**Syntax**

`float:arcsinh(float n)`

**Usage**

Return hyperbolic arcsine of float n.

---

**ArcTan**

**Syntax**

`float:arctan(float n)`

**Usage**

Return arctangent of float n.

---

**ArcTanH**

**Syntax**

`float:arctanh(float n)`

**Usage**

Return hyperbolic arctangent of float n.

**Avg**

---

**Syntax**

```
int:avg(int n1, int n2)
```

**Usage**

Return the average of n1 and n2.

**Syntax**

```
float:avg(float n1, float n2)
```

**Usage**

Return the average of n1 and n2.

**Ceil**

---

**Syntax**

```
float:ceil(float n)
```

**Usage**

Return float n rounded to the smallest integer that can contain n.

**Clamp**

---

**Syntax**

```
int:clamp(int n, int low, int high)
```

**Usage**

Clamp the integer n to be in the range  $low \leq n \leq high$ .

**Syntax**

```
float:clamp(float n, float low, float high)
```

**Usage**

Clamp the float n to be in the range  $low \leq n \leq high$ .

**Cos**

---

**Syntax**

```
float:cos(float n)
```

**Usage**

Return cosine of float n.

**Cosh**

---

**Syntax**

```
float:cosh(float n)
```

**Usage**

Return hyperbolic cosine of float n.

## **Deg**

---

### **Syntax**

`float:deg(float n)`

### **Usage**

Convert radians  $n$  to degrees and return.

## **Distance**

---

### **Syntax**

`float:distance(float x1, float y1, float x2, float y2)`

### **Usage**

Return distance between  $(x1,y1)$  and  $(x2,y2)$ .

## **Even**

---

### **Syntax**

`boolean:even(int n)`

### **Usage**

Return TRUE if  $n$  is even.

## **Exp**

---

### **Syntax**

`float:exp(float n)`

### **Usage**

Return  $n$ , where  $e^n$ .

## **Exp2**

---

### **Syntax**

`float:exp2(float n)`

### **Usage**

Return  $n$ , where  $2^n$ .

## **Exp10**

---

### **Syntax**

`float:exp10(float n)`

### **Usage**

Return  $n$ , where  $10^n$ .

## **Fix**

---

### **Syntax**

`float:fix(float n)`

**Usage**

Return float n without the fractional part (no rounding is performed).

**Frac**

---

**Syntax**

```
float:frac(float n)
```

**Usage**

Return the fractional portion of the float n.

**Log**

---

**Syntax**

```
float:log(float n)
```

**Usage**

Return base e (natural) logarithm of float n.

**Log2**

---

**Syntax**

```
float:log2(float n)
```

**Usage**

Return base 2 logarithm of float n.

**Log10**

---

**Syntax**

```
float:log10(float n)
```

**Usage**

Return base 10 logarithm of float n.

**Max**

---

**Syntax**

```
int:max(int n1, int n2)
```

**Usage**

Return whichever number is greater, n1 or n2.

**Syntax**

```
float:max(float n1, float n2)
```

**Usage**

Return whichever number is greater, n1 or n2.

**Min**

---

**Syntax**

```
int:min(int n1, int n2)
```

**Usage**

Return whichever number is lower, n1 or n2.

**Syntax**

```
float:min(float n1, float n2)
```

**Usage**

Return whichever number is lower, n1 or n2.

---

**Odd****Syntax**

```
boolean:odd(int n)
```

**Usage**

Return TRUE if n is odd.

---

**Rad****Syntax**

```
float:rad(float n)
```

**Usage**

Convert degrees n to radians and return.

---

**Round****Syntax**

```
float:round(float n)
```

**Usage**

Return float n rounded to the closest integer.

---

**Sgn****Syntax**

```
int:sgn(int n)
```

**Usage**

Return -1 if n is negative, 0 if n is zero, or 1 if n is positive.

**Syntax**

```
float:sgn(float n)
```

**Usage**

Return -1 if n is negative, 0 if n is zero, or 1 if n is positive.

---

**Sin****Syntax**

```
float:sin(float n)
```

**Usage**

Return sine of float n.

---

## Sinh

### Syntax

```
float:sinh(float n)
```

### Usage

Return hyperbolic sine of float n.

---

## Sqr

### Syntax

```
float:sqr(float n)
```

### Usage

Return square root of float n.

---

## Tan

### Syntax

```
float:tan(float n)
```

### Usage

Return tangent of float n.

---

## Tanh

### Syntax

```
float:tanh(float n)
```

### Usage

Return hyperbolic tangent of float n.

---

## Mouse

---

### ClipMouse

#### Syntax

```
clipmouse(int x1, int y1, int x2, int y2)
```

#### Usage

Clip the mouse pointer to the rectangular region defined by (x1, y1) – (x2, y2).

---

### GetMouseButton

#### Syntax

```
int:getmousebutton()
```

#### Usage

Return the status of the mouse button:

- 0 no mouse button down
- 1 left mouse button down

### **GetMouseClickedX**

---

**Syntax**

`int:getmouseclickedx()`

**Usage**

Return the X coordinate of where the mouse was clicked.

### **GetMouseClickedY**

---

**Syntax**

`int:getmouseclickedy()`

**Usage**

Return the Y coordinate of where the mouse was clicked.

### **GetMouseDesktopX**

---

**Syntax**

`int:getmousedesktopx()`

**Usage**

Return the X coordinate of the mouse, where (0, 0) is the upper left corner of the Windows Desktop.

### **GetMouseDesktopY**

---

**Syntax**

`int:getmousedesktopy()`

**Usage**

Return the Y coordinate of the mouse, where (0, 0) is the upper left corner of the Windows Desktop.

### **GetMousePointer**

---

**Syntax**

`int:getmousepointer()`

**Usage**

Return the current mouse pointer:

- arrow
- cross
- I-Beam
- arrow
- sizing pointer (all directions)
- sizing pointer (NE-SW directions)
- sizing pointer (vertical)
- sizing pointer (NW-SE directions)
- sizing pointer (horizontal)
- up arrow
- hourglass

- no mouse pointer
- arrow with an hourglass

## GetMouseX

---

### Syntax

```
int:getmousex()
```

### Usage

Return the X coordinate of the mouse, where (0, 0) is the upper left corner of the screen.

## GetMouseY

---

### Syntax

```
int:getmousey()
```

### Usage

Return the Y coordinate of the mouse, where (0, 0) is the upper left corner of the screen.

## HideMouse

---

### Syntax

```
hidemouse()
```

### Usage

Hide the mouse pointer.

## MouseFeedback

---

### Syntax

```
mousefeedback(boolean state)
```

### Usage

Enable mouse feedback if state is TRUE, or disable mouse feedback if state is FALSE.

Whenever the mouse button is down Liquid Studio detects what entity and node the mouse pointer is over. To always detect what entity and node the mouse pointer is over, regardless of the mouse button, enable mouse feedback. Be aware that mouse feedback can incur a performance penalty, because it must stall the program until the entire scene is rendered before it can detect what Entity it is over. By default mouse feedback is disabled.

## MoveMouse

---

### Syntax

```
movemouse(int x, int y)
```

### Usage

Move the mouse pointer to the specified (x, y) location.

## NoClipMouse

---

### Syntax

```
noclipmouse()
```

**Usage**

Disable mouse clipping.

**SetMousePointer**

---

**Syntax**

```
setmousepointer(int ptr)
```

**Usage**

Set the mouse pointer:

- 1      arrow
- 2      cross
- 3      I-Beam
- 4      arrow
- 5      sizing pointer (all directions)
- 6      sizing pointer (NE-SW directions)
- 7      sizing pointer (vertical)
- 8      sizing pointer (NW-SE directions)
- 9      sizing pointer (horizontal)
- 10     up arrow
- 11     hourglass
- 12     no mouse pointer
- 13     arrow with an hourglass

**ShowMouse**

---

**Syntax**

```
showmouse ()
```

**Usage**

Show the mouse pointer.

**Parsing**

---

**ParseCSV**

---

**Syntax**

```
string:parsecsv(string z, int n)
```

**Usage**

Return the  $n^{\text{th}}$  field in the comma-separated string  $z$ .

**ParseTag**

---

**Syntax**

```
string:parsetag(string z, string tag)
```

**Usage**

Return the field in the comma-separated string  $z$ , which has the specified tag. A tag is specified in a field by using the format `<tag>=<value>`. For example, `ParseCSV("a=123, b=555, c=Hello world!", "c")` returns "Hello world!"

## Random Numbers

---

### Range

---

**Syntax**

```
int:range(int lo, int hi)
```

**Usage**

Return a random int between lo and hi, inclusive. Negative numbers can be used. lo is automatically swapped with hi if lo is greater than hi.

**Syntax**

```
float:range(float lo, float hi)
```

**Usage**

Return a random float between lo and hi, inclusive. Negative numbers can be used. lo is automatically swapped with hi if lo is greater than hi.

### Rnd

---

**Syntax**

```
float:rnd()
```

**Usage**

Return a random float between 0 and 1.

## Sound

---

### Beep

---

**Syntax**

```
beep()
```

**Usage**

Play the Windows "Default Beep" sound. The "Default Beep" sound can be configured in the Windows' Control Panel.

### PlayWave

---

**Syntax**

```
playwave(string filename)
```

**Usage**

Play a WAV file.

### StopWave

---

**Syntax**

```
stopwave()
```

**Usage**

Stop playing a WAV file.

## Strings

---

### Asc

---

**Syntax**

```
int:asc(string z)
```

**Usage**

Return the first ASCII character in string z. If z is null then -1 is returned.

**Syntax**

```
int:asc(string z, int pos)
```

**Usage**

Return the ASCII character in string z at position pos. If position is less than 0 or greater than the length of the string, -1 is returned.

### Chr

---

**Syntax**

```
string:chr(int n)
```

**Usage**

Return ASCII character n as a string.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if n is less than 0 or greater than 255.

### CSet

---

**Syntax**

```
string:cset(string z, int n)
```

**Usage**

Return string z centered and n characters long.

### EndsWith

---

**Syntax**

```
boolean:endswith(string z, string x)
```

**Usage**

Return TRUE if string z ends with string x.

### ExtractByte

---

**Syntax**

```
int:extractbyte(string z, int pos)
```

**Usage**

Return byte in string z at position pos.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if pos is less than 0 or greater than the length of string z - 1.

## ExtractFloat

---

### Syntax

```
int:extractfloat(string z, int pos)
```

### Usage

Return float in string z at position pos.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if pos is less than 0 or greater than the length of string z - 4.

## ExtractInt

---

### Syntax

```
int:extractint(string z, int pos)
```

### Usage

Return integer in string z at position pos.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if pos is less than 0 or greater than the length of string z - 4.

## Format

---

### Syntax

```
string:format(float n, string z)
```

### Usage

Format float n according to string z and return.

## Instr

---

### Syntax

```
int:instr(string z, string find, int pos)
```

### Usage

Return position of string find inside string z starting at position pos.

## LCase

---

### Syntax

```
string:lowercase(string z)
```

### Usage

Return string z converted to lowercase.

## Left

---

### Syntax

```
string:left(string z, int n)
```

### Usage

Return n characters on left of string z.

---

## Len

### Syntax

```
int:len(string z)
```

### Usage

Return the length of string z.

---

## LSet

### Syntax

```
string:lset(string z, int n)
```

### Usage

Return string z left justified and n characters long.

---

## LTrim

### Syntax

```
string:ltrim(string z)
```

### Usage

Return string z with all spaces trimmed from the left.

---

## MCase

### Syntax

```
string:mcase(string z)
```

### Usage

Return string z converted to mixed case.

---

## Mid

### Syntax

```
string:mid(string z, int n)
```

### Usage

Return string z from position n on.

### Syntax

```
string:mid(string z, int pos, int n)
```

### Usage

Return n characters starting at position pos in string z.

---

## RegExpr

### Syntax

```
int:regexpr(string z, string pattern)
```

## Usage

Return the position inside string `z` where the regular expression pattern matches, or 0 if there was no match.

The following table shows the characters that can be used inside a regular expression pattern:

Character	Meaning
<code>c</code>	Matches any literal character <code>c</code> .
<code>.</code> (period)	Matches any single character.
<code>^</code>	Matches the beginning of the input string.
<code>\$</code>	Matches the end of the input string.
<code>*</code>	Matches zero or more occurrences of the previous character.
<code>?</code>	Matches zero or one occurrence of the previous character.
<code>+</code>	Matches one or more occurrences of the previous character.

## Repeat

---

### Syntax

```
string:repeat(string z, int n)
```

### Usage

Return string `z` repeated `n` times.

## Replace

---

### Syntax

```
string:replace(string z, string old, string new)
```

### Usage

Return string `z` with `old` replaced with `new`.

## Right

---

### Syntax

```
string:right(string z, int n)
```

### Usage

Return `n` characters on right of string `z`.

## RSet

---

### Syntax

```
string:rset(string z, int n)
```

### Usage

Return string `z` right justified and `n` characters long.

## RTrim

---

### Syntax

```
string:rtrim(string z)
```

**Usage**

Return string `z` with all spaces trimmed from the right.

**Space**

---

**Syntax**

```
string:space(int n)
```

**Usage**

Return string with `n` number of spaces.

**StartsWith**

---

**Syntax**

```
boolean:startswith(string z, string x)
```

**Usage**

Return `TRUE` if string `z` starts with string `x`.

**StrDelete**

---

**Syntax**

```
string:strdelete(string z, int pos, int count)
```

**Usage**

Return string `z` with `count` characters deleted at position `pos`.

**StrInsert**

---

**Syntax**

```
string:strinsert(string z, string new, int pos)
```

**Usage**

Return string `new` inserted into string `z` at position `pos`.

**StrRemove**

---

**Syntax**

```
string:strremove(string z, string old)
```

**Usage**

Return string `z` with all occurrences of string `old` removed.

**StrReverse**

---

**Syntax**

```
string:strreverse(string z)
```

**Usage**

Return string `z` reversed from front to back.

## Trim

---

### Syntax

```
string:trim(string z)
```

### Usage

Return string z with all spaces trimmed from the left and right.

## UCase

---

### Syntax

```
string:ucase(string z)
```

### Usage

Return string z converted to uppercase.

## Timers

---

### GetAtomicClock

---

#### Syntax

```
int:getatomicclock()
```

#### Usage

Return the number of milliseconds that have elapsed since the computer was turned on.

### GetDate

---

#### Syntax

```
string:getdate()
```

#### Usage

Return the current date in the format "MM-DD-YYYY".

### GetDay

---

#### Syntax

```
string:getday()
```

#### Usage

Return the current day of the month.

### GetDayOfTheWeek

---

#### Syntax

```
string:getdayoftheweek()
```

#### Usage

Return the current day of the week (Monday, Tuesday, etc.).

### GetElapsedTime

---

#### Syntax

```
int:getelapsedtime(int n)
```

**Usage**

Return the number of milliseconds that have elapsed since n.

**GetFramesPerSecond**

---

**Syntax**

`int:getframespersecond()`

**Usage**

Return the number of frames per second.

**GetMonth**

---

**Syntax**

`string:getmonth()`

**Usage**

Return the current month.

**GetTime**

---

**Syntax**

`string:gettime()`

**Usage**

Return the current time in the format "HH:MM:SS".

**GetYear**

---

**Syntax**

`string:getyear()`

**Usage**

Return the current year.

**IsLeapYear**

---

**Syntax**

`boolean:isleapyear(int y)`

**Usage**

Return TRUE if year y is a leap year, otherwise return FALSE.

**Timer**

---

**Syntax**

`float:timer()`

**Usage**

Return the number of seconds that have elapsed since midnight.

## Words

### HiWord

---

#### Syntax

```
int:hiword(int n)
```

#### Usage

Return the high word (upper 16 bits) of the 32-bit integer n.

### LoWord

---

#### Syntax

```
int:loword(int n)
```

#### Usage

Return the low word (lower 16 bits) of the 32-bit integer n.

## Class Reference

### App

---

An app (short for application) is a process that provides direct 2D/3D hardware accelerated graphics and an "evolution" programming model.

Inherits from PROCESS.

### AutoSort2D

---

#### Syntax

```
autosort2d(int m)
```

#### Usage

Set the 2D automatic sort mode:

0	DS_NONE	do not sort 2D entities
1	DS_UP	sort 2D entities based on depth, from minimum to maximum (default)
2	DS_DOWN	sort 2D entities based on depth, from maximum to minimum

### AutoSort3D

---

#### Syntax

```
autosort3d(int m)
```

#### Usage

Set the 3D automatic sort mode:

0	DS_NONE	do not sort 3D entities
1	DS_UP	sort 3D entities based on Z position, from minimum to maximum (default)
2	DS_DOWN	sort 3D entities based on Z position, from maximum to minimum

## ClearMessageQueue

---

### Syntax

```
clearmessagequeue ()
```

### Usage

Clear the program's message queue of any pending messages.

## CloseScreen

---

### Syntax

```
closescreen ()
```

### Usage

Close the screen. A screen can not switch between windowed and fullscreen without first closing. When a screen is closed all the fonts and textures must be reloaded.

## DelayMessage

---

### Syntax

```
int:delaymessage(object dest, int msg, string data, int delay)
```

### Usage

Send a delayed message to object dest from the program. The delay is specified in milliseconds. Optional data can be passed with the message. A handle to the router transaction responsible for delivering the delayed message is returned.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if delay is less than 100 (1/10<sup>th</sup> of a second) or greater than 86400000 (24 hours).

## DequeueMessage

---

### Syntax

```
message:dequeuemessage ()
```

### Usage

Dequeue the next message in the program's message queue and return it.

## DispatchMessage

---

### Syntax

```
dispatchmessage (message m)
```

### Usage

Dispatch the message to an object. When a message is dispatched, the message is sent to the receiver object's behavior, where it is processed. If the object's behavior returns a non-zero int, the message is passed up to the object's parent.

## Evolve

---

### Syntax

```
evolve (entity)
```

### Usage

Evolve the entity by running its Update component.

**Syntax**

`evolve (entity3d)`

**Usage**

Evolve the 3D entity by running its Update component.

**Syntax**

`evolve ()`

**Usage**

Evolve all running entities and 3D entities by running their Update component.

**Flip**

---

**Syntax**

`flip ()`

**Usage**

Perform the following:

- dispatch all enqueued messages in the message queue
- run the router (delivers any pending delayed and pulsed messages)
- run the sweeper (frees any unused objects)
- run the Windows message pump
- flip the workscreen to the visible screen
- clear the workscreen
- update the plasma color
- update the mouse

All graphics in a program are double buffered, meaning the graphics are drawn to a workscreen in memory. When the frame is completed, the entire workscreen is copied over to the visible screen. This creates smooth, flicker-free animation.

**GetMode**

---

**Syntax**

`int : getmode ()`

**Usage**

Return the current mode (1-ortho, 2-perspective).

**GetScreenHeight**

---

**Syntax**

`int : getscreenheight ()`

**Usage**

Return the height (in pixels) of the screen.

**GetScreenWidth**

---

**Syntax**

`int : getscreenwidth ()`

## Usage

Return the width (in pixels) of the screen.

---

## gxActiveTexture

### Syntax

```
gxactivetexture(int n)
```

### Usage

Set the active texture to n. Liquid Studio supports "multitexturing", a technique of combining multiple textures in a single pass on the graphics card. Most graphics cards support at least 2 textures (meaning that 2 textures can be combined at once), many support 4, 8, or more textures.

This command is ignored if not in perspective mode.

---

## gxAlignPrint

### Syntax

```
gxalignprint(font f, int ha, int va, int x, int y, string msg)
```

### Usage

Print the string msg in font f at (x, y). The string is aligned according to ha (HA\_LEFT, HA\_CENTER, or HA\_RIGHT) and va (VA\_TOP, VA\_CENTER, VA\_RIGHT).

This command is ignored if not in ortho mode.

---

## gxAlphaFunc

### Syntax

```
gxalphafunc(int func, float ref)
```

### Usage

Set the alpha test function. If alpha testing is enabled, the alpha test function specifies the test that each pixel's alpha component must pass in order to be rendered to the screen. func is the appropriate function to use, and ref is the reference value to compare the pixel's alpha component.

0	GX_NEVER	never passes
1	GX_LESS	passes if alpha component is less than ref
2	GX_EQUAL	passes if alpha component is equal to ref
3	GX_LEQUAL	passes if alpha component is less than or equal to ref
4	GX_GREATER	passes if alpha component is greater than ref
5	GX_NOTEQUAL	passes if alpha component is not equal to ref
6	GX_GEQUAL	passes if alpha component is greater than or equal to ref
7	GX_ALWAYS	always passes

By default, alpha testing is enabled and the alpha test function is (GX\_NOTEQUAL, 0), meaning that a pixel is only rendered to the screen if the alpha component is not equal to 0 (transparent).

---

## gxAlphaTest

### Syntax

```
gxalphatest(boolean state)
```

### Usage

If state is TRUE alpha testing is enabled. If state is FALSE alpha testing is disabled.

## **gxBegin**

---

### **Syntax**

```
gxbegin(int m)
```

### **Usage**

Begin a submission to the graphics card. m is the mode:

0	GX_POINTS	draw submitted vertexes as points
1	GX_LINES	draw submitted vertexes as lines
2	GX_LINE_LOOP	draw submitted vertexes as a line loop
3	GX_LINE_STRIP	draw submitted vertexes as a line strip
4	GX_TRIANGLES	draw submitted vertexes as triangles
5	GX_TRIANGLES_STRIP	draw submitted vertexes as a triangle strip
6	GX_TRIANGLE_FAN	draw submitted vertexes as a triangle fan
7	GX_QUADS	draw submitted vertexes as quads
8	GX_QUAD_STRIP	draw submitted vertexes as a quad strip
9	GX_POLYGON	draw submitted vertexes as a polygon

This command is ignored if not in perspective mode.

## **gxBezierCurve**

---

### **Syntax**

```
gxbeziercurve(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4)
```

### **Usage**

Draw a Bezier curve using the four control points specified: (x1, y1), (x2, y2), (x3, y3), and (x4, y4).

## **gxBlend**

---

### **Syntax**

```
gxblend(boolean state)
```

### **Usage**

If state is TRUE blending is enabled. If state is FALSE blending is disabled.

## **gxBlendFunc**

---

### **Syntax**

```
gxblendfunc(int sfactor, int dfactor)
```

### **Usage**

Set the blending test function.

sfactor is the source blending factor, and can be: GX\_ZERO, GX\_ONE, GX\_DST\_COLOR, GX\_ONE\_MINUS\_DST\_COLOR, GX\_SRC\_ALPHA, GX\_ONE\_MINUS\_SRC\_ALPHA, GX\_DST\_ALPHA, GX\_ONE\_MINUS\_DST\_ALPHA, and GX\_SRC\_ALPHA\_SATURATE.

dfactor is the destination blending factor, and can be: GX\_ZERO, GX\_ONE, GX\_SRC\_COLOR, GX\_ONE\_MINUS\_SRC\_COLOR, GX\_SRC\_ALPHA, GX\_ONE\_MINUS\_SRC\_ALPHA, GX\_DST\_ALPHA, and GX\_ONE\_MINUS\_DST\_ALPHA.

By default, blending is enabled and the blending test function is (GX\_SRC\_ALPHA, GX\_ONE\_MINUS\_SRC\_ALPHA). According to the "OpenGL Reference Manual: Third Edition", page 114, this is the best blending test function for implementing transparency.

---

### **gxCallList**

#### **Syntax**

```
gxcalllist(int d)
```

#### **Usage**

Draw display list d.

---

### **gxCenterChar**

#### **Syntax**

```
gxcenterchar(font f, int x, int y, int ch)
```

#### **Usage**

Print character ch in font f, centered at (x, y).

---

### **gxCircle**

#### **Syntax**

```
gxcircle(int x, int y, int radius, int fillmode)
```

#### **Usage**

Draw a circle at (x, y) with the specified radius. The circle is filled if fillmode is not zero.

This command is ignored if not in ortho mode.

---

### **gxClipRect**

#### **Syntax**

```
gxcliprect(int x1, int y1, int x2, int y2, int mode, int c)
```

#### **Usage**

Add a clipping region on the screen from (x1, y1) to (x2, y2). Mode is as follows:

- 0 draw a filled rectangle in the color c
- 1 draw a filled rectangle in the color c, with a thin black outline
- 2 draw a filled rectangle in the color c, with a raised frame
- 3 draw a filled rectangle in the color c, with a lowered frame

All 2D graphics are clipped to the clipping region while it is active.

This command is ignored if not in ortho mode.

---

### **gxCls**

#### **Syntax**

```
gxcls()
```

#### **Usage**

Clear the screen. This method is almost never used, as the screen clears automatically during the Flip method.

## **gxClsDepth**

---

### **Syntax**

```
gxclsdepth()
```

### **Usage**

Clear the depth buffer.

## **gxCylinder**

---

### **Syntax**

```
gxcylinder(float base, float top, float height, int slice, int stacks)
```

### **Usage**

Draw a 3D cylinder. base specifies the radius of the cylinder at  $z = 0$ . top specifies the radius of the cylinder at  $z = \text{height}$ . height specifies the height of the cylinder. slices specifies the number of subdivisions around the z axis. stacks specifies the number of subdivisions along the z axis.

This command is ignored if not in perspective mode.

## **gxDepthFunc**

---

### **Syntax**

```
gxdepthfunc(int func)
```

### **Usage**

Set the depth test function. If depth testing is enabled, the depth test function specifies the test that each pixel's depth value must pass in order to be rendered to the screen. func is the appropriate function to use.

0	GX_NEVER	never passes
1	GX_LESS	passes if incoming depth value is less than stored depth value
2	GX_EQUAL	passes if incoming depth value is equal to stored depth value
3	GX_LEQUAL	passes if incoming depth value is less than or equal to stored depth value
4	GX_GREATER	passes if incoming depth value is greater than stored depth value
5	GX_NOTEQUAL	passes if incoming depth value is not equal to stored depth value
6	GX_GEQUAL	passes if incoming depth value is greater than or equal to stored depth value
7	GX_ALWAYS	always passes

By default, depth testing is enabled and the depth test function is (GX\_LEQUAL) for entities, and (GX\_LESS) for 3D entities.

## **gxDepthTest**

---

### **Syntax**

```
gxdepthtest(boolean state)
```

### **Usage**

If state is TRUE depth testing is enabled. If state is FALSE depth testing is disabled.

## **gxDisk**

---

### **Syntax**

```
gxdisk(float inner, float outer, int slices, int loops)
```

### **Usage**

Draw a 3D disk. inner specifies the inner radius of the disk (may be 0). outer specifies the outer radius of the disk. slices specifies the number of subdivisions around the z axis. loops specifies the number of concentric rings around the origin into which the disk is subdivided.

This command is ignored if not in perspective mode.

## **gxEdgeFlag**

---

### **Syntax**

```
gxedgeflag(boolean state)
```

### **Usage**

Enable the edge flag if state is TRUE, disable the edge flag if state is FALSE.

This command is ignored if not in perspective mode.

## **gxEllipse**

---

### **Syntax**

```
gxellipse(int x, int y, int xradius, int yradius, int fillmode)
```

### **Usage**

Draw an ellipse at (x, y) with the specified x radius and y radius. The ellipse is filled if fillmode is not zero.

This command is ignored if not in ortho mode.

## **gxEnd**

---

### **Syntax**

```
gxend()
```

### **Usage**

End the current submission to the graphics card.

This command is ignored if not in perspective mode.

## **gxEndList**

---

### **Syntax**

```
gxendlist()
```

### **Usage**

End the current display list definition.

## **gxFrame**

---

### **Syntax**

```
gxframe(int x1, int y1, int x2, int y2, int c1, int c2, int thickness)
```

**Usage**

Draw a frame from (x1, y1) to (x2, y2). The color c1 is used for the left and top of the frame. The color c2 is used for the right and bottom of the frame. Thickness is how many pixels thick the frame is.

This command is ignored if not in ortho mode.

**gxGetInk**

---

**Syntax**

```
int:gxgetink()
```

**Usage**

Return the current ink (foreground) color.

**gxGetLineWidth**

---

**Syntax**

```
float:gxgetlinewidth()
```

**Usage**

Return the current line width.

**gxGetPaper**

---

**Syntax**

```
int:gxgetpaper()
```

**Usage**

Return the current paper (background) color.

**gxGetPointSize**

---

**Syntax**

```
float:gxgetpointsize()
```

**Usage**

Return the current point size (default is 1).

**gxGouraudLine**

---

**Syntax**

```
gxgouraudline(int x1, int y1, int x2, int y2, int c1, int c2)
```

**Usage**

Draw a Gouraud-shaded line from (x1, y1) to (x2, y2). The color c1 is used at (x1, y1), and gradually fades to c2 as the line approaches (x2, y2).

This command is ignored if not in ortho mode.

**gxGouraudRect**

---

**Syntax**

```
gxxgouraudrect(int x1, int y1, int x2, int y2, int c1, int c2, int c3, int c4)
```

**Usage**

Draw a Gouraud-shaded rectangle from (x1, y1) to (x2, y2). The color c1 is used at (x1, y1), and gradually fades to c2 as the rectangle approaches (x2, y1), and gradually fades to c3 as the rectangle approaches (x2, y2), and gradually fades to c4 as the rectangle approaches (x1, y2).

This command is ignored if not in ortho mode.

**gxInk**

---

**Syntax**

```
gxink(int c)
```

**Usage**

Set the ink (foreground) color (the color used to draw to the screen).

**Syntax**

```
gxink(float r, float g, float b)
```

**Usage**

Set the ink (foreground) color (the color used to draw to the screen) using the given red, green, and blue components (the alpha component is 1.0, or maxed).

**Syntax**

```
gxink(float r, float g, float b, float a)
```

**Usage**

Set the ink (foreground) color (the color used to draw to the screen) using the given red, green, blue and alpha components.

**gxIsAlphaTest**

---

**Syntax**

```
boolean:gxisalphatest()
```

**Usage**

Return TRUE if alpha testing is enabled, or FALSE if alpha testing is disabled.

**gxIsBlend**

---

**Syntax**

```
boolean:gxisblend()
```

**Usage**

Return TRUE if blending is enabled, or FALSE if blending is disabled.

**gxIsDepthTest**

---

**Syntax**

```
boolean:gxisdepthtest()
```

**Usage**

Return TRUE if depth testing is enabled, or FALSE if depth testing is disabled.

## **gxIsLighting**

---

### **Syntax**

```
boolean:gxIsLighting()
```

### **Usage**

Return TRUE if lighting is enabled, or FALSE if lighting is disabled.

This command is ignored if not in perspective mode.

## **gxLighting**

---

### **Syntax**

```
gxLighting(boolean state)
```

### **Usage**

If state is TRUE lighting is enabled. If state is FALSE lighting is disabled.

This command is ignored if not in perspective mode.

## **gxLine**

---

### **Syntax**

```
gxLine(int x1, int y1, int x2, int y2)
```

### **Usage**

Draw a line from (x1, y1) to (x2, y2).

This command is ignored if not in ortho mode.

## **gxLineStipple**

---

### **Syntax**

```
gxLineStipple(int factor, int pattern)
```

### **Usage**

Enable line stipple mode and set the current line stipple. factor specifies a multiplier for each bit in the line stipple pattern. If factor is 3, for example, each bit in the pattern is used three times before the next bit in the pattern is used. factor must be between 1 and 256 and defaults to 1. pattern specifies a 16-bit integer whose bit pattern determines which fragments of a line will be drawn when the line is rendered. Bit zero is used first; the default pattern is all 1's.

## **gxLineWidth**

---

### **Syntax**

```
gxLineWidth(float w)
```

### **Usage**

Set the current line width (default is 1).

## **gxLoadIdentity**

---

### **Syntax**

```
gxLoadIdentity()
```

**Usage**

Replace the current matrix with the identity matrix.

**gxLookAt**

---

**Syntax**

```
gxlookat(float eyex, float eyey, float eyez, float centerx, float centery, float centerz, float upx, float upy, float upz)
```

**Usage**

Replace the current matrix with a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an "up vector".

This command is ignored if not in perspective mode.

**gxMultiTexCoord**

---

**Syntax**

```
gxmultiplexcoord(int n, float u, float v)
```

**Usage**

Submit multitexture coordinate (u, v) for texture n to the graphics card.

This command is ignored if not in perspective mode.

**gxNewList**

---

**Syntax**

```
int:gxnewlist()
```

**Usage**

Begin a display list definition and return the display list ID. All of the gx commands used between the call to the gxNewList method and the gxEndList method are stored in a display list. Display lists are very efficient; they are stored on the graphics card, reducing traffic on the bus between the CPU and the graphics card. The display list can be rendered by passing the display list ID to the gxCallList method.

**gxNoClipRect**

---

**Syntax**

```
gxnocliprect()
```

**Usage**

Remove the last clipping region from the screen.

This command is ignored if not in ortho mode.

**gxNode**

---

**Syntax**

```
gxnode(int n)
```

**Usage**

Set the current node being drawn. Each entity can have one or more "nodes", which can then be used to tell what part of an entity is clicked if the entity is clicked on.

This command is ignored if not in ortho mode.

---

### **gxNoLineStipple**

#### **Syntax**

```
gxnolinestipple ()
```

#### **Usage**

Disable line stipple mode, so all lines are solid. Only enable line stipple if it is needed, for performance reasons.

---

### **gxNormal**

#### **Syntax**

```
gxnormal(float x, float y, float z)
```

#### **Usage**

Submit normal (x, y, z) to the graphics card. Normals are required to light 3D objects correctly.

This command is ignored if not in perspective mode.

---

### **gxPaper**

#### **Syntax**

```
gxpaper(int c)
```

#### **Usage**

Set the paper (background) color (the color used by the Flip method to clear the screen).

---

### **gxPartialDisk**

#### **Syntax**

```
gxpartialdisk(float inner, float outer, int slices, int loops, float start, float sweep)
```

#### **Usage**

Draw a 3D arc of a disk. inner specifies the inner radius of the partial disk (can be 0). outer specifies the outer radius of the partial disk. slices specifies the number of subdivisions around the z axis. loops specifies the number of concentric rings around the origin into which the partial disk is subdivided. start specifies the starting angle, in degrees, of the disk portion. sweep specifies the sweep angle, in degrees, of the disk portion.

This command is ignored if not in perspective mode.

---

### **gxPlot**

#### **Syntax**

```
gxplot(int x, int y)
```

#### **Usage**

Plot a dot at (x, y).

This command is ignored if not in ortho mode.

## **gxPointSize**

---

### **Syntax**

```
gxpointsize(float s)
```

### **Usage**

Set the current point size.

## **gxPopMatrix**

---

### **Syntax**

```
gxpopmatrix()
```

### **Usage**

Pop the current matrix from the matrix stack.

## **gxPrint**

---

### **Syntax**

```
gxprint(font f, int x, int y, string msg)
```

### **Usage**

Print the string msg in font f starting at (x, y).

This command is ignored if not in ortho mode.

### **Syntax**

```
gxprint(font3d f, float x, float y, string msg)
```

### **Usage**

Print the string msg using the 3D font f starting at (x, y).

This command is ignored if not in perspective mode.

### **Syntax**

```
gxprint(font3d f, texture t, float x, float y, string msg)
```

### **Usage**

Print the string msg using the 3D font f starting at (x, y). The string is textured using the texture t.

This command is ignored if not in perspective mode.

## **gxPrintChar**

---

### **Syntax**

```
gxprintchar(font f, int x, int y, int ch)
```

### **Usage**

Print the character ch in font f at (x, y).

This command is ignored if not in ortho mode.

### **Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 0 or greater than 255.

## **gxPushMatrix**

---

### **Syntax**

`gxpushmatrix()`

### **Usage**

Push the current matrix onto the matrix stack.

## **gxRect**

---

### **Syntax**

`gxrect(int x1, int y1, int x2, int y2, int fillmode)`

### **Usage**

Draw a rectangle from (x1, y1) to (x2, y2). The rectangle is filled if fillmode is not zero.

This command is ignored if not in ortho mode.

## **gxRender**

---

### **Syntax**

`gxRender(entity)`

### **Usage**

Render the entity.

This command is ignored if not in ortho mode.

### **Syntax**

`gxRender(entity3d)`

### **Usage**

Render the 3D entity.

This command is ignored if not in perspective mode.

### **Syntax**

`gxRender()`

### **Usage**

If in ortho mode the 2D entities are sorted according to their depth (if 2D autosort is enabled) and rendered. If in perspective mode the 3D entities are sorted according to their z position (if 3D autosort is enabled) and rendered.

## **gxRotate**

---

### **Syntax**

`gxrotate(float deg)`

### **Usage**

Rotate the current matrix by deg degrees.

This command is ignored if not in ortho mode.

### **Syntax**

`gxrotate(float deg, float x, float y, float z)`

**Usage**

Rotate the current matrix by deg degrees, using the specified X, Y, and Z factors.

This command is ignored if not in perspective mode.

**gxScale**

---

**Syntax**

```
gxscale(float x, float y)
```

**Usage**

Scale the current matrix by (x, y).

This command is ignored if not in ortho mode.

**Syntax**

```
gxscale(float x, float y, float z)
```

**Usage**

Scale the current matrix by (x, y, z).

This command is ignored if not in perspective mode.

**gxSphere**

---

**Syntax**

```
gxsphere(float radius, int slices, int stacks)
```

**Usage**

Draw a 3D sphere. radius specifies the radius of the sphere. slices specifies the number of subdivisions around the z axis (similar to lines of longitude). stacks specifies the number of subdivisions along the z axis (similar to lines of latitude).

This command is ignored if not in perspective mode.

**gxStamp**

---

**Syntax**

```
gxstamp(texture t, int x, int y)
```

**Usage**

Draw the texture t at (x, y).

This command is ignored if not in ortho mode.

**gxTexCoord**

---

**Syntax**

```
gxtexcoord(float u, float v)
```

**Usage**

Submit texture coordinate (u, v) to the graphics card.

This command is ignored if not in perspective mode.

## **gxTexMap**

---

### **Syntax**

```
gxtexturemap(texture t)
```

### **Usage**

Set the active texture map to t.

This command is ignored if not in perspective mode.

## **gxTexQuad**

---

### **Syntax**

```
gxtexturequad(texture t, int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4)
```

### **Usage**

Draw the texture t, stretched to fit the quad from (x1, y1) to (x2, y2) to (x3, y3) to (x4, y4).

This command is ignored if not in ortho mode.

## **gxTexRect**

---

### **Syntax**

```
gxtexturerect(texture t, int x1, int y1, int x2, int y2)
```

### **Usage**

Draw the texture t, stretched to fit the rectangle from (x1, y1) to (x2, y2).

This command is ignored if not in ortho mode.

## **gxTranslate**

---

### **Syntax**

```
gxtranslate(int x, int y)
```

### **Usage**

Translate the current matrix by (x, y).

This command is ignored if not in ortho mode.

### **Syntax**

```
gxtranslate(float x, float y, float z)
```

### **Usage**

Translate the current matrix by (x, y, z).

This command is ignored if not in perspective mode.

## **gxTriangle**

---

### **Syntax**

```
gxtriangle(int x1, int y1, int x2, int y2, int x3, int y3, int fillmode)
```

**Usage**

Draw a triangle from (x1, y1) to (x2, y2) to (x3, y3). The triangle is filled if fillmode is not zero.

This command is ignored if not in ortho mode.

**gxVertex**

---

**Syntax**

```
gxvertex(float x, float y, float z)
```

**Usage**

Submit vertex (x, y, z) to the graphics card.

This command is ignored if not in perspective mode.

**gxvSync**

---

**Syntax**

```
gxvsync(boolean state)
```

**Usage**

Enable vertical synchronization if state is TRUE, disable vertical synchronization if state is FALSE. Vertical synchronization ensures that during the Flip method the workscreen is copied to the visible screen no faster than the monitor refresh rate. If vertical synchronization is disabled, the Flip method will run as fast as possible, without waiting for the monitor to refresh. While this might at first seem optimal, it can result in "tearing", where two or more separate frames are showing at once, because the screen is being refreshed faster than the monitor can display the results.

**OrthoMode**

---

**Syntax**

```
orthomode()
```

**Usage**

Set the program to Ortho mode. Ortho mode is a 2D mode.

This command is ignored if already in ortho mode.

**PeekMessageQueue**

---

**Syntax**

```
boolean:peekmessagequeue()
```

**Usage**

Return TRUE if there is a message pending in the program's message queue.

**PerspectiveMode**

---

**Syntax**

```
perspectivemode()
```

**Usage**

Set the program to Perspective mode. Perspective mode is a 3D mode, with a 24-bit depth buffer.

This command is ignored if already in perspective mode.

## PulseMessage

---

### Syntax

```
int:pulsemessage(object dest, int msg, string data, int delay, int count)
```

### Usage

Send a pulsed message to object dest from the program. The delay is specified in milliseconds. Count indicates the number of times to pulse the message (if count is zero then the message is pulsed until the object is freed). Optional data can be passed with the message. A handle to the router transaction responsible for delivering the pulsed message is returned.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if delay is less than 100 (1/10<sup>th</sup> of a second) or greater than 86400000 (24 hours).

## RemoveMessage

---

### Syntax

```
removemessage(int h)
```

### Usage

Remove a pending router transaction (for a delayed or pulsed message). h is the handle returned by DelayMessage or PulseMessage.

## Screen

---

### Syntax

```
screen()
```

### Usage

Switch to fullscreen mode.

### Exceptions

Throws VM\_DENIED if the operation failed.

### Syntax

```
screen(int w, int h)
```

### Usage

Switch to fullscreen mode with w pixels across and h pixels high.

### Exceptions

Throws VM\_DENIED if the operation failed.

### Syntax

```
screen(int w, int h, int mode)
```

### Usage

Resize the screen to w pixels across and h pixels high, and sets the window's mode (1 – fullscreen, 2 – window).

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if mode is less than 1 or greater than 3.

Throws VM\_DENIED if the operation failed.

## SendMessage

---

### Syntax

`sendMessage(object dest, int msg, string data)`

### Usage

Send a message to object `dest` from the program. Optional data can be passed with the message.

## Sort2D

---

### Syntax

`sort2d()`

### Usage

Sort any 2D entities based on the entity's depth.

## Sort3D

---

### Syntax

`sort3d()`

### Usage

Sort any 3D entities based on the entity's Z position.

## Array

---

From Wikipedia:

Arrays hold a series of objects. Individual elements are accessed by their position in the array. The position is given by an index, which is also called a subscript. The index is a consecutive range of integers, from 0 to  $s$ , where  $s$  is the size of the array.

ACCESS	$O(1)$
FIND	$O(n)$
DELETE	$O(n)$
INSERT	$O(n)$

Inherits from COLLECTION.

## Clear

---

### Syntax

`clear()`

### Usage

Clear the array.

### Exceptions

Throws `VM_NOT_DIMENSIONED` if the array has not been dimensioned yet.

## Contains

---

### Syntax

```
boolean:contains(item i)
```

### Usage

Return TRUE if item *i* is in the array, otherwise returns FALSE.

### Exceptions

Throws VM\_NOT\_DIMENSIONED if the array has not been dimensioned yet.

## Delete

---

### Syntax

```
delete(int inx)
```

### Usage

Delete the item at index *inx*.

### Exceptions

Throws VM\_NOT\_DIMENSIONED if the array has not been dimensioned yet.

Throws VM\_BAD\_SUBSCRIPT if the index *inx* is less than 0 or greater than *s*, where *s* is the size of the array.

## Dim

---

### Syntax

```
dim(int s)
```

### Usage

Dimension the array to size *s*.

### Exceptions

Throws VM\_DENIED if the array has already been dimensioned.

Throws VM\_ILLEGAL\_QUANTITY if *s* is less than 0.

Throws VM\_OUT\_OF\_MEMORY if the array is too big to fit into memory.

## Exchange

---

### Syntax

```
exchange(int inx1, int inx2)
```

### Usage

Exchange the item at index *inx1* with the item at index *inx2*.

### Exceptions

Throws VM\_NOT\_DIMENSIONED if the array has not been dimensioned yet.

Throws VM\_BAD\_SUBSCRIPT if the index *inx1* or *inx2* is less than 0 or greater than *s*, where *s* is the size of the array.

## Fill

---

### Syntax

```
fill(item i)
```

### Usage

Fill the array with item *i*.

**Exceptions**

Throws VM\_NOT\_DIMENSIONED if the array has not been dimensioned yet.

**Find**

---

**Syntax**

```
int:find(item i)
```

**Usage**

Return the index where item i resides, or -1 if the item was not found.

**Exceptions**

Throws VM\_NOT\_DIMENSIONED if the array has not been dimensioned yet.

**Insert**

---

**Syntax**

```
insert(int inx, item i)
```

**Usage**

Insert item i into the array at index inx.

**Exceptions**

Throws VM\_NOT\_DIMENSIONED if the array has not been dimensioned yet.

Throws VM\_BAD\_SUBSCRIPT if the index inx is less than 0 or greater than s, where s is the size of the array.

**LBound**

---

**Syntax**

```
int:lbound()
```

**Usage**

Return the lowest index the array can have (always returns zero).

**ReDim**

---

**Syntax**

```
redim(int s)
```

**Usage**

Redimension an already dimensioned array to size s.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if s is less than 0.

Throws VM\_OUT\_OF\_MEMORY if the array is too big to fit into memory.

**ReDimPreserve**

---

**Syntax**

```
redimpreserve(int s)
```

**Usage**

Redimension an already dimensioned array to size s while preserving the original contents of the array.

**Exceptions**

Throws `VM_ILLEGAL_QUANTITY` if `s` is less than 0.

Throws `VM_OUT_OF_MEMORY` if the array is too big to fit into memory.

**Reverse**

---

**Syntax**

```
reverse(int inx1, int inx2)
```

**Usage**

Reverse the items in the array from index `inx1` to index `inx2`.

**Exceptions**

Throws `VM_NOT_DIMENSIONED` if the array has not been dimensioned yet.

Throws `VM_BAD_SUBSCRIPT` if the index `inx1` or `inx2` is less than 0 or greater than `s`, where `s` is the size of the array.

**Shuffle**

---

**Syntax**

```
shuffle(int inx1, int inx2)
```

**Usage**

Shuffle the items in the array from index `inx1` to index `inx2`.

**Exceptions**

Throws `VM_NOT_DIMENSIONED` if the array has not been dimensioned yet.

Throws `VM_BAD_SUBSCRIPT` if the index `inx1` or `inx2` is less than 0 or greater than `s`, where `s` is the size of the array.

**Sort**

---

**Syntax**

```
sort(int inx1, int inx2)
```

**Usage**

Sort the items in the array from index `inx1` to index `inx2`.

**Exceptions**

Throws `VM_NOT_DIMENSIONED` if the array has not been dimensioned yet.

Throws `VM_BAD_SUBSCRIPT` if the index `inx1` or `inx2` is less than 0 or greater than `s`, where `s` is the size of the array.

**UBound**

---

**Syntax**

```
int:ubound()
```

**Usage**

Return the highest index the array can have.

**UnDim**

---

**Syntax**

```
undim()
```

## Usage

Undimension an array.

## Audio

---

Audio is an audio object, such as a sound effect or music.

## Constructor

---

### Syntax

```
audio(string fn)
```

### Usage

Load audio data into memory from disk. The filename is passed in `fn`. The following file types are supported:

- WAV            Wave file
- MID            MIDI file
- MP3            MPEG layer 3 file

## Cast

---

### Syntax

```
string:cast()
```

### Usage

Cast the audio object to a string. The name of the audio is returned.

## GetLength

---

### Syntax

```
int:getlength()
```

### Usage

Return the length (in milliseconds) of the audio object.

## GetPosition

---

### Syntax

```
int:getposition()
```

### Usage

Return the current position (in milliseconds) in the audio object.

## IsPaused

---

### Syntax

```
boolean:ispaused()
```

### Usage

Return TRUE if the audio object is currently paused, otherwise return FALSE.

## IsPlaying

---

### Syntax

`boolean:isplaying()`

### Usage

Return TRUE if the audio object is currently playing, otherwise return FALSE.

## IsStopped

---

### Syntax

`boolean:isstopped()`

### Usage

Return TRUE if the audio object is currently stopped, otherwise return FALSE.

## Pause

---

### Syntax

`pause()`

### Usage

Pause the audio object.

## Play

---

### Syntax

`play()`

### Usage

Play the audio object. The audio object will play in the background and the program will continue.

## Seek

---

### Usage

`seek(int p)`

### Usage

Seek to position p (in milliseconds) in the audio object.

## Stop

---

### Syntax

`stop()`

### Usage

Stop the audio object.

## BinarySearchTree

Inherits from COLLECTION.

## Clear

---

### Syntax

`clear()`

### Usage

Clear the binary search tree.

## Delete

---

### Syntax

`delete(item i)`

### Usage

Delete item *i* from the binary search tree.

### Exceptions

Throws `VM_DENIED` if the node to delete has two children.

Throws `VM_ITEM_NOT_FOUND` if the item to delete could not be found.

## Find

---

### Syntax

`boolean:find(item i)`

### Usage

Return `TRUE` if item *i* exists in the binary search tree, otherwise return `FALSE`.

## Insert

---

### Syntax

`insert(item i)`

### Usage

Insert item *i* into the binary search tree.

### Exceptions

Throws `VM_DUPLICATE_ITEM` if the item is already in the binary search tree.

## BinaryTree

A binary tree is a data structure where each node can have zero, one, or two children.

Inherits from `COLLECTION`.

## Clear

---

### Syntax

`clear()`

### Usage

Clear the binary tree.

## Delete

---

### Syntax

```
delete(int node)
```

### Usage

Delete node from the binary tree.

### Exceptions

Throws VM\_BAD\_HANDLE if the node is invalid.

Throws VM\_DENIED if the node to delete has two children.

## InsertAt

---

### Syntax

```
int:insertat(int parent, int childtype, item i)
```

### Usage

Insert item *i* into the binary tree, under the parent node. Child type must be BT\_LEFT or BT\_RIGHT. Set parent to zero and childtype to BT\_LEFT to insert the root node. Return the handle of the node item *i* was inserted into.

### Exceptions

Throws VM\_BAD\_HANDLE if the node is invalid.

Throws VM\_DENIED if the parent already has a node as childtype.

## Root

---

### Syntax

```
int:root()
```

### Usage

Returns the handle of the root node.

## BitArray

Provides a fast, efficient array of bits.

Inherits from OBJECT.

## Constructor

---

### Syntax

```
bitarray(int s)
```

### Usage

Allocate memory for the bitarray. *s* is the number of bits the bitarray should hold.

## Cast

---

### Syntax

```
int:cast()
```

### Usage

Cast the bitarray object to an int. The lowest 32-bits of the bitarray are returned.

**Exceptions**

Throws VM\_DENIED if the bitarray is smaller than 32 bits.

**AndArray**

---

**Syntax**

```
andarray(bitarray ba)
```

**Usage**

AND bitarray ba with the current bitarray.

**Exceptions**

Throws VM\_DENIED if the size of bitarray ba does not match the size of the current bitarray.

**CalcBit**

---

**Syntax**

```
calcbit(int b, int s)
```

**Usage**

Set bit b if s is not zero, otherwise resets bit b.

**Exceptions**

Throws VM\_BAD\_SUBSCRIPT if b is less than 0 or greater than the number of bits allocated in the bitarray.

**FirstFalse**

---

**Syntax**

```
int:firstfalse()
```

**Usage**

Return the index of the first false bit, or -1 if not found.

**FirstTrue**

---

**Syntax**

```
int:firsttrue()
```

**Usage**

Return the index of the first true bit, or -1 if not found.

**GetBit**

---

**Syntax**

```
boolean:getbit(int b)
```

**Usage**

Return TRUE if bit b is set, otherwise returns FALSE.

**Exceptions**

Throws VM\_BAD\_SUBSCRIPT if b is less than 0 or greater than the number of bits allocated in the bitarray.

## **LastFalse**

---

### **Syntax**

`int:lastfalse()`

### **Usage**

Return the index of the last false bit, or -1 if not found.

## **LastTrue**

---

### **Syntax**

`int:lasttrue()`

### **Usage**

Return the index of the last true bit, or -1 if not found.

## **NextFalse**

---

### **Syntax**

`int:nextfalse(int i)`

### **Usage**

Return the index of the next false bit from the given bit, or -1 if not found.

## **NextTrue**

---

### **Syntax**

`int:nexttrue(int i)`

### **Usage**

Return the index of the next true bit from the given bit, or -1 if not found.

## **OrArray**

---

### **Syntax**

`orarray(bitarray ba)`

### **Usage**

OR bitarray ba with the current bitarray.

### **Exceptions**

Throws VM\_DENIED if the size of bitarray ba does not match the size of the current bitarray.

## **PrevFalse**

---

### **Syntax**

`int:prevfalse(int i)`

### **Usage**

Return the index of the previous false bit from the given bit, or -1 if not found.

## **PrevTrue**

---

### **Syntax**

```
int:prevtrue(int i)
```

**Usage**

Return the index of the previous true bit from the given bit, or -1 if not found.

**ResetAll**

---

**Syntax**

```
resetall()
```

**Usage**

Reset all of the bits in the bitarray.

**ResetBit**

---

**Syntax**

```
resetbit(int b)
```

**Usage**

Reset bit b.

**Exceptions**

Throws VM\_BAD\_SUBSCRIPT if b is less than 0 or greater than the number of bits allocated in the bitarray.

**SetAll**

---

**Syntax**

```
setall()
```

**Usage**

Set all of the bits in the bitarray.

**SetBit**

---

**Syntax**

```
setbit(int b)
```

**Usage**

Set bit b.

**Exceptions**

Throws VM\_BAD\_SUBSCRIPT if b is less than 0 or greater than the number of bits allocated in the bitarray.

**ToggleAll**

---

**Syntax**

```
toggleall()
```

**Usage**

Toggle all of the bits in the bitarray.

## ToggleBit

---

### Syntax

```
togglebit(int b)
```

### Usage

Toggle bit b.

### Exceptions

Throws VM\_BAD\_SUBSCRIPT if b is less than 0 or greater than the number of bits allocated in the bitarray.

## XorArray

---

### Syntax

```
xorarray(bitarray ba)
```

### Usage

XOR bitarray ba with the current bitarray.

### Exceptions

Throws VM\_DENIED if the size of bitarray ba does not match the size of the current bitarray.

## BitmapLayer

A bitmap layer is a layer that encapsulates a texture.

Inherits from LAYER. Can be rendered in console in multiscreen mode.

## Constructor

---

### Syntax

```
bitmaplayer(int width, int height, int filter)
```

### Usage

Create a new bitmap layer. width specifies the width (in pixels), and height specifies the height (in pixels). filter is one of the following:

1	TX_NEAREST	pixels are scaled
2	TX_LINEAR	pixels are scaled and averaged with nearby pixels, smoothing the graphics
3	TX_MIPMAP	same as TX_LINEAR, but creates multiple textures of varying sizes

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if width is less than 1 or greater than 1024, height is less than 1 or greater than 1024, or filter is not one of the constants above.

Throws VM\_DENIED if the operation failed.

### Syntax

```
bitmaplayer(string filename, int filter)
```

### Usage

Load an image from disk. filename specifies the file on disk to load.

### Syntax

```
bitmaplayer(string filename, int width, int height, int filter)
```

**Usage**

Load an image from disk, and resize it to the specified width and height.

**Syntax**

```
bitmaplayer(string filename, int size, int filter)
```

**Usage**

Load an image from disk, and resize it to fit inside a square bitmap layer. The square bitmap layer is size pixels across and size pixels high. The aspect ratio of the image is preserved. This form of the constructor is useful for making "thumbnails" of images.

**Syntax**

```
bitmaplayer(texture t)
```

**Usage**

Create a new bitmap layer, by encapsulating texture t.

**AutoRefresh**

---

**Syntax**

```
autorefresh(boolean state)
```

**Usage**

If state is TRUE, enable auto refresh. If state is FALSE, disable auto refresh.

When auto refresh is enabled, any time the bitmap layer is rendered on screen it will automatically refresh (this is similar to single buffering). If auto refresh is disabled, the bitmap layer will not be refreshed until an explicit call to the Refresh method (this is similar to double buffering).

When rendering to a bitmap layer, it is often faster to disable auto refresh, draw to the bitmap layer as much as needed, and then explicitly call the Refresh method to copy the bitmap layer into video memory.

**BezierCurve**

---

**Syntax**

```
beziercurve(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4)
```

**Usage**

Draw a Bezier curve using the four control points specified: (x1, y1), (x2, y2), (x3, y3), and (x4, y4). The brush and clipping are used when drawing pixels.

**Brush**

---

**Syntax**

```
brush(int op)
```

**Usage**

Set the brush operation to op.

**Syntax**

```
brush(int op, int mask)
```

**Usage**

Set the brush operation to op and the brush mask to mask.

## Circle

---

### Syntax

```
circle(int x, int y, int r, int fillmode)
```

### Usage

Draw a circle at (x, y) using the current ink color with the radius r. Fillmode is 0 if the circle is not filled, or 1 if the circle is filled. The brush and clipping method are used when drawing pixels.

## Clip

---

### Syntax

```
clip(int x1, int y1, int x2, int y2)
```

### Usage

Clip the bitmap layer to the region from (x1, y1) to (x2, y2).

## Cls

---

### Syntax

```
cls()
```

### Usage

Clear the bitmap layer.

## Copy

---

### Syntax

```
copy(texture t)
```

### Usage

Copy texture t to the bitmap layer. If t is larger than the bitmap layer, t is clipped to the bitmap layer. The brush is used and clipping is ignored.

## DrawText

---

### Syntax

```
drawto(charset cs, int x, int y, string s)
```

### Usage

Print the string s using character set cs at (x, y) using the current ink color. The brush and clipping method are used when drawing pixels.

## DrawTo

---

### Syntax

```
drawto(int x, int y)
```

### Usage

Draw a line from the current graphics cursor to location (x, y) using the current ink color. The brush and clipping method are used when drawing pixels.

## Ellipse

---

### Syntax

```
ellipse(int x, int y, int xr, int yr, int fillmode)
```

### Usage

Draw an ellipse at (x, y) using the current ink color with a horizontal radius of xr and a vertical radius of yr. Fillmode is 0 if the ellipse is not filled, or 1 if the ellipse is filled. The brush and clipping method are used when drawing pixels.

## Fill

---

### Syntax

```
fill(int color)
```

### Usage

Fill the bitmap layer with a color. The brush is used, but clipping is ignored.

## Flip180

---

### Syntax

```
flip180()
```

### Usage

Rotate the bitmap layer 180 degrees.

## FloodFill

---

### Syntax

```
floodfill(int x, int y)
```

### Usage

Flood fill an area at (x, y) using the current ink color. The current pixel at (x, y) is used as the "old color". All pixels with the old color to the left, right, top, or bottom of (x, y) are colored, and this repeats with those pixels. This is often used to fill in an irregular shape. The shape must be enclosed, or the color will "spill out" of the shape. The brush and clipping method are used when drawing pixels.

## GetBrushMask

---

### Syntax

```
int:getbrushmask()
```

### Usage

Return the current brush mask.

## GetBrushOp

---

### Syntax

```
int:getbrushop()
```

### Usage

Return the current brush operation.

## GetCursorX

---

### Syntax

`int:getcursorx()`

### Usage

Return the cursor's X location.

## GetCursorY

---

### Syntax

`int:getcursory()`

### Usage

Return the cursor's Y location.

## GetFilter

---

### Syntax

`int:getfilter()`

### Usage

Return the current graphics filter:

1	TX_NEAREST	pixels are scaled
2	TX_LINEAR	pixels are scaled and averaged with nearby pixels, smoothing the graphics
3	TX_MIPMAP	same as TX_LINEAR, but creates multiple textures of varying sizes

## GetHeading

---

### Syntax

`float:getheading()`

### Usage

Return the turtle's heading, in degrees (0 to 359).

## GetHeight

---

### Syntax

`int:getheight()`

### Usage

Return the height (in pixels) of the bitmap layer.

## GetInk

---

### Syntax

`int:getink()`

### Usage

Return the ink color.

## GetTurtleX

---

### Syntax

```
float:getturtleX()
```

### Usage

Return the turtle's X coordinate.

## GetTurtleY

---

### Syntax

```
float:getturtleY()
```

### Usage

Return the turtle's Y coordinate.

## GetWidth

---

### Syntax

```
int:getwidth()
```

### Usage

Return the width (in pixels) of the bitmap layer.

## GoBackward

---

### Syntax

```
gobackward(float units)
```

### Usage

Move the turtle backward the specified number of units.

## GoForward

---

### Syntax

```
goforward(float units)
```

### Usage

Move the turtle forward the specified number of units.

## GrayScale

---

### Syntax

```
grayscale()
```

### Usage

Filter all of the pixels in the bitmap layer to a gray scale, effectively making color pictures black and white.

## hLine

---

### Syntax

```
hline(int x1, int x2, int y)
```

### Usage

Draw a horizontal line from (x1, y) to (x2, y) using the current ink color. The brush and clipping are used when drawing pixels.

## Home

---

### Syntax

```
home()
```

### Usage

Return the turtle to the center of the bitmap layer.

## Ink

---

### Syntax

```
ink(int color)
```

### Usage

Set the ink (foreground) color. The ink color is used when drawing pixels.

## Line

---

### Syntax

```
line(int x1, int y1, int x2, int y2)
```

### Usage

Draw a line from (x1, y1) to (x2, y2) using the current ink color. The brush and clipping are used when drawing pixels.

### Syntax

```
line(float deg, float n)
```

### Usage

Draw a line from the current graphics cursor in the direction deg (degrees), with the length n. The brush and clipping are used when drawing pixels.

## LoadImage

---

### Syntax

```
loadimage(string f)
```

### Usage

Load the image in file f. The image is automatically scaled to fit the bitmap layer.

### Exceptions

Throws VM\_DENIED if the operation failed.

Throws VM\_FILE\_NOT\_FOUND if the file f does not exist.

Throws VM\_BAD\_FILE\_FORMAT if the file f is corrupted or not a supported image type.

## Mask

---

### Syntax

```
mask(int color, int alpha)
```

### Usage

Scan the entire bitmap layer, and wherever the RGB components of color are present, set the alpha component to alpha. This

is often used to load a 24-bit BMP (which treats all pixels as opaque), and making all pixels of a set color (for example, black) transparent.

---

### **MirrorX**

#### **Syntax**

```
mirrorx()
```

#### **Usage**

Mirror the bitmap layer along the X axis.

---

### **MirrorY**

#### **Syntax**

```
mirrory()
```

#### **Usage**

Mirror the bitmap layer along the Y axis.

---

### **Peek**

#### **Syntax**

```
int:peek(int d)
```

#### **Usage**

Return the pixel stored at position d. Pixels are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the bitmap layer.

#### **Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if d is less than zero or greater than or equal to the number of pixels in the bitmap layer.

---

### **PenDown**

#### **Syntax**

```
pendown()
```

#### **Usage**

Place the turtle's pen down.

---

### **PenUp**

#### **Syntax**

```
penup()
```

#### **Usage**

Place the turtle's pen up.

---

### **Plot**

#### **Syntax**

```
plot(int x, int y)
```

### Usage

Plot a pixel at location (x, y) using the current ink color. The brush and clipping are used when drawing pixels.

### Point

---

#### Syntax

```
int:point(int x, int y)
```

#### Usage

Return the pixel at location (x, y).

### Poke

---

#### Syntax

```
poke(int d, int c)
```

#### Usage

Set the pixel stored at position d to color c. Pixels are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the bitmap layer.

#### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if d is less than zero or greater than or equal to the number of pixels in the bitmap layer.

### Rect

---

#### Syntax

```
rect(int x1, int y1, int x2, int y2, int fillmode)
```

#### Usage

Draw a rectangle from (x1, y1) to (x2, y2) using the current ink color. Fillmode is 0 if the rectangle is not filled, or 1 if the rectangle is filled. The brush and clipping are used when drawing pixels.

### Refresh

---

#### Syntax

```
refresh()
```

#### Usage

"Refresh" the bitmap layer, meaning that the bitmap layer in memory is copied into video memory.

### Rescale

---

#### Syntax

```
rescale(int w, int h, int filter)
```

#### Usage

Resize the bitmap layer to the specified width and height. Filter determines the algorithm to use when resizing:

0	TXF_BOX	Box, pulse, Fourier window, 1st order (constant) b-spline
1	TXF_BICUBIC	Mitchell & Netravali's two-param cubic filter
2	TXF_BILINEAR	Bilinear filter
3	TXF_BSPLINE	4th order (cubic) b-spline

4	TXF_CATMULLROM	Catmull-Rom spline, Overhauser spline
5	TXF_LANCZOS3	Lanczos3 filter

---

## SaveImage

### Syntax

`saveimage(string f)`

### Usage

Save the bitmap layer to file f.

### Exceptions

Throws VM\_DENIED if the operation failed.

Throws VM\_BAD\_FILE\_FORMAT if the file f is not a supported image type.

---

## Scroll

### Syntax

`scroll(int dir, int count, boolean wrap)`

### Usage

Scroll the entire bitmap layer in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of pixels in count. If wrap is TRUE then pixels that scroll off the bitmap layer appear on the other side, if wrap is FALSE then they do not.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

### Syntax

`scroll(int dir, int count, int x1, int y1, int x2, int y2)`

### Usage

Scroll the region from (x1, y1) to (x2, y2) in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of pixels in count. If wrap is TRUE then pixels that scroll off the region appear on the other side, if wrap is FALSE then they do not.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

---

## Sepia

### Syntax

`sepia()`

### Usage

Filter all of the pixels in the bitmap layer to a sepia tone (similar to an 1800's photograph).

---

## SetFilter

### Syntax

`setfilter(int f)`

### Usage

Set the graphics filter.

1	TX_NEAREST	pixels are scaled
---	------------	-------------------

2	TX_LINEAR	pixels are scaled and averaged with nearby pixels, smoothing the graphics
3	TX_MIPMAP	same as TX_LINEAR, but creates multiple textures of varying sizes

---

## SetHeading

### Syntax

`setheading(float h)`

### Usage

Set the turtle's heading, in degrees (0 to 359).

---

## SetPosition

### Syntax

`setposition(int x, int y)`

### Usage

Set the turtle's X and Y coordinates.

---

## Stamp

### Syntax

`stamp(texture t, int x, int y)`

### Usage

Stamp texture t onto the bitmap layer at location (x, y). The brush and clipping are used when drawing pixels.

---

## Triangle

### Syntax

`triangle(int x1, int y1, int x2, int y2, int x3, int y3, int fillmode)`

### Usage

Draw a triangle from (x1, y1) to (x2, y2) to (x3, y3) using the current ink color. Fillmode is 0 if the triangle is not filled, or 1 if the triangle is filled. The brush and clipping are used when drawing pixels.

---

## TurnLeft

### Syntax

`turnleft(float deg)`

### Usage

Rotate the turtle counter clockwise by the specified number of degrees.

---

## TurnRight

### Syntax

`turnright(float deg)`

### Usage

Rotate the turtle clockwise by the specified number of degrees.

## vLine

---

### Syntax

```
vline(int x, int y1, int y2)
```

### Usage

Draw a vertical line from (x, y1) to (x, y2) using the current ink color. The brush and clipping are used when drawing pixels.

## CharSet

A char set is a fixed width custom character set.

Inherits from FONT.

## Constructor

---

### Syntax

```
charset(texture t, int tx, int ty, int x, int y)
```

### Usage

Create a new custom character set. Texture t is the texture to build the character set from. tx is the width (in pixels) of each character tile in the texture, and ty is the height (in pixels) of each character tile in the texture. x is the width (in pixels) to draw the character, and y is the height (in pixels) to draw the character.

The constructor simply binds the texture to the character set and defines the size of the character tiles and the size to draw each character. Each ASCII character must be "mapped" to a character tile before it can be printed.

### Example



The texture is 128x128 pixels. Each character tile is 8x8 pixels big (tx=8, ty=8), so the texture has 16 characters across (128÷8=16) and 16 characters down (128÷8=16), for a total of 256 characters.

Setting x=8 and y=8 would draw each character the proper size. Setting x=16 and y=8 would draw each character double width. Setting x=8 and y=16 would draw each character double height. Setting x=16 and y=16 would draw each character double size.

## Clear

---

### Syntax

```
clear(int ch)
```

### Usage

Clear the character ch.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 0 or greater than 255.

## MapAll

---

### Syntax

`mapall()`

### Usage

Map all 256 ASCII characters to each character tile in the texture, starting with character tile 0 in the upper left hand corner.

### Exceptions

Throw `VM_DENIED` if the operation failed.

## MapChar

---

### Syntax

`mapchar(int ch, int block)`

### Usage

Map character `ch` to character tile `block`. Character tiles are numbered sequentially, with 0 in the upper left hand corner.

### Exceptions

Throws `VM_ILLEGAL_QUANTITY` if `ch` is less than 0 or greater than 255.

Throws `VM_DENIED` if the operation failed.

## MapChars

---

### Syntax

`mapchars(int ch, int block, int size)`

### Usage

Map multiple sequential characters starting at character `ch` to character tile `block`. Character tiles are numbered sequentially, with 0 in the upper left hand corner. `size` is the number of character blocks to map.

### Exceptions

Throws `VM_ILLEGAL_QUANTITY` if `ch` is less than 0 or greater than 255.

Throws `VM_DENIED` if the operation failed.

## MapTheseChars

---

### Syntax

`mapthesechars(string chars, int block)`

### Usage

Map the characters in string `chars` to a character tile, starting at character tile `block`. Character tiles are numbered sequentially, with 0 in the upper left hand corner.

### Exceptions

Throws `VM_DENIED` if the operation failed.

## Palette

---

### Syntax

`palette(int ch, int color)`

### Usage

Each character set has its own independent "palette" of 256 colors. Each color is represented with an ASCII character, that can then be used in the Write method to write colored pixels to a character. In effect this allows "custom character sets", where simple graphics and animation can be achieved by directly altering characters, affecting the way they are rendered.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 0 or greater than 255.

### Refresh

---

#### Syntax

```
refresh()
```

### Usage

Refresh the character set, by copying it from memory to video memory. Any changes made to a character set will not appear on-screen until the character set is refreshed.

### Scroll

---

#### Syntax

```
scroll(int ch, int dir, int count, boolean wrap)
```

### Usage

Scroll character ch in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of pixels in count. If wrap is TRUE then pixels that scroll off the character appear on the other side, if wrap is FALSE then they do not.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 0 or greater than 255, dir is less than 1 or greater than 4, or count is less than 1.

### TranslateChar

---

#### Syntax

```
translatechar(int ch)
```

### Usage

Tell a character to translate one tile width to the right when rendered. This is used to designate a character as a space. For example, TranslateChar(32) tells character 32 (the space character) to move one tile width over to the right when rendered.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 0 or greater than 255.

### Write

---

#### Syntax

```
write(int ch, int scanline, string p)
```

### Usage

Write directly to a character in the character set. ch is the ASCII character to write to. scanline is the scan line to write to; scan lines are numbered starting with 0 at the top. p is a string of ASCII characters, where each character represents a color in the character sets palette (see the Palette method).

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 0 or greater than 255, or scanline is less than 0 or greater than or equal to the height of the character.

## Collection

A collection is used to hold zero, one, or more objects. Some collections have defined rules, to implement certain data structures (for example, a stack can only push and pop objects). The efficiency of accessing, finding, deleting, and inserting into a collection depends on these rules and whether the collection is sorted or not.

Inherits from OBJECT.

## Console

A console is a linear process that uses an interrupt to automatically handle the message pump.

Inherits from PROCESS.

---

## AddTimer

### Syntax

```
addtimer(int timer, int ms)
```

### Usage

Add a timer to the console. A console can have up to 256 independent timers. A timer is used to enqueue a TIMER event at set intervals, ms milliseconds apart. AddTimer will "overwrite" an existing timer if defined.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if timer is less than 0 or greater than 255, or ms is less than 100 (1/10<sup>th</sup> of a second) or greater than 86400000 (24 hours).

---

## AutoScroll

### Syntax

```
autoscroll(boolean state)
```

### Usage

Enable auto scrolling if state is TRUE, disable auto scrolling if state is FALSE.

---

## AutoSort2D

### Syntax

```
autosort2d(int m)
```

### Usage

Set the 2D automatic sort mode:

0	DS_NONE	do not sort 2D entities
1	DS_UP	sort 2D entities based on depth, from minimum to maximum (default)
2	DS_DOWN	sort 2D entities based on depth, from maximum to minimum

---

## BezierCurve

### Syntax

```
beziercurve(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4)
```

**Usage**

Draw a Bezier curve using the four control points specified: (x1, y1), (x2, y2), (x3, y3), and (x4, y4). The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

---

**BitmapMode****Syntax**

```
bitmapmode(int w, int h)
```

**Usage**

Switch the console to bitmap mode, using the specified width and height. The console window is automatically resized to accommodate the bitmap screen.

**Exceptions**

Throws VM\_DENIED if the operation failed.

**Syntax**

```
bitmapmode(int w, int h, int x, int y)
```

**Usage**

Switch the console to bitmap mode, using the specified width and height, and scaling the bitmap to (x, y). The console window is automatically resized to (x,y).

**Exceptions**

Throws VM\_DENIED if the operation failed.

---

**Brush****Syntax**

```
brush(int op)
```

**Usage**

Set the brush operation to op.

This command is ignored if not in bitmap mode.

**Syntax**

```
brush(int op, int mask)
```

**Usage**

Set the brush operation to op and the brush mask to mask.

This command is ignored if not in bitmap mode.

---

**CenterText****Syntax**

```
centertext(int n)
```

**Usage**

Center the int n at the current cursor location.

This command is ignored if not in text mode.

**Syntax**

```
centertext(float n)
```

**Usage**

Center the float *n* at the current cursor location.

This command is ignored if not in text mode.

**Syntax**

```
centertext(string s)
```

**Usage**

Center the string *s* at the current cursor location.

This command is ignored if not in text mode.

**Circle**

---

**Syntax**

```
circle(int x, int y, int r, int fillmode)
```

**Usage**

Draw a circle at (*x*, *y*) using the current ink color with the radius *r*. Fillmode is 0 if the circle is not filled, or 1 if the circle is filled. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

**ClearEventQueue**

---

**Syntax**

```
cleareventqueue()
```

**Usage**

Clear the event queue.

**Clip**

---

**Syntax**

```
clip(int x1, int y1, int x2, int y2)
```

**Usage**

Clip the bitmap screen to the region from (*x1*, *y1*) to (*x2*, *y2*).

This command is ignored if not in bitmap mode.

**Cls**

---

**Syntax**

```
cls()
```

**Usage**

Clear the text screen or bitmap screen, whichever is active.

## **Copy**

---

### **Syntax**

```
copy(texture t)
```

### **Usage**

Copy texture *t* to the bitmap screen. If *t* is larger than the bitmap screen, *t* is clipped to the bitmap screen. The brush is used and clipping is ignored.

This command is ignored if not in bitmap mode.

## **DoubleBuffer**

---

### **Syntax**

```
doublebuffer()
```

### **Usage**

Set the bitmap screen to double buffer mode. Any graphics drawn will be rendered to a workscreen in memory. When the graphics are finished, this workscreen is "flipped" to the active screen, using the Flip method, where it becomes visible. This allows for smooth animation, as the entire screen is only displayed when drawing is completed. Without double buffering, the graphics would flicker as the screen was drawn and redrawn.

This command is ignored if not in bitmap mode.

## **DrawText**

---

### **Syntax**

```
drawto(charset cs, int x, int y, string s)
```

### **Usage**

Print the string *s* using character set *cs* at (*x*, *y*) using the current ink color. The brush and clipping method are used when drawing pixels.

This command is ignored if not in bitmap mode.

## **DrawTo**

---

### **Syntax**

```
drawto(int x, int y)
```

### **Usage**

Draw a line from the current graphics cursor to location (*x*, *y*) using the current ink color. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

## **Ellipse**

---

### **Syntax**

```
ellipse(int x, int y, int xr, int yr, int fillmode)
```

### **Usage**

Draw an ellipse at (x, y) using the current ink color with a horizontal radius of xr and a vertical radius of yr. Fillmode is 0 if the ellipse is not filled, or 1 if the ellipse is filled. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

---

## Fill

### Syntax

```
fill(int c)
```

### Usage

Fill the bitmap screen with a color. The brush is used, but clipping is ignored.

This command is ignored if not in bitmap mode.

---

## Flip

### Syntax

```
flip()
```

### Usage

In double buffered bitmap mode, copy the hidden workscreen that is being rendered to the visible screen.

In all modes, flip also runs an "interrupt", so the Windows message pump runs and the window is redrawn.

---

## Flip180()

### Syntax

```
flip180()
```

### Usage

Rotate the bitmap screen 180 degrees.

This command is ignored if not in bitmap mode.

---

## FloodFill

### Syntax

```
floodfill(int x, int y)
```

### Usage

Flood fill an area at (x, y) using the current ink color. The current pixel at (x, y) is used as the "old color". All pixels with the old color to the left, right, top, or bottom of (x, y) are colored, and this repeats with those pixels. This is often used to fill in an irregular shape. The shape must be enclosed, or the color will "spill out" of the shape. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

---

## GetBrushMask

### Syntax

```
int:getbrushmask()
```

### Usage

Return the current brush mask.

This command is ignored if not in bitmap mode.

---

### GetBrushOp

---

#### Syntax

`int:getbrushop()`

#### Usage

Return the current brush operation.

This command is ignored if not in bitmap mode.

---

### GetCharSet

---

#### Syntax

`charset:getcharset()`

#### Usage

Return the current character set.

This command is ignored if not in text mode.

---

### GetConsoleHeight

---

#### Syntax

`int:getconsoleheight()`

#### Usage

Return the console's height in pixels.

---

### GetConsoleName

---

#### Syntax

`string:getconsolename()`

#### Usage

Return the console's name.

---

### GetConsoleWidth

---

#### Syntax

`int:getconsolewidth()`

#### Usage

Return the console's width in pixels.

---

### GetConsoleX

---

#### Syntax

`int:getconsolex()`

**Usage**

Return the console's X position, where (0, 0) is the upper left corner of the Windows Desktop.

**GetConsoleY**

---

**Syntax**

```
int:getconsoley()
```

**Usage**

Return the console's Y position, where (0, 0) is the upper left corner of the Windows Desktop.

**GetCursorColor**

---

**Syntax**

```
int:getcursorcolor()
```

**Usage**

Return the cursor color.

This command is ignored if not in text mode.

**GetCursorX**

---

**Syntax**

```
int:getcursorx()
```

**Usage**

Return the cursor's X location.

This command is ignored if not in text mode or bitmap mode.

**GetCursorY**

---

**Syntax**

```
int:getcursory()
```

**Usage**

Return the cursor's Y location.

This command is ignored if not in text mode or bitmap mode.

**GetEvent**

---

**Syntax**

```
int:getevent()
```

**Usage**

Return the next event in the event queue, or 0 if the event queue is empty.

**GetEventData**

---

**Syntax**

```
int:geteventdata(int e)
```

### Usage

Return the data encapsulated in event e. The data is determined by the actual event type:

EVT_KEYDOWN	ASCII code of the key down
EVT_MOUSEDOWN	the mouse button (1- left, 2-right, 4-middle)
EVT_MOUSEUP	the mouse button
EVT_CLICKED	the mouse button
EVT_DBLCLICKED	the mouse button
EVT_MOUSEWHEEL	the mouse wheel position (<128 down, >128 up)
EVT_TIMER	the timer that fired

### GetEventType

---

#### Syntax

```
int:geteventtype(int e)
```

#### Usage

Return the type encapsulated in event e. The event type is one of the following:

EVT_KEYDOWN	a key is down
EVT_MOUSEMOVE	the mouse has moved
EVT_MOUSEDOWN	the mouse button is down
EVT_MOUSEUP	the mouse button (that was down) is now up
EVT_CLICKED	the mouse has been clicked (down and up)
EVT_DBLCLICKED	the mouse has been double clicked
EVT_MOUSEWHEEL	the mouse wheel has changed
EVT_TIMER	a timer fired

### GetFilter

---

#### Syntax

```
int:getfilter()
```

#### Usage

Return the current graphics filter:

1	TX_NEAREST	pixels are scaled
2	TX_LINEAR	pixels are scaled and averaged with nearby pixels, smoothing the graphics
3	TX_MIPMAP	same as TX_LINEAR, but creates multiple textures of varying sizes

This command is ignored if not in bitmap mode.

### GetHeading

---

#### Syntax

```
float:getheading()
```

#### Usage

Return the turtle's heading, in degrees (0 to 359).

This command is ignored if not in bitmap mode.

## GetHeight

---

### Syntax

`int:getheight()`

### Usage

Return the console's height (characters in text mode, pixels in bitmap mode).

## GetHighlight

---

### Syntax

`int:gethighlight()`

### Usage

Return the highlight color.

This command is ignored if not in text mode.

## GetInk

---

### Syntax

`int:getink()`

### Usage

Return the ink color.

## GetKey

---

### Syntax

`int:getkey()`

### Usage

Consume all events until a KEYDOWN event is found. Return the KEYDOWN event's ASCII code, or 0 if no KEYDOWN event is in the event queue.

## GetMode

---

### Syntax

`int:getmode()`

### Usage

Return the current mode (1-text mode, 2-bitmap mode, 3-multiscreen mode).

## GetPaper

---

### Syntax

`int:getpaper()`

### Usage

Return the paper color.

## GetTurtleX

---

### Syntax

`float:getturtleX()`

**Usage**

Return the turtle's X coordinate.

This command is ignored if not in bitmap mode.

---

**GetTurtleY**

**Syntax**

`float:getturtleY()`

**Usage**

Return the turtle's Y coordinate.

This command is ignored if not in bitmap mode.

---

**GetWidth**

**Syntax**

`int:getwidth()`

**Usage**

Return the console's width (characters in text mode, pixels in bitmap mode).

---

**GoBackward**

**Syntax**

`gobackward(float units)`

**Usage**

Move the turtle backward the specified number of units.

This command is ignored if not in bitmap mode.

---

**GoForward**

**Syntax**

`goforward(float units)`

**Usage**

Move the turtle forward the specified number of units.

This command is ignored if not in bitmap mode.

---

**GrayScale**

**Syntax**

`grayscale()`

**Usage**

Filter all of the pixels in the bitmap screen to a gray scale, effectively making color pictures black and white.

This command is ignored if not in bitmap mode.

## HideCursor

---

### Syntax

```
hidecursor()
```

### Usage

Hide the cursor if in text mode.

This command is ignored if not in text mode.

## Highlight

---

### Syntax

```
highlight(int c)
```

### Usage

Set the highlight color.

This command is ignored if not in text mode.

## hLine

---

### Syntax

```
hline(int x1, int x2, int y)
```

### Usage

Draw a horizontal line from (x1, y) to (x2, y) using the current ink color. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

## Home

---

### Syntax

```
home()
```

### Usage

Return the turtle to the center of the bitmap screen.

This command is ignored if not in bitmap mode.

## Ink

---

### Syntax

```
ink(int fg)
```

### Usage

Set the ink (foreground) color. The ink color is used when printing characters in text mode, or drawing pixels in bitmap mode.

## Input

---

### Syntax

```
string:input()
```

### Usage

Input a string from the user. The console will wait until ENTER is pressed, and then return the string.

This command is ignored if not in text mode.

## InputNum

---

### Syntax

```
float:inputnum()
```

### Usage

Input a floating point number from the user. Only digits and one decimal point and an optional negative sign are allowed, other keys are ignored. The console will wait until ENTER is pressed, and then return the number. If no number is entered 0 is returned.

This command is ignored if not in text mode.

## IsCursorVisible

---

### Syntax

```
boolean:iscursorvisible()
```

### Usage

Return TRUE if the cursor is visible, FALSE if the cursor is not visible.

This command is ignored if not in text mode.

## Line

---

### Syntax

```
line(int x1, int y1, int x2, int y2)
```

### Usage

Draw a line from (x1, y1) to (x2, y2) using the current ink color. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

### Syntax

```
line(float deg, float n)
```

### Usage

Draw a line from the current graphics cursor in the direction deg (degrees), with the length n. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

## LineFeed

---

### Syntax

```
linefeed()
```

**Usage**

Print a line feed at the current cursor location, scrolling the screen if the cursor is on the last line of the text screen.

This command is ignored if not in text mode.

---

**LoadImage****Syntax**

```
loadimage(string f)
```

**Usage**

Load the image in file *f*. The image is automatically scaled to fit the bitmap screen.

This command is ignored if not in bitmap mode.

**Exceptions**

Throws VM\_DENIED if the operation failed.

Throws VM\_FILE\_NOT\_FOUND if the file *f* does not exist.

Throws VM\_BAD\_FILE\_FORMAT if the file *f* is corrupted or not a supported image type.

---

**Locate****Syntax**

```
locate(int x, int y)
```

**Usage**

Set the cursor's X and Y location.

This command is ignored if not in text mode.

---

**Mask****Syntax**

```
mask(int color, int alpha)
```

**Usage**

Scan the entire bitmap screen, and wherever the RGB components of color are present, set the alpha component to alpha.

This is often used to load a 24-bit BMP (which treats all pixels as opaque), and making all pixels of a set color (for example, black) transparent.

This command is ignored if not in bitmap mode.

---

**MirrorX****Syntax**

```
mirrorx()
```

**Usage**

Mirror the bitmap screen along the X axis.

This command is ignored if not in bitmap mode.

## MirrorY

---

### Syntax

```
mirrorY()
```

### Usage

Mirror the bitmap screen along the Y axis.

This command is ignored if not in bitmap mode.

## MoveConsole

---

### Syntax

```
moveconsole(int x, int y)
```

### Usage

Move the console to (x, y), where (0, 0) is the upper left corner of the Windows Desktop.

## MultiscreenMode

---

### Syntax

```
multiscreenmode(int w, int h)
```

### Usage

Switch the console to multiscreen mode, using the specified width and height. The console window is automatically resized to accommodate the multiscreen. Multiscreen mode is a very powerful mode, inspired by the Super Nintendo's classic "Mode 7". In multiscreen mode, individual text and bitmap layers can be stacked on top of each other, similar to the traditional cel animation in cartoons. Individual layers can be tinted, blended, scaled, and rotated, allowing for very creative graphics.

### Exceptions

Throws VM\_DENIED if the operation failed.

## Paper

---

### Syntax

```
paper(int bg)
```

### Usage

Set the paper (background) color.

## Peek

---

### Syntax

```
int:peek(int d)
```

### Usage

Return the character stored at position d in text mode. Return the pixel stored at position d in bitmap mode. Characters and pixels are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the screen.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if d is less than zero or greater than or equal to the number of characters (text mode) or pixels (bitmap mode).

## PenDown

---

### Syntax

`pendown ()`

### Usage

Place the turtle's pen down.

This command is ignored if not in bitmap mode.

## PenUp

---

### Syntax

`penup ()`

### Usage

Place the turtle's pen up.

This command is ignored if not in bitmap mode.

## Play

---

### Syntax

`play(string music, boolean wait)`

### Usage

Play a music string through the speaker.

#### Octave and Tone commands

- `o n` Set current octave ( $n = 0-6$ )
- `< or >` Move up or down one octave
- `A - G` Play A, B, ... G note in current octave (+ or # = sharp, - = flat)
- `Nn` Play a note ( $n = 0-84$ , where 0 is a rest)

#### Duration and Tempo commands

- `Ln` Sets length of a note ( $n = 1-64$ , where 64 is the 64<sup>th</sup> note)
- `ML` Each note plays full length
- `MN` Each note plays 7/8 of length
- `MS` Each note plays 3/4 of length
- `Pn` Pause for the duration of  $n$  quarternotes ( $n = 1-64$ )
- `Tn` Sets tempo ( $n = 32-255$ , 120 quarter notes per minute default)

If `wait` is `TRUE` the console will wait until the music finishes playing. If `wait` is `FALSE` the music will play in the background and the console will continue.

### Example

```
; Deck the Halls
Play("mnt255o2L4A.L8GL4F#EDEF#DL8EF#GEL4F#.L8EL4DC#L2DL4A.L8GL4F#EDEF#DL8EF#GEL4F#.
L8EL4DC#L2DL4E.L8F#L4GEF#.L8GL4AEL8F#G#L4AL8B>C#L4DC#<BL2AL4A.L8GL4F#EDEF#DL8BBBBL4
A.L8GL4F#EL2D ")
```

## Plot

---

### Syntax

```
plot(int x, int y)
```

**Usage**

Plot a pixel at location (x, y) using the current ink color. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

---

**Point****Syntax**

```
int:point(int x, int y)
```

**Usage**

Return the pixel at location (x, y).

This command is ignored if not in bitmap mode.

---

**Poke****Syntax**

```
poke(int d, int c)
```

**Usage**

Set the character stored at position d to ASCII character c in text mode. Set the pixel stored at position d to color c in bitmap mode. Characters and pixels are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the screen.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if d is less than zero or greater than or equal to the number of characters (text mode) or pixels (bitmap mode).

---

**Print****Syntax**

```
print(int n)
```

**Usage**

Print int n at the current cursor location.

This command is ignored if not in text mode.

**Syntax**

```
print(float n)
```

**Usage**

Print float n at the current cursor location.

This command is ignored if not in text mode.

**Syntax**

```
print(string s)
```

**Usage**

Print string s at the current cursor location.

This command is ignored if not in text mode.

## **PrintAt**

---

### **Syntax**

```
printat(int x, int y, int n)
```

### **Usage**

Print int n at the cursor location (x, y).

This command is ignored if not in text mode.

### **Syntax**

```
printat(int x, int y, float n)
```

### **Usage**

Print float n at the cursor location (x, y).

This command is ignored if not in text mode.

### **Syntax**

```
printat(int x, int y, string s)
```

### **Usage**

Print string s at the cursor location (x, y).

This command is ignored if not in text mode.

## **PrintChar**

---

### **Syntax**

```
printchar(int ch)
```

### **Usage**

Print a single character at the current cursor location. ch is the ASCII character to print, and must be in the range of 0 to 255.

This command is ignored if not in text mode.

### **Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if key is less than 0 or greater than 255.

## **PrintLn**

---

### **Syntax**

```
println(int n)
```

### **Usage**

Print int n at the current cursor location, followed by a carriage return.

This command is ignored if not in text mode.

### **Syntax**

```
println(float n)
```

### **Usage**

Print float `n` at the current cursor location, followed by a carriage return.

This command is ignored if not in text mode.

#### **Syntax**

```
println(string s)
```

#### **Usage**

Print string `s` at the current cursor location, followed by a carriage return.

This command is ignored if not in text mode.

---

### **Rect**

#### **Syntax**

```
rect(int x1, int y1, int x2, int y2, int fillmode)
```

#### **Usage**

Draw a rectangle from `(x1, y1)` to `(x2, y2)` using the current ink color. Fillmode is 0 if the rectangle is not filled, or 1 if the rectangle is filled. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

---

### **RemoveTimer**

#### **Syntax**

```
removetimer(int timer)
```

#### **Usage**

Stop the given timer.

#### **Exceptions**

Throws `VM_ILLEGAL_QUANTITY` if timer is less than 0 or greater than 255.

---

### **RenameConsole**

#### **Syntax**

```
renameconsole(string caption)
```

#### **Usage**

Rename the console to caption.

---

### **Rescale**

#### **Syntax**

```
rescale(int w, int h, int filter)
```

#### **Usage**

Resize the bitmap screen to the specified width and height. Filter determines the algorithm to use when resizing:

0	<code>TXF_BOX</code>	Box, pulse, Fourier window, 1st order (constant) b-spline
1	<code>TXF_BICUBIC</code>	Mitchell & Netravali's two-param cubic filter
2	<code>TXF_BILINEAR</code>	Bilinear filter
3	<code>TXF_BSPLINE</code>	4th order (cubic) b-spline

4	TXF_CATMULLROM	Catmull-Rom spline, Overhauser spline
5	TXF_LANCZOS3	Lanczos3 filter

This command is ignored if not in bitmap mode.

---

## ResizeConsole

### Syntax

```
resizeconsole(int w, int h)
```

### Usage

Resize the console to the specified width and height.

---

## SaveImage

### Syntax

```
saveimage(string f)
```

### Usage

Save the bitmap screen to file f.

This command is ignored if not in bitmap mode.

### Exceptions

Throws VM\_DENIED if the operation failed.

Throws VM\_BAD\_FILE\_FORMAT if the file f is not a supported image type.

---

## Say

### Syntax

```
say(string p, boolean wait)
```

### Usage

Speak the specified string.

If wait is TRUE the console will wait until the speech finishes. If wait is FALSE the speech will play in the background and the console will continue.

### Example

```
Say("Hello world!")
```

---

## Scroll

### Syntax

```
scroll(int dir, int count, boolean wrap)
```

### Usage

Scroll the entire screen in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of character cells (in text mode) or pixels (in bitmap mode) in count. If wrap is TRUE then characters or pixels that scroll off the screen appear on the other side, if wrap is FALSE then they do not.

This command is ignored if not in text mode.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

**Syntax**

```
scroll(int dir, int count, int x1, int y1, int x2, int y2, boolean wrap)
```

**Usage**

Scroll the region from (x1, y1) to (x2, y2) in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of character cells (in text mode) or pixels (in bitmap mode) in count. If wrap is TRUE then characters or pixels that scroll off the region appear on the other side, if wrap is FALSE then they do not.

This command is ignored if not in text mode.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

---

**Sepia****Syntax**

```
sepia()
```

**Usage**

Filter all of the pixels in the bitmap screen to a sepia tone (similar to an 1800's photograph).

This command is ignored if not in bitmap mode.

---

**SetCharSet****Syntax**

```
setcharset(charset cs)
```

**Usage**

Set the character set to cs.

This command is ignored if not in text mode.

---

**SetCursorColor****Syntax**

```
setcursorcolor(int c)
```

**Usage**

Set the cursor color.

This command is ignored if not in text mode.

---

**SetFilter****Syntax**

```
setfilter(int f)
```

**Usage**

Set the graphics filter.

1	TX_NEAREST	pixels are scaled
---	------------	-------------------

- |   |           |   |
|---|-----------|---|
| 2 | TX_LINEAR | pixels are scaled and averaged with nearby pixels, smoothing the graphics |
| 3 | TX_MIPMAP | same as TX_LINEAR, but creates multiple textures of varying sizes         |

This command is ignored if not in bitmap mode.

---

### SetHeading

#### Syntax

```
setheading(float h)
```

#### Usage

Set the turtle's heading, in degrees (0 to 359).

This command is ignored if not in bitmap mode.

---

### SetPosition

#### Syntax

```
setposition(int x, int y)
```

#### Usage

Set the turtle's X and Y coordinates.

This command is ignored if not in bitmap mode.

---

### ShowCursor

#### Syntax

```
showcursor()
```

#### Usage

Show the cursor if in text mode.

This command is ignored if not in text mode.

---

### SingleBuffer

#### Syntax

```
singlebuffer()
```

#### Usage

Set the bitmap screen to single buffer mode, so any graphics drawn will appear immediately.

This command is ignored if not in bitmap mode.

---

### Sleep

#### Syntax

```
sleep(int ms)
```

#### Usage

Pause the console by ms milliseconds.

## Sort2D

---

### Syntax

```
sort2d()
```

### Usage

Sort any console entities based on the entity's depth.

## Sound

---

### Syntax

```
sound(int freq, int duration)
```

### Usage

Play a simple tone on the speaker. `freq` is the tone frequency, in hertz, and must be between 37 and 32,767. `duration` is the number of milliseconds to play the tone, and must be between 1 and 65,535.

### Exceptions

Throws `VM_ILLEGAL_QUANTITY` if hertz is less than 37 or greater than 32,767, or if duration is less than 1 or greater than 65,535.

## Stamp

---

### Syntax

```
stamp(texture t, int x, int y)
```

### Usage

Stamp texture `t` onto the bitmap screen at location `(x, y)`. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

## TextMode

---

### Syntax

```
textmode(int w, int h)
```

### Usage

Switch the console to text mode, using the default console font and the specified width and height. The console window is automatically resized to accommodate the text screen.

### Exceptions

Throws `VM_DENIED` if the operation failed.

### Syntax

```
textmode(charset cs, int w, int h)
```

### Usage

Switch the console to text mode, using the specified character set, width, and height. The console window is automatically resized to accommodate the text screen.

### Exceptions

Throws `VM_DENIED` if the operation failed.

## Triangle

---

### Syntax

```
triangle(int x1, int y1, int x2, int y2, int x3, int y3, int fillmode)
```

### Usage

Draw a triangle from (x1, y1) to (x2, y2) to (x3, y3) using the current ink color. Fillmode is 0 if the triangle is not filled, or 1 if the triangle is filled. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

## TurnLeft

---

### Syntax

```
turnleft(float deg)
```

### Usage

Rotate the turtle counter clockwise by the specified number of degrees.

This command is ignored if not in bitmap mode.

## TurnRight

---

### Syntax

```
turnright(float deg)
```

### Usage

Rotate the turtle clockwise by the specified number of degrees.

This command is ignored if not in bitmap mode.

## vLine

---

### Syntax

```
vline(int x, int y1, int y2)
```

### Usage

Draw a vertical line from (x, y1) to (x, y2) using the current ink color. The brush and clipping are used when drawing pixels.

This command is ignored if not in bitmap mode.

## WaitEvent

---

### Syntax

```
int:waitevent()
```

### Usage

Wait until an event is enqueued in the event queue and return it.

## WaitKey

---

### Syntax

```
int:waitkey()
```

**Usage**

Clear the event queue and wait until a KEYDOWN event happens. Return the KEYDOWN event's ASCII code.

**Databank**

---

Provides a memory bank to store and retrieve information.

Inherits from OBJECT.

**Constructor**

---

**Syntax**

```
databank(int s)
```

**Usage**

Allocate memory for the databank. s is the number of bytes the databank should hold.

**Cast**

---

**Syntax**

```
string:cast()
```

**Usage**

Cast the databank object to a string. The entire contents of the databank are returned.

**Clear**

---

**Syntax**

```
clear()
```

**Usage**

Clear the databank.

**Peek**

---

**Syntax**

```
int:peek(int d)
```

**Usage**

Return the byte at memory location d.

**Exceptions**

Throws VM\_BAD\_SUBSCRIPT if d is less than 0 or greater than the number of bytes allocated in the databank.

**PeekFloat**

---

**Syntax**

```
float:peekfloat(int d)
```

**Usage**

Return the float at memory location d.

**Exceptions**

Throws VM\_BAD\_SUBSCRIPT if d is less than 0 or greater than the number of bytes allocated in the databank.

---

### PeekInt

#### Syntax

```
int:peekint(int d)
```

#### Usage

Return the integer at memory location d.

#### Exceptions

Throws VM\_BAD\_SUBSCRIPT if d is less than 0 or greater than the number of bytes allocated in the databank.

---

### Poke

#### Syntax

```
poke(int d, int e)
```

#### Usage

Poke byte e at memory location d.

#### Exceptions

Throws VM\_BAD\_SUBSCRIPT if d is less than 0 or greater than the number of bytes allocated in the databank.

---

### PokeFloat

#### Syntax

```
pokefloat(int d, float e)
```

#### Usage

Poke float e at memory location d.

#### Exceptions

Throws VM\_BAD\_SUBSCRIPT if d is less than 0 or greater than the number of bytes allocated in the databank.

---

### PokeInt

#### Syntax

```
pokeint(int d, int e)
```

#### Usage

Poke integer e at memory location d.

#### Exceptions

Throws VM\_BAD\_SUBSCRIPT if d is less than 0 or greater than the number of bytes allocated in the databank.

---

## Database

A database object provides access to a database, a collection of related data. Liquid Studio currently supports the SQLite database. More information on SQLite can be found at [www.sqlite.org](http://www.sqlite.org).

Inherits from OBJECT.

## Close

---

### Syntax

`close()`

### Usage

Close the database.

### Exceptions

Throws `VM_DENIED` if the operation failed.

## Open

---

### Syntax

`open(string dbname)`

### Usage

Open the specified database. `dbname` is the filename of the SQLite database to open.

### Exceptions

Throws `VM_FILE_OPEN` if the database has been already opened.

Throws `VM_DENIED` if the operation failed.

## Submit

---

### Syntax

`submit(string cmd)`

### Usage

Submit a set of SQL commands (`cmd`) to execute on the database.

### Exceptions

Throws `VM_FILE_NOT_OPEN` if the database has not been opened yet.

Throws `VM_DATABASE_ERROR` if the operation failed. The exception data holds the actual error message.

## Deque

A deque (double ended queue) can add and remove objects from the head (front) or tail (back) of the deque.

Inherits from `COLLECTION`.

## Clear

---

### Syntax

`clear()`

### Usage

Clear the deque.

## GetCount

---

### Syntax

`int:getcount()`

**Usage**

Return the number of items in the deque.

**IsEmpty**

---

**Syntax**

`boolean: isempty ()`

**Usage**

Return TRUE if the deque is empty, otherwise return FALSE.

**PeekBack**

---

**Syntax**

`item: peekback ()`

**Usage**

Return the item at the back of the deque.

**PeekFront**

---

**Syntax**

`item: peekfront ()`

**Usage**

Return the item at the front of the deque.

**PopBack**

---

**Syntax**

`item: popback ()`

**Usage**

Pop the item at the back of the deque and return it.

**PopFront**

---

**Syntax**

`item: popfront ()`

**Usage**

Pop the item at the front of the deque and return it.

**PushBack**

---

**Syntax**

`pushback (item i)`

**Usage**

Push item *i* onto the back of the deque.

## PushFront

---

### Syntax

`pushfront(item i)`

### Usage

Push item *i* onto the front of the deque.

## DoubleLinkedList

A double link list stores objects in nodes. Each node has a pointer to the prior node in the list and the next node in the list.

ACCESS	O(1)
FIND	O(n) if unsorted, O(log(n)) if sorted
DELETE	O(1)
INSERT	O(1)

Inherits from COLLECTION.

## Add

---

### Syntax

`int:add(item i)`

### Usage

Add item *i* to the double link list. Return the index of the added item.

## Clear

---

### Syntax

`clear()`

### Usage

Clear the double link list.

## Contains

---

### Syntax

`boolean:contains(item i)`

### Usage

Return TRUE if item *i* is in the double link list, otherwise return FALSE.

## Delete

---

### Syntax

`delete(int inx)`

### Usage

Delete the item at index *inx*.

## Exceptions

Throws VM\_BAD\_SUBSCRIPT if the index `inx` is less than 0 or greater than `s`, where `s` is the size of the double link list.

---

### DeleteAtCursor

---

#### Syntax

```
deleteatcursor()
```

#### Usage

Delete the item at the cursor.

#### Exceptions

Throws VM\_DENIED if the cursor is on the head node or the tail node.

---

### Enqueue

---

#### Syntax

```
enqueue(item i)
```

#### Usage

Enqueue item `i` into the sorted double link list.

#### Exceptions

Throws VM\_NOT\_SORTED if the double link list is not sorted.

---

### Find

---

#### Syntax

```
int:find(item i)
```

#### Usage

Find item `i` in the single link list and return its location, or `-1` if the item can't be found.

---

### First

---

#### Syntax

```
item:first()
```

#### Usage

Return the first item in the double link list.

#### Exceptions

Throws VM\_ITEM\_NOT\_FOUND if the double link list is empty.

---

### GetCount

---

#### Syntax

```
int:getcount()
```

#### Usage

Return the number of items in the double link list.

## **Insert**

---

### **Syntax**

```
insert(int inx, item i)
```

### **Usage**

Insert item *i* into the double link list at index *inx*.

### **Exceptions**

Throws `VM_BAD_SUBSCRIPT` if the index *inx* is less than 0 or greater than *s*, where *s* is the size of the double link list.

## **InsertAtCursor**

---

### **Syntax**

```
insert(item i)
```

### **Usage**

Insert item *i* at the cursor.

## **InsertionSort**

---

### **Syntax**

```
insertionsort()
```

### **Usage**

Sort the double link list using an insertion sort.

## **IsAfterLast**

---

### **Syntax**

```
boolean:isafterlast()
```

### **Usage**

Return `TRUE` if the cursor is after the last item in the double link list, otherwise return `FALSE`.

## **IsBeforeFirst**

---

### **Syntax**

```
boolean:isbeforefirst()
```

### **Usage**

Return `TRUE` if the cursor is before the first item in the double link list, otherwise return `FALSE`.

## **IsEmpty**

---

### **Syntax**

```
boolean:isempty()
```

### **Usage**

Return `TRUE` if the double link list is empty, otherwise return `FALSE`.

## **Last**

---

### **Syntax**

`item:last()`

**Usage**

Return the last item in the double link list.

**Exceptions**

Throws `VM_ITEM_NOT_FOUND` if the double link list is empty.

---

**Locate**

**Syntax**

`int:locate(item i)`

**Usage**

Locate item `i` in the double link list and return its location, or `-1` if the item can't be located.

---

**MoveAfterLast**

**Syntax**

`moveafterlast()`

**Usage**

Move the cursor after the last item in the double link list.

---

**MoveBeforeFirst**

**Syntax**

`movebeforefirst()`

**Usage**

Move the cursor before the first item in the double link list.

---

**MoveNext**

**Syntax**

`movenext()`

**Usage**

Move the cursor to the next node in the double link list.

---

**MovePrev**

**Syntax**

`moveprev()`

**Usage**

Move the cursor to the previous node in the double link list.

---

**Peek**

**Syntax**

`item:peek()`

**Usage**

Return the item at the cursor.

**Exceptions**

Throws VM\_DENIED if the double link list is empty.

**Remove**

---

**Syntax**

```
remove(item i)
```

**Usage**

Remove item i from the double link list.

**Sort**

---

**Syntax**

```
sort()
```

**Usage**

Sort the double link list.

**Entity**

An entity is a 2D object. Each entity can have four additional components not found in other objects: render (how to draw the object), update (how to update the object), instinct (how the object acts in its environment), and behavior (how the object reacts to its environment).

Inherits from OBJECT.

**AllowDbClicked**

---

**Syntax**

```
allowdbclicked(boolean state)
```

**Usage**

Allow the entity to receive MSG\_DBLCLICKED messages if state is TRUE. If state is FALSE then send two MSG\_CLICKED messages instead of one MSG\_DBLCLICKED message.

**Blend**

---

**Syntax**

```
blend(float b)
```

**Usage**

Set the entity's blend (0 = transparent,  $0 < x < 1$  = translucent, 1 = opaque).

**Center**

---

**Syntax**

```
center()
```

**Usage**

Center the entity on-screen.

---

## DelayMessage

---

### Syntax

```
int:delaymessage(object dest, int msg, string data, int delay)
```

### Usage

Send a delayed message to object dest from the entity. The delay is specified in milliseconds. Optional data can be passed with the message. A handle to the router transaction responsible for delivering the delayed message is returned.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if delay is less than 100 (1/10<sup>th</sup> of a second) or greater than 86400000 (24 hours).

---

## Disable

---

### Syntax

```
disable ()
```

### Usage

Disable an entity (will not receive any messages that are dispatched to it).

---

## Enable

---

### Syntax

```
enable ()
```

### Usage

Enable an entity (will receive any messages that are dispatched to it by running the Behavior component).

---

## GetBlend

---

### Syntax

```
float:getblend ()
```

### Usage

Return the entity's blend (0 = transparent,  $0 < x < 1$  = translucent, 1 = opaque).

---

## GetDepth

---

### Syntax

```
int:getdepth ()
```

### Usage

Return the entity's depth.

---

## GetMouseOverNode

---

### Syntax

```
int:getmouseovernode ()
```

### Usage

Return the node the mouse is over.

The `GetMouseOverNode` method only works if the mouse button is down or mouse feedback is enabled (see the `MouseFeedback` method in the API).

---

### **GetNodeClicked**

#### **Syntax**

```
int:getnodeclicked()
```

#### **Usage**

Return the node the mouse clicked.

---

### **GetRotate**

#### **Syntax**

```
float:getrotate()
```

#### **Usage**

Return the entity's rotation (in degrees).

---

### **GetTint**

#### **Syntax**

```
int:gettint()
```

#### **Usage**

Return the entity's tint.

---

### **GetToolTip**

#### **Syntax**

```
string:gettooltip()
```

#### **Usage**

Return the entity's ToolTip.

---

### **GetXPos**

#### **Syntax**

```
int:getxpos()
```

#### **Usage**

Return the entity's X position on-screen.

---

### **GetXScale**

#### **Syntax**

```
float:getxscale()
```

#### **Usage**

Return the entity's X scale.

## GetYPos

---

### Syntax

`int:getypos()`

### Usage

Return the entity's Y position on-screen.

## GetYScale

---

### Syntax

`float:getyscale()`

### Usage

Return the entity's Y scale.

## GotFocus

---

### Syntax

`boolean:gotfocus()`

### Usage

Return TRUE if the entity has the keyboard focus.

## Hide

---

### Syntax

`hide()`

### Usage

Remove the entity from the program's render queue.

## IsClicked

---

### Syntax

`boolean:isclicked()`

### Usage

Return TRUE if the entity is clicked.

## IsEnabled

---

### Syntax

`boolean:isenabled()`

### Usage

Return TRUE if the entity is enabled.

## IsMouseOver

---

### Syntax

`boolean:ismouseover()`

### Usage

Return TRUE if the mouse is over the entity.

The `IsMouseOver` method only works if the mouse button is down or mouse feedback is enabled (see the `MouseFeedback` method in the API).

---

### **IsMouseOverNode**

---

#### **Syntax**

```
boolean:ismouseovernode(int n)
```

#### **Usage**

Return TRUE if the mouse is over the entity's node `n`.

The `IsMouseOverNode` method only works if the mouse button is down or mouse feedback is enabled (see the `MouseFeedback` method in the API).

---

### **IsRunning**

---

#### **Syntax**

```
boolean:isrunning()
```

#### **Usage**

Return TRUE if the entity is running.

---

### **IsVisible**

---

#### **Syntax**

```
boolean:isvisible()
```

#### **Usage**

Return TRUE if the entity is visible.

---

### **Move**

---

#### **Syntax**

```
move(int x, int y)
```

#### **Usage**

Move the entity to the specified X and Y position.

---

### **MoveDir**

---

#### **Syntax**

```
movedir(float d, float s)
```

#### **Usage**

Move the entity `s` pixels in the direction `d` (degrees) from its current location.

---

### **MoveRel**

---

#### **Syntax**

```
moverel(int x, int y)
```

**Usage**

Move the entity to the relative X and Y position.

**PulseMessage**

---

**Syntax**

```
int:pulsemessage(object dest, int msg, string data, int delay, int count)
```

**Usage**

Send a pulsed message to object dest from the entity. The delay is specified in milliseconds. Count indicates the number of times to pulse the message (if count is zero then the message is pulsed until the object is freed). Optional data can be passed with the message. A handle to the router transaction responsible for delivering the pulsed message is returned.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if delay is less than 100 (1/10<sup>th</sup> of a second) or greater than 86400000 (24 hours).

**Rotate**

---

**Syntax**

```
rotate(float r)
```

**Usage**

Set the entity's rotation (in degrees).

**Scale**

---

**Syntax**

```
scale(float s)
```

**Usage**

Set the entity's X and Y scale.

**Syntax**

```
scale(float xs, float ys)
```

**Usage**

Set the entity's X and Y scale.

**SendMessage**

---

**Syntax**

```
sendmessage(object dest, int msg, string data)
```

**Usage**

Send a message to object dest from the entity. Optional data can be passed with the message.

**SetDepth**

---

**Syntax**

```
setdepth(int d)
```

**Usage**

Set the entity's depth. Entities are sorted by depth from lowest to highest when rendered.

## **SetFocus**

---

### **Syntax**

`setfocus()`

### **Usage**

Give the entity the keyboard focus.

## **SetRotate**

---

### **Syntax**

`setrotate(float r)`

### **Usage**

Set the entity's rotation (in degrees).

## **SetToolTip**

---

### **Syntax**

`settooltip(string tt)`

### **Usage**

Set the entity's ToolTip.

## **SetXPos**

---

### **Syntax**

`setxpos(int x)`

### **Usage**

Set the entity's X position on-screen.

## **SetXScale**

---

### **Syntax**

`setxscale(float xs)`

### **Usage**

Set the entity's X scale.

## **SetYPos**

---

### **Syntax**

`setypos(int y)`

### **Usage**

Set the entity's Y position on-screen.

## **SetYScale**

---

### **Syntax**

`setyscale(float ys)`

### **Usage**

Set the entity's Y scale.

---

## Show

### Syntax

```
show()
```

### Usage

Add the entity to the program's render queue.

---

## Start

### Syntax

```
start()
```

### Usage

Add the entity to the program's update queue.

---

## Stop

### Syntax

```
stop()
```

### Usage

Remove the entity from the program's update queue.

---

## Tint

### Syntax

```
tint(int t)
```

### Usage

Set the entity's tint.

---

## Entity3D

An entity3d is a 3D object. Each entity can have four additional components not found in other objects: render (how to draw the object), update (how to update the object), instinct (how the object acts in its environment), and behavior (how the object reacts to its environment).

Inherits from OBJECT.

---

## Blend

### Syntax

```
blend(float b)
```

### Usage

Set the entity's blend (0 = transparent,  $0 < x < 1$  = translucent, 1 = opaque).

---

## Center

### Syntax

`center()`

**Usage**

Center the entity on-screen.

---

**DelayMessage**

**Syntax**

`int:delaymessage(object dest, int msg, string data, int delay)`

**Usage**

Send a delayed message to object `dest` from the entity. The delay is specified in milliseconds. Optional data can be passed with the message. A handle to the router transaction responsible for delivering the delayed message is returned.

**Exceptions**

Throws `VM_ILLEGAL_QUANTITY` if delay is less than 100 (1/10<sup>th</sup> of a second) or greater than 86400000 (24 hours).

---

**Disable**

**Syntax**

`disable()`

**Usage**

Disable an entity (will not receive any messages that are dispatched to it).

---

**Enable**

**Syntax**

`enable()`

**Usage**

Enable an entity (will receive any messages that are dispatched to it by running the Behavior component).

---

**GetBlend**

**Syntax**

`float:getblend()`

**Usage**

Return the entity's blend (0 = transparent,  $0 < x < 1$  = translucent, 1 = opaque).

---

**GetTint**

**Syntax**

`int:gettint()`

**Usage**

Return the entity's tint.

---

**GetXPos**

**Syntax**

`float:getxpos()`

**Usage**

Return the entity's X position on-screen.

**GetXRotate**

---

**Syntax**

```
float:getxrotate()
```

**Usage**

Return the entity's X rotation (in degrees).

**GetXScale**

---

**Syntax**

```
float:getxscale()
```

**Usage**

Return the entity's X scale.

**GetYPos**

---

**Syntax**

```
float:getypos()
```

**Usage**

Return the entity's Y position on-screen.

**GetYRotate**

---

**Syntax**

```
float:getyrotate()
```

**Usage**

Return the entity's Y rotation (in degrees).

**GetYScale**

---

**Syntax**

```
float:getyscale()
```

**Usage**

Return the entity's Y scale.

**GetZPos**

---

**Syntax**

```
float:getzpos()
```

**Usage**

Return the entity's Z position on-screen.

## GetZRotate

---

### Syntax

```
float: getzrotate ()
```

### Usage

Return the entity's Z rotation (in degrees).

## GetZScale

---

### Syntax

```
float: getzscale ()
```

### Usage

Return the entity's Z scale.

## Hide

---

### Syntax

```
hide ()
```

### Usage

Remove the entity from the program's render queue.

## IsEnabled

---

### Syntax

```
boolean: isenabled ()
```

### Usage

Return TRUE if the entity is enabled.

## IsRunning

---

### Syntax

```
boolean: isrunning ()
```

### Usage

Return TRUE if the entity is running.

## IsVisible

---

### Syntax

```
boolean: isvisible ()
```

### Usage

Return TRUE if the entity is visible.

## Move

---

### Syntax

```
move(float x, float y, float z)
```

### Usage

Move the entity to the specified (X, Y, Z) position.

---

### MoveRel

#### Syntax

```
moverel(float x, float y, float z)
```

#### Usage

Move the entity to the relative (X, Y, Z) position.

---

### PulseMessage

#### Syntax

```
int:pulsemessage(object dest, int msg, string data, int delay, int count)
```

#### Usage

Send a pulsed message to object dest from the entity. The delay is specified in milliseconds. Count indicates the number of times to pulse the message (if count is zero then the message is pulsed until the object is freed). Optional data can be passed with the message. A handle to the router transaction responsible for delivering the pulsed message is returned.

#### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if delay is less than 100 (1/10<sup>th</sup> of a second) or greater than 86400000 (24 hours).

---

### Rotate

#### Syntax

```
rotate(float xr, float yr, float zr)
```

#### Usage

Set the entity's X, Y, and Z rotation (in degrees).

---

### Scale

#### Syntax

```
scale(float s)
```

#### Usage

Set the entity's X, Y, and Z scale.

#### Syntax

```
scale(float xs, float ys, float zs)
```

#### Usage

Set the entity's X, Y, and Z scale.

---

### SendMessage

#### Syntax

```
sendmessage(object dest, int msg, string data)
```

#### Usage

Send a message to object dest from the entity. Optional data can be passed with the message.

## **SetXPos**

---

### **Syntax**

`setxpos(float x)`

### **Usage**

Set the entity's X position on-screen.

## **SetXRotate**

---

### **Syntax**

`setxrotate(float xr)`

### **Usage**

Set the entity's X rotation (in degrees).

## **SetXScale**

---

### **Syntax**

`setxscale(float xs)`

### **Usage**

Set the entity's X scale.

## **SetYPos**

---

### **Syntax**

`setypos(float y)`

### **Usage**

Set the entity's Y position on-screen.

## **SetYRotate**

---

### **Syntax**

`setyrotate(float yr)`

### **Usage**

Set the entity's Y rotation (in degrees).

## **SetYScale**

---

### **Syntax**

`setyscale(float ys)`

### **Usage**

Set the entity's Y scale.

## **SetZPos**

---

### **Syntax**

`setzpos(float z)`

### **Usage**

Set the entity's Z position on-screen.

---

### SetZRotate

---

#### Syntax

```
setzrotate(float zr)
```

#### Usage

Set the entity's Z rotation (in degrees).

---

### SetZScale

---

#### Syntax

```
setzscale(float zs)
```

#### Usage

Set the entity's Z scale.

---

### Show

---

#### Syntax

```
show()
```

#### Usage

Add the entity to the program's render queue.

---

### Start

---

#### Syntax

```
start()
```

#### Usage

Add the entity to the program's update queue.

---

### Stop

---

#### Syntax

```
stop()
```

#### Usage

Remove the entity to the program's update queue.

---

### Tint

---

#### Syntax

```
tint(int t)
```

#### Usage

Set the entity's tint.

---

### Exception

An exception is an error that occurs during runtime. When an error happens, Liquid Studio "throws" an exception to the

task. The exception can be trapped by the task and acted upon. If error trapping is not enabled in the task, then the exception is reported to the user and the task automatically unloads.

Inherits from OBJECT.

---

## GetData

### Syntax

string:getdata ()

### Usage

Return any optional data passed along with the error code in the exception.

---

## GetError

### Syntax

int:geterror ()

### Usage

Return the error code of the exception.

Liquid Studio has several constants defined for various errors. Listed below are the exceptions in Liquid Studio (bolded exceptions are exceptions that can not be trapped, even with error trapping enabled):

VM_NONE	no reported exception
<b>VM_INTERNAL_ERROR</b>	an internal error has occurred
VM_USER	a user generated exception has occurred
<b>VM_OUT_OF_MEMORY</b>	Liquid Studio has run out of memory
VM_DENIED	the requested operation was denied
VM_ILLEGAL_QUANTITY	an illegal quantity was used
VM_NOT_DIMENSIONED	a collection was used before it was properly dimensioned
VM_BAD_SUBSCRIPT	a bad subscript in a collection was used
VM_KEY_NOT_FOUND	the specified key was not found in a collection
VM_DUPLICATE_KEY	the specified key already exists in a collection
VM_NOT_SORTED	a collection is not sorted
VM_BAD_ITEM	a bad item was used in a collection
VM_ITEM_NOT_FOUND	the specified item was not found in a collection
VM_DUPLICATE_ITEM	the specified item already exists in a collection
<b>VM_TIMEOUT</b>	Liquid Studio has timed out (stopped responding)
<b>VM_STACK_OVERFLOW</b>	the stack has overflowed (ran out of space)
VM_OVERFLOW	an int or float overflowed
VM_DIVISION_BY_ZERO	an int or float was divided by zero
VM_NULL_OBJECT	an attempt to use a Null (empty) object was made
VM_BAD_HANDLE	a bad handle was used
VM_BAD_FILE_MODE	a bad file mode was used
VM_FILE_NOT_FOUND	the specified file was not found
VM_FILE_OPEN	a file is already open
VM_FILE_NOT_OPEN	a file is not open
VM_NOT_BINARY_FILE	a binary operation was made on a file that is not a binary file
VM_NOT_INPUT_FILE	an input operation was made on a file that is not an input file
VM_NOT_OUTPUT_FILE	an output operation was made on a file that is not an output file
VM_BAD_FILE_FORMAT	a file is in a bad (unsupported) file format
VM_TCP_TIMEOUT	a TCP/IP operation timed out

VM_TCP_OPEN	a TCP/IP socket is already open
VM_TCP_NOT_OPEN	a TCP/IP socket is not open
VM_FTP_ERROR	an FTP specific error occurred, the exception data holds the error message
VM_DATABASE_ERROR	a database specific error occurred, the exception data holds the error message
VM_XML_ERROR	an XML specific error occurred, the exception data holds the error message

---

## GetFilename

### Syntax

```
string:getfilename()
```

### Usage

Return the filename of the program the error occurred in.

---

## GetLineNumber

### Syntax

```
int:getlinenumber()
```

### Usage

Return the line number where the error occurred.

---

## File

A file is simply a file stored on disk.

Inherits from OBJECT.

---

## Close

### Syntax

```
close()
```

### Usage

Close an open file.

---

## EOF

### Syntax

```
boolean:eof()
```

### Usage

Return TRUE if the end of the file has been reached, otherwise return FALSE.

### Exceptions

Throws VM\_FILE\_NOT\_OPEN if the file is not open.

Throws VM\_DENIED if the operation failed.

---

## Get

### Syntax

```
string:get(int length)
```

**Usage**

Return the specified number of bytes as a string from a binary file.

**Exceptions**

Throws `VM_FILE_NOT_OPEN` if the file is not open.

Throws `VM_NOT_BINARY_FILE` if the open file is not a binary file.

Throws `VM_DENIED` if the operation failed.

---

**GetFilename**

---

**Syntax**

```
string:getfilename()
```

**Usage**

Return the filename of the file.

---

**IsClosed**

---

**Syntax**

```
boolean:isclosed()
```

**Usage**

Return `TRUE` if the file is closed, otherwise return `FALSE`.

---

**IsOpen**

---

**Syntax**

```
boolean:isopen()
```

**Usage**

Return `TRUE` if the file is open, otherwise return `FALSE`.

---

**LOF**

---

**Syntax**

```
int:lof()
```

**Usage**

Return the length (in bytes) of the file.

**Exceptions**

Throws `VM_FILE_NOT_OPEN` if the file is not open.

Throws `VM_DENIED` if the operation failed.

---

**Open**

---

**Syntax**

```
open(string fn, string m)
```

**Usage**

Open a file named `fn` using mode `m`, where `m` is one of the following:

- `b`      open the file for binary
- `i`      open the file for input
- `o`      open the file for output

a      open the file for append  
r      open the file for random

### Exceptions

Throws VM\_FILE\_NOT\_FOUND if the file fn could not be opened for input.

Throws VM\_DENIED if the operation failed.

Throws VM\_FILE\_OPEN if the file is already opened.

## Put

---

### Syntax

```
put(string msg)
```

### Usage

Write the specified string to a binary file.

### Exceptions

Throws VM\_FILE\_NOT\_OPEN if the file is not open.

Throws VM\_NOT\_BINARY\_FILE if the open file is not a binary file.

Throws VM\_DENIED if the operation failed.

## Read

---

### Syntax

```
string:read()
```

### Usage

Read the next line in an open input file.

### Exceptions

Throws VM\_FILE\_NOT\_OPEN if the file is not open.

Throws VM\_NOT\_INPUT\_FILE if the open file is not an input file.

Throws VM\_DENIED if the operation failed.

## Seek

---

### Syntax

```
seek(int pos)
```

### Usage

Move the cursor within a file to pos. pos is specified in bytes.

### Exceptions

Throws VM\_FILE\_NOT\_OPEN if the file is not open.

Throws VM\_DENIED if the operation failed.

## Write

---

### Syntax

```
write(string)
```

### Usage

Write a line to an open output or append file.

### Exceptions

Throws VM\_FILE\_NOT\_OPEN if the file is not open.

Throws VM\_NOT\_OUTPUT\_FILE if the open file is not an output or append file.

Throws VM\_DENIED if the operation failed.

### WriteLn

---

#### Syntax

```
writeln(string)
```

#### Usage

Write a line followed by a carriage return to an open output or append file.

### Exceptions

Throws VM\_FILE\_NOT\_OPEN if the file is not open.

Throws VM\_NOT\_OUTPUT\_FILE if the open file is not an output or append file.

Throws VM\_DENIED if the operation failed.

### Fog

---

Fog is fog for use in a three-dimensional perspective scene (in PROGRAM).

Inherits from ENTITY3D.

### Constructor

---

#### Syntax

```
fog(float start, float end, float density)
```

#### Usage

Creates a new fog object, with the specified range (from start to end) and the specified density. By default, the fog has its filter set to GX\_EXP and its color set to gray.

### GetColor

---

#### Syntax

```
int:getcolor()
```

#### Usage

Return the fog's color.

### GetDensity

---

#### Syntax

```
int:getdensity()
```

#### Usage

Return the fog's density.

### GetEnd

---

#### Syntax

```
float:getend()
```

**Usage**

Return the fog's end range.

---

**GetFilter****Syntax**

```
int:getfilter()
```

**Usage**

Return the fog's filter.

---

**GetStart****Syntax**

```
float:getstart()
```

**Usage**

Return the fog's start range.

---

**IsTurnedOn****Syntax**

```
boolean:isturnedon()
```

**Usage**

Return TRUE if the fog is turned on, otherwise return FALSE.

---

**SetColor****Syntax**

```
setcolor(color c)
```

**Usage**

Set the fog's color.

---

**SetDensity****Syntax**

```
setdensity(float d)
```

**Usage**

Set the fog's density.

---

**SetFilter****Syntax**

```
setfilter(int f)
```

**Usage**

Set the fog's filter:

1	GX_EXP	fullscreen fog
---	--------	----------------

2	GX_EXP2	fullscreen fog, with added depth
3	GX_LINEAR	depth based fog

---

## SetRange

### Syntax

```
setrange(float start, float end)
```

### Usage

Define the fog's range, or the depth of the fog. The fog will appear from the depth start to the depth end.

---

## TurnOff

### Syntax

```
turnoff()
```

### Usage

Turn the fog off.

---

## TurnOn

### Syntax

```
turnon()
```

### Usage

Turn the fog on.

---

## Font

A font is a font.

Inherits from OBJECT.

---

## Constructor

### Syntax

```
font(string fontname, int pointsize, boolean bold, boolean italics, boolean underline)
```

### Usage

Build a font using Windows. fontname is the Windows TrueType font to load and pointsize is the point size of the font. If bold is TRUE the font is bolded. If italics is TRUE the font is italicized. If underline is TRUE the font is underlined.

### Exceptions

Throws VM\_DENIED if the operation failed.

### Syntax

```
font(string fontname, int pointsize)
```

### Usage

Build a font using the FreeType library. fontname is the Windows TrueType font to load and pointsize is the point size of the font.

### Exceptions

Throws VM\_FILE\_NOT\_FOUND if the file fontname does not exist.  
Throws VM\_ILLEGAL\_QUANTITY if pointsize is less than 4 or greater than 256.  
Throws VM\_DENIED if the operation failed.

---

## Cast

### Syntax

```
string:cast()
```

### Usage

Cast the font object to a string. The name of the font is returned.

---

## GetCharWidth

### Syntax

```
int:getcharwidth(int ch)
```

### Usage

Return the width (in pixels) of the specified character.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 0 or greater than 255.

---

## GetHeight

### Syntax

```
int:getheight()
```

### Usage

Return the height (in pixels) of the font.

---

## GetPointSize

### Syntax

```
int:getpointsize()
```

### Usage

Return the font's point size.

---

## GetStringWidth

### Syntax

```
int:getstringwidth(string msg)
```

### Usage

Return the width (in pixels) of the specified string.

---

## IsAvailable

### Syntax

```
boolean:isavailable(int ch)
```

### Usage

Return TRUE if the specified character is defined in the font.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 0 or greater than 255.

### IsBold

---

#### Syntax

```
boolean:isbold()
```

#### Usage

Return TRUE if the font is bolded.

### IsFixedWidth

---

#### Syntax

```
boolean:isfixedwidth()
```

#### Usage

Return TRUE if the font is fixed width. Return FALSE if the font is proportionally spaced.

### IsItalics

---

#### Syntax

```
boolean:isitalics()
```

#### Usage

Return TRUE if the font is italicized.

### IsUnderline

---

#### Syntax

```
boolean:isunderline()
```

#### Usage

Return TRUE if the font is underlined.

### Strip

---

#### Syntax

```
string:strip(string msg)
```

#### Usage

Strip out any characters in msg that are not defined in the font and return the results.

### Font3D

---

A Font3D is a three dimensional OpenGL font, for use in perspective scenes (in PROGRAM).

Inherits from OBJECT.

## Constructor

---

### Syntax

```
font3d(string fontname, int pointsize, boolean bold, boolean italics, float deviationlevel, float thickness)
```

### Usage

Create a new 3D font. fontname is a valid Windows font name, pointsize is the point size of the font. If bold is TRUE then the font is bolded. If italics is TRUE then the font is italicized. deviationlevel is the amount of deviation allowed from the font (the higher the number, the "sloppier" the font will look, however the lower the number the slower the font will render). thickness is how thick the font should be.

## GetCharHeight

---

### Syntax

```
float:getcharheight(int ch)
```

### Usage

Return the height of the specified character ch. ch must be between 32 and 127.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 32 or greater than 127.

## GetCharWidth

---

### Syntax

```
float:getcharwidth(int ch)
```

### Usage

Return the width of the specified character ch. ch must be between 32 and 127.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if ch is less than 32 or greater than 127.

## GetStringHeight

---

### Syntax

```
float:getstringheight(string msg)
```

### Usage

Return the height of the specified string msg.

## GetStringWidth

---

### Syntax

```
float:getstringwidth(string msg)
```

### Usage

Return the width of the specified string msg.

## FTP

FTP is a communications protocol used on the Internet to transfer files.

Inherits from OBJECT.

---

### Constructor

#### Syntax

```
ftp(string server, string user, string password)
```

#### Usage

Connect to an FTP server using the specified user name and password.

#### Exceptions

Throws VM\_FTP\_ERROR if the operation failed. The exception data holds the actual error message.

---

### ChDir

#### Syntax

```
chdir(string p)
```

#### Usage

Change to a new directory on the remote FTP server.

#### Exceptions

Throws VM\_FTP\_ERROR if the operation failed. The exception data holds the actual error message.

---

### CurDir

#### Syntax

```
string:curdir()
```

#### Usage

Return the current directory on the remote FTP server.

#### Exceptions

Throws VM\_FTP\_ERROR if the operation failed. The exception data holds the actual error message.

---

### GetDirList

#### Syntax

```
string:getdirlist()
```

#### Usage

Return a delimited string of all the directories in the current directory on the remote FTP server.

#### Exceptions

Throws VM\_FTP\_ERROR if the operation failed. The exception data holds the actual error message.

---

### GetFile

#### Syntax

```
getfile(string remotefile, string localfile)
```

#### Usage

Download remotefile from the remote FTP server and stores it locally as localfile.

**Exceptions**

Throws VM\_FTP\_ERROR if the operation failed. The exception data holds the actual error message.

**GetFileList**

---

**Syntax**

```
string:getfilelist()
```

**Usage**

Return a delimited string of all the files in the current directory on the remote FTP server.

**Exceptions**

Throws VM\_FTP\_ERROR if the operation failed. The exception data holds the actual error message.

**MkDir**

---

**Syntax**

```
mkdir(string p)
```

**Usage**

Create a directory on the remote FTP server.

**Exceptions**

Throws VM\_FTP\_ERROR if the operation failed. The exception data holds the actual error message.

**MoveFile**

---

**Syntax**

```
movefile(string source, string dest)
```

**Usage**

Rename or move a file on the remote FTP server. If dest does not exist, the file source is renamed. If dest exists and is a valid directory, the file source is moved.

**Exceptions**

Throws VM\_FTP\_ERROR if the operation failed. The exception data holds the actual error message.

**PutFile**

---

**Syntax**

```
putfile(string remotefile, string localfile)
```

**Usage**

Upload localfile to the remote FTP server and store it remotely as remotefile.

**Exceptions**

Throws VM\_FTP\_ERROR if the operation failed. The exception data holds the actual error message.

**RemoveFile**

---

**Syntax**

```
removefile(string f)
```

**Usage**

Remove (delete) the file `f` on the remote FTP server.

### Exceptions

Throws `VM_FTP_ERROR` if the operation failed. The exception data holds the actual error message.

## HashTable

A hash table is similar to an array but is indexed using a string key instead of an integer.

ACCESS	O(1) to O(n)
FIND	N/A
DELETE	O(1) to O(n)
INSERT	O(1) to O(n)

Inherits from `COLLECTION`.

### Clear

---

#### Syntax

```
clear()
```

#### Usage

Clear the hash table.

### Delete

---

#### Syntax

```
delete(string key)
```

#### Usage

Delete the item in the hash table linked to key.

#### Exceptions

Throws `VM_KEY_NOT_FOUND` if the key does not exist.

### Find

---

#### Syntax

```
boolean:find(string key)
```

#### Usage

Return `TRUE` if key exists in the hash table, otherwise returns `FALSE`.

### GetCount

---

#### Syntax

```
int:getcount()
```

#### Usage

Return the number of items in the hash table.

## Insert

---

### Syntax

```
insert(string key, item i)
```

### Usage

Insert the item *i* into the hash table and links item *i* to key.

### Exceptions

Throws VM\_DUPLICATE\_KEY if the key already exists.

Throws VM\_OUT\_OF\_MEMORY if the hash table is too big to fit into memory.

## Layer

A layer is similar to a "cel" used in traditional animation. Layers can be stacked, one on top of one another, in a console when it is in multiscreen mode. Each layer can be independently moved, tinted, blended, scaled, or rotated.

Inherits from ENTITY. Can be rendered in console in multiscreen mode.

## Light

A light is a light for use in a three-dimensional perspective scene (in PROGRAM).

Inherits from ENTITY3D.

## Constructor

---

### Syntax

```
light(float x, float y, float z, float w)
```

### Usage

Create a new light object. Coordinates are given in eye coordinates. By default, the light's ambient color is black, and the light's diffuse and specular color are white.

## GetAmbient

---

### Syntax

```
int:getambient()
```

### Usage

Return the light's ambient intensity.

## GetDiffuse

---

### Syntax

```
int:getdiffuse()
```

### Usage

Return the light's diffuse intensity.

## GetSpecular

---

### Syntax

```
int:getspecular()
```

**Usage**

Return the light's specular intensity.

**GetW**

---

**Syntax**

```
float:getw()
```

**Usage**

Return the W coordinate of the light.

**GetX**

---

**Syntax**

```
float:getx()
```

**Usage**

Return the X coordinate of the light.

**GetY**

---

**Syntax**

```
float:gety()
```

**Usage**

Return the Y coordinate of the light.

**GetZ**

---

**Syntax**

```
float:getz()
```

**Usage**

Return the Z coordinate of the light.

**IsTurnedOn**

---

**Syntax**

```
boolean:isturnedon()
```

**Usage**

Return TRUE if the light is turned on, otherwise return FALSE.

**SetAmbient**

---

**Syntax**

```
setambient(int c)
```

**Usage**

Set the light's ambient intensity.

## SetDiffuse

---

### Syntax

```
setdiffuse(int c)
```

### Usage

Set the light's diffuse intensity.

## SetPosition

---

### Syntax

```
setposition(float w, float x, float y, float z)
```

### Usage

Set the position of the light to (x, y, z). If w is zero, the light is treated as a directional source (lighting calculations take the direction of the light into account, and not its position). Otherwise, if w is not zero, the lighting calculations take both the direction and position of the light into account.

## SetSpecular

---

### Syntax

```
setspecular(int c)
```

### Usage

Set the light's specular intensity.

## TurnOff

---

### Syntax

```
turnoff()
```

### Usage

Turn the light off.

## TurnOn

---

### Syntax

```
turnon()
```

### Usage

Turn the light on.

## List

A list is used to store objects in nodes. Each node has a pointer to the previous and next node in the list.

ACCESS	O(n)
FIND	O(n) if unsorted, O(log(n)) if sorted
DELETE	O(1)
INSERT	O(1)

Can be used with the FOR EACH statement.

Inherits from COLLECTION.

---

### AddHead

#### Syntax

```
addhead(item i)
```

#### Usage

Add item *i* to the head (beginning) of the list.

---

### AddTail

#### Syntax

```
addtail(item i)
```

#### Usage

Add item *i* to the tail (end) of the list.

---

### Clear

#### Syntax

```
clear()
```

#### Usage

Clear the list.

---

### Contains

#### Syntax

```
boolean:contains(item i)
```

#### Usage

Return TRUE if item *i* is in the list, otherwise return FALSE.

---

### Delete

#### Syntax

```
delete(int inx)
```

#### Usage

Delete the item at index *inx*.

#### Exceptions

Throws VM\_BAD\_SUBSCRIPT if the index *inx* is less than 0 or greater than *s*, where *s* is the size of the list.

---

### Enqueue

#### Syntax

```
enqueue(item i)
```

#### Usage

Enqueue item *i* in the list. When an object is enqueued in a list it is sorted, whereas when an object is added it is not sorted.

**Exceptions**

Throws VM\_NOT\_SORTED if the list is not sorted.

**Find**

---

**Syntax**

```
int:find(item i)
```

**Usage**

Return the index where item *i* resides, or -1 if the item can't be found.

**GetCount**

---

**Syntax**

```
int:getcount ()
```

**Usage**

Return the number of items in the list.

**GetHead**

---

**Syntax**

```
item:gethead ()
```

**Usage**

Return the item at the head (beginning) of the list.

**Exceptions**

Throws VM\_ITEM\_NOT\_FOUND if the list is empty.

**GetTail**

---

**Syntax**

```
item:gettail ()
```

**Usage**

Return the item at the tail (end) of the list.

**Exceptions**

Throws VM\_ITEM\_NOT\_FOUND if the list is empty.

**Insert**

---

**Syntax**

```
insert(int inx, item i)
```

**Usage**

Insert item *i* into the list at index *inx*.

**Exceptions**

Throws VM\_BAD\_SUBSCRIPT if the index *inx* is less than 0 or greater than *s*, where *s* is the size of the single link list.

## IsEmpty

---

### Syntax

```
boolean: isempty()
```

### Usage

Return TRUE if the list is empty, otherwise return FALSE.

## Locate

---

### Syntax

```
int: locate(item i)
```

### Usage

Locate item *i* in the list and return its location, or -1 if the item can't be located.

## MoveItemDown

---

### Syntax

```
moveitemdown(item i)
```

### Usage

Move item *i* down the list.

## MoveItemToHead

---

### Syntax

```
moveitemtohead(item i)
```

### Usage

Move item *i* to the head (beginning) of the list.

## MoveItemToTail

---

### Syntax

```
moveitemtotail(item i)
```

### Usage

Move item *i* to the tail (end) of the list.

## MoveItemUp

---

### Syntax

```
moveitemup(item i)
```

### Usage

Move item *i* up the list.

## Remove

---

### Syntax

```
remove(item i)
```

### Usage

Remove item *i* from the list.

---

### RemoveHead

---

#### Syntax

`item:removehead()`

#### Usage

Remove and return the item at the head (beginning) of the list.

#### Exceptions

Throws `VM_DENIED` if the list is empty.

---

### RemoveTail

---

#### Syntax

`item:removetail()`

#### Usage

Remove and return the item at the tail (end) of the list.

#### Exceptions

Throws `VM_DENIED` if the list is empty.

---

### Sort

---

#### Syntax

`sort()`

#### Usage

Sort the list.

---

### Matrix

---

A matrix is a two dimensional array. It uses two indexes to access an element in a matrix.

ACCESS	O(1)
FIND	N/A
DELETE	N/A
INSERT	N/A

Inherits from `COLLECTION`.

---

### Add

---

#### Syntax

`add(matrix m1, matrix m2)`

#### Usage

Add matrix *m1* and matrix *m2* together. *m1* and *m2* must be either integer or float matrices.

#### Exceptions

Throws `VM_NOT_DIMENSIONED` if the matrix has not been dimensioned yet.

Throws `VM_DENIED` if matrix `m1` or `m2` is not an integer or float matrix, or if matrix `m1` and `m2` and the matrix object are not the same size.

---

## Clear

### Syntax

```
clear()
```

### Usage

Clear the matrix.

### Exceptions

Throws `VM_NOT_DIMENSIONED` if the matrix has not been dimensioned yet.

---

## Dim

### Syntax

```
dim(int x, int y)
```

### Usage

Dimension the matrix to `x` elements across and `y` elements down.

### Exceptions

Throws `VM_DENIED` if the matrix has already been dimensioned.

Throws `VM_ILLEGAL_QUANTITY` if `x` is less than 0 or `y` is less than 0.

Throws `VM_OUT_OF_MEMORY` if the matrix is too big to fit into memory.

---

## Exchange

### Syntax

```
exchange(int x1, int y1, int x2, int y2)
```

### Usage

Exchange the item at index `(x1, y1)` with the item at index `(x2, y2)`.

### Exceptions

Throws `VM_NOT_DIMENSIONED` if the matrix has not been dimensioned yet.

Throws `VM_BAD_SUBSCRIPT` if the index `x1` or `x2` is less than 0 or greater than the width of the array, or if the index `y1` or `y2` is less than 0 or greater than the height of the array.

---

## Fill

### Syntax

```
fill(item i)
```

### Usage

Fill the matrix with item `i`.

### Exceptions

Throws `VM_NOT_DIMENSIONED` if the matrix has not been dimensioned yet.

---

## GetHeight

### Syntax

`int: getheight()`

**Usage**

Return the height of the matrix.

---

**GetWidth**

**Syntax**

`int: getwidth()`

**Usage**

Return the width of the matrix.

---

**Identity**

**Syntax**

`identity()`

**Usage**

Create an identity matrix. The matrix must be either a square integer or square float matrix.

**Exceptions**

Throws `VM_NOT_DIMENSIONED` if the matrix has not been dimensioned yet.

Throws `VM_DENIED` if the matrix is not a square integer or square float matrix.

---

**Invert**

**Syntax**

`invert(matrix m)`

**Usage**

Invert the matrix. The matrix must be a square float matrix.

**Exceptions**

Throws `VM_NOT_DIMENSIONED` if the matrix has not been dimensioned yet.

Throws `VM_DENIED` if the matrix is not a square float matrix, or if matrix `m` and the matrix object are not the same size.

---

**Multiply**

**Syntax**

`multiply(matrix m1, matrix m2)`

**Usage**

Multiply matrix `m1` by matrix `m2`. `m1` and `m2` must be float matrices.

**Exceptions**

Throws `VM_NOT_DIMENSIONED` if the matrix has not been dimensioned yet.

Throws `VM_DENIED` if matrix `m1` or `m2` is not a float matrix, or if matrix `m1` and `m2` and the matrix object are not the same size.

---

**ReDim**

**Syntax**

`redim(int x, int y)`

**Usage**

Redimension an already dimensioned matrix to x elements across and y elements down.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if x is less than 0 or y is less than 0.

Throws VM\_OUT\_OF\_MEMORY if the matrix is too big to fit into memory.

**ReDimPreserve**

---

**Syntax**

```
redimpreserve(int x, int y)
```

**Usage**

Redimension an already dimensioned matrix to x elements across and y elements down while preserving the original contents of the matrix.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if x is less than 0 or y is less than 0.

Throws VM\_OUT\_OF\_MEMORY if the matrix is too big to fit into memory.

**ScalarMultiply**

---

**Syntax**

```
scalarmultiply(matrix m, float f)
```

**Usage**

Multiply matrix m by the scalar float f.

**Exceptions**

Throws VM\_NOT\_DIMENSIONED if the matrix has not been dimensioned yet.

Throws VM\_DENIED if matrix m is not a float matrix, or if matrix m and the matrix object are not the same size.

**Subtract**

---

**Syntax**

```
subtract(matrix m1, matrix m2)
```

**Usage**

Subtract matrix m2 from matrix m1. m1 and m2 must be either integer or float matrices.

**Exceptions**

Throws VM\_NOT\_DIMENSIONED if the matrix has not been dimensioned yet.

Throws VM\_DENIED if matrix m1 or m2 is not an integer or float matrix, or if matrix m1 and m2 and the matrix object are not the same size.

**Transpose**

---

**Syntax**

```
transpose(matrix m)
```

**Usage**

Transpose matrix m. m must be either an integer or float matrix.

## Exceptions

Throws `VM_NOT_DIMENSIONED` if the matrix has not been dimensioned yet.

Throws `VM_DENIED` if matrix `m` is not an integer or float matrix, or if matrix `m` and the matrix object are not the same size.

---

## UnDim

### Syntax

```
undim()
```

### Usage

Undimension a matrix.

---

## Message

Encapsulate communication between two objects.

Inherits from `OBJECT`.

---

## GetBody

### Syntax

```
int:getbody()
```

### Usage

Return the message body. Liquid Studio has several predefined message bodies:

<code>MSG_EXIT</code>	the user has ended the program
<code>MSG_KEYDOWN</code>	a key is down
<code>MSG_KEYUP</code>	a key (that was down) is now up
<code>MSG_KEYPRESSED</code>	a key has been pressed (down and up)
<code>MSG_MOUSEMOVE</code>	the mouse has moved
<code>MSG_MOUSEOVER</code>	the mouse is over an entity
<code>MSG_MOUSEDOWN</code>	the mouse button is down
<code>MSG_MOUSEUP</code>	the mouse button (that was down) is now up
<code>MSG_CLICKED</code>	the mouse has been clicked (down and up)
<code>MSG_DBLCLICKED</code>	the mouse has been double clicked
<code>MSG_MOUSEWHEEL</code>	the mouse wheel has changed

Unique constants for additional message bodies can be defined using the `CONST` keyword.

---

## GetData

### Syntax

```
string:getdata()
```

### Usage

Return the message data.

---

## GetTimeStamp

### Syntax

```
int:gettimestamp()
```

**Usage**

Return the time (according to the atomic clock) the message was sent.

**IsFrom**

---

**Syntax**

```
boolean:isfrom(object d)
```

**Usage**

Return TRUE if the message was sent by object d.

**IsTo**

---

**Syntax**

```
boolean:isto(object d)
```

**Usage**

Return TRUE if the message is addressed to object d.

**Object**

---

All objects derive from the class Object.

**GetClass**

---

**Syntax**

```
string:getclass()
```

**Usage**

Return the name of the class the object belongs to.

**GetTag**

---

**Syntax**

```
string:gettag()
```

**Usage**

Return an object's "tag" (name).

**Tag**

---

**Syntax**

```
tag(string name)
```

**Usage**

"Tag" the object with the given name.

**PriorityQueue**

---

A priority queue object is similar to a queue, but places the "largest" object at the beginning of the queue.

Inherits from COLLECTION.

## Clear

---

### Syntax

`clear()`

### Usage

Clear the priority queue.

## Dequeue

---

### Syntax

`item: dequeue()`

### Usage

Dequeue the item at the front of the priority queue and return it.

## Enqueue

---

### Syntax

`enqueue(item i)`

### Usage

Enqueue item *i* into the priority queue.

## GetCount

---

### Syntax

`int: getcount()`

### Usage

Return the number of items in the priority queue.

## IsEmpty

---

### Syntax

`boolean: isempty()`

### Usage

Return TRUE if the priority queue is empty, otherwise return FALSE.

## Peek

---

### Syntax

`item: peek()`

### Usage

Return the item at the front of the priority queue.

## PriorityQueueEx

An extended priority queue object is similar to a queue, but places the "largest" object at the beginning of the queue. Objects can change their priority, causing the extended priority queue to reshuffle so the largest object is always at the beginning of

the queue. An object can also be removed from the extended priority queue, regardless of where it is in the queue.

Inherits from COLLECTION.

---

### **ChangePriority**

---

#### **Syntax**

`changepriority(int node)`

#### **Usage**

Reshuffle the extended priority queue after an individual node has changed its priority.

#### **Exceptions**

Throws VM\_BAD\_HANDLE if node is not a valid node handle.

---

### **Clear**

---

#### **Syntax**

`clear()`

#### **Usage**

Clear the priority queue.

---

### **Dequeue**

---

#### **Syntax**

`item:dequeue()`

#### **Usage**

Dequeue the item at the front of the priority queue and return it.

---

### **Enqueue**

---

#### **Syntax**

`int:enqueue(item i)`

#### **Usage**

Enqueue item i into the priority queue and return its node.

---

### **GetCount**

---

#### **Syntax**

`int:getcount()`

#### **Usage**

Return the number of items in the priority queue.

---

### **IsEmpty**

---

#### **Syntax**

`boolean:isempty()`

#### **Usage**

Return TRUE if the priority queue is empty, otherwise return FALSE.

## Peek

---

### Syntax

```
item:peek()
```

### Usage

Return the item at the front of the priority queue.

## Remove

---

### Syntax

```
remove(int node)
```

### Usage

Remove an individual node from the extended priority queue.

### Exceptions

Throws VM\_BAD\_HANDLE if node is not a valid node handle.

## Process

---

A process is a task that has access to the file system.

Inherits from TASK.

## Arg

---

### Syntax

```
string:arg(int n)
```

### Usage

Return argument n in the command line as a string.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if n is less than 1 or greater than the number of arguments passed in the command line.

### Example

```
myprog.ldx -n "InFile.txt" OutFile.txt
```

arg(1) returns -n

arg(2) returns InFile.txt

arg(3) returns OutFile.txt

## ArgCount

---

### Syntax

```
int:argcount()
```

### Usage

Return the number of arguments passed in the command line.

**Example**

```
myprog.ldx -n "InFile.txt" OutFile.txt
```

argcount() returns 3

**Chain**

---

**Syntax**

```
chain(string ldx)
```

**Usage**

Run a compiled Liquid executable in a new window.

**Exceptions**

Throws VM\_FILE\_NOT\_FOUND if the compiled Liquid executable does not exist.

**ChDir**

---

**Syntax**

```
chdir(string d)
```

**Usage**

Change the current directory to d.

**Exceptions**

Throws VM\_PATH\_NOT\_FOUND if the directory d does not exist.

**ChDrive**

---

**Syntax**

```
chdrive(string d)
```

**Usage**

Change the current drive to d.

**Exceptions**

Throws VM\_DENIED if the drive d does not exist.

**CommandLine**

---

**Syntax**

```
string:commandline()
```

**Usage**

Return the command line passed to the thread.

**CopyFile**

---

**Syntax**

```
copyfile(string f1, string f2)
```

**Usage**

Copy file f1 to f2, which can either be a path to a directory or a new filename.

**Exceptions**

Throws VM\_FILE\_NOT\_FOUND if the file f1 does not exist.

Throws VM\_DENIED if the operation failed.

---

**CurDir****Syntax**

string:curdir()

**Usage**

Return the current directory.

---

**Exist****Syntax**

boolean:exist(string f)

**Usage**

Return TRUE if file f exists, otherwise returns FALSE.

---

**GetDirList****Syntax**

string:getdirlist(string d)

**Usage**

Return a delimited string list of sub-directories (name only) in directory d.

---

**GetDriveList****Syntax**

string:getdrivelist(string d)

**Usage**

Return a delimited string list of the available drives.

---

**GetDriveType****Syntax**

string:getdrivetype(int d)

**Usage**

Return the type of drive d, the ASCII character of the drive (A thru Z): Unknown drive, Removable drive, Fixed drive, Network drive, CDROM drive, or RAM disk drive.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if d is less than 65 ('A') or greater than 90 ('Z').

Throws VM\_DENIED if the operation failed.

---

**GetFileDate****Syntax**

string:getfiledate(string f)

**Usage**

Return file f's file date.

**Exceptions**

Throws VM\_FILE\_NOT\_FOUND if the file f does not exist.

---

**GetFileExtension**

---

**Syntax**

```
string:getfileextension(string f)
```

**Usage**

Return the file extension of file f.

---

**GetFileList**

---

**Syntax**

```
string:getfilelist(string d)
```

**Usage**

Return a delimited string list of files (name only) in directory d.

---

**GetFilename**

---

**Syntax**

```
string:getfilename(string f)
```

**Usage**

Return the filename (without the path) of string f.

---

**GetFilepath**

---

**Syntax**

```
string:getfilepath(string f)
```

**Usage**

Return the path (without the filename) of string f.

---

**GetFileSize**

---

**Syntax**

```
int:getfilesize(string f)
```

**Usage**

Return the size (in bytes) of file f.

**Exceptions**

Throws VM\_FILE\_NOT\_FOUND if the file f does not exist.

---

**GetFileSubType**

---

**Syntax**

```
string:getfilesubtype (string f)
```

**Usage**

Return a detailed description of file f based on f's file extension.

**GetFileTime**

---

**Syntax**

```
string:getfiletime (string f)
```

**Usage**

Return file f's file time.

**Exceptions**

Throws VM\_FILE\_NOT\_FOUND if the file f does not exist.

**GetFileType**

---

**Syntax**

```
string:getfiletype (string f)
```

**Usage**

Return "Folder" if f is a directory. Otherwise returns a description of file f based on f's file extension.

**GetHomeDir**

---

**Syntax**

```
string:gethomedir ()
```

**Usage**

Return the home directory. The home directory is the directory where the compiled Liquid executable resides.

**GetLineCount**

---

**Syntax**

```
int:getlinecount (string f)
```

**Usage**

Return the number of lines in file f.

**Exceptions**

Throws VM\_FILE\_NOT\_FOUND if the file f does not exist.

**GetVerboseDirList**

---

**Syntax**

```
string:getverboosedirlist (string d)
```

**Usage**

Return a delimited string list of sub-directories (full path and name) in directory d.

## GetVerboseFileList

---

### Syntax

`string:getverbosefilelist(string d)`

### Usage

Return a delimited string list of files (full path and name) in directory d.

## MkDir

---

### Syntax

`mkdir(string d)`

### Usage

Create a new directory named d.

### Exceptions

Throws VM\_PATH\_ERROR if the directory d already exists.

Throws VM\_DENIED if the operation failed.

## MoveFile

---

### Syntax

`movefile(string f1, string f2)`

### Usage

Move file f1 to directory f2, or rename file f1 to file f2.

### Exceptions

Throws VM\_FILE\_NOT\_FOUND if the file f1 does not exist.

Throws VM\_DENIED if the operation failed.

## OpenBrowser

---

### Syntax

`openbrowser(string url)`

### Usage

Open the web page url in the user's default web browser.

## OpenMedia

---

### Syntax

`openmedia(string m)`

### Usage

Open a file using the Windows application registered to file m's file extension.

### Exceptions

Throws VM\_FILE\_NOT\_FOUND if the media does not exist.

## ReadEntireFile

---

### Syntax

```
string:readentirefile(string f)
```

**Usage**

Return the entire file *f* as a string.

**Exceptions**

Throws `VM_FILE_NOT_FOUND` if the file *f* does not exist.

---

**RemoveFile**

---

**Syntax**

```
removefile(string f1)
```

**Usage**

Remove file *f1*.

**Exceptions**

Throws `VM_FILE_NOT_FOUND` if the file *f1* does not exist.  
Throws `VM_DENIED` if the operation failed.

---

**Rmdir**

---

**Syntax**

```
rmdir(string d)
```

**Usage**

Remove directory *d*.

**Exceptions**

Throws `VM_PATH_ERROR` if the directory *d* does not exist.  
Throws `VM_DENIED` if the operation failed.

---

**StripFileExtension**

---

**Syntax**

```
string:stripfileextension(string f)
```

**Usage**

Return the name of file *f* with the extension stripped.

---

**WriteEntireFile**

---

**Syntax**

```
writeentirefile(string f, string data)
```

**Usage**

Write the string *data* to file *f*.

**Exceptions**

Throws `VM_DENIED` if the operation failed.

---

**Queue**

---

A queue stores objects in a First In First Out fashion.

Inherits from COLLECTION.

---

### **Clear**

#### **Syntax**

`clear()`

#### **Usage**

Clear the queue.

---

### **Dequeue**

#### **Syntax**

`item:pop()`

#### **Usage**

Dequeue the item at the front of the queue and returns it.

---

### **Enqueue**

#### **Syntax**

`enqueue(item i)`

#### **Usage**

Enqueue item *i* onto the back of the queue.

---

### **GetCount**

#### **Syntax**

`int:getcount()`

#### **Usage**

Return the number of items in the queue.

---

### **IsEmpty**

#### **Syntax**

`boolean:isempty()`

#### **Usage**

Return TRUE if the queue is empty, otherwise returns FALSE.

---

### **Peek**

#### **Syntax**

`item:peek()`

#### **Usage**

Return the item at the front of the queue.

## RecordSet

Record sets are used to hold data returned from a database query. A record set is a collection of fields, arranged in a table. The table is made up of rows of columns. An internal cursor is maintained to point to the current row of the record set. Individual columns within the row can then be read.

### Constructor

---

#### Syntax

```
recordset(database db, string query)
```

#### Usage

Submit a query to the database and create a record set to hold the results.

#### Exceptions

Throws VM\_FILE\_NOT\_OPEN if the SQL database has not been opened yet.

Throws VM\_DATABASE\_ERROR if the operation failed. The exception data holds the actual error message.

### BOF

---

#### Syntax

```
boolean:bof()
```

#### Usage

Return TRUE if the cursor is before the beginning of the table.

### EOF

---

#### Syntax

```
boolean:eof()
```

#### Usage

Return TRUE if the cursor is after the end of the table.

### GetField

---

#### Syntax

```
string:getfield(int n)
```

#### Usage

Return the data stored in the current row at the  $n^{\text{th}}$  column.

#### Exceptions

Throws VM\_DENIED if the cursor is before the first row or after the last row.

Throws VM\_BAD\_SUBSCRIPT if  $n$  is less than 1 or  $n$  is greater than the number of columns in the record set.

#### Syntax

```
string:getfield(string key)
```

#### Usage

Return the data stored in the current row at the column named `key`.

#### Exceptions

Throws VM\_DENIED if the cursor is before the first row or after the last row.

Throws VM\_BAD\_SUBSCRIPT if the `key` is not a valid column name.

## GetColumnCount

---

### Syntax

`int:getcolumncount ()`

### Usage

Return the number of columns in the record set.

## GetColumnName

---

### Syntax

`string:getcolumnname (int n)`

### Usage

Return the name of the  $n^{\text{th}}$  column.

### Exceptions

Throws `VM_BAD_SUBSCRIPT` if  $n$  is less than 1 or  $n$  is greater than the number of columns in the record set.

## GetRowCount

---

### Syntax

`int:getrowcount ()`

### Usage

Return the number of rows in the record set.

## MoveFirst

---

### Syntax

`movefirst ()`

### Usage

Move the cursor to the first row.

## MoveLast

---

### Syntax

`movelast ()`

### Usage

Move the cursor to the last row.

## MoveNext

---

### Syntax

`movenext ()`

### Usage

Move the cursor to the next row.

## MovePrev

---

### Syntax

`moveprev()`

### Usage

Move the cursor to the previous row.

## MoveTo

---

### Syntax

`moveto(int n)`

### Usage

Move the cursor to the  $n^{\text{th}}$  row.

### Exceptions

Throws `VM_BAD_SUBSCRIPT` if  $n$  is less than 1 or  $n$  is greater than the number of rows in the record set.

## RedBlackTree

---

A Red Black Tree is a binary tree that is always balanced.

Inherits from `BINARYSEARCHTREE`.

## Clear

---

### Syntax

`clear()`

### Usage

Clear the red black tree.

## Delete

---

### Syntax

`delete(item i)`

### Usage

Delete item  $i$  from the red black tree.

### Exceptions

Throws `VM_DENIED` if the node to delete has two children.

Throws `VM_ITEM_NOT_FOUND` if the item to delete could not be found.

## Find

---

### Syntax

`boolean:find(item i)`

### Usage

Return `TRUE` if item  $i$  exists in the red black tree, otherwise return `FALSE`.

## Insert

---

### Syntax

```
insert(item i)
```

### Usage

Insert item *i* into the red black tree.

### Exceptions

Throws `VM_DUPLICATE_ITEM` if the item is already in the splay tree.

## RNG

---

Generates random numbers.

Inherits from `OBJECT`.

## Constructor

---

### Syntax

```
rng(int seed)
```

### Usage

Set the random number generator's initial seed. If seed is less than or equal to zero then the atomic clock is used as a seed. The random number generator will always produce the exact same random sequence when seeded with a positive number.

## GetSeed

---

### Syntax

```
int:getseed()
```

### Usage

Return the random number generator's seed.

## Range

---

### Syntax

```
int:range(int lower, int upper)
```

### Usage

Return a random integer between lower and upper. Negative numbers can be used. lower is automatically swapped with upper if lower is greater than upper.

### Syntax

```
float:range(float lower, float upper)
```

### Usage

Return a random float between lower and upper. Negative numbers can be used. lower is automatically swapped with upper if lower is greater than upper.

## Rnd

---

### Syntax

```
float:rnd()
```

**Usage**

Return a random number between 0 and 1.

**SetSeed**

---

**Syntax**

```
setseed(int seed)
```

**Usage**

Set the random number generator's seed.

**Shape**

---

A shape is a geometric shape, such as a rectangle or ellipse. Shapes can be hollow or filled.

Inherits from ENTITY. Can be rendered in console in any mode.

**Constructor**

---

**Syntax**

```
shape(int t, int x1, int y1, int x2, int y2)
```

**Usage**

Create a new shape. t specifies the type of shape. The shape is bound by the rectangle from (x1, y1) to (x2, y2).

1	SHP_LINE	line
2	SHP_RECT	rectangle
3	SHP_RRECT	rounded rectangle
4	SHP_ELLIPSE	ellipse

**GetFillColor**

---

**Syntax**

```
int:getfillcolor()
```

**Usage**

Return the fill color of the shape (default is 0, meaning the shape is hollow).

**GetLineColor**

---

**Syntax**

```
int:getlinecolor()
```

**Usage**

Return the line color of the shape.

**GetLineWidth**

---

**Syntax**

```
int:getlinewidth()
```

**Usage**

Return the line width.

---

### SetCoords

#### Syntax

```
setcoords(int x1, int y1, int x2, int y2)
```

#### Usage

Set the shape's bounding box to the rectangle from (x1, y1) to (x2, y2).

---

### SetFillColor

#### Syntax

```
setfillcolor(int)
```

#### Usage

Set the fill color of the shape.

---

### SetLineColor

#### Syntax

```
setlinecolor(int)
```

#### Usage

Set the line color of the shape.

---

### SetLineWidth

#### Syntax

```
setlinewidth(int)
```

#### Usage

Set the line width.

---

## SingleLinkedList

A single link list stores objects in nodes. Each node has a pointer to the next node in the list.

ACCESS	O(n)
FIND	O(n) if unsorted, O(log(n)) if sorted
DELETE	O(1)
INSERT	O(1)

Inherits from COLLECTION.

---

### Add

#### Syntax

```
int:add(item i)
```

#### Usage

Add item *i* to the single link list. Return the index of the added item.

---

## Clear

### Syntax

```
clear()
```

### Usage

Clear the single link list.

---

## Contains

### Syntax

```
boolean:contains(item i)
```

### Usage

Return TRUE if item *i* is in the single link list, otherwise return FALSE.

---

## Delete

### Syntax

```
delete(int inx)
```

### Usage

Delete the item at index *inx*.

### Exceptions

Throws VM\_BAD\_SUBSCRIPT if the index *inx* is less than 0 or greater than *s*, where *s* is the size of the single link list.

---

## DeleteAtCursor

### Syntax

```
deleteatcursor()
```

### Usage

Delete the item at the cursor.

### Exceptions

Throws VM\_DENIED if the cursor is on the head node.

---

## Enqueue

### Syntax

```
enqueue(item i)
```

### Usage

Enqueue item *i* into the sorted single link list.

### Exceptions

Throws VM\_NOT\_SORTED if the single link list is not sorted.

## Find

---

### Syntax

```
int:find(item i)
```

### Usage

Find item *i* in the single link list and return its location, or  $-1$  if the item can't be found.

## First

---

### Syntax

```
item:first()
```

### Usage

Return the first item in the single link list.

### Exceptions

Throws `VM_ITEM_NOT_FOUND` if the single link list is empty.

## GetCount

---

### Syntax

```
int:getcount()
```

### Usage

Return the number of items in the single link list.

## Insert

---

### Syntax

```
insert(int inx, item i)
```

### Usage

Insert item *i* into the single link list at index *inx*.

### Exceptions

Throws `VM_BAD_SUBSCRIPT` if the index *inx* is less than 0 or greater than *s*, where *s* is the size of the single link list.

## InsertAtCursor

---

### Syntax

```
insert(item i)
```

### Usage

Insert item *i* at the cursor.

## InsertionSort

---

### Syntax

```
insertionsort()
```

### Usage

Sort the single link list using an insertion sort.

## IsAfterLast

---

### Syntax

boolean:isafterlast ()

### Usage

Return TRUE if the cursor is after the last item in the single link list, otherwise return FALSE.

## IsBeforeFirst

---

### Syntax

boolean:isbeforefirst ()

### Usage

Return TRUE if the cursor is before the first item in the single link list, otherwise return FALSE.

## IsEmpty

---

### Syntax

boolean:isempty ()

### Usage

Return TRUE if the single link list is empty, otherwise return FALSE.

## Last

---

### Syntax

item:last ()

### Usage

Return the last item in the single link list.

### Exceptions

Throws VM\_ITEM\_NOT\_FOUND if the single link list is empty.

## Locate

---

### Syntax

int:locate (item i)

### Usage

Locate item i in the single link list and return its location, or -1 if the item can't be located.

## MoveBeforeFirst

---

### Syntax

movebeforefirst ()

### Usage

Move the cursor before the first item in the single link list.

## MoveNext

---

### Syntax

`movenext ()`

**Usage**

Move the cursor to the next node in the single link list.

**Peek**

---

**Syntax**

`item:peek ()`

**Usage**

Return the item at the cursor.

**Exceptions**

Throws `VM_DENIED` if the single link list is empty.

**Remove**

---

**Syntax**

`remove (item i)`

**Usage**

Remove item `i` from the single link list.

**Sort**

---

**Syntax**

`sort ()`

**Usage**

Sort the single link list.

**SkipList**

A skip list stores objects in nodes. Each node has a pointer to the prior node in the list and the next node in the list, as well as several pointers to nodes further down the list. These additional pointers act as "expressways" when traversing the list.

ACCESS	N/A
FIND	$O(\log(n))$
DELETE	$O(1)$
INSERT	$O(1)$

Inherits from `COLLECTION`.

**Add**

---

**Syntax**

`add (item i)`

**Usage**

Add item `i` to the skip list.

### Exceptions

Throws VM\_DUPLICATE\_ITEM if the item is already in the skip list.

### Clear

---

#### Syntax

```
clear ()
```

#### Usage

Clear the skip list.

### Delete

---

#### Syntax

```
delete ()
```

#### Usage

Delete the item at the cursor.

### Exceptions

Throws VM\_DENIED if the cursor is on the head node or the tail node.

### Find

---

#### Syntax

```
boolean:find(item i)
```

#### Usage

Move the cursor to item i and return TRUE if item i is in the skip list, otherwise return FALSE.

### GetCount

---

#### Syntax

```
int:getcount ()
```

#### Usage

Return the number of items in the skip list.

### IsAfterLast

---

#### Syntax

```
boolean:isafterlast ()
```

#### Usage

Return TRUE if the cursor is after the last item in the skip list, otherwise return FALSE.

### IsBeforeFirst

---

#### Syntax

```
boolean:isbeforefirst ()
```

#### Usage

Return TRUE if the cursor is before the first item in the skip list, otherwise return FALSE.

## **IsEmpty**

---

### **Syntax**

`boolean: isempty ()`

### **Usage**

Return TRUE if the skip list is empty, otherwise return FALSE.

## **MoveAfterLast**

---

### **Syntax**

`moveafterlast ()`

### **Usage**

Move the cursor after the last item in the skip list.

## **MoveBeforeFirst**

---

### **Syntax**

`movebeforefirst ()`

### **Usage**

Move the cursor before the first item in the skip list.

## **MoveNext**

---

### **Syntax**

`movenext ()`

### **Usage**

Move the cursor to the next node in the skip list.

## **MovePrev**

---

### **Syntax**

`moveprev ()`

### **Usage**

Move the cursor to the previous node in the skip list.

## **Peek**

---

### **Syntax**

`item: peek ()`

### **Usage**

Return the item at the cursor.

### **Exceptions**

Throws VM\_DENIED if the skip list is empty.

## Remove

---

### Syntax

`remove(item i)`

### Usage

Remove item *i* from the skip list.

### Exceptions

Throws `VM_ITEM_NOT_FOUND` if the item to remove could not be found.

## SplayTree

Inherits from `BINARYSEARCHTREE`.

## Clear

---

### Syntax

`clear()`

### Usage

Clear the splay tree.

## Delete

---

### Syntax

`delete(item i)`

### Usage

Delete item *i* from the splay tree.

### Exceptions

Throws `VM_DENIED` if the node to delete has two children.

Throws `VM_ITEM_NOT_FOUND` if the item to delete could not be found.

## Find

---

### Syntax

`boolean:find(item i)`

### Usage

Return `TRUE` if item *i* exists in the splay tree, otherwise return `FALSE`.

## Insert

---

### Syntax

`insert(item i)`

### Usage

Insert item *i* into the splay tree.

### Exceptions

Throws `VM_DUPLICATE_ITEM` if the item is already in the splay tree.

## Sprite

A sprite is a two dimensional bitmap that can be displayed and moved around the screen.

Inherits from ENTITY. Can be rendered in console in any mode.

---

### Constructor

#### Syntax

```
sprite(texture t)
```

#### Usage

Create a new sprite using texture t. The sprite is sized to the size of the texture.

#### Syntax

```
sprite(texture t, int width, int height)
```

#### Usage

Create a new sprite using texture t. width is the width (in pixels) of the sprite. height is the height (in pixels) of the sprite. Note that the texture itself is not resized, only the sprite is resized (allowing you to have multiple sprites of all different sizes using the same texture).

---

### GetHeight

#### Syntax

```
int:getheight()
```

#### Usage

Return the height of the sprite (not the texture).

---

### GetTexture

#### Syntax

```
texture:gettexture()
```

#### Usage

Return the texture being used by the sprite.

---

### GetWidth

#### Syntax

```
int:getwidth()
```

#### Usage

Return the width of the sprite (not the texture).

---

### Hit

#### Syntax

```
boolean:hit(sprite s)
```

#### Usage

Return TRUE if the sprite is hitting sprite s. Only a simple bounding box is used, which is fast but not accurate to the pixel level.

The two sprites must not be scaled or rotated in order for Hit to work properly.

---

## HitEx

### Syntax

```
boolean:hitex(sprite s)
```

### Usage

Return TRUE if the sprite is hitting sprite s. Pixel perfect collision is used. Any pixel that does not have an alpha component of zero will trigger a collision.

The two sprites must not be scaled or rotated in order for HitEx to work properly.

---

## MoveSpr

### Syntax

```
movespr(float deg, float step)
```

### Usage

Move the sprite in the direction deg by the number of pixels specified in step. deg is in degrees, and should be between 0 and 360.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if step is less than zero or greater than 15.

---

## Resize

### Syntax

```
resize(int width, int height)
```

### Usage

Set the sprite's size. The sprite's texture is not affected, only the size of the sprite.

---

## SetTexture

### Syntax

```
settexture(texture t)
```

### Usage

Set the sprite's texture.

---

## Stack

A stack stores objects in a Last In First Out fashion.

Inherits from COLLECTION.

---

## Clear

### Syntax

```
clear()
```

**Usage**

Clear the stack.

**GetCount**

---

**Syntax**

`int:getcount()`

**Usage**

Return the number of items on the stack.

**IsEmpty**

---

**Syntax**

`boolean:isempty()`

**Usage**

Return TRUE if the stack is empty, otherwise returns FALSE.

**Peek**

---

**Syntax**

`item:peek()`

**Usage**

Return the item at the top of the stack.

**Pop**

---

**Syntax**

`item:pop()`

**Usage**

Pop the item at the top of the stack and returns it.

**Push**

---

**Syntax**

`push(item i)`

**Usage**

Push item i onto the top of the stack.

**Task**

A task is an independent job, with its own register set, instruction stream, and dataspace (memory). Since every running program is technically a task, its methods can be used anywhere in any program.

Inherits from OBJECT.

**CheckTrap**

---

**Syntax**

`exception:checktrap()`

**Usage**

Return an exception that has been trapped, or Null if an exception hasn't been thrown.

---

**EmptyTrap**

**Syntax**

`exception:emptytrap()`

**Usage**

Return an exception that has been trapped, or Null if an exception hasn't been thrown. Also clear the error trap.

---

**Trap**

**Syntax**

`trap(boolean state)`

**Usage**

Enable error trapping if state is TRUE, disable error trapping if state is FALSE.

---

**TCP**

TCP/IP is a communications protocol used on the Internet.

Inherits from OBJECT.

---

**Close**

**Syntax**

`close()`

**Usage**

Close an open TCP/IP connection.

---

**EOF**

**Syntax**

`boolean:eof()`

**Usage**

Return TRUE if the end of the stream has been reached, otherwise return FALSE.

**Exceptions**

Throws VM\_TCP\_NOT\_OPEN if the TCP/IP socket is not open.

Throws VM\_DENIED if the operation failed.

---

**Open**

**Syntax**

`open(string serv, string host)`

**Usage**

Open a TCP/IP client connection. `serv` specifies the TCP/IP service (e.g. "http", "smtp", etc.). `host` is either a domain name (e.g. "www.globalheavyindustries.com") or a dotted IP address.

#### Exceptions

Throws `VM_TCP_OPEN` if the TCP/IP socket is already open.

Throws `VM_TCP_TIMEOUT` if the operation times out.

### OpenPort

---

#### Syntax

```
openport(int port, string host)
```

#### Usage

Open a TCP/IP client connection. `port` is the server port the client should connect to. `host` is either a domain name (e.g. "www.globalheavyindustries.com") or a dotted IP address.

#### Exceptions

Throws `VM_TCP_OPEN` if the TCP/IP socket is already open.

Throws `VM_TCP_TIMEOUT` if the operation times out.

### Read

---

#### Syntax

```
string:read()
```

#### Usage

Read a string from the TCP/IP connection and return them as a string.

#### Exceptions

Throws `VM_TCP_NOT_OPEN` if the TCP/IP socket is not open.

Throws `VM_DENIED` if the operation failed.

### Recv

---

#### Syntax

```
string:recv(int length)
```

#### Usage

Receive the specified length of bytes from the TCP/IP connection and return them as a string.

#### Exceptions

Throws `VM_TCP_NOT_OPEN` if the TCP/IP socket is not open.

Throws `VM_TCP_TIMEOUT` if the operation times out.

Throws `VM_DENIED` if the operation failed.

### Send

---

#### Syntax

```
send(string msg)
```

#### Usage

Send the specified string to the TCP/IP connection.

### Exceptions

Throws VM\_TCP\_NOT\_OPEN if the TCP/IP socket is not open.

Throws VM\_DENIED if the operation failed.

### Write

---

#### Syntax

```
write(string msg)
```

#### Usage

Write the specified string to the TCP/IP connection.

### Exceptions

Throws VM\_TCP\_NOT\_OPEN if the TCP/IP socket is not open.

Throws VM\_DENIED if the operation failed.

### WriteLn

---

#### Syntax

```
writeln(string msg)
```

#### Usage

Write the specified string (followed by a carriage return) to the TCP/IP connection.

### Exceptions

Throws VM\_TCP\_NOT\_OPEN if the TCP/IP socket is not open.

Throws VM\_DENIED if the operation failed.

### Text

---

Text is text.

Inherits from ENTITY. Can be rendered in console in any mode.

### Constructor

---

#### Syntax

```
text(string msg)
```

#### Usage

Create a new text object.

#### Syntax

```
text(font f, string msg)
```

#### Usage

Create a new text object using the font f.

### GetBgColor

---

#### Syntax

```
int:getbgcolor()
```

#### Usage

Return the background color of the text.

---

### **GetColor**

#### **Syntax**

`int:getcolor()`

#### **Usage**

Return the foreground color of the text.

---

### **GetFont**

#### **Syntax**

`font:getfont()`

#### **Usage**

Return the font used by the text.

---

### **GethAlign**

#### **Syntax**

`int:gethalign()`

#### **Usage**

Return the current horizontal alignment.

---

### **GetHeight**

#### **Syntax**

`int:getheight()`

#### **Usage**

Return the height of the text.

---

### **GetNumber**

#### **Syntax**

`float:getnumber()`

#### **Usage**

Return the text as a float.

---

### **GetText**

#### **Syntax**

`string:gettext()`

#### **Usage**

Return the text.

---

### **GetvAlign**

#### **Syntax**

`int: getvalign()`

**Usage**

Return the current vertical alignment.

---

**GetWidth**

**Syntax**

`int: getwidth()`

**Usage**

Return the width of the text.

---

**hAlign**

**Syntax**

`halign(int alignment)`

**Usage**

Set the horizontal alignment:

- `HA_LEFT`                   left justified
- `HA_CENTER`               centered
- `HA_RIGHT`                 right justified

---

**SetBgColor**

**Syntax**

`setbgcolor(int bg)`

**Usage**

Set the background color of the text.

---

**SetColor**

**Syntax**

`setcolor(color fg)`

**Usage**

Set the foreground color of the text.

---

**SetFont**

**Syntax**

`setFont(font f)`

**Usage**

Set the font used by the text.

---

**SetNumber**

**Syntax**

`setnumber(float f)`

**Usage**

Set the text to the float f.

**SetText**

---

**Syntax**

```
settext(string msg)
```

**Usage**

Set the text.

**vAlign**

---

**Syntax**

```
valign(int alignment)
```

**Usage**

Set the vertical alignment:

1	VA_TOP	top justified
2	VA_CENTER	centered
3	VA_BOTTOM	bottom justified

**Text3D**

---

Text3D is three dimensional text.

Inherits from ENTITY3D.

**Constructor**

---

**Syntax**

```
text3d(string msg)
```

**Usage**

Create a new 3D text object.

**Syntax**

```
text3d(font3d f, string msg)
```

**Usage**

Create a new 3D text object using the 3D font f.

**GetColor**

---

**Syntax**

```
color:getcolor()
```

**Usage**

Return the text's color.

**GetFont3D**

---

**Syntax**

`font:getfont3d()`

**Usage**

Return the 3D font used by the text.

---

**GethAlign**

**Syntax**

`int:getalign()`

**Usage**

Return the current horizontal alignment.

---

**GetHeight**

**Syntax**

`float:getheight()`

**Usage**

Return the height of the text.

---

**GetNumber**

**Syntax**

`float:getnumber()`

**Usage**

Return the text as a float.

---

**GetText**

**Syntax**

`string:gettext()`

**Usage**

Return the text.

---

**GetTexture**

**Syntax**

`texture:gettexture()`

**Usage**

Return the texture mapped onto the text.

---

**GetvAlign**

**Syntax**

`int:getvalign()`

**Usage**

Return the current vertical alignment.

## GetWidth

---

### Syntax

```
float:getwidth()
```

### Usage

Return the width of the text.

## hAlign

---

### Syntax

```
halign(int alignment)
```

### Usage

Set the horizontal alignment:

1	HA_LEFT	left justified
2	HA_CENTER	centered
3	HA_RIGHT	right justified

## SetColor

---

### Syntax

```
setcolor(color c)
```

### Usage

Set the color of the text.

## SetFont3D

---

### Syntax

```
setFont3d(font3d f)
```

### Usage

Set the 3D font used by the text.

## SetNumber

---

### Syntax

```
setnumber(float f)
```

### Usage

Set the text to the float f.

## SetText

---

### Syntax

```
settext(string msg)
```

### Usage

Set the text.

## SetTexture

---

### Syntax

```
settexture(texture t)
```

**Usage**

Set the texture mapped onto the text.

**vAlign**

---

**Syntax**

```
valign(int alignment)
```

**Usage**

Set the vertical alignment:

1	VA_TOP	top justified
2	VA_CENTER	centered
3	VA_BOTTOM	bottom justified

**TextLayer**

---

A text layer is a layer that encapsulates a text map.

Inherits from LAYER. Can be rendered in console in multiscreen mode.

**Constructor**

---

**Syntax**

```
textlayer(int width, int height)
```

**Usage**

Create a new text layer, using the default console font and the specified width and height.

**Exceptions**

Throws VM\_DENIED if the operation failed.

**Syntax**

```
textlayer(charset f, int width, int height)
```

**Usage**

Create a new text layer, using the specified character set, width, and height.

**Exceptions**

Throws VM\_DENIED if the operation failed.

**Syntax**

```
textlayer(textmap t)
```

**Usage**

Create a new text layer, by encapsulating the textmap t.

**AutoScroll**

---

**Syntax**

```
autoscroll(boolean state)
```

**Usage**

Enable auto scrolling if state is TRUE, disable auto scrolling if state is FALSE.

## **CenterText**

---

### **Syntax**

```
centertext(int n)
```

### **Usage**

Center the int n at the current cursor location.

### **Syntax**

```
centertext(float n)
```

### **Usage**

Center the float n at the current cursor location.

### **Syntax**

```
centertext(string s)
```

### **Usage**

Center the string s at the current cursor location.

## **Cls**

---

### **Syntax**

```
cls()
```

### **Usage**

Clear the text layer.

## **GetCharSet**

---

### **Syntax**

```
charset:getcharset()
```

### **Usage**

Return the cursor character set.

## **GetCursorColor**

---

### **Syntax**

```
int:getcursorcolor()
```

### **Usage**

Return the cursor color.

## **GetCursorX**

---

### **Syntax**

```
int:getcursorx()
```

### **Usage**

Return the cursor's X location.

## GetCursorY

---

### Syntax

`int:getcursory()`

### Usage

Return the cursor's Y location.

## GetHeight

---

### Syntax

`int:getheight()`

### Usage

Return the height (in characters) of the text layer.

## GetHighlight

---

### Syntax

`int:gethighlight()`

### Usage

Return the highlight color.

## GetInk

---

### Syntax

`int:getink()`

### Usage

Return the ink color.

## GetWidth

---

### Syntax

`int:getwidth()`

### Usage

Return the width (in characters) of the text layer.

## HideCursor

---

### Syntax

`hidecursor()`

### Usage

Hide the cursor.

## Highlight

---

### Syntax

`highlight(int c)`

### Usage

Set the highlight color.

---

## **Ink**

### **Syntax**

```
ink(int fg)
```

### **Usage**

Set the ink (foreground) color. The ink color is used when printing characters.

---

## **IsCursorVisible**

### **Syntax**

```
boolean:iscursorvisible()
```

### **Usage**

Return TRUE if the cursor is visible, FALSE if the cursor is not visible.

---

## **LineFeed**

### **Syntax**

```
linefeed()
```

### **Usage**

Print a line feed at the current cursor location, scrolling the text layer if the cursor is on the last line of the text layer.

---

## **Locate**

### **Syntax**

```
locate(int x, int y)
```

### **Usage**

Set the cursor's X and Y location.

---

## **Peek**

### **Syntax**

```
int:peek(int d)
```

### **Usage**

Return the character stored at position d. Characters are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the text layer.

### **Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if d is less than zero or greater than or equal to the number of characters in the text layer.

---

## **Poke**

### **Syntax**

```
poke(int d, int c)
```

### **Usage**

Set the character stored at position `d` to ASCII character `c` in text mode. Characters are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the text layer.

#### Exceptions

Throws `VM_ILLEGAL_QUANTITY` if `d` is less than zero or greater than or equal to the number of characters in the text layer.

### Print

---

#### Syntax

```
print(int n)
```

#### Usage

Print the `int n` at the current cursor location.

#### Syntax

```
print(float n)
```

#### Usage

Print the `float n` at the current cursor location.

#### Syntax

```
print(string s)
```

#### Usage

Print the `string s` at the current cursor location.

### PrintAt

---

#### Syntax

```
printat(int x, int y, int n)
```

#### Usage

Print the `int n` at the cursor location `(x, y)`.

#### Syntax

```
printat(int x, int y, float n)
```

#### Usage

Print the `float n` at the cursor location `(x, y)`.

#### Syntax

```
printat(int x, int y, string s)
```

#### Usage

Print the `string s` at the cursor location `(x, y)`.

### PrintChar

---

#### Syntax

```
printchar(int key)
```

#### Usage

Print a single character at the current cursor location. `ch` is the ASCII character to print, and must be in the range of 0 to 255.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if key is less than 0 or greater than 255.

**PrintLn**

---

**Syntax**

```
println(int n)
```

**Usage**

Print the int n at the current cursor location, followed by a carriage return.

**Syntax**

```
println(float n)
```

**Usage**

Print the float n at the current cursor location, followed by a carriage return.

**Syntax**

```
println(string s)
```

**Usage**

Print the string s at the current cursor location, followed by a carriage return.

**Scroll**

---

**Syntax**

```
scroll(int dir, int count, boolean wrap)
```

**Usage**

Scroll the entire text layer in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of character cells in count. If wrap is TRUE then characters that scroll off the text layer appear on the other side, if wrap is FALSE then they do not.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

**Syntax**

```
scroll(int dir, int count, int x1, int y1, int x2, int y2, boolean wrap)
```

**Usage**

Scroll the region from (x1, y1) to (x2, y2) in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of character cells in count. If wrap is TRUE then characters that scroll off the region appear on the other side, if wrap is FALSE then they do not.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

**SetCharSet**

---

**Syntax**

```
setcharset(charset cs)
```

**Usage**

Set the character set to cs.

## SetCursorColor

---

### Syntax

```
setcursorcolor(int c)
```

### Usage

Set the cursor color.

## ShowCursor

---

### Syntax

```
showcursor()
```

### Usage

Show the cursor.

## TextMap

A text map is a two dimensional matrix that is filled with character cells.

Inherits from ENTITY.

## Constructor

---

### Syntax

```
textmap(int width, int height)
```

### Usage

Create a new text map, using the default console font and the specified width and height.

### Exceptions

Throws VM\_DENIED if the operation failed.

### Syntax

```
textmap(charset f, int width, int height)
```

### Usage

Create a new text map, using the specified character set, width, and height.

### Exceptions

Throws VM\_DENIED if the operation failed.

## AutoScroll

---

### Syntax

```
autoscroll(boolean state)
```

### Usage

Enable auto scrolling if state is TRUE, disable auto scrolling if state is FALSE.

## CenterText

---

### Syntax

```
centertext(int n)
```

**Usage**

Center the int n at the current cursor location.

**Syntax**

```
centertext(float n)
```

**Usage**

Center the float n at the current cursor location.

**Syntax**

```
centertext(string s)
```

**Usage**

Center the string s at the current cursor location.

**Cls**

---

**Syntax**

```
cls()
```

**Usage**

Clear the text map.

**GetCharSet**

---

**Syntax**

```
charset:getcharset()
```

**Usage**

Return the cursor character set.

**GetCursorColor**

---

**Syntax**

```
int:getcursorcolor()
```

**Usage**

Return the cursor color.

**GetCursorX**

---

**Syntax**

```
int:getcursorx()
```

**Usage**

Return the cursor's X location.

**GetCursorY**

---

**Syntax**

```
int:getcursory()
```

**Usage**

Return the cursor's Y location.

---

### GetHeight

#### Syntax

```
int: getheight ()
```

#### Usage

Return the height (in characters) of the text map.

---

### GetHighlight

#### Syntax

```
int: gethighlight ()
```

#### Usage

Return the highlight color.

---

### GetInk

#### Syntax

```
int: getink ()
```

#### Usage

Return the ink color.

---

### GetWidth

#### Syntax

```
int: getwidth ()
```

#### Usage

Return the width (in characters) of the text map.

---

### HideCursor

#### Syntax

```
hidecursor ()
```

#### Usage

Hide the cursor.

---

### Highlight

#### Syntax

```
highlight (int c)
```

#### Usage

Set the highlight color.

---

### Ink

#### Syntax

```
ink(int fg)
```

**Usage**

Set the ink (foreground) color. The ink color is used when printing characters.

**IsCursorVisible**

---

**Syntax**

```
boolean:iscursorvisible()
```

**Usage**

Return TRUE if the cursor is visible, FALSE if the cursor is not visible.

**LineFeed**

---

**Syntax**

```
linefeed()
```

**Usage**

Print a line feed at the current cursor location, scrolling the text map if the cursor is on the last line of the text map.

**Locate**

---

**Syntax**

```
locate(int x, int y)
```

**Usage**

Set the cursor's X and Y location.

**Peek**

---

**Syntax**

```
int:peek(int d)
```

**Usage**

Return the character stored at position d. Characters are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the text map.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if d is less than zero or greater than or equal to the number of pixels in the text map.

**Poke**

---

**Syntax**

```
poke(int d, int c)
```

**Usage**

Set the character stored at position d to ASCII character c. Characters are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the text map.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if d is less than zero or greater than or equal to the number of characters in the text map.

## Print

---

### Syntax

```
print(int n)
```

### Usage

Print the int n at the current cursor location.

### Syntax

```
print(float n)
```

### Usage

Print the float n at the current cursor location.

### Syntax

```
print(string s)
```

### Usage

Print the string s at the current cursor location.

## PrintAt

---

### Syntax

```
printat(int x, int y, int n)
```

### Usage

Print the int n at the cursor location (x, y).

### Syntax

```
printat(int x, int y, float n)
```

### Usage

Print the float n at the cursor location (x, y).

### Syntax

```
printat(int x, int y, string s)
```

### Usage

Print the string s at the cursor location (x, y).

## PrintChar

---

### Syntax

```
printchar(int key)
```

### Usage

Print a single character at the current cursor location. ch is the ASCII character to print, and must be in the range of 0 to 255.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if key is less than 0 or greater than 255.

## PrintLn

---

### Syntax

```
println(int n)
```

**Usage**

Print the int n at the current cursor location, followed by a carriage return.

**Syntax**

```
println(float n)
```

**Usage**

Print the float n at the current cursor location, followed by a carriage return.

**Syntax**

```
println(string s)
```

**Usage**

Print the string s at the current cursor location, followed by a carriage return.

**Scroll**

---

**Syntax**

```
scroll(int dir, int count, boolean wrap)
```

**Usage**

Scroll the entire text map in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of character cells in count. If wrap is TRUE then characters that scroll off the text map appear on the other side, if wrap is FALSE then they do not.

This command is ignored if not in text mode.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

**Syntax**

```
scroll(int dir, int count, int x1, int y1, int x2, int y2, boolean wrap)
```

**Usage**

Scroll the region from (x1, y1) to (x2, y2) in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of character cells in count. If wrap is TRUE then characters that scroll off the region appear on the other side, if wrap is FALSE then they do not.

This command is ignored if not in text mode.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

**SetCharSet**

---

**Syntax**

```
setcharset(charset cs)
```

**Usage**

Set the character set to cs.

**SetCursorColor**

---

**Syntax**

```
setcursorcolor(int c)
```

**Usage**

Set the cursor color.

**ShowCursor**

---

**Syntax**

```
showcursor()
```

**Usage**

Show the cursor.

**Texture**

---

A texture is a two dimensional bitmap.

Inherits from OBJECT.

**Constructor**

---

**Syntax**

```
texture(int width, int height, int filter)
```

**Usage**

Create a new texture. width specifies the width (in pixels), and height specifies the height (in pixels). filter is one of the following:

1	TX_NEAREST	pixels are scaled
2	TX_LINEAR	pixels are scaled and averaged with nearby pixels, smoothing the graphics
3	TX_MIPMAP	same as TX_LINEAR, but creates multiple textures of varying sizes

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if width is less than 1 or greater than 1024, height is less than 1 or greater than 1024, or filter is not one of the constants above.

Throws VM\_DENIED if the operation failed.

**Syntax**

```
texture(string filename, int filter)
```

**Usage**

Load an image from disk. filename specifies the file on disk to load.

**Syntax**

```
texture(string filename, int width, int height, int filter)
```

**Usage**

Load an image from disk, and resize it to the specified width and height.

**Syntax**

```
texture(string filename, int size, int filter)
```

**Usage**

Load an image from disk, and resize it to fit inside a square texture. The square texture is size pixels across and size pixels high. The aspect ratio of the image is preserved. This form of the constructor is useful for making "thumbnails" of images.

## Cast

---

### Syntax

```
string:cast()
```

### Usage

Cast the texture object to a string. The name of the texture is returned.

## AdjustRGB

---

### Syntax

```
adjustrgb(float r, float g, float b)
```

### Usage

Adjust the red, green, and blue components of a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
adjustrgb(10, 100, 1000)
```



## AdjustHSV

---

### Syntax

```
adjusthsv(float h, float s, float v)
```

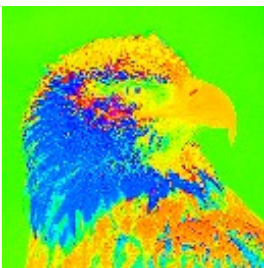
### Usage

Adjust the hue, saturation, and brightness components of a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
adjusthsv(10, 100, 1000)
```



## AutoRefresh

---

### Syntax

```
autorefresh(boolean state)
```

### Usage

If state is TRUE, enable auto refresh. If state is FALSE, disable auto refresh.

When auto refresh is enabled, any time the texture is rendered on screen it will automatically refresh (this is similar to single buffering). If auto refresh is disabled, the texture will not be refreshed until an explicit call to the Refresh method (this is similar to double buffering).

When rendering to a texture, it is often faster to disable auto refresh, draw to the texture as much as needed, and then explicitly call the Refresh method to copy the texture into video memory.

## BezierCurve

---

### Syntax

```
beziercurve(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4)
```

### Usage

Draw a Bezier curve using the four control points specified: (x1, y1), (x2, y2), (x3, y3), and (x4, y4). The brush and clipping are used when drawing pixels.

## Blobs

---

### Syntax

```
blobs(int seed, int amount, boolean rgbflag)
```

### Usage

Create a bunch of blobs.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
blobs(12, 34, TRUE)
```



## Blur

---

### Syntax

```
blur()
```

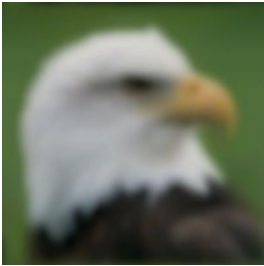
### Usage

Blur a texture.

This command is ignored if the texture is not 256x256 pixels.

**Example**

```
do 50 times
  blur()
loop
```

**Brush**

---

**Syntax**

```
brush(int op)
```

**Usage**

Set the brush operation to op.

**Syntax**

```
brush(int op, int mask)
```

**Usage**

Set the brush operation to op and the brush mask to mask.

**CellMachine**

---

**Syntax**

```
cellmachine(int seed, int rule)
```

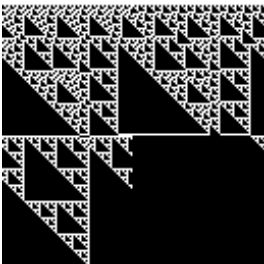
**Usage**

Create a cell machine.

This command is ignored if the texture is not 256x256 pixels.

**Example**

```
cellmachine(500, 60)
```

**Checkerboard**

---

**Syntax**

```
checkerboard(int c1, int c2, int w, int h)
```

**Usage**

Create a checkerboard pattern on the texture, by creating blocks that are w pixels wide and h pixels high, and alternating between colors c1 and c2.

This command is ignored if the texture is not 256x256 pixels.

---

**Circle****Syntax**

```
circle(int x, int y, int r, int fillmode)
```

**Usage**

Draw a circle at (x, y) using the current ink color with the radius r. Fillmode is 0 if the circle is not filled, or 1 if the circle is filled. The brush and clipping method are used when drawing pixels.

---

**Clip****Syntax**

```
clip(int x1, int y1, int x2, int y2)
```

**Usage**

Clip the texture to the region from (x1, y1) to (x2, y2).

---

**Cls****Syntax**

```
cls()
```

**Usage**

Clear the texture.

---

**Copy****Syntax**

```
copy(texture t)
```

**Usage**

Copy texture t to the texture. If t is larger than the texture, t is clipped to the texture. The brush is used and clipping is ignored.

---

**Dilate****Syntax**

```
dilate()
```

**Usage**

Dilate a texture (bright stuff grows bigger).

This command is ignored if the texture is not 256x256 pixels.

**Example**

```
do 5 times
```

```
dilate()  
loop
```



---

## DrawText

### Syntax

```
drawto(charset cs, int x, int y, string s)
```

### Usage

Print the string *s* using character set *cs* at (*x*, *y*) using the current ink color. The brush and clipping method are used when drawing pixels.

---

## DrawTo

### Syntax

```
drawto(int x, int y)
```

### Usage

Draw a line from the current graphics cursor to location (*x*, *y*) using the current ink color. The brush and clipping method are used when drawing pixels.

---

## EdgeHorizontal

### Syntax

```
edgehorizontal()
```

### Usage

Find horizontal edges in a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
edgehorizontal()
```



## EdgeVertical

---

### Syntax

```
edgevertical ()
```

### Usage

Find vertical edges in a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
edgevertical ()
```



## Ellipse

---

### Syntax

```
ellipse(int x, int y, int xr, int yr, int fillmode)
```

### Usage

Draw an ellipse at (x, y) using the current ink color with a horizontal radius of xr and a vertical radius of yr. Fillmode is 0 if the ellipse is not filled, or 1 if the ellipse is filled. The brush and clipping method are used when drawing pixels.

## Emboss

---

### Syntax

```
emboss ()
```

### Usage

Emboss a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
emboss ()
```



## EqualizeFull

---

### Syntax

```
equalizefull ()
```

### Usage

Equalize the texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
equalizefull ()
```



## EqualizeStretch

---

### Syntax

```
equalizestretch ()
```

### Usage

Equalize stretch the texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
equalizestretch ()
```



## Erode

---

### Syntax

```
erode ()
```

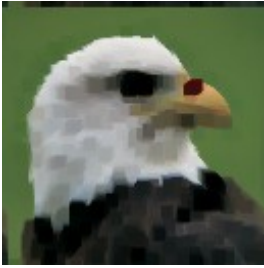
### Usage

Erode a texture (dark stuff grows bigger).

This command is ignored if the texture is not 256x256 pixels.

### Example

```
do 5 times
  erode()
loop
```



---

## Fill

### Syntax

```
fill(int color)
```

### Usage

Fill the texture with a color. The brush is used, but clipping is ignored.

---

## Flip180

### Syntax

```
flip180()
```

### Usage

Rotate the texture 180 degrees.

---

## FloodFill

### Syntax

```
floodfill(int x, int y)
```

### Usage

Flood fill an area at (x, y) using the current ink color. The current pixel at (x, y) is used as the "old color". All pixels with the old color to the left, right, top, or bottom of (x, y) are colored, and this repeats with those pixels. This is often used to fill in an irregular shape. The shape must be enclosed, or the color will "spill out" of the shape. The brush and clipping method are used when drawing pixels.

---

## GetBrushMask

### Syntax

```
int:getbrushmask()
```

### Usage

Return the current brush mask.

---

## GetBrushOp

### Syntax

```
int:getbrushop()
```

**Usage**

Return the current brush operation.

---

**GetCursorX****Syntax**

`int:getcursorex()`

**Usage**

Return the cursor's X location.

---

**GetCursorY****Syntax**

`int:getcursory()`

**Usage**

Return the cursor's Y location.

---

**GetFilter****Syntax**

`int:getfilter()`

**Usage**

Return the current graphics filter:

1	TX_NEAREST	pixels are scaled
2	TX_LINEAR	pixels are scaled and averaged with nearby pixels, smoothing the graphics
3	TX_MIPMAP	same as TX_LINEAR, but creates multiple textures of varying sizes

---

**GetHeading****Syntax**

`float:getheading()`

**Usage**

Return the turtle's heading, in degrees (0 to 359).

---

**GetHeight****Syntax**

`int:getheight()`

**Usage**

Return the height (in pixels) of the texture.

---

**GetImageHeight****Syntax**

`int:getimageheight()`

**Usage**

Return the height (in pixels) of the image if an image was loaded in the constructor. Otherwise return 0.

---

### GetImageWidth

#### Syntax

```
int:getimagewidth()
```

#### Usage

Return the width (in pixels) of the image if an image was loaded in the constructor. Otherwise return 0.

---

### GetInk

#### Syntax

```
int:getink()
```

#### Usage

Return the ink color.

---

### GetTurtleX

#### Syntax

```
float:getturtleX()
```

#### Usage

Return the turtle's X coordinate.

---

### GetTurtleY

#### Syntax

```
float:getturtleY()
```

#### Usage

Return the turtle's Y coordinate.

---

### GetWidth

#### Syntax

```
int:getwidth()
```

#### Usage

Return the width (in pixels) of the texture.

---

### GoBackward

#### Syntax

```
gobackward(float units)
```

#### Usage

Move the turtle backward the specified number of units.

---

### GoForward

#### Syntax

```
goforward(float units)
```

**Usage**

Move the turtle forward the specified number of units.

**GrayScale**

---

**Syntax**

```
grayscale()
```

**Usage**

Filter all of the pixels in the texture to a gray scale, effectively making color pictures black and white.

**hLine**

---

**Syntax**

```
hline(int x1, int x2, int y)
```

**Usage**

Draw a horizontal line from (x1, y) to (x2, y) using the current ink color. The brush and clipping are used when drawing pixels.

**Home**

---

**Syntax**

```
home()
```

**Usage**

Return the turtle to the center of the texture.

**Ink**

---

**Syntax**

```
ink(int color)
```

**Usage**

Set the ink (foreground) color. The ink color is used when drawing pixels.

**Invert**

---

**Syntax**

```
invert()
```

**Usage**

Invert the texture.

This command is ignored if the texture is not 256x256 pixels.

**Example**

```
invert()
```



## Kaleidoscope

---

### Syntax

```
kaleidoscope(int corner)
```

### Usage

Kaleidoscope a texture (mirror and resize 2x2).

This command is ignored if the texture is not 256x256 pixels.

### Example

```
kaleidoscope(123)
```



## Line

---

### Syntax

```
line(int x1, int y1, int x2, int y2)
```

### Usage

Draw a line from (x1, y1) to (x2, y2) using the current ink color. The brush and clipping are used when drawing pixels.

### Syntax

```
line(float deg, float n)
```

### Usage

Draw a line from the current graphics cursor in the direction deg (degrees), with the length n. The brush and clipping are used when drawing pixels.

## LoadImage

---

### Syntax

```
loadimage(string f)
```

### Usage

Load the image in file f. The image is automatically scaled to fit the texture.

The extension of file *f* determines the image format:

<b>File Extension</b>	<b>Image Format</b>
BMP	Windows or OS/2 Bitmap
CUT	Dr. Halo
DDS	DirectX Surface
EXR	ILM OpenEXR
G3	Raw fax format CCITT G.3
GIF	Graphics Interchange Format
HDR	High Dynamic Range
ICO	Windows Icon
IFF, LBM	IFF Interleaved Bitmap
J2K, J2C	JPEG-2000 codestream
JNG	JPEG Network Graphics
JP2	JPEG-2000 File Format
JPG, JIF, JPEG, JPE	Independent JPEG Group
KOA	C64 Koala Graphics
MNG	Multiple Network Graphics
PBM	Portable Bitmap (ASCII or binary)
PCD	Kodak PhotoCD
PCX	Zsoft Paintbrush
PGM	Portable Greymap (ASCII or binary)
PNG	Portable Network Graphics
PPM	Portable Pixelmap (ASCII or binary)
PSD	Adobe Photoshop
RAS	Sun Raster Image
SGI	Silicon Graphics SGI image format
TGA, TARGA	Truevision Targa
TIF, TIFF	Tagged Image File Format
WAP, WBMP, WBM	Wireless Bitmap
XBM	X11 Bitmap Format
XPM	X11 Pixmap Format

### Exceptions

Throws VM\_DENIED if the operation failed.

Throws VM\_FILE\_NOT\_FOUND if the file *f* does not exist.

Throws VM\_BAD\_FILE\_FORMAT if the file *f* is corrupted or not a supported image type.

---

## LogPolar

### Syntax

```
logpolar()
```

**Usage**

Log polar effect.

This command is ignored if the texture is not 256x256 pixels.

**Example**

```
logpolar()
```



---

**MakeTilable****Syntax**

```
maketilable(int strength)
```

**Usage**

Make a texture tilable.

This command is ignored if the texture is not 256x256 pixels.

**Example**

```
maketilable(75)
```



---

**Mask****Syntax**

```
mask(int color, int alpha)
```

**Usage**

Scan the entire texture, and wherever the RGB components of color are present, set the alpha component to alpha. This is often used to load a 24-bit BMP (which treats all pixels as opaque), and making all pixels of a set color (for example, black) transparent.

---

**Median****Syntax**

`median()`

**Usage**

Median a texture (similar to blur).

This command is ignored if the texture is not 256x256 pixels.

**Example**

`median()`



---

**MirrorX**

**Syntax**

`mirrorx()`

**Usage**

Mirror the texture along the X axis.

---

**MirrorY**

**Syntax**

`mirrory()`

**Usage**

Mirror the texture along the Y axis.

---

**MoveDistort**

**Syntax**

`movedistort(int dx, int dy)`

**Usage**

Move a texture.

This command is ignored if the texture is not 256x256 pixels.

**Example**

`movedistort(128, 128)`



## Noise

---

### Syntax

```
noise(int seed, int radius)
```

### Usage

Add noise to a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
noise(123, 645)
```



## Particle

---

### Syntax

```
particle(float rad)
```

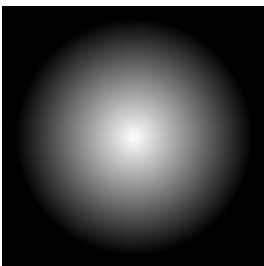
### Usage

Create a particle, where rad is the radius.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
particle(100)
```



## Peek

---

### Syntax

```
int:peek(int d)
```

### Usage

Return the pixel stored at position d. Pixels are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the texture.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if d is less than zero or greater than or equal to the number of pixels in the texture.

## PenDown

---

### Syntax

```
pendown ()
```

### Usage

Place the turtle's pen down.

## PenUp

---

### Syntax

```
penup ()
```

### Usage

Place the turtle's pen up.

## PerlinNoise

---

### Syntax

```
perlinnoise(int dist, int seed, int amplitude, int octaves, int persistence,  
boolean rgbflag)
```

### Usage

Create perlin noise.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
perlinnoise(128, 9876543, 256, 8, 150, TRUE)
```



## Pixels

---

### Syntax

```
pixels(int color, int count)
```

**Usage**

Randomly draw a number of pixels (count) in the given color.

This command is ignored if the texture is not 256x256 pixels.

---

**Plot****Syntax**

```
plot(int x, int y)
```

**Usage**

Plot a pixel at location (x, y) using the current ink color. The brush and clipping are used when drawing pixels.

---

**Point****Syntax**

```
int:point(int x, int y)
```

**Usage**

Return the pixel at location (x, y).

---

**Poke****Syntax**

```
poke(int d, int c)
```

**Usage**

Set the pixel stored at position d to color c. Pixels are stored sequentially, from left to right and top to bottom, starting with 0 at the upper left corner of the texture.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if d is less than zero or greater than or equal to the number of pixels in the texture.

---

**Rect****Syntax**

```
rect(int x1, int y1, int x2, int y2, int fillmode)
```

**Usage**

Draw a rectangle from (x1, y1) to (x2, y2) using the current ink color. Fillmode is 0 if the rectangle is not filled, or 1 if the rectangle is filled. The brush and clipping are used when drawing pixels.

---

**Refresh****Syntax**

```
refresh()
```

**Usage**

"Refresh" the texture, meaning that the texture in memory is copied into video memory.

## Rescale

---

### Syntax

```
rescale(int w, int h, int filter)
```

### Usage

Resize the texture to the specified width and height. Filter determines the algorithm to use when resizing:

0	TXF_BOX	Box, pulse, Fourier window, 1st order (constant) b-spline
1	TXF_BICUBIC	Mitchell & Netravali's two-param cubic filter
2	TXF_BILINEAR	Bilinear filter
3	TXF_BSPLINE	4th order (cubic) b-spline
4	TXF_CATMULLROM	Catmull-Rom spline, Overhauser spline
5	TXF_LANCZOS3	Lanczos3 filter

## Savelmage

---

### Syntax

```
saveimage(string f)
```

### Usage

Save the texture to file f.

The extension of file f determines the image format:

File Extension	Image Format
BMP	Windows or OS/2 Bitmap
EXR	ILM OpenEXR
GIF	Graphics Interchange Format
HDR	High Dynamic Range
ICO	Windows Icon
J2K, J2C	JPEG-2000 codestream
JP2	JPEG-2000 File Format
JPG, JIF, JPEG, JPE	Independent JPEG Group
PBM	Portable Bitmap (ASCII or binary)
PGM	Portable Greymap (ASCII or binary)
PNG	Portable Network Graphics
PPM	Portable Pixelmap (ASCII or binary)
TGA, TARGA	Truevision Targa
TIF, TIFF	Tagged Image File Format
WAP, WBMP, WBM	Wireless Bitmap
XPM	X11 Pixmap Format

### Exceptions

Throws VM\_DENIED if the operation failed.

Throws VM\_BAD\_FILE\_FORMAT if the file f is not a supported image type.

## ScaleHSV

---

### Syntax

```
scalehsv(float h, float s, float v)
```

### Usage

Scale the hue, saturation, and brightness components of a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
scalehsv(1, 2, 3)
```



## ScaleRGB

---

### Syntax

```
scalergb(float r, float g, float b)
```

### Usage

Scale the red, green, and blue components of a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
scalergb(2, 2, 2)
```



## Scroll

---

### Syntax

```
scroll(int dir, int count, boolean wrap)
```

### Usage

Scroll the entire texture in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of pixels in count. If wrap is TRUE then pixels that scroll off the texture appear on the other side, if wrap is FALSE then they do not.

### Exceptions

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

**Syntax**

```
scroll(int dir, int count, int x1, int y1, int x2, int y2)
```

**Usage**

Scroll the region from (x1, y1) to (x2, y2) in the direction dir (1-up, 2-down, 3-left, 4-right) by the number of pixels in count. If wrap is TRUE then pixels that scroll off the region appear on the other side, if wrap is FALSE then they do not.

**Exceptions**

Throws VM\_ILLEGAL\_QUANTITY if direction is less than 1 or greater than 4, or if count is less than 1.

**Sculpture**

---

**Syntax**

```
sculpture ()
```

**Usage**

Sculpture a texture.

This command is ignored if the texture is not 256x256 pixels.

**Example**

```
sculpture ()
```

**Sepia**

---

**Syntax**

```
sepia ()
```

**Usage**

Filter all of the pixels in the texture to a sepia tone (similar to an 1800's photograph).

**SetFilter**

---

**Syntax**

```
setfilter(int f)
```

**Usage**

Set the graphics filter.

1	TX_NEAREST	pixels are scaled
2	TX_LINEAR	pixels are scaled and averaged with nearby pixels, smoothing the graphics
3	TX_MIPMAP	same as TX_LINEAR, but creates multiple textures of varying sizes

## SetHeading

---

### Syntax

```
setheading(float h)
```

### Usage

Set the turtle's heading, in degrees (0 to 359).

## SetPosition

---

### Syntax

```
setposition(int x, int y)
```

### Usage

Set the turtle's X and Y coordinates.

## Sharpen

---

### Syntax

```
sharpen()
```

### Usage

Sharpen a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
sharpen()
```



## SineDistort

---

### Syntax

```
sinedistort(int dx, int depthx, int dy, int depthy)
```

### Usage

Sine distort the texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
sinedistort(50, 25, 50, 25)
```



## SinePlasma

---

### Syntax

```
sineplasma(float dx, float dy, float amplitude)
```

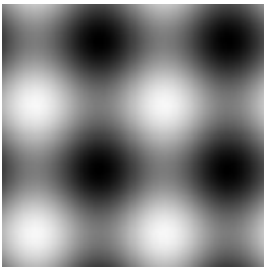
### Usage

Create a sine plasma.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
sineplasma(50, 50, 250)
```



## SineRGB

---

### Syntax

```
sinergb(float r, float g, float b)
```

### Usage

Sine the red, green, and blue components in a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
sinergb(12, 34, 56)
```



## Stamp

---

### Syntax

```
stamp(texture t, int x, int y)
```

### Usage

Stamp texture t onto the texture at location (x, y). The brush and clipping are used when drawing pixels.

## SubPlasma

---

### Syntax

```
subplasma(int dist, int seed, int amplitude, boolean rgbflag)
```

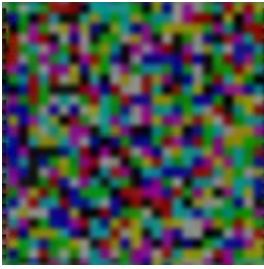
### Usage

Create a sub plasma.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
subplasma(10, 50, 150, TRUE)
```



## Tile

---

### Syntax

```
tile()
```

### Usage

Tile a texture (copy and resize 2x2).

This command is ignored if the texture is not 256x256 pixels.

### Example

```
tile()
```



## Triangle

---

### Syntax

```
triangle(int x1, int y1, int x2, int y2, int x3, int y3, int fillmode)
```

### Usage

Draw a triangle from (x1, y1) to (x2, y2) to (x3, y3) using the current ink color. Fillmode is 0 if the triangle is not filled, or 1 if the triangle is filled. The brush and clipping are used when drawing pixels.

## Tunnel

---

### Syntax

```
tunnel(int zoom)
```

### Usage

Tunnel a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
tunnel(25)
```



## TurnLeft

---

### Syntax

```
turnleft(float deg)
```

### Usage

Rotate the turtle counter clockwise by the specified number of degrees.

## TurnRight

---

### Syntax

```
turnright(float deg)
```

### Usage

Rotate the turtle clockwise by the specified number of degrees.

## Twirl

---

### Syntax

```
twirl(int rot, int scale)
```

### Usage

Twirl a texture.

This command is ignored if the texture is not 256x256 pixels.

### Example

```
twirl(123, 1000)
```



### vLine

---

#### Syntax

```
vline(int x, int y1, int y2)
```

#### Usage

Draw a vertical line from (x, y1) to (x, y2) using the current ink color. The brush and clipping are used when drawing pixels.

### Vector

A vector is an array that is treated as a list that can grow and shrink in size. The index is a consecutive range of integers, from 0 to s, where s is the size of the vector.

ACCESS	O(1)
FIND	O(n) if unsorted, O(log(n)) if sorted
DELETE	O(n)
INSERT	O(n)

Can be used with the FOR EACH statement.

Inherits from COLLECTION.

### Add

---

#### Syntax

```
int:add(item i)
```

#### Usage

Add item i to the vector. Return the index of the added item.

#### Exceptions

Throws VM\_BAD\_ITEM if item i is a null object.

Throws VM\_OUT\_OF\_MEMORY if the vector is too big to fit into memory.

### Clear

---

#### Syntax

`clear()`

**Usage**

Clear the vector.

**Contains**

---

**Syntax**

`boolean:contains(item i)`

**Usage**

Return TRUE if item *i* is in the vector, otherwise return FALSE.

**Delete**

---

**Syntax**

`delete(int inx)`

**Usage**

Delete the item at index *inx*.

**Exceptions**

Throws VM\_BAD\_SUBSCRIPT if the index *inx* is less than 0 or greater than *s*, where *s* is the size of the vector.

**Enqueue**

---

**Syntax**

`enqueue(item i)`

**Usage**

Enqueue item *i* into the sorted vector.

**Exceptions**

Throws VM\_NOT\_SORTED if the vector is not sorted.

Throws VM\_BAD\_ITEM if item *i* is a null object.

**Exchange**

---

**Syntax**

`exchange(int inx1, int inx2)`

**Usage**

Exchange the item at index *inx1* with the item at index *inx2*.

**Exceptions**

Throws VM\_BAD\_SUBSCRIPT if the index *inx1* or *inx2* is less than 0 or greater than *s*, where *s* is the size of the vector.

**Find**

---

**Syntax**

`int:find(item i)`

**Usage**

Return the index where item *i* resides, or -1 if the item can't be found.

## First

---

### Syntax

`item:first()`

### Usage

Return the first item in the vector.

### Exceptions

Throws `VM_ITEM_NOT_FOUND` if the vector is empty.

## GetCount

---

### Syntax

`int:getcount()`

### Usage

Return the number of items in the vector.

## Insert

---

### Syntax

`insert(int inx, item i)`

### Usage

Insert item `i` into the vector at index `inx`.

### Exceptions

Throws `VM_BAD_SUBSCRIPT` if the index `inx` is less than 0 or greater than `s`, where `s` is the size of the vector.  
Throws `VM_BAD_ITEM` if item `i` is a null object.

## IsEmpty

---

### Syntax

`boolean:isempty()`

### Usage

Return `TRUE` if the vector is empty, otherwise return `FALSE`.

## Last

---

### Syntax

`item:last()`

### Usage

Return the last item in the vector.

### Exceptions

Throws `VM_ITEM_NOT_FOUND` if the vector is empty.

## Move

---

### Syntax

```
move(int inx1, int inx2)
```

### Usage

Move the item at index `inx1` by removing the item and then inserting it at index `inx2`.

### Exceptions

Throws `VM_BAD_SUBSCRIPT` if the index `inx1` or `inx2` is less than 0 or greater than `s`, where `s` is the size of the vector.

## Remove

---

### Syntax

```
remove(item i)
```

### Usage

Remove item `i` from the vector.

## Reverse

---

### Syntax

```
reverse(int inx1, int inx2)
```

### Usage

Reverse the items in the vector from index `inx1` to index `inx2`.

### Exceptions

Throws `VM_BAD_SUBSCRIPT` if the index `inx1` or `inx2` is less than 0 or greater than `s`, where `s` is the size of the vector.

## Shuffle

---

### Syntax

```
shuffle()
```

### Usage

Shuffles the items in the vector.

## Sort

---

### Syntax

```
sort()
```

### Usage

Sorts the items in the vector.

## XML

---

XML (eXtensible Markup Language) is a markup language.

Inherits from `OBJECT`.

## Constructor

---

### Syntax

```
xml(string data)
```

### Usage

Parse the XML passed to it into a tree. The tree is a collection of nodes, starting with a root node that can have children node attached to it. Each node is assigned a unique handle so it can be referenced. Navigating through the nodes allows the XML file to be effectively parsed.

### Exceptions

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

## AddChild

---

### Syntax

```
addchild(int parent, int child)
```

### Usage

Add a node as a child to another node. If parent is zero then the child is added to the root.

### Exceptions

Throws VM\_BAD\_HANDLE if child or parent is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

## AddSimpleChild

---

### Syntax

```
int:addsimplechild(int parent, string tag, string value)
```

### Usage

Add a simple node with a defined tag and value to parent. If parent is zero then the child is added to the root. Return a handle to the node that was added.

### Exceptions

Throws VM\_BAD\_HANDLE if parent is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

## AttribCount

---

### Syntax

```
int:attribcount(int node)
```

### Usage

Return the number of attributes in a node.

### Exceptions

Throws VM\_BAD\_HANDLE if node is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

## AttribExists

---

### Syntax

```
boolean:attribexists(int node, string attrib)
```

**Usage**

Return TRUE if the attribute name exists in a given node, otherwise return FALSE if the attribute name is not found in the given node.

**Exceptions**

Throws VM\_BAD\_HANDLE if node is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

---

**AttribName**

---

**Syntax**

```
string:attribname(int node, int index)
```

**Usage**

Return the attribute's name given its number (from 1 to AttribCount).

**Exceptions**

Throws VM\_BAD\_HANDLE if node is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

---

**AttribToString**

---

**Syntax**

```
string:attribtostring(int node)
```

**Usage**

Return the node's attributes as a string.

**Exceptions**

Throws VM\_BAD\_HANDLE if node is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

---

**AttribValue**

---

**Syntax**

```
string:attribvalue(int node, string attrib)
```

**Usage**

Return the node's attribute's value.

**Exceptions**

Throws VM\_BAD\_HANDLE if handle is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

---

**Child**

---

**Syntax**

```
int:child(int node)
```

**Usage**

Return a node handle to the first child of the given node.

**Exceptions**

Throws VM\_BAD\_HANDLE if node is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

---

## ChildData

### Syntax

```
string:childdata(int node)
```

### Usage

Return the first child of the given node as a string.

### Exceptions

Throws VM\_BAD\_HANDLE if node is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

---

## Clean

### Syntax

```
string:clean(string inp)
```

### Usage

Clean up the string inp and return it.

&	is replaced with	&amp
<	is replaced with	&lt;
>	is replaced with	&gt;
"	is replaced with	&quot;
'	is replaced with	&apos;

### Exceptions

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

---

## FindNode

### Syntax

```
string:findnode(int root, int level, string tag, string attrib, string value)
```

### Usage

Return a node handle to the node that has the specified tag and optionally the specified attribute set to value.

### Exceptions

Throws VM\_BAD\_HANDLE if root is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

---

## GetChildTagData

### Syntax

```
string:getchildtagdata(int node, string tag)
```

### Usage

Return a given node's child data.

### Example

If the input XML is as follows:

```
<root><item1>Hello World</item1></root>
```

Then `GetChildTagData(0, "item1")` would return Hello World.

### Exceptions

Throws `VM_BAD_HANDLE` if node is an invalid node handle.

Throws `VM_XML_ERROR` if the operation failed. The exception data holds the actual error message.

## GetTag

---

### Syntax

```
string:gettag(int node)
```

### Usage

Return a given node's tag.

### Exceptions

Throws `VM_BAD_HANDLE` if node is an invalid node handle.

Throws `VM_XML_ERROR` if the operation failed. The exception data holds the actual error message.

## NextSibling

---

### Syntax

```
int:nextsibling(int node)
```

### Usage

Return a node handle to a given node's sibling.

### Exceptions

Throws `VM_BAD_HANDLE` if node is an invalid node handle.

Throws `VM_XML_ERROR` if the operation failed. The exception data holds the actual error message.

## NodeToString

---

### Syntax

```
string:nodetostring(int node)
```

### Usage

Return a given node as a string, without the end tag.

### Exceptions

Throws `VM_BAD_HANDLE` if node is an invalid node handle.

Throws `VM_XML_ERROR` if the operation failed. The exception data holds the actual error message.

## Parent

---

### Syntax

```
int:parent(int node)
```

### Usage

Return a node handle to a given node's parent.

### Exceptions

Throws `VM_BAD_HANDLE` if node is an invalid node handle.

Throws `VM_XML_ERROR` if the operation failed. The exception data holds the actual error message.

## SetAttrib

---

### Syntax

```
setattrib(int node, string attrib, string value)
```

### Usage

Set an existing attribute to value for the given node, or add the attribute if it does not exist.

### Exceptions

Throws VM\_BAD\_HANDLE if node is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

## TreeToString

---

### Syntax

```
string:treetostring(int root, boolean crlf)
```

### Usage

Convert to a single string all of the nodes from a given root node on down. A carriage return and linefeed (\$0C\$0A) is added onto the end of each line if crlf is TRUE.

### Exceptions

Throws VM\_BAD\_HANDLE if root is an invalid node handle.

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.

## Unclean

---

### Syntax

```
string:unclean(string inp)
```

### Usage

"Uncleans" the string inp and returns it.

&amp;	is replaced with	&
&lt;	is replaced with	<
&gt;	is replaced with	>
&quot;	is replaced with	"
&apos;	is replaced with	'

### Exceptions

Throws VM\_XML\_ERROR if the operation failed. The exception data holds the actual error message.