Squirrel 3.0 Reference Manual

version 3.0.7 release stable

Alberto Demichelis

Extensive review: Wouter Van Oortmersern

Chapter 1. Introduction

Squirrel is a high level imperative-OO programming language, designed to be a powerful scripting tool that fits in the size, memory bandwidth, and real-time requirements of applications like games. Although Squirrel offers a wide range of features like dynamic typing, delegation, higher order functions, generators, tail recursion, exception handling, automatic memory management, both compiler and virtual machine fit together in about 6k lines of C++ code.

Chapter 2. The language

This part of the document describes the syntax and semantics of the language.

Lexical structure

Identifiers

Identifiers start with a alphabetic character or '_' followed by any number of alphabetic characters, '_' or digits ([0-9]). Squirrel is a case sensitive language, this means that the lowercase and uppercase representation of the same alphabetic character are considered different characters. For instance "foo", "Foo" and "fOo" will be treated as 3 distinct identifiers.

```
id:= [a-zA-Z_]+[a-zA-Z_0-9]*
```

Keywords

The following words are reserved words by the language and cannot be used as identifiers:

base	break	case	catch	class	clone
continue	const	default	delete	else	enum
extends	for	foreach	function	if	in
local	null	resume	return	switch	this
throw	try	typeof	while	yield	constructor
instanceof	true	false	static		

Keywords are covered in detail later in this document.

Operators

Squirrel recognizes the following operators:

!	!=	||	==	&&	<=	=>	>
<=>	+	+=	-	-=	/	/=	*
*=	%	%=	++	--	<-	=	
&	^	|	~	>>	<<	>>>	

Other tokens

Other used tokens are:

{ } [ ] . : :: ' ; " @"

Literals

Squirrel accepts integer numbers, floating point numbers and strings literals.

34	Integer number(base 10)
0xFF00A120	Integer number(base 16)
0753	Integer number(base 8)
'a'	Integer number
1.52	Floating point number
1.e2	Floating point number
1.e-2	Floating point number
"I'm a string"	String
@"I'm a verbatim string"	String
@" I'm a multiline verbatim string "	String

```
IntegerLiteral := [1-9][0-9]* | '0x' [0-9A-Fa-f]+ | ''' [.]+ ''' | 0[0-7]+
FloatLiteral := [0-9]+ '.' [0-9]+
FloatLiteral := [0-9]+ '.' 'e'|'E' '+'|'-' [0-9]+
StringLiteral:= '"'[.]* '"'
VerbatimStringLiteral:= '@''"'[.]* '"'
```

Comments

A comment is text that the compiler ignores but that is useful for programmers. Comments are normally used to embed annotations in the code. The compiler treats them as white space.

The /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters. This syntax is the same as ANSI C.

```
/*
this is
```

```
a multiline comment.
this lines will be ignored by the compiler
*/
```

The // (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. It is commonly called a “single-line comment.”

```
//this is a single line comment. this line will be ignored by the compiler
```

The character # is an alternative syntax for single line comment.

```
#this is also a single line comment.
```

This to facilitate the use of squirrel in UNIX-style shell scripts.

Values and Data types

Squirrel is a dynamically typed language so variables do not have a type, although they refer to a value that does have a type. Squirrel basic types are integer, float, string, null, table, array, function, generator, class, instance, bool, thread and userdata.

Integer

An Integer represents a 32 bits (or better) signed number.

```
local a = 123 //decimal
local b = 0x0012 //hexadecimal
local c = 075 //octal
local d = 'w' //char code
```

Float

A float represents a 32 bits (or better) floating point number.

```
local a=1.0
local b=0.234
```

String

Strings are an immutable sequence of characters to modify a string is necessary create a new one.

Squirrel's strings, behave like C or C++, are delimited by quotation marks(") and can contain escape sequences(\t,\a,\b,\n,\r,\v,\f,\\,\",\',\0,\x*hhhh*).

Verbatim string literals begin with @" and end with the matching quote. Verbatim string literals also can extend over a line break. If they do, they include any white space characters between the quotes:

```
local a = "I'm a wonderful string\n"
```

```
// has a newline at the end of the string
local x = @"I'm a verbatim string\n"
// the \n is copied in the string same as \\n in a regular string "I'm a verbatim strin
```

The only exception to the "no escape sequence" rule for verbatim string literals is that you can put a double quotation mark inside a verbatim string by doubling it:

```
local multiline = @"
	this is a multiline string
	it will ""embed"" all the new line
	characters
"
```

Null

The null value is a primitive value that represents the null, empty, or non-existent reference. The type Null has exactly one value, called null.

```
local a=null
```

Bool

the bool data type can have only two. They are the literals true and false. A bool value expresses the validity of a condition (tells whether the condition is true or false).

```
local a = true;
```

Table

Tables are associative containers implemented as pairs of key/value (called a slot).

```
local t={}
local test=
{
   a=10
   b=function(a) { return a+1; }
}
```

Array

Arrays are simple sequence of objects, their size is dynamic and their index starts always from 0.

```
local a=["I'm","an","array"]
local b=[null]
b[0]=a[2];
```

Function

Functions are similar to those in other C-like languages and to most programming languages in general, however there are a few key differences (see below).

Class

Classes are associative containers implemented as pairs of key/value. Classes are created through a 'class expression' or a 'class statement'. class members can be inherited from another class object at creation time. After creation members can be added until a instance of the class is created.

Class instance

Class instances are created by calling a class object. Instances, as tables, are implemented as pair of key/value. Instances members cannot be dyncamically added or removed however the value of the members can be changed.

Generator

Generators are functions that can be suspended with the statement 'yield' and resumed later (see Generators).

Userdata

Userdata objects are blobs of memory(or pointers) defined by the host application but stored into Squirrel variables (See Userdata and UserPointers).

Thread

Threads are objects that represents a cooperative thread of execution, also known as coroutines.

Weak References

Weak References are objects that point to another(non scalar) object but do not own a strong reference to it. (See Weak References).

Execution Context

The execution context is the union of the function stack frame and the function environment object(this). The stack frame is the portion of stack where the local variables declared in its body are stored. The environment object is an implicit parameter that is automatically passed by the function caller (see Functions). During the execution, the body of a function can only transparently refer to his execution context. This mean that a single identifier can refer either to a local variable or to an environment object slot; Global variables require a special syntax (see Variables). The environment object can be explicitly accessed by the keyword this.

Variables

There are two types of variables in Squirrel, local variables and tables/arrays slots. Because global variables are stored in a table, they are table slots.

A single identifier refers to a local variable or a slot in the environment object.

derefexp := id;

```
_table["foo"]
_array[10]
```

with tables we can also use the '.' syntax

derefexp := exp '.' id

```
_table.foo
```

Squirrel first checks if an identifier is a local variable (function arguments are local variables) if not it checks if it is a member of the environment object (this).

For instance:

```
function testy(arg)
{
   local a=10;
   print(a);
   return arg;
}
```

will access to local variable 'a' and prints 10.

```
function testy(arg)
{
   local a=10;
   return arg+foo;
}
```

in this case 'foo' will be equivalent to 'this.foo' or this["foo"].

Global variables are stored in a table called the root table. Usually in the global scope the environment object is the root table, but to explicitly access the global table from another scope, the slot name must be prefixed with '::' (::foo).

exp:= '::' id

For instance:

```
function testy(arg)
{
   local a=10;
   return arg+::foo;
}
```

accesses the global variable 'foo'.

However (since squirrel 2.0) if a variable is not local and is not found in the 'this' object Squirrel will search it in the root table.

```
function test() {
	foo = 10;
}
```

is equivalent to write

```
function test() {
	if("foo" in this) {
		this.foo = 10;
	}else {
		::foo = 10;
	}
}
```

Statements

A squirrel program is a simple sequence of statements.

stats := stat [';'|'\n'] stats

Statements in squirrel are comparable to the C-Family languages (C/C++, Java, C# etc...): assignment, function calls, program flow control structures etc.. plus some custom statement like yield, table and array constructors (All those will be covered in detail later in this document). Statements can be separated with a new line or ';' (or with the keywords case or default if inside a switch/case statement), both symbols are not required if the statement is followed by '}'.

Block

stat := '{' stats '}'

A sequence of statements delimited by curly brackets ({ }) is called block; a block is a statement itself.

Control Flow Statements

squirrel implements the most common control flow statements: if, while, do-while, switch-case, for, foreach.

true and false

Squirrel has a boolean type(bool) however like C++ it considers null, 0(integer) and 0.0(float) as *false*, any other value is considered *true*.

if/else

stat:= 'if' '(' exp ')' stat ['else' stat]

Conditionally execute a statement depending on the result of an expression.

```
if(a>b)
    a=b;
else
```

```
   b=a;
////
if(a==10)
{
   b=a+b;
   return a;
}
```

while

stat:= 'while' '(' exp ')' stat

Executes a statement while the condition is true.

```
function testy(n)
{
   local a=0;
   while(a<n) a+=1;

      while(1)
      {
      if(a<0) break;
      a-=1;
   }
}
```

do/while

stat:= 'do' stat 'while' '(' expression ')'

Executes a statement once, and then repeats execution of the statement until a condition expression evaluates to false.

```
local a=0;
do
{
    print(a+"\n");
    a+=1;
} while(a>100)
```

switch

```
stat := 'switch' ''( exp ')' '{'
        'case' case_exp ':'
                stats
        ['default' ':'
                stats]
'}'
```

Is a control statement allows multiple selections of code by passing control to one of the case statements within its body. The control is transferred to the case label whose case_exp matches with exp if none of the case match will jump to the default label (if present). A switch statement can contain any number if case instances, if 2 case have the same expression result the first one will be taken in account first. The default label is only allowed once and must be the last one. A break statement will jump outside the switch block.

Loops

for

```
stat:= 'for' '(' [initexp] ';' [condexp] ';' [incexp] ')' statement
```

Executes a statement as long as a condition is different than false.

```
for(local a=0;a<10;a+=1)
    print(a+"\n");
//or
glob <- null
for(glob=0;glob<10;glob+=1){
    print(glob+"\n");
}
//or
for(;;){
    print(loops forever+"\n");
}
```

foreach

'foreach' '(' [index_id','] value_id 'in' exp ')' stat

Executes a statement for every element contained in an array, table, class, string or generator. If exp is a generator it will be resumed every iteration as long as it is alive; the value will be the result of 'resume' and the index the sequence number of the iteration starting from 0.

```
local a=[10,23,33,41,589,56]
foreach(idx,val in a)
    print("index="+idx+" value="+val+"\n");
//or
foreach(val in a)
    print("value="+val+"\n");
```

break

stat := 'break'

The break statement terminates the execution of a loop (for, foreach, while or do/while) or jumps out of switch statement;

continue

stat := 'continue'

The continue operator jumps to the next iteration of the loop skipping the execution of the following statements.

return

stat:= return [exp]

The return statement terminates the execution of the current function/generator and optionally returns the result of an expression. If the expression is omitted the function will return null. If the return statement is used inside a generator, the generator will not be resumable anymore.

yield

stat := yield [exp]

(see Generators).

Local variables declaration

initz := id [= exp][',' initz]
stat := 'local' initz

Local variables can be declared at any point in the program; they exist between their declaration to the end of the block where they have been declared. EXCEPTION: a local declaration statement is allowed as first expression in a for loop.

```
for(local a=0;a<10;a+=1)
    print(a);
```

Function declaration

funcname := id ['::' id]
stat:= 'function' id ['::' id]+ '(' args ')'[':' '(' args ')'] stat

creates a new function.

Class declaration

```
memberdecl := id '=' exp [';'] |      '[' exp ']' '=' exp [';'] |    functionstat | 'constru
stat:= 'class' derefexp ['extends' derefexp] '{'
                    [memberdecl]
             '}'
```

creates a new class.

try/catch

```
stat:= 'try' stat 'catch' '(' id ')' stat
```

The try statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a throw statement. The catch clause provides the exceptionhandling code. When a catch clause catches an exception, its id is bound to that exception.

throw

```
stat:= 'throw' exp
```

Throws an exception. Any value can be thrown.

const

```
stat:= 'const' id '=' 'Integer | Float | StringLiteral
```

Declares a constant (see Constants & Enumerations).

enum

```
        enumerations := ( ‘id’ '=' Integer | Float | StringLiteral ) [‘,’]
        stat:= 'enum' id '{' enumerations '}'
```

Declares an enumeration (see Constants & Enumerations).

expression statement

```
stat := exp
```

In Squirrel every expression is also allowed as statement, if so, the result of the expression is thrown away.

Expressions

Assignment(=) & new slot(<-)

```
exp := derefexp '=' exp
exp:= derefexp '<-' exp
```

squirrel implements 2 kind of assignment: the normal assignment(=)

```
a=10;
```

and the "new slot" assignment.

```
a <- 10;
```

The new slot expression allows to add a new slot into a table(see Tables). If the slot already exists in the table it behaves like a normal assignment.

Operators

?: Operator

```
exp := exp_cond '?' exp1 ':' exp2
```

conditionally evaluate an expression depending on the result of an expression.

Arithmetic

```
exp:= 'exp' op 'exp'
```

Squirrel supports the standard arithmetic operators +, -, *, / and %. Other than that is also supports compact operators (+=,-=,*=,/=,%=) and increment and decrement operators(++ and --);

```
a+=2;
//is the same as writing
a=a+2;
```

```
x++
//is the same as writing
x=x+1
```

All operators work normally with integers and floats; if one operand is an integer and one is a float the result of the expression will be float. The + operator has a special behavior with strings; if one of the operands is a string the operator + will try to convert the other operand to string as well and concatenate both together. For instances and tables, _tostring is invoked.

Relational

```
exp:= 'exp' op 'exp'
```

Relational operators in Squirrel are : == < <= > >= !=

These operators return true if the expression is false and a value different than true if the expression is true. Internally the VM uses the integer 1 as true but this could change in the future.

3 ways compare

```
exp:= 'exp' op 'exp'
```

the 3 ways compare operator <=> compares 2 values A and B and returns an integer less than 0 if A < B, 0 if A == B and an integer greater than 0 if A > B.

Logical

```
exp := exp op exp
exp := '!' exp
```

Logical operators in Squirrel are : && || !

The operator && (logical and) returns null if its first argument is null, otherwise returns its second argument. The operator || (logical or) returns its first argument if is different than null, otherwise returns the second argument.

The '!' operator will return null if the given value to negate was different than null, or a value different than null if the given value was null.

in operator

exp:= keyexp 'in' tableexp

Tests the existence of a slot in a table. Returns true if keyexp is a valid key in tableexp

```
local t=
{
    foo="I'm foo",
    [123]="I'm not foo"
}

if("foo" in t) dostuff("yep");
if(123 in t) dostuff();
```

instanceof operator

exp:= instanceexp 'instanceof' classexp

Tests if a class instance is an instance of a certain class. Returns true if instanceexp is an instance of classexp.

typeof operator

exp:= 'typeof' exp

returns the type name of a value as string.

```
local a={},b="squirrel"
print(typeof a); //will print "table"
print(typeof b); //will print "string"
```

comma operator

```
exp:= exp ',' exp
```

The comma operator evaluates two expression left to right, the result of the operator is the result of the expression on the right; the result of the left expression is discarded.

```
local j=0,k=0;
for(local i=0; i<10; i++ , j++)
{
	k = i + j;
}
local a,k;
a = (k=1,k+2); //a becomes 3
```

Bitwise Operators

```
exp:= 'exp' op 'exp'
exp := '~' exp
```

Squirrel supports the standard c-like bit wise operators &,|,^,~,<<,>> plus the unsigned right shift operator >>>. The unsigned right shift works exactly like the normal right shift operator(>>) except for treating the left operand as an unsigned integer, so is not affected by the sign. Those operators only work on integers values, passing of any other operand type to these operators will cause an exception.

Operators precedence

-,~,!,typeof ,++,--	highest
/, *, %	...
+, -	
<<, >>,>>>	
<, <=, >, >=	
==, !=, <=>	
&	
^	

|	
&&, in	
||	
?:	
+=,=,-=	...
,(comma operator)	lowest

Table constructor

```
tslots := ( ‘id’ ‘=’ exp | ‘[‘ exp ‘]’ ‘=’ exp ) [‘,’]
exp := ‘{’ [tslots] ‘}’
```

Creates a new table.

```
local a={} //create an empty table
```

A table constructor can also contain slots declaration; With the syntax:

```
id = exp [',']
```

a new slot with id as key and exp as value is created

```
local a=
{
   slot1="I'm the slot value"
}
```

An alternative syntax can be

```
'[' exp1 ']' = exp2 [',']
```

A new slot with exp1 as key and exp2 as value is created

```
local a=
{
    [1]="I'm the value"
}
```

both syntaxes can be mixed

```
local table=
{
    a=10,
    b="string",
    [10]={},
    function bau(a,b)
    {
        return a+b;
    }
}
```

The comma between slots is optional.

Table with JSON syntax

Since Squirrel 3.0 is possible to declare a table using JSON syntax(see http://www.wikipedia.org/wiki/JSON).

the following JSON snippet:

```
local x = {
  "id": 1,
  "name": "Foo",
  "price": 123,
  "tags": ["Bar","Eek"]
}
```

is equivalent to the following squirrel code:

```
local x = {
  id = 1,
  name = "Foo",
  price = 123,
  tags = ["Bar","Eek"]
}
```

clone

exp:= 'clone' exp

Clone performs shallow copy of a table, array or class instance (copies all slots in the new object without recursion). If the source table has a delegate, the same delegate will be assigned as delegate (not copied) to the new table (see Delegation).

After the new object is ready the "_cloned" meta method is called (see Metamethods).

When a class instance is cloned the constructor is not invoked(initializations must rely on _cloned instead

Array constructor

exp := '[' [explist] ']'

Creates a new array.

```
a <- [] //creates an empty array
```

arrays can be initialized with values during the construction

```
a <- [1,"string!",[],{}] //creates an array with 4 elements
```

Tables

Tables are associative containers implemented as pairs of key/value (called slot); values can be any possible type and keys any type except 'null'. Tables are squirrel's skeleton, delegation and many other features are all implemented through this type; even the environment, where global variables are stored, is a table (known as root table).

Construction

Tables are created through the table constructor (see Table constructor)

Slot creation

Adding a new slot in a existing table is done through the "new slot" operator '<-'; this operator behaves like a normal assignment except that if the slot does not exists it will be created.

```
local a={}
```

The following line will cause an exception because the slot named 'newslot' does not exist in the table ‘a’

```
a.newslot = 1234
```

this will succeed:

```
a.newslot <- 1234;
```

or

```
a[1] <- "I'm the value of the new slot";
```

Slot deletion

exp:= delete derefexp

Deletion of a slot is done through the keyword delete; the result of this expression will be the value of the deleted slot.

```
a <- {
    test1=1234
    deleteme="now"
}

delete a.test1
print(delete a.deleteme); //this will print the string "now"
```

Arrays

An array is a sequence of values indexed by a integer number from 0 to the size of the array minus 1. Arrays elements can be obtained through their index.

```
local a=[“I’m a string”, 123]
print(typeof a[0]) //prints "string"
print(typeof a[1]) //prints "integer"
```

Resizing, insertion, deletion of arrays and arrays elements is done through a set of standard functions (see built-in functions).

Functions

Functions are first class values like integer or strings and can be stored in table slots, local variables, arrays and passed as function parameters. Functions can be implemented in Squirrel or in a native language with calling conventions compatible with ANSI C.

Function declaration

Functions are declared through the function expression

```
local a= function(a,b,c) {return a+b-c;}
```

or with the syntactic sugar

```
function ciao(a,b,c)
{
   return a+b-c;
}
```

that is equivalent to

```
this.ciao <- function(a,b,c)
{
   return a+b-c;
}
```

a local function can be declared with this syntactic sugar

```

local function tuna(a,b,c)
{
   return a+b-c;
}
```

that is equivalent to

```
local tuna = function(a,b,c)
{
    return a+b-c;
}
```

is also possible to declare something like

```
T <- {}
function T::ciao(a,b,c)
{
    return a+b-c;
}

//that is equivalent to write

T.ciao <- function(a,b,c)
{
    return a+b-c;
}

//or

T <- {
        function ciao(a,b,c)
        {
                return a+b-c;
        }
}
```

Default Paramaters
Squirrel's functions can have default parameters.

A function with default parameters is declared as follows:

```
function test(a,b,c = 10, d = 20)
{
        ....
}
```

when the function test is invoked and the parameter c or d are not specified, the VM autometically assigns the default value to the unspecified parameter. A default parameter can be any valid squirrel expression. The expression is evaluated at runtime.

Function with variable number of paramaters
Squirrel's functions can have variable number of parameters(varargs functions).

A vararg function is declared by adding three dots (`...´) at the end of its parameter list.

When the function is called all the extra parameters will be accessible through the *array* called vargv, that is passed as implicit parameter.

vargv is a regular squirrel array and can be used accordingly.

```
function test(a,b,...)
{
        for(local i = 0; i< vargv.len(); i++)
        {
                ::print("varparam "+i+" = "+vargv[i]+"\n");
        }
  foreach(i,val in vargv)
        {
                ::print("varparam "+i+" = "+val+"\n");
        }
}
```

```
test("goes in a","goes in b",0,1,2,3,4,5,6,7,8);
```

Function calls

exp:= derefexp ‘(‘ explist ‘)’

The expression is evaluated in this order: derefexp after the explist (arguments) and at the end the call.

Every function call in Squirrel passes the environment object ‘this’ as hidden parameter to the called function. The ‘this’ parameter is the object where the function was indexed from.

If we call a function with this syntax

```
table.foo(a)
```

the environment object passed to foo will be ‘table’

```
foo(x,y) // equivalent to this.foo(x,y)
```

The environment object will be ‘this’ (the same of the caller function).

Binding an environment to a function

while by default a squirrel function call passes as environment object 'this', the object where the function was indexed from. However, is also possible to statically bind an evironment to a closure using the built-in method closure.bindenv(env_obj). The method bindenv() returns a new instance of a closure with the environment bound to it. When an environment object is bound to a function, every time the function is invoked, its 'this' parameter will always be the previously bound environent. This mechanism is useful to implement callbacks systems similar to C# delegates.

Note
The closure keeps a weak reference to the bound environmet object, because of this if the object is deleted, the next call to the closure

will result in a null environment object.

Lambda expressions

exp := '@' '(' paramlist ')' exp

Lambda expressions are a synctactic sugar to quickly define a function that consists of a single expression. This feature comes handy when functional programming patterns are applied, like map/reduce or passing a compare method to array.sort().

here a lambda expression

```
local myexp = @(a,b) a + b
```

that is equivalent to

```
local myexp = function(a,b) { return a + b; }
```

a more useful usage could be

```
local arr = [2,3,5,8,3,5,1,2,6];
arr.sort(@(a,b) a <=> b);
arr.sort(@(a,b) -(a <=> b));
```

that could have been written as

```
local arr = [2,3,5,8,3,5,1,2,6];
arr.sort(function(a,b) { return a <=> b; } );
arr.sort(function(a,b) { return -(a <=> b); } );
```

other than being limited to a single expression lambdas support all features of regular functions. in fact are implemented as a compile time feature.

Free variables

A free variable is a variable external from the function scope as is not a local variable or parameter of the function. Free variables reference a local variable from a outer scope. In the following example the variables 'testy', 'x' and 'y' are bound to the function 'foo'.

```
local x=10,y=20
local testy="I'm testy"

function foo(a,b)
{
    ::print(testy);
    return a+b+x+y;
}
```

A program can read or write a free variable.

Tail recursion

Tail recursion is a method for partially transforming a recursion in a program into an iteration: it applies when the recursive calls in a function are the last executed statements in that function (just before the return). If this happenes the squirrel interpreter collapses the caller stack frame before the recursive call; because of that very deep recursions are possible without risk of a stack overflow.

```
function loopy(n)
{
    if(n>0){
        ::print("n="+n+"\n");
        return loopy(n-1);
    }
}

loopy(1000);
```

Classes

Squirrel implements a class mechanism similar to languages like Java/C++/etc... however because of its dynamic nature it differs in several aspects. Classes are first class objects like integer or strings and can be stored in table slots local variables, arrays and passed as function parameters.

Class declaration

A class object is created through the keyword 'class' . The class object follows the same declaration syntax of a table(see tables) with the only difference of using ';' as optional separator rather than ','.

For instance:

```
class Foo {
	//constructor
	constructor(a)
	{
		testy = ["stuff",1,2,3,a];
	}
	//member function
	function PrintTesty()
	{
		foreach(i,val in testy)
		{
			::print("idx = "+i+" = "+val+" \n");
		}
	}
	//property
	testy = null;

}
```

the previous code examples is a syntactic sugar for:

```
Foo <- class {
	//constructor
	constructor(a)
	{
		testy = ["stuff",1,2,3,a];
	}
	//member function
	function PrintTesty()
	{
		foreach(i,val in testy)
		{
			::print("idx = "+i+" = "+val+" \n");
		}
	}
	//property
	testy = null;

}
```

in order to emulate namespaces, is also possible to declare something like this

```
//just 2 regular nested tables
FakeNamespace <- {
	Utils = {}
}

class FakeNamespace.Utils.SuperClass {
	constructor()
	{
		::print("FakeNamespace.Utils.SuperClass")
	}
	function DoSomething()
	{
		::print("DoSomething()")
	}
}

function FakeNamespace::Utils::SuperClass::DoSomethingElse()
{
	::print("FakeNamespace::Utils::SuperClass::DoSomethingElse()")
}
```

```
local testy = FakeNamespace.Utils.SuperClass();
testy.DoSomething();
testy.DoSomethingElse();
```

After its declaration, methods or properties can be added or modified by following the same rules that apply to a table(operator <- and =).

```
//adds a new property
Foo.stuff <- 10;

//modifies the default value of an existing property
Foo.testy = "I'm a string";

//adds a new method
function Foo::DoSomething(a,b)
{
	return a+b;
}
```

After a class is instantiated is no longer possible to add new properties however is possible to add or replace methods.

Static variables

Squirrel's classes support static member variables. A static variable shares its value between all instances of the class. Statics are declared by prefixing the variable declaration with the keyword static; the declaration must be in the class body.

> **Note**
>
> Statics are read-only.

```
class Foo {
	constructor()
	{
		//..stuff
	}
	name = "normal variable";
	//static variable
	static classname = "The class name is foo";
};
```

Class attributes
Classes allow to associate attributes to it's members. Attributes are a form of metadata that can be used to store application specific informations, like documentations strings, properties for IDEs, code generators etc... Class attributes are declared in the class body by preceding the member declaration and are delimited by the symbol </ and />. Here an example:

```
class Foo </ test = "I'm a class level attribute" />{
	</ test = "freakin attribute" /> //attributes of PrintTesty
	function PrintTesty()
	{
		foreach(i,val in testy)
		{
			::print("idx = "+i+" = "+val+" \n");
		}
	}
	</ flippy = 10 , second = [1,2,3] /> //attributes of testy
	testy = null;

}
```

Attributes are, matter of fact, a table. Squirrel uses </ /> syntax instead of curly brackets {} for the attribute declaration to increase readability.

This means that all rules that apply to tables apply to attributes.

Attributes can be retrieved through the built-in function classobj.getattributes(membername) (see built-in functions). and can be modified/added through the built-in function classobj.setattributes(membername,val).

the following code iterates through the attributes of all Foo members.

```
foreach(member,val in Foo)
{
	::print(member+"\n");
	local attr;
	if((attr = Foo.getattributes(member)) != null) {
		foreach(i,v in attr)
		{
			::print("\t"+i+" = "+(typeof v)+"\n");
		}
	}
	else {
		::print("\t<no attributes>\n")
	}
}
```

Class instances

The class objects inherits several of the table's feature with the difference that multiple instances of the same class can be created. A class instance is an object that share the same structure of the table that created it but holds is own values. Class *instantiation* uses function notation. A class instance is created by calling a class object. Can be useful to imagine a class like a function that returns a class instance.

```
//creates a new instance of Foo
local inst = Foo();
```

When a class instance is created its member are initialized *with the same value* specified in the class declaration. The values are copied verbatim, *no cloning is performed* even if the value is a container or a class instances.

Note

FOR C# and Java programmers:

Squirrel doesn't clone member's default values nor executes the member declaration for each instace(as C# or java). So consider this example:

```
class Foo {
  myarray = [1,2,3]
  mytable = {}
}
```

```
local a = Foo();
local b = Foo();
```

In the snippet above both instances will refer to the same array and same table.To archieve what a C# or Java programmer would exepect, the following approach should be taken.

```
class Foo {
  myarray = null
  mytable = null
  constructor()
  {
    myarray = [1,2,3]
    mytable = {}
  }
}

local a = Foo();
local b = Foo();
```

When a class defines a method called 'constructor', the class instantiation operation will automatically invoke it for the newly created instance. The constructor method can have parameters, this will impact on the number of parameters that the *instantiation operation* will require. Constructors, as normal functions, can have variable number of parameters (using the parameter ...).

```
class Rect {
	constructor(w,h)
	{
		width = w;
		height = h;
	}
	x = 0;
	y = 0;
	width = null;
	height = null;
}

//Rect's constructor has 2 parameters so the class has to be 'called'
//with 2 parameters
local rc = Rect(100,100);
```

After an instance is created, its properties can be set or fetched following the same rules that apply to tables. Methods cannot be set.

Instance members cannot be removed.

The class object that created a certain instance can be retrieved through the built-in function instance.getclass()(see built-in functions)

The operator instanceof tests if a class instance is an instance of a certain class.

```
local rc = Rect(100,100);
if(rc instanceof ::Rect) {
	::print("It's a rect");
}
else {
	::print("It isn't a rect");
}
```

Note
Since Squirrel 3.x instanceof doesn't throw an exception if the left expression is not a class, it simply fails

Inheritance

Squirrel's classes support single inheritance by adding the keyword extends, followed by an expression, in the class declaration. The syntax for a derived class is the following:

```
class SuperFoo extends Foo {
	function DoSomething() {
		::print("I'm doing something");
	}
}
```

When a derived class is declared, Squirrel first copies all base's members in the new class then proceeds with evaluating the rest of the declaration.

A derived class inherit all members and properties of it's base, if the derived class overrides a base function the base implementation is shadowed. It's possible to access a overridden method of the base class by fetching the method from through the 'base' keyword.

Here an example:

```
class Foo {
	function DoSomething() {
		::print("I'm the base");
	}
};

class SuperFoo extends Foo {
	//overridden method
	function DoSomething() {
		//calls the base method
		base.DoSomething();
		::print("I'm doing something");
	}
}
```

Same rule apply to the constructor. The constructor is a regular function (apart from being automatically invoked on contruction).

```

class BaseClass {
	constructor()
	{
		::print("Base constructor\n");
	}
}

class ChildClass extends BaseClass {
	constructor()
	{
		base.constructor();
		::print("Child constructor\n");
	}
}

local test = ChildClass();
```

The base class of a derived class can be retrieved through the built-in method getbase().

```
local thebaseclass = SuperFoo.getbase();
```

Note that because methods do not have special protection policies when calling methods of the same objects, a method of a base class that calls a method of the same class can end up calling a overridden method of the derived class.

A method of a base class can be explicitly invoked by a method of a derived class though the keyword base(as in base.MyMethod()).

```
class Foo {
	function DoSomething() {
		::print("I'm the base");
	}
	function DoIt()
	{
		DoSomething();
	}
};

class SuperFoo extends Foo {
	//overridden method
	function DoSomething() {
		::print("I'm the derived");
	}
	function DoIt() {
		base.DoIt();
	}
}

//creates a new instance of SuperFoo
local inst = SuperFoo();

//prints "I'm the derived"
inst.DoIt();
```

Metamethods

Class instances allow the customization of certain aspects of the their semantics through metamethods(see Metamethods). For C++ programmers: "metamethods behave roughly like overloaded operators". The metamethods supported by classes are _add, _sub, _mul, _div, _unm, _modulo, _set, _get, _typeof, _nexti, _cmp, _call, _delslot,_tostring

Class objects instead support only 2 metamethods : _newmember and _inherited

the following example show how to create a class that implements the metamethod _add.

```
class Vector3 {
	constructor(...)
	{
		if(vargv.len() >= 3) {
			x = vargv[0];
			y = vargv[1];
			z = vargv[2];
		}
	}
	function _add(other)
	{
		return ::Vector3(x+other.x,y+other.y,z+other.z);
	}

	x = 0;
	y = 0;
	z = 0;
}

local v0 = Vector3(1,2,3)
local v1 = Vector3(11,12,13)
local v2 = v0 + v1;
::print(v2.x+","+v2.y+","+v2.z+"\n");
```

Since version 2.1, classes support 2 metamethods _inherited and _newmember. _inherited is invoked when a class inherits from the one that implements _inherited. _newmember is invoked for each member that is added to the class(at declaration time).

Generators

A function that contains a yield statement is called ‘generator function’. When a generator function is called, it does not execute the function body, instead it returns a new suspended generator. The returned generator can be resumed through the resume statement while it is alive. The yield keyword, suspends the execution of a generator and optionally returns the result of an expression to the function that resumed the generator. The generator dies when it returns, this can happen through an explicit return statement or by exiting the function body; If an unhandled exception (or runtime error) occurs while a generator is running, the generator will automatically die. A dead generator cannot be resumed anymore.

```
function geny(n)
{
    for(local i=0;i<n;i+=1)
        yield i;
    return null;
}

local gtor=geny(10);
local x;
while(x=resume gtor) print(x+”\n”);
```

the output of this program will be

```
0
1
2
3
4
5
6
7
8
9
```

generators can also be iterated using the foreach statement. When a generator is evaluated by foreach, the generator will be resumed for each iteration until it returns. The value returned by the return statement will be ignored.

Note

A suspended generator will hold a strong reference to all the values stored in it's local variables except the this object that is only a weak reference. A running generator hold a strong reference also to the this object.

Constants & Enumerations

Squirrel allows to bind constant values to an identifier that will be evaluated compile-time. This is archieved though constants and enumarations.

Constants

Constants bind a specific value to an indentifier. Constants are similar to global values, except that they are evaluated compile time and their value cannot be changed.

constants values can only be integers, floats or string literals. No expression are allowed. are declared with the following syntax.

```
const foobar = 100;
const floatbar = 1.2;
const stringbar = "I'm a contant string";
```

constants are always globally scoped, from the moment they are declared, any following code can reference them. Constants will shadow any global slot with the same name(the global slot will remain visible by using the :: syntax).

```
local x = foobar * 2;
```

Enumerations

As Constants, Enumerations bind a specific value to a name. Enumerations are also evaluated compile time and their value cannot be changed.

An enum declaration introduces a new enumeration into the program. Enumerations values can only be integers, floats or string literals. No expression are allowed.

```
enum Stuff {
  first, //this will be 0
  second, //this will be 1
  third //this will be 2
}
```

or

```
enum Stuff {
  first = 10
  second = "string"
  third = 1.2
}
```

An enum value is accessed in a manner that's similar to accessing a static class member. The name of the member must be qualified with the name of the enumeration, for example Stuff.second. Enumerations will shadow any global slot with the same name(the global slot will remain visible by using the :: syntax).

```
local x = Stuff.first * 2;
```

Implementation notes

Enumerations and Contants are a compile-time feature. Only integers, string and floats can be declared as const/enum; No expressions are allowed(because they would have to be evaluated compile time). When a const or an enum is declared, it is added compile time to the consttable. This table is stored in the VM shared state and is shared by the VM and all its threads. The consttable is a regular squirrel table; In the same way as the roottable it can be modified runtime. You can access the consttable through the built-in function getconsttable() and also change it through the built-in function setconsttable()

here some example:

```
//create a constant
getconsttable()["something"] <- 10"
//create an enumeration
getconsttable()["somethingelse"] <- { a = "10", c = "20", d = "200"};
//deletes the constant
delete getconsttable()["something"]
//deletes the enumeration
delete getconsttable()["somethingelse"]
```

This system allows to procedurally declare constants and enumerations, it is also possible to assign any squirrel type to a constant/enumeration(function,classes etc...). However this will make serialization of a code chunk impossible.

Threads

Squirrel supports cooperative threads(also known as coroutines). A cooperative thread is a subroutine that can suspended in mid-execution and provide a value to the caller without returning program flow, then its execution can be resumed later from the same point where it was suspended. At first look a Squirrel thread can be confused with a generator, in fact their behaviour is quite similar. However while a generator runs in the caller stack and can suspend only the local routine stack a thread has its own execution stack, global table and error handler; This allows a thread to suspend nested calls and have it's own error policies.

Using threads

Threads are created through the built-in function 'newthread(func)'; this function gets as parameter a squirrel function and bind it to the new thread objecs(will be the thread body). The returned thread object is initially in 'idle' state. the thread can be started with the function 'threadobj.call()'; the parameters passed to 'call' are passed to the thread function.

A thread can be be suspended calling the function suspend(), when this happens the function that wokeup(or started) the thread returns (If a parametrer is passed to suspend() it will be the return value of the wakeup function , if no parameter is passed the return value will be null). A suspended thread can be resumed calling the funtion 'threadobj.wakeup', when this happens the function that suspended the thread will return(if a parameter is passed to wakeup it will be the return value of the suspend function, if no parameter is passed the return value will be null).

A thread terminates when its main function returns or when an unhandled exception occurs during its execution.

```
function coroutine_test(a,b)
{
	::print(a+" "+b+"\n");
	local ret = ::suspend("suspend 1");
	::print("the coroutine says "+ret+"\n");
	ret = ::suspend("suspend 2");
	::print("the coroutine says "+ret+"\n");
	ret = ::suspend("suspend 3");
	::print("the coroutine says "+ret+"\n");
	return "I'm done"
}

local coro = ::newthread(coroutine_test);

local susparam = coro.call("test","coroutine"); //starts the coroutine

local i = 1;
do
{
	::print("suspend passed ("+susparam+")\n")
	susparam = coro.wakeup("ciao "+i);
	++i;
}while(coro.getstatus()=="suspended")

::print("return passed ("+susparam+")\n")
```

the result of this program will be

```
test coroutine
suspend passed (suspend 1)
the coroutine says ciao 1
suspend passed (suspend 2)
the coroutine says ciao 2
suspend passed (suspend 3)
the coroutine says ciao 3
return passed (I'm done).
```

the following is an interesting example of how threads and tail recursion can be combined.

```

function state1()
{
	::suspend("state1");
	return state2(); //tail call
}

function state2()
{
	::suspend("state2");
	return state3(); //tail call
}

function state3()
{
	::suspend("state3");
	return state1(); //tail call
}

local statethread = ::newthread(state1)

::print(statethread.call()+"\n");

for(local i = 0; i < 10000; i++)
	::print(statethread.wakeup()+"\n");
```

Weak References

The weak references allows the programmers to create references to objects without influencing the lifetime of the object itself. In squirrel Weak references are first-class objects created through the built-in method obj.weakref(). All types except null implement the weakref() method; however in bools,integers and float the method simply returns the object itself(this because this types are always passed by value). When a weak references is assigned to a container (table slot,array,class or instance) is treated differently than other objects; When a container slot that hold a weak reference is fetched, it always returns the value pointed by the weak reference instead of the weak reference object. This allow the programmer to ignore the fact that the value handled is weak. When the object pointed by weak reference is destroyed, the weak reference is automatically set to null.

```
local t = {}
local a = ["first","second","third"]
//creates a weakref to the array and assigns it to a table slot
t.thearray <- a.weakref();
```

The table slot 'thearray' contains a weak reference to an array. The following line prints "first", because tables(and all other containers) always return the object pointed by a weak ref

```
print(t.thearray[0]);
```

the only strong reference to the array is owned by the local variable 'a', so because the following line assigns a integer to 'a' the array is destroyed.

```
a = 123;
```

When an object pointed by a weak ref is destroyed the weak ref is automatically set to null, so the following line will print "null".

```
::print(typeof(t.thearray))
```

Handling weak references explicitly

If a weak reference is assigned to a local variable, then is treated as any other value.

```
local t = {}
local weakobj = t.weakref();
```

the following line prints "weakref".

```
::print(typeof(weakobj))
```

the object pointed by the weakref can be obtained through the built-in method weakref.ref().

The following line prints "table".

```
::print(typeof(weakobj.ref()))
```

Delegation

Squirrel supports implicit delegation. Every table or userdata can have a parent table (delegate). A parent table is a normal table that allows the definition of special behaviors for his child. When a table (or userdata) is indexed with a key that doesn’t correspond to one of its slots, the interpreter automatically delegates the get (or set) operation to its parent.

```
Entity <- {
}

function Entity::DoStuff()
{
    ::print(_name);
}

local newentity = {
    _name=”I’m the new entity”
}
newentity.setdelegate(Entity)

newentity.DoStuff(); //prints “I’m the new entity”
```

The delegate of a table can be retreived through built-in method table.getdelegate().

```
local thedelegate = newentity.getdelegate();
```

Metamethods

Metamethods are a mechanism that allows the customization of certain aspects of the language semantics. Those methods are normal functions placed in a table parent(delegate) or class declaration; Is possible to change many aspect of a table/class instance behavior by just defining a metamethod. Class objects(not instances) supports only 2 metamethods _newmember,_inherited.

For example when we use relational operators other than ‘==’ on 2 tables, the VM will check if the table has a method in his parent called ‘_cmp’ if so it will call it to determine the relation between the tables.

```

local comparable={
    _cmp = function (other)
    {
        if(name<other.name)return –1;
        if(name>other.name)return 1;
        return 0;
    }
}

local a={ name="Alberto" }.setdelegate(comparable);
local b={ name="Wouter" }.setdelegate(comparable);

if(a>b)
    print("a>b")
else
    print("b<=a");
```

for classes the previous code become:

```
class Comparable {
        constructor(n)
        {
                name = n;
        }
        function _cmp(other)
        {
                if(name<other.name) return -1;
                if(name>other.name) return 1;
                return 0;
        }
        name = null;
}

local a = Comparable("Alberto");
local b = Comparable("Wouter");

if(a>b)
    print("a>b")
else
    print("b<=a");
```

_set

invoked when the index idx is not present in the object or in its delegate chain. _set must 'throw null' to notify that a key wasn't found but the there were not runtime errors(clean failure). This allows the program to defferentieate between a runtime error and a 'index not found'.

```
function _set(idx,val) //returns val
```

_get

invoked when the index idx is not present in the object or in its delegate chain. _get must 'throw null' to notify that a key wasn't found but the there were not runtime errors(clean failure). This allows the program to defferentieate between a runtime error and a 'index not found'.

```
function _get(idx) //return the fetched values
```

_newslot

invoked when a script tries to add a new slot in a table.

```
function _newslot(key,value) //returns val
```

if the slot already exists in the target table the method will not be invoked also if the “new slot” operator is used.

_delslot

invoked when a script deletes a slot from a table.

if the method is invoked squirrel will not try to delete the slot himself

```
function _delslot(key)
```

_add

the + operator

```
function _add(op) //returns this+op
```

_sub

the – operator (like _add)

_mul

the * operator (like _add)

_div

the / operator (like _add)

_modulo

the % operator (like _add)

_unm

the unary minus operator

```
function _unm()
```

_typeof

invoked by the typeof operator on tables ,userdata and class instances

```
function _typeof() //returns the type of this as string
```

_cmp

invoked to emulate the < > <= >= operators

```
function _cmp(other)
```

returns an integer:

```
>0 if this > other
0  if this == other
<0 if this < other
```

_call

invoked when a table, userdata or class instance is called

```
function _call(original_this,params…)
```

_cloned

invoked when a table or class instance is cloned(in the cloned table)

```
function _cloned(original)
```

_nexti

invoked when a userdata or class instance is iterated by a foreach loop

```
function _nexti(previdx)
```

if previdx==null it means that it is the first iteration. The function has to return the index of the ‘next’ value.

_tostring

invoked when during string conacatenation or when the print function prints a table, instance or userdata. The method is also invoked by the sq_tostring() api

```
function _tostring()
```

must return a string representation of the object.

_inherited

invoked when a class object inherits from the class implementing _inherited the this contains the new class.

```
function _inherited(attributes)
```

return value is ignored.

_newmember

invoked for each member declared in a class body(at declaration time).

```
function _newmember(index,value,attributes,isstatic)
```

if the function is implemented, members will not be added to the class.

Built-in functions

The squirrel virtual machine has a set of built utility functions.

Global symbols

array(size,[fill])

create and returns array of a specified size.if the optional parameter *`fill`* is specified its value will be used to fill the new array's slots. If the *`fill`* paramter is omitted null is used instead.

seterrorhandler(func)

sets the runtime error handler

callee()

returns the currently running closure

setdebughook(hook_func)

sets the debug hook

enabledebuginfo(enable)

enable/disable the debug line information generation at compile time. enable != null enables . enable == null disables.

getroottable()

returns the root table of the VM.

setroottable(table)

sets the root table of the VM. And returns the previous root table.

getconsttable()

returns the const table of the VM.

setconsttable(table)

sets the const table of the VM. And returns the previous const table.

assert(exp)

throws an exception if exp is null

print(x)

prints x in the standard output

error(x)

prints x in the standard error output

compilestring(string,[buffername])

compiles a string containing a squirrel script into a function and returns it

```
local compiledscript=compilestring("::print(\"ciao\")");
//run the script
compiledscript();
```

collectgarbage()

runs the garbage collector and returns the number of reference cycles found(and deleted) This function only works on garbage collector builds.

resurrectunreachable()

runs the garbage collector and returns an array containing all unreachable object found. If no unreachable object is found, null is returned instead. This function is meant to help debugging reference cycles. This function only works on garbage collector builds.

type(obj)

return the 'raw' type of an object without invoking the metatmethod '_typeof'.

getstackinfos(level)

returns the stack informations of a given call stack level. returns a table formatted as follow:

```
{
        func="DoStuff",     //function name

        src="test.nut", //source file

        line=10,          //line number

        locals = {        //a table containing the local variables

                a=10,

                testy="I'm a string"
        }
}
```

level = 0 is the current function, level = 1 is the caller and so on. If the stack level doesn't exist the function returns null.

newthread(threadfunc)

creates a new cooperative thread object(coroutine) and returns it

versionnumber

integer values describing the version of VM and compiler. eg. for Squirrel 3.0.1 this value will be 301

version

string values describing the version of VM and compiler.

charsize

size in bytes of the internal VM rapresentation for characters(1 for ASCII builds

2 for UNICODE builds).

intsize

size in bytes of the internal VM rapresentation for integers(4 for 32bits builds 8 for 64bits builds).

floatsize

size in bytes of the internal VM rapresentation for floats(4 for single precision builds 8 for double precision builds).

Default delegates

Except null and userdata every squirrel object has a default delegate containing a set of functions to manipulate and retrieve information from the object itself.

Integer

tofloat()

convert the number to float and returns it

tostring()

converts the number to string and returns it

tointeger()

returns the value of the integer(dummy function)

tochar()

returns a string containing a single character rapresented by the integer.

weakref()

dummy function, returns the integer itself.

Float

tofloat()

returns the value of the float(dummy function)

tointeger()

converts the number to integer and returns it

tostring()

converts the number to string and returns it

tochar()

returns a string containing a single character rapresented by the integer part of the float.

weakref()

dummy function, returns the float itself.

Bool

tofloat()

returns 1.0 for true 0.0 for false

tointeger()

returns 1 for true 0 for false

tostring()

returns "true" for true "false" for false

weakref()

dummy function, returns the bool itself.

String

len()

returns the string length

tointeger()

converts the string to integer and returns it

tofloat()

converts the string to float and returns it

tostring()

returns the string(dummy function)

slice(start,[end])

returns a section of the string as new string. Copies from start to the end (not included). If start is negative the index is calculated as length + start, if end is negative the index is calculated as length + end. If end is omitted end is equal to the string length.

find(substr,[startidx])

search a sub string(substr) starting from the index startidx and returns the index of its first occurrence. If startidx is omitted the search operation starts from the beginning of the string. The function returns null if substr is not found.

tolower()

returns a lowercase copy of the string.

toupper()

returns a uppercase copy of the string.

weakref()

returns a weak reference to the object.

Table

len()

returns the number of slots contained in a table

rawget(key)

tries to get a value from the slot ‘key’ without employing delegation

rawset(key,val)

sets the slot ‘key’ with the value ‘val’ without employing delegation. If the slot does not exists , it will be created.

rawdelete()

deletes the slot key without emplying delegetion and retunrs his value. if the slo does not exists returns always null.

rawin(key)

returns true if the slot ‘key’ exists. the function has the same eddect as the operator 'in' but does not employ delegation.

weakref()

returns a weak reference to the object.

tostring()

tries to invoke the _tostring metamethod, if failed. returns "(table : pointer)".

clear()

removes all the slot from the table

setdelegate(table)

sets the delegate of the table, to remove a delegate 'null' must be passed to the function. The function returns the table itself (eg. a.setdelegate(b) in this case 'a' is the return value).

getdelegate()

returns the table's delegate or null if no delegate was set.

Array

len()

returns the length of the array

append(val)

appends the value ‘val’ at the end of the array

push(val)

appends the value ‘val’ at the end of the array

extend(array)

Extends the array by appending all the items in the given array.

pop()

removes a value from the back of the array and returns it.

top()

returns the value of the array with the higher index

insert(idx,val)

inserst the value ‘val’ at the position ‘idx’ in the array

remove(idx)

removes the value at the position ‘idx’ in the array

resize(size,[fill])

resizes the array, if the optional parameter *`fill`* is specified its value will be used to fill the new array's slots(if the size specified is bigger than the previous size) . If the *`fill`* paramter is omitted null is used instead.

sort([compare_func])

sorts the array. a custom compare function can be optionally passed.The function prototype as to be the following.

```
function custom_compare(a,b)
{
        if(a>b) return 1
        else if(a<b) return -1
        return 0;
}
```

a more compact version of a custom compare can be written using a lambda expression and the operator <=>

```
arr.sort(@(a,b) a <=> b);
```

reverse()

reverse the elements of the array in place

slice(start,[end])

returns a section of the array as new array. Copies from start to the end (not included). If start is negative the index is calculated as length + start, if end is negative the index is calculated as length + end. If end is omitted end is equal to the array length.

weakref()

returns a weak reference to the object.

tostring()

returns the string "(array : pointer)".

clear()

removes all the items from the array

map(func(a))

creates a new array of the same size. for each element in the original array invokes the function 'func' and assigns the return value of the function to the corresponding element of the newly created array.

apply(func(a))

for each element in the array invokes the function 'func' and replace the original value of the element with the return value of the function.

reduce(func(prevval,curval))

Reduces an array to a single value. For each element in the array invokes the function 'func' passing the initial value (or value from the previous callback call) and the value of the current element. the return value of the function is then used as 'prevval' for the next element. Given an array of length 0, returns null. Given an array of length 1, returns the first element. Given an array with 2 or more elements calls the function with the first two elements as the parameters, gets that result, then calls the function with that result and the third element, gets that result, calls the function with that result and the fourth parameter and so on until all element have been processed. Finally returns the return value of the last invocation of func.

filter(func(index,val))

Creates a new array with all elements that pass the test implemented by the provided function. In detail, it creates a new array, for each element in the original array invokes the specified function passing the index of the element and it's value; if the function returns 'true', then the value of the corresponding element is added on the newly created array.

find(value)

Performs a linear search for the value in the array. Returns the index of the value if it was found null otherwise.

Function

call(_this,args…)

calls the function with the specified environment object(’this’) and parameters

pcall(_this,args…)

calls the function with the specified environment object(’this’) and parameters, this function will not invoke the error callback in case of failure(pcall stays for 'protected call')

acall(array_args)

calls the function with the specified environment object(’this’) and parameters. The function accepts an array containing the parameters that will be passed to the called function.Where array_args has to contain the required 'this' object at the [0] position.

pacall(array_args)

calls the function with the specified environment object(’this’) and parameters. The function accepts an array containing the parameters that will be passed to the called function.Where array_args has to contain the required 'this' object at the [0] position. This function will not invoke the error callback in case of failure(pacall stays for 'protected array call')

weakref()

returns a weak reference to the object.

tostring()

returns the string "(closure : pointer)".

bindenv(env)

clones the function(aka closure) and bind the enviroment object to it(table,class or instance). the this parameter of the newly create function will always be set to env. Note that the created function holds a weak reference to its environment object so cannot be used to control its lifetime.

getinfos()

returns a table containing informations about the function, like parameters, name and source name;

```
//the data is returned as a table is in form
//pure squirrel function
{
  native = false
  name = "zefuncname"
  src = "/somthing/something.nut"
  parameters = ["a","b","c"]
  defparams = [1,"def"]
  varargs = 2
}
//native C function
{
  native = true
  name = "zefuncname"
  paramscheck = 2
  typecheck = [83886082,83886384] //this is the typemask (see C defines OT_IN
}
```

Class

instance()

returns a new instance of the class. this function does not invoke the instance constructor. The constructor must be explicitly called(eg. class_inst.constructor(class_inst)).

getattributes(membername)

returns the attributes of the specified member. if the parameter member is null the function returns the class level attributes.

setattributes(membername,attr)

sets the attribute of the specified member and returns the previous attribute value. if the parameter member is null the function sets the class level attributes.

rawin(key)

returns true if the slot ‘key’ exists. the function has the same eddect as the operator 'in' but does not employ delegation.

weakref()

returns a weak reference to the object.

tostring()

returns the string "(class : pointer)".

rawget(key)

tries to get a value from the slot ‘key’ without employing delegation

rawset(key,val)

sets the slot ‘key’ with the value ‘val’ without employing delegation. If the slot does not exists , it will be created.

newmember(key,val,[attrs],[bstatic])

sets/adds the slot ‘key’ with the value ‘val’ and attributes 'attrs' and if present invokes the _newmember metamethod. If bstatic is true the slot will be added as static. If the slot does not exists , it will be created.

rawnewmember(key,val,[attrs],[bstatic])

sets/adds the slot ‘key’ with the value ‘val’ and attributes 'attrs'.If bstatic is true the slot will be added as static. If the slot does not exists , it will be created. It doesn't invoke any metamethod.

Class Instance

getclass()

returns the class that created the instance.

rawin(key)

returns true if the slot ‘key’ exists. the function has the same eddect as the operator 'in' but does not employ delegation.

weakref()

returns a weak reference to the object.

tostring()

tries to invoke the _tostring metamethod, if failed. returns "(insatnce : pointer)".

rawget(key)

tries to get a value from the slot ‘key’ without employing delegation

rawset(key,val)

sets the slot ‘key’ with the value ‘val’ without employing delegation. If the slot does not exists , it will be created.

Generator

getstatus()

returns the status of the generator as string : “running”, ”dead” or ”suspended”.

weakref()

returns a weak reference to the object.

tostring()

returns the string "(generator : pointer)".

Thread

call(...)

starts the thread with the specified parameters

wakeup([wakeupval])

wakes up a suspended thread, accepts a optional parameter that will be used as return value for the function that suspended the thread(usually suspend())

getstatus()

returns the status of the thread ("idle","running","suspended")

weakref()

returns a weak reference to the object.

tostring()

returns the string "(thread : pointer)".

getstackinfos(stacklevel)

returns the stack frame informations at the given stack level (0 is the current function 1 is the caller and so on).

Weak Reference

ref()

returns the object that the weak reference is pointing at, null if the object that was point at was destroyed.

weakref()

returns a weak reference to the object.

tostring()

returns the string "(weakref : pointer)".

Chapter 3. Embedding Squirrel

This section describes how to embed Squirrel in a host application, C language knowledge is required to understand this part of the manual.

Because of his nature of extension language, Squirrel’s compiler and virtual machine are implemented as C library. The library exposes a set of functions to compile scripts, call functions, manipulate data and extend the virtual machine. All declarations needed for embedding the language in an application are in the header file ‘squirrel.h’.

Memory management

Squirrel uses reference counting (RC) as primary system for memory management; however, the virtual machine (VM) has an auxiliary mark and sweep garbage collector that can be invoked on demand.

There are 2 possible compile time options:

- The default configuration consists in RC plus a mark and sweep garbage collector. The host program can call the function sq_collectgarbage() and perform a garbage collection cycle during the program execution. The garbage collector isn’t invoked by the VM and has to be explicitly called by the host program.

- The second a situation consists in RC only(define NO_GARBAGE_COLLECTOR); in this case is impossible for the VM to detect reference cycles, so is the programmer that has to solve them explicitly in order to avoid memory leaks.

The only advantage introduced by the second option is that saves 2 additional pointers that have to be stored for each object in the default configuration with garbage collector(8 bytes for 32 bits systems). The types involved are: tables, arrays, functions, threads, userdata and generators; all other types are untouched. These options do not affect execution speed.

Unicode

By default Squirrel strings are plain 8-bits ASCII characters; however if the symbol 'SQUNICODE' is defined the VM, compiler and API will use 16-bits characters.

Squirrel on 64 bits architectures

Squirrel can be compiled on 64 bits architectures by defining '_SQ64' in the C++ preprocessor. This flag should be defined in any project that includes 'squirrel.h'.

Userdata alignment

Both class instances and userdatas can have a buffer associated to them. Squirrel specifies the alignment(in bytes) through the peroprocessor defining 'SQ_ALIGNMENT'. By default SQ_ALIGNMENT is defined as 4 for 32 bits builds and 8 for 64bits builds and builds that use 64bits floats. It is possible to override the value of SQ_ALIGNMENT respecting the following rules. SQ_ALIGNMENT shall be less than or equal to SQ_MALLOC alignments, and it shall be power of 2.

> **Note**
>
> This only applies for userdata allocated by the VM, specified via sq_setclassudsize() or belonging to a userdata object. userpointers specified by the user are not affected by alignemnt rules.

Stand-alone VM without compiler

Squirrel's VM can be compiled without it's compiler by defining 'NO_COMPILER' in the C++ preprocessor. When 'NO_COMPILER' is defined all function related to the compiler (eg. sq_compile) will fail. Other functions that conditionally load precompiled bytecode or compile a file (eg. sqstd_dofile) will only work with precompiled bytecode.

Error conventions

Most of the functions in the API return a SQRESULT value; SQRESULT indicates if a function completed successfully or not. The macros SQ_SUCCEEDED() and SQ_FAILED() are used to test the result of a function.

```
if(SQ_FAILED(sq_getstring(v,-1,&s)))
    printf("getstring failed");
```

Initializing Squirrel

The first thing that a host application has to do, is create a virtual machine. The host application can create any number of virtual machines through the function sq_open().

Every single VM has to be released with the function sq_close() when it is not needed anymore.

```
int main(int argc, char* argv[])
{
    HSQUIRRELVM v;
    v = sq_open(1024); //creates a VM with initial stack size 1024

    //do some stuff with squirrel here

    sq_close(v);
}
```

The Stack

Squirrel exchanges values with the virtual machine through a stack. This mechanism has been inherited from the language Lua. For instance to call a Squirrel function from C it is necessary to push the function and the arguments in the stack and then invoke the function; also when Squirrel calls a C function the parameters will be in the stack as well.

Stack indexes

Many API functions can arbitrarily refer to any element in the stack through an index. The stack indexes follow those conventions:

- 1 is the stack base
- Negative indexes are considered an offset from top of the stack. For instance –1 is the top of the stack.
- 0 is an invalid index

Here an example (let’s pretend that this table is the VM stack)

STACK		*positive index*	*negative index*
"test"		4	-1(top)
1		3	-2
0.5		2	-3
"foo"		1(base)	-4

In this case, the function sq_gettop would return 4;

Stack manipulation

The API offers several functions to push and retrieve data from the Squirrel stack.

To push a value that is already present in the stack in the top position

```
void sq_push(HSQUIRRELVM v,SQInteger idx);
```

To pop an arbitrary number of elements

```
void sq_pop(HSQUIRRELVM v,SQInteger nelemstopop);
```

To remove an element from the stack

```
void sq_remove(HSQUIRRELVM v,SQInteger idx);
```

To retrieve the top index (and size) of the current virtual stack you must call sq_gettop

```
SQInteger sq_gettop(HSQUIRRELVM v);
```

To force the stack to a certain size you can call sq_settop

```
void sq_settop(HSQUIRRELVM v,SQInteger newtop);
```

If the newtop is bigger than the previous one, the new posistions in the stack will be filled with null values.

The following function pushes a C value into the stack

```
void sq_pushstring(HSQUIRRELVM v,const SQChar *s,SQInteger len);
void sq_pushfloat(HSQUIRRELVM v,SQFloat f);
void sq_pushinteger(HSQUIRRELVM v,SQInteger n);
void sq_pushuserpointer(HSQUIRRELVM v,SQUserPointer p);
void sq_pushbool(HSQUIRRELVM v,SQBool b);
```

this function pushes a null into the stack

```
void sq_pushnull(HSQUIRRELVM v);
```

returns the type of the value in a arbitrary position in the stack

```
SQObjectType sq_gettype(HSQUIRRELVM v,SQInteger idx);
```

the result can be one of the following values:

```
OT_NULL,OT_INTEGER,OT_FLOAT,OT_STRING,OT_TABLE,OT_ARRAY,
OT_CLOSURE,OT_NATIVECLOSURE,OT_GENERATOR,OT_USERPOINTE
```

The following functions convert a squirrel value in the stack to a C value

```
SQRESULT sq_getstring(HSQUIRRELVM v,SQInteger idx,const SQChar **c);
SQRESULT sq_getinteger(HSQUIRRELVM v,SQInteger idx,SQInteger *i);
SQRESULT sq_getfloat(HSQUIRRELVM v,SQInteger idx,SQFloat *f);
SQRESULT sq_getuserpointer(HSQUIRRELVM v,SQInteger idx,SQUserPointe
SQRESULT sq_getuserdata(HSQUIRRELVM v,SQInteger idx,SQUserPointer *
SQRESULT sq_getbool(HSQUIRRELVM v,SQInteger idx,SQBool *p);
```

The function sq_cmp compares 2 values from the stack and returns their relation (like strcmp() in ANSI C).

```
SQInteger sq_cmp(HSQUIRRELVM v);
```

Runtime error handling

When an exception is not handled by Squirrel code with a try/catch statement, a runtime error is raised and the execution of the current program is interrupted. It is possible to set a call back function to intercept the runtime error from the host program; this is useful to show meaningful errors to the script writer and for implementing visual debuggers. The following API call pops a Squirrel function from the stack and sets it as error handler.

```
SQUIRREL_API void sq_seterrorhandler(HSQUIRRELVM v);
```

The error handler is called with 2 parameters, an environment object (this) and a object. The object can be any squirrel type.

Compiling a script

You can compile a Squirrel script with the function sq_compile.

```
typedef SQInteger (*SQLEXREADFUNC)(SQUserPointer userdata);

SQRESULT sq_compile(HSQUIRRELVM v,SQREADFUNC read,SQUserPointe
        const SQChar *sourcename,SQBool raiseerror);
```

In order to compile a script is necessary for the host application to implement a reader function (SQLEXREADFUNC); this function is used to feed the compiler with the script data. The function is called every time the compiler needs a character; It has to return a character code if succeed or 0 if the source is finished.

If sq_compile succeeds, the compiled script will be pushed as Squirrel function in the stack.

> **Note**
> In order to execute the script, the function generated by sq_compile() has to be called through sq_call()

Here an example of a ‘read’ function that read from a file:

```
SQInteger file_lexfeedASCII(SQUserPointer file)
{
    int ret;
    char c;
    if( ( ret=fread(&c,sizeof(c),1,(FILE *)file )>0) )
        return c;
    return 0;
}

int compile_file(HSQUIRRELVM v,const char *filename)
{
    FILE *f=fopen(filename,”rb”);
    if(f)
    {
         sq_compile(v,file_lexfeedASCII,f,filename,1);
         fclose(f);
         return 1;
    }
    return 0;
}
```

When the compiler fails for a syntax error it will try to call the ‘compiler error handler’; this function must be declared as follow

```
typedef void (*SQCOMPILERERROR)(HSQUIRRELVM /*v*/,const SQChar *
/*source*/,SQInteger /*line*/,SQInteger /*column*/);
```

and can be set with the following API call

```
void sq_setcompilererrorhandler(HSQUIRRELVM v,SQCOMPILERERROR f);
```

Calling a function

To call a squirrel function it is necessary to push the function in the stack followed by the parameters and then call the function sq_call. The function will pop the parameters and push the return value if the last sq_call parameter is >0.

```
sq_pushroottable(v);
sq_pushstring(v,“foo”,-1);
sq_get(v,-2); //get the function from the root table
sq_pushroottable(v); //’this’ (function environment object)
sq_pushinteger(v,1);
sq_pushfloat(v,2.0);
sq_pushstring(v,”three”,-1);
sq_call(v,4,SQFalse);
sq_pop(v,2); //pops the roottable and the function
```

this is equivalent to the following Squirrel code

```
foo(1,2.0,”three”);
```

If a runtime error occurs (or a exception is thrown) during the squirrel code execution the sq_call will fail.

Create a C function

A native C function must have the following prototype:

```
typedef SQInteger (*SQFUNCTION)(HSQUIRRELVM);
```

The parameters is an handle to the calling VM and the return value is an integer respecting the following rules:

- 1 if the function returns a value
- 0 if the function does not return a value
- SQ_ERROR runtime error is thrown

In order to obtain a new callable squirrel function from a C function pointer, is necessary to call sq_newclosure() passing the C function to it; the new Squirrel function will be pushed in the stack.

When the function is called, the stackbase is the first parameter of the function and the top is the last. In order to return a value the function has to push it in the stack and return 1.

Function parameters are in the stack from postion 1 ('this') to n. sq_gettop() can be used to determinate the number of parameters.

If the function has free variables, those will be in the stack after the explicit parameters an can be handled as normal parameters. Note also that the value returned bysq_gettop() will be affected by free variables. sq_gettop() will return the number of parameters plus number of free variables.

Here an example, the following function print the value of each argument and return the number of arguments.

```

SQInteger print_args(HSQUIRRELVM v)
{
   SQInteger nargs = sq_gettop(v); //number of arguments
```

```
for(SQInteger n=1;n<=nargs;n++)
{
    printf("arg %d is ",n);
    switch(sq_gettype(v,n))
    {
        case OT_NULL:
            printf("null");
            break;
        case OT_INTEGER:
            printf("integer");
            break;
        case OT_FLOAT:
            printf("float");
            break;
        case OT_STRING:
            printf("string");
            break;
        case OT_TABLE:
            printf("table");
            break;
        case OT_ARRAY:
            printf("array");
            break;
        case OT_USERDATA:
            printf("userdata");
            break;
        case OT_CLOSURE:
            printf("closure(function)");
            break;
        case OT_NATIVECLOSURE:
            printf("native closure(C function)");
            break;
        case OT_GENERATOR:
            printf("generator");
            break;
        case OT_USERPOINTER:
            printf("userpointer");
            break;
        case OT_CLASS:
            printf("class");
            break;
        case OT_INSTANCE:
            printf("instance");
            break;
        case OT_WEAKREF:
            printf("weak reference");
            break;
        default:
            return sq_throwerror(v,"invalid param"); //throws an exception
        }
    }
    printf("\n");
    sq_pushinteger(v,nargs); //push the number of arguments as return value
    return 1; //1 because 1 value is returned
}
```

Here an example of how to register a function

```
SQInteger register_global_func(HSQUIRRELVM v,SQFUNCTION f,const char
{
    sq_pushroottable(v);
    sq_pushstring(v,fname,-1);
    sq_newclosure(v,f,0,0); //create a new function
    sq_newslot(v,-3,SQFalse);
    sq_pop(v,1); //pops the root table
}
```

Tables and arrays manipulation

A new table is created calling sq_newtable, this function pushes a new table in the stack.

```
void sq_newtable (HSQUIRRELVM v);
```

To create a new slot

```
SQRESULT sq_newslot(HSQUIRRELVM v,SQInteger idx,SQBool bstatic);
```

To set or get the table delegate

```
SQRESULT sq_setdelegate(HSQUIRRELVM v,SQInteger idx);
SQRESULT sq_getdelegate(HSQUIRRELVM v,SQInteger idx);
```

A new array is created calling sq_newarray, the function pushes a new array in the stack; if the parameters size is bigger than 0 the elements are initialized to null.

```
void sq_newarray (HSQUIRRELVM v,SQInteger size);
```

To append a value to the back of the array

```
SQRESULT sq_arrayappend(HSQUIRRELVM v,SQInteger idx);
```

To remove a value from the back of the array

```
SQRESULT sq_arraypop(HSQUIRRELVM v,SQInteger idx,SQInteger pushval);
```

To resize the array

```
SQRESULT sq_arrayresize(HSQUIRRELVM v,SQInteger idx,SQInteger newsiz
```

To retrieve the size of a table or an array you must use sq_getsize()

```
SQInteger sq_getsize(HSQUIRRELVM v,SQInteger idx);
```

To set a value in an array or table

```
SQRESULT sq_set(HSQUIRRELVM v,SQInteger idx);
```

To get a value from an array or table

```
SQRESULT sq_get(HSQUIRRELVM v,SQInteger idx);
```

To get or set a value from a table without employ delegation

```
SQRESULT sq_rawget(HSQUIRRELVM v,SQInteger idx);
SQRESULT sq_rawset(HSQUIRRELVM v,SQInteger idx);
```

To iterate a table or an array

```
SQRESULT sq_next(HSQUIRRELVM v,SQInteger idx);
```

Here an example of how to perform an iteration:

```
//push your table/array here
sq_pushnull(v)  //null iterator
while(SQ_SUCCEEDED(sq_next(v,-2)))
{
    //here -1 is the value and -2 is the key

    sq_pop(v,2); //pops key and val before the nex iteration
}

sq_pop(v,1); //pops the null iterator
```

Userdata and UserPointers

Squirrel allows the host application put arbitrary data chunks into a Squirrel value, this is possible through the data type userdata.

```
SQUserPointer sq_newuserdata (HSQUIRRELVM v,SQUnsignedInteger size);
```

When the function sq_newuserdata is called, Squirrel allocates a new userdata with the specified size, returns a pointer to his payload buffer and push the object in the stack; at this point the application can do whatever it want with this memory chunk, the VM will automatically take cake of the memory deallocation like for every other built-in type. A userdata can be passed to a function or stored in a table slot. By default Squirrel cannot manipulate directly userdata; however is possible to assign a delegate to it and define a behavior like it would be a table. Because the application would want to do something with the data stored in a userdata object when it get deleted, is possible to assign a callback that will be called by the VM just before deleting a certain userdata. This is done through the API call sq_setreleasehook.

```
typedef SQInteger (*SQRELEASEHOOK)(SQUserPointer,SQInteger size);

void sq_setreleasehook(HSQUIRRELVM v,SQInteger idx,SQRELEASEHOOK
```

Another kind of userdata is the userpointer; this type is not a memory chunk like the normal userdata, but just a ‘void*’ pointer. It cannot have a delegate and is passed by value, so pushing a userpointer doesn’t cause any memory allocation.

```
void sq_pushuserpointer(HSQUIRRELVM v,SQUserPointer p);
```

The registry table

The registry table is an hidden table shared between vm and all his thread(friend vms). This table is accessible only through the C API and is ment to be an utility structure for native C library implementation. For instance the sqstdlib(squirrel standard library)uses it to store configuration and shared objects delegates. The registry is accessible through the API call sq_pushregistrytable.

```
void sq_pushregistrytable(HSQUIRRELVM v);
```

Mantaining strong references to Squirrel values from the C API

Squirrel allows to reference values through the C API; the function sq_getstackobj() gets a handle to a squirrel object(any type). The object handle can be used to control the lifetime of an object by adding or removing references to it(see sq_addref() and sq_release()). The object can be also re-pushed in the VM stack using sq_pushobject().

```
HSQOBJECT obj;

sq_resetobject(v,&obj) //initialize the handle
sq_getstackobj(v,-2,&obj); //retrieve an object handle from the pos –2
sq_addref(v,&obj); //adds a reference to the object

… //do stuff

sq_pushobject(v,&obj); //push the object in the stack
sq_release(v,&obj); //relese the object
```

Debug Interface

The squirrel VM exposes a very simple debug interface that allows to easily built a full featured debugger. Through the functions sq_setdebughook and sq_setnativedebughook is possible in fact to set a callback function that will be called every time the VM executes an new line of a script or if a function get called/returns. The callback will pass as argument the current line the current source and the current function name (if any).

```
SQUIRREL_API void sq_setdebughook(HSQUIRRELVM v);
```

or

```
SQUIRREL_API void sq_setnativedebughook(HSQUIRRELVM v,SQDEBUGH
```

The following code shows how a debug hook could look like(obviously is possible to implement this function in C as well).

```
function debughook(event_type,sourcefile,line,funcname)
{
    local fname=funcname?funcname:"unknown";
    local srcfile=sourcefile?sourcefile:"unknown"
    switch (event_type) {
    case 'l': //called every line(that contains some code)
        ::print("LINE line [" + line + "] func [" + fname + "]");
        ::print("file [" + srcfile + "]\n");
                break;
    case 'c': //called when a function has been called
        ::print("LINE line [" + line + "] func [" + fname + "]");
        ::print("file [" + srcfile + "]\n");
                break;
    case 'r': //called when a function returns
        ::print("LINE line [" + line + "] func [" + fname + "]");
        ::print("file [" + srcfile + "]\n");
                break;
        }
}
```

The parameter event_type can be 'l' ,'c' or 'r' ; a hook with a 'l' event is called for each line that gets executed, 'c' every time a function gets called and 'r' every time a function returns.

A full-featured debugger always allows displaying local variables and calls stack. The call stack information are retrieved through sq_getstackinfos()

```
SQInteger sq_stackinfos(HSQUIRRELVM v,SQInteger level,SQStackInfos *si);
```

While the local variables info through sq_getlocal()

```
SQInteger sq_getlocal(HSQUIRRELVM v,SQUnsignedInteger level,SQUnsigne
```

In order to receive line callbacks the scripts have to be compiled with debug infos enabled this is done through sq_enabledebuginfo();

```
void sq_enabledebuginfo(HSQUIRRELVM v, SQInteger debuginfo);
```

Chapter 4. API Reference

Virtual Machine

sq_close

void **sq_close**(*HSQUIRRELVM v*);

releases a squirrel VM and all related friend VMs

parameters:

`HSQUIRRELVM v`

the target VM

sq_geterrorfunc

SQPRINTFUNCTION **sq_geterrorfunc**(*HSQUIRRELVM v*);

returns the current error function of the given Virtual machine. (see sq_setprintfunc())

parameters:

`HSQUIRRELVM v`

the target VM

return:

a pointer to a SQPRINTFUNCTION, or NULL if no function has been set.

sq_getforeignptr

SQUserPointer **sq_getforeignptr**(*HSQUIRRELVM v*);

Returns the foreign pointer of a VM instance.

parameters:

`HSQUIRRELVM v`

the target VM

return:

the current VMs foreign pointer.

sq_getprintfunc

SQPRINTFUNCTION **sq_getprintfunc**(*HSQUIRRELVM v*);

returns the current print function of the given Virtual machine. (see sq_setprintfunc())

parameters:

`HSQUIRRELVM v`

the target VM

return:

a pointer to a SQPRINTFUNCTION, or NULL if no function has been set.

sq_getversion

SQInteger **sq_getversion**();

returns the version number of the vm.

return:

version number of the vm(as in SQUIRREL_VERSION_NUMBER).

sq_getvmstate

SQInteger **sq_getvmstate**(*HSQUIRRELVM v*);

returns the execution state of a virtual machine

parameters:

`HSQUIRRELVM v`

the target VM

return:

the state of the vm encoded as integer value. The following constants are defined: SQ_VMSTATE_IDLE, SQ_VMSTATE_RUNNING, SQ_VMSTATE_SUSPENDED.

sq_move

void **sq_move**(*HSQUIRRELVM dest*, *HSQUIRRELVM src*, *SQInteger idx*);

pushes the object at the position 'idx' of the source vm stack in the destination vm stack.

parameters:

`HSQUIRRELVM dest`

the destination VM

`HSQUIRRELVM src`

the source VM

`SQInteger idx`

the index in the source stack of the value that has to be moved

sq_newthread

HSQUIRRELVM **sq_newthread**(*HSQUIRRELVM friendvm, SQInteger initialstacksize*);

creates a new vm friendvm of the one passed as first parmeter and pushes it in its stack as "thread" object.

parameters:

`HSQUIRRELVM friendvm`

a friend VM

`SQInteger initialstacksize`

the size of the stack in slots(number of objects)

return:

a pointer to the new VM.

remarks:

By default the roottable is shared with the VM passed as first parameter. The new VM lifetime is bound to the "thread" object pushed in the stack and behave like a normal squirrel object.

sq_open

HSQUIRRELVM **sq_open**(*SQInteger initialstacksize*);

creates a new instance of a squirrel VM that consists in a new execution stack.

parameters:

`SQInteger initialstacksize`

the size of the stack in slots(number of objects)

return:

an handle to a squirrel vm

remarks:

the returned VM has to be released with sq_releasevm sq_pushconsttable

void **sq_pushconsttable**(*HSQUIRRELVM v*);

pushes the current const table in the stack

parameters:

HSQUIRRELVM v

the target VM

sq_pushregistrytable

void **sq_pushregistrytable**(*HSQUIRRELVM v*);

pushes the registry table in the stack

parameters:

`HSQUIRRELVM v`

the target VM

sq_pushroottable

void **sq_pushroottable**(*HSQUIRRELVM v*);

pushes the current root table in the stack

parameters:

HSQUIRRELVM v

the target VM

sq_setconsttable

void **sq_setconsttable**(*HSQUIRRELVM v*);

pops a table from the stack and set it as const table

parameters:

HSQUIRRELVM v

the target VM

sq_seterrorhandler

void **sq_seterrorhandler**(*HSQUIRRELVM v*);

pops from the stack a closure or native closure an sets it as runtime-error handler.

parameters:

`HSQUIRRELVM v`

the target VM

remarks:

the error handler is shared by friend VMs

sq_setforeignptr

void **sq_setforeignptr**(*HSQUIRRELVM v*, *SQUserPointer p*);

Sets the foreign pointer of a certain VM instance. The foreign pointer is an arbitrary user defined pointer associated to a VM (by default is value id 0). This pointer is ignored by the VM.

parameters:

`HSQUIRRELVM v`

the target VM

`SQUserPointer p`

The pointer that has to be set

sq_setprintfunc

void **sq_setprintfunc**(*HSQUIRRELVM v*, *SQPRINTFUNCTION printfunc*, *SQPRINTFUNCTION errorfunc*);

sets the print function of the virtual machine. This function is used by the built-in function '::print()' to output text.

parameters:

`HSQUIRRELVM v`

the target VM

`SQPRINTFUNCTION printfunc`

a pointer to the print func or NULL to disable the output.

`SQPRINTFUNCTION errorfunc`

a pointer to the error func or NULL to disable the output.

remarks:

the print func has the following prototype: void printfunc(HSQUIRRELVM v,const SQChar *s,...)

sq_setroottable

void **sq_setroottable**(*HSQUIRRELVM v*);

pops a table from the stack and set it as root table

parameters:

`HSQUIRRELVM v`

the target VM

sq_suspendvm

HRESULT **sq_suspendvm**(*HSQUIRRELVM v*);

Suspends the execution of the specified vm.

parameters:

`HSQUIRRELVM v`

the target VM

return:

an SQRESULT(that has to be returned by a C function)

remarks:

sq_result can only be called as return expression of a C function. The function will fail is the suspension is done through more C calls or in a metamethod.

eg.

```
SQInteger suspend_vm_example(HSQUIRRELVM v)
{
	return sq_suspendvm(v);
}
```

sq_wakeupvm

HRESULT **sq_wakeupvm**(*HSQUIRRELVM v*, *SQBool resumedret*, *SQBool retval*, *SQBool raiseerror*, *SQBool throwerror*);

Wake up the execution a previously suspended virtual machine.

parameters:

`HSQUIRRELVM v`

the target VM

`SQBool resumedret`

if true the function will pop a value from the stack and use it as return value for the function that has previously suspended the virtual machine.

`SQBool retval`

if true the function will push the return value of the function that suspend the excution or the main function one.

`SQBool raiseerror`

if true, if a runtime error occurs during the execution of the call, the vm will invoke the error handler.

`SQBool throwerror`

if true, the vm will thow an exception as soon as is resumed. the exception payload must be set beforehand invoking sq_thowerror().

return:

an HRESULT.

Compiler

sq_compile

SQRESULT **sq_compile**(*HSQUIRRELVM v*, *HSQLEXREADFUNC read*, *SQUserPointer p*, *const SQChar * sourcename*, *SQBool raiseerror*);

compiles a squirrel program; if it succeeds, push the compiled script as function in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`HSQLEXREADFUNC read`

a pointer to a read function that will feed the compiler with the program.

`SQUserPointer p`

a user defined pointer that will be passed by the compiler to the read function at each invocation.

`const SQChar * sourcename`

the symbolic name of the program (used only for more meaningful runtime errors)

`SQBool raiseerror`

if this value is true the compiler error handler will be called in case of an error

return:

a SQRESULT. If the sq_compile fails nothing is pushed in the stack.

remarks:

in case of an error the function will call the function set by sq_setcompilererrorhandler().

sq_compilebuffer

SQRESULT **sq_compilebuffer**(*HSQUIRRELVM v*, *const SQChar* s*, *SQInteger size*, *const SQChar * sourcename*, *SQBool raiseerror*);

compiles a squirrel program from a memory buffer; if it succeeds, push the compiled script as function in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`const SQChar* s`

a pointer to the buffer that has to be compiled.

`SQInteger size`

size in characters of the buffer passed in the parameter 's'.

`const SQChar * sourcename`

the symbolic name of the program (used only for more meaningful runtime errors)

`SQBool raiseerror`

if this value true the compiler error handler will be called in case of an error

return:

a SQRESULT. If the sq_compilebuffer fails nothing is pushed in the stack.

remarks:

in case of an error the function will call the function set by sq_setcompilererrorhandler().

sq_enabledebuginfo

void **sq_enabledebuginfo**(*HSQUIRRELVM v*, *SQBool enable*);

enable/disable the debug line information generation at compile time.

parameters:

`HSQUIRRELVM v`

the target VM

`SQBool enable`

if true enables the debug info generation, if == 0 disables it.

remarks:

The function affects all threads as well.

sq_notifyallexceptions

void **sq_notifyallexceptions**(*HSQUIRRELVM v*, *SQBool enable*);

enable/disable the error callback notification of handled exceptions.

parameters:

`HSQUIRRELVM v`

the target VM

`SQBool enable`

if true enables the error callback notification of handled exceptions.

remarks:

By default the VM will invoke the error callback only if an exception is not handled (no try/catch traps are present in the call stack). If notifyallexceptions is enabled, the VM will call the error callback for any exception even if between try/catch blocks. This feature is useful for implementing debuggers.

sq_setcompilererrorhandler

void **sq_setcompilererrorhandler**(*HSQUIRRELVM v*, *SQCOMPILERERROR f*);

sets the compiler error handler function

parameters:

`HSQUIRRELVM v`

the target VM

`SQCOMPILERERROR f`

A pointer to the error handler function

remarks:

if the parameter f is NULL no function will be called when a compiler error occurs. The compiler error handler is shared between friend VMs.

Stack Operations

sq_cmp

SQInteger **sq_cmp**(*HSQUIRRELVM v*);

compares 2 object from the stack and compares them.

parameters:

`HSQUIRRELVM v`

the target VM

return:

> 0 if obj1>obj2
== 0 if obj1==obj2
< 0 if obj1<obj2

sq_gettop

SQInteger **sq_gettop**(*HSQUIRRELVM v*);

returns the index of the top of the stack

parameters:

HSQUIRRELVM v

the target VM

return:

an integer representing the index of the top of the stack

sq_pop

void **sq_pop**(*HSQUIRRELVM v*, *SQInteger nelementstopop*);

pops n elements from the stack

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger nelementstopop`

the number of elements to pop

sq_poptop

void **sq_poptop**(*HSQUIRRELVM v*);

pops 1 object from the stack

parameters:

`HSQUIRRELVM v`

the target VM

sq_push

void **sq_push**(*HSQUIRRELVM v*, *SQInteger idx*);

pushes in the stack the value at the index idx

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

the index in the stack of the value that has to be pushed

sq_remove

void **sq_remove**(*HSQUIRRELVM v*, *SQInteger idx*);

removes an element from an arbitrary position in the stack

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the element that has to be removed

sq_reservestack

SQRESULT **sq_reservestack**(*HSQUIRRELVM v*, *SQInteger nsize*);

ensure that the stack space left is at least of a specified size.If the stack is smaller it will automatically grow. if there's a memtamethod currently running the function will fail and the stack will not be resized, this situatuation has to be considered a "stack overflow".

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger nsize`

required stack size

return:

a SQRESULT

sq_settop

void **sq_settop**(*HSQUIRRELVM v*, *SQInteger v*);

resize the stack, if new top is bigger then the current top the function will push nulls.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger v`

the new top index

Object creation and handling

sq_bindenv

SQRESULT **sq_bindenv**(*HSQUIRRELVM v*, *SQInteger idx*);

pops an object from the stack(must be a table,instance or class) clones the closure at position idx in the stack and sets the popped object as environment of the cloned closure. Then pushes the new cloned closure on top of the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target closure

return:

a SQRESULT

remarks:

the cloned closure holds the environment object as weak reference

sq_createinstance

SQRESULT **sq_createinstance**(*HSQUIRRELVM v*, *SQInteger idx*);

creates an instance of the class at 'idx' position in the stack. The new class instance is pushed on top of the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target class

return:

a SQRESULT

remarks:

the function doesn't invoke the instance contructor. To create an instance and automatically invoke its contructor, sq_call must be used instead.

sq_getbool

SQRESULT **sq_getbool**(*HSQUIRRELVM v*, *SQInteger idx*, *SQBool * b*);

gets the value of the bool at the idx position in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQBool * b`

A pointer to the bool that will store the value

return:

a SQRESULT

sq_getbyhandle

SQRESULT **sq_getbyhandle**(*HSQUIRRELVM v*, *SQInteger idx*, *HSQMEMBERHANDLE* handle*);

pushes the value of a class or instance member using a member handle (see sq_getmemberhandle)

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack pointing to the class

`HSQMEMBERHANDLE* handle`

a pointer the member handle

return:

a SQRESULT

sq_getclosureinfo

SQRESULT **sq_getclosureinfo**(*HSQUIRRELVM v*, *SQInteger idx*, *SQUnsignedInteger * nparams, SQUnsignedInteger * nfreevars*);

retrieves number of parameters and number of freevariables from a squirrel closure.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target closure

`SQUnsignedInteger * nparams`

a pointer to an unsigned integer that will store the number of parameters

`SQUnsignedInteger * nfreevars`

a pointer to an unsigned integer that will store the number of free variables

return:

an SQRESULT

sq_getclosurename

SQRESULT **sq_getclosurename**(*HSQUIRRELVM v*, *SQInteger idx*);

pushes the name of the closure at poistion idx in the stack. Note that the name can be a string or null if the closure is anonymous or a native closure with no name assigned to it.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target closure

return:

an SQRESULT

sq_getfloat

SQRESULT **sq_getfloat**(*HSQUIRRELVM v*, *SQInteger idx*, *SQFloat * f*);

gets the value of the float at the idx position in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQFloat * f`

A pointer to the float that will store the value

return:

a SQRESULT

sq_gethash

SQHash **sq_gethash**(*HSQUIRRELVM v*, *SQInteger idx*);

returns the hash key of a value at the idx position in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

return:

the hash key of the value at the position idx in the stack

remarks:

the hash value function is the same used by the VM.

sq_getinstanceup

SQRESULT **sq_getinstanceup**(*HSQUIRRELVM v*, *SQInteger idx*, *SQUserPointer * up*, *SQUSerPointer typetag*);

gets the userpointer of the class instance at position idx in the stack. if the parameter 'typetag' is different than 0, the function checks that the class or a base class of the instance is tagged with the specified tag; if not the function fails. If 'typetag' is 0 the function will ignore the tag check.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQUserPointer * up`

a pointer to the userpointer that will store the result

`SQUSerPointer typetag`

the typetag that has to be checked, if this value is set to 0 the typetag is ignored.

return:

a SQRESULT

sq_getinteger

SQRESULT **sq_getinteger**(*HSQUIRRELVM v*, *SQInteger idx*, *SQInteger * i*);

gets the value of the integer at the idx position in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQInteger * i`

A pointer to the integer that will store the value

return:

a SQRESULT

sq_getmemberhandle

SQRESULT **sq_getmemberhandle**(*HSQUIRRELVM v*, *SQInteger idx*, *HSQMEMBERHANDLE* handle*);

pops a value from the stack and uses it as index to fetch the handle of a class member. The handle can be later used to set or get the member value using sq_getbyhandle(),sq_setbyhandle().

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack pointing to the class

`HSQMEMBERHANDLE* handle`

a pointer to the variable that will store the handle

return:

a SQRESULT

remarks:

This method works only with classes and instances. A handle retrieved through a class can be later used to set or get values from one of the class instances and vice-versa. Handles retrieved from base classes are still valid in derived classes and respect inheritance rules.

sq_getscratchpad

SQChar * **sq_getscratchpad**(*HSQUIRRELVM v*, *SQInteger minsize*);

returns a pointer to a memory buffer that is at least as big as minsize.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger minsize`

the requested size for the scratchpad buffer

remarks:

the buffer is valid until the next call to sq_getscratchpad

sq_getsize

SQObjectType **sq_getsize**(*HSQUIRRELVM v*, *SQInteger idx*);

returns the size of a value at the idx position in the stack, if the value is a class or a class instance the size returned is the size of the userdata buffer(see sq_setclassudsize).

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

return:

the size of the value at the position idx in the stack

remarks:

this function only works with strings,arrays,tables,classes,instances and userdata if the value is not a valid type types the function will return –1.

sq_getstring

SQRESULT **sq_getstring**(*HSQUIRRELVM v*, *SQInteger idx*, *const SQChar ** c*);

gets a pointer to the string at the idx position in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`const SQChar ** c`

a pointer to the pointer that will point to the string

return:

a SQRESULT

sq_getthread

SQRESULT **sq_getthread**(*HSQUIRRELVM v*, *SQInteger idx*, *HSQUIRRELVM* v*);

gets a a pointer to the thread the idx position in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`HSQUIRRELVM* v`

A pointer to the variable that will store the thread pointer

return:

a SQRESULT

sq_gettype

SQObjectType **sq_gettype**(*HSQUIRRELVM v*, *SQInteger idx*);

returns the type of the value at the position idx in the stack

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

return:

the type of the value at the position idx in the stack

sq_gettypetag

SQRESULT **sq_gettypetag**(*HSQUIRRELVM v*, *SQInteger idx*, *SQUserPointer * typetag*);

gets the typetag of the object(userdata or class) at position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQUserPointer * typetag`

a pointer to the variable that will store the tag

return:

a SQRESULT

remarks:

the function works also with instances. if the taget object is an instance, the typetag of it's base class is fetched.

sq_getuserdata

SQRESULT **sq_getuserdata**(*HSQUIRRELVM v*, *SQInteger idx*, *SQUserPointer * p*, *SQUserPointer * typetag*);

gets a pointer to the value of the userdata at the idx position in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQUserPointer * p`

A pointer to the userpointer that will point to the userdata's payload

`SQUserPointer * typetag`

A pointer to a SQUserPointer that will store the userdata tag(see sq_settypetag). The parameter can be NULL.

return:

a SQRESULT

sq_getuserpointer

SQRESULT **sq_getuserpointer**(*HSQUIRRELVM v*, *SQInteger idx*, *SQUserPointer * p*);

gets the value of the userpointer at the idx position in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQUserPointer * p`

A pointer to the userpointer that will store the value

return:

a SQRESULT

sq_newarray

void **sq_newarray**(*HSQUIRRELVM v*, *SQInteger size*);

creates a new array and pushes it in the stack

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger size`

the size of the array that as to be created

sq_newclass

SQRESULT **sq_newclass**(*HSQUIRRELVM v*, *SQBool hasbase*);

creates a new class object. If the parameter 'hasbase' is different than 0, the function pops a class from the stack and inherits the new created class from it.

parameters:

`HSQUIRRELVM v`

the target VM

`SQBool hasbase`

if the parameter is true the function expects a base class on top of the stack.

return:

a SQRESULT

sq_newclosure

void **sq_newclosure**(*HSQUIRRELVM v*, *HSQFUNCTION func*, *SQInteger nfreevars*);

create a new native closure, pops n values set those as free variables of the new closure, and push the new closure in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`HSQFUNCTION func`

a pointer to a native-function

`SQInteger nfreevars`

number of free variables(can be 0)

sq_newtable

void **sq_newtable**(*HSQUIRRELVM v*);

creates a new table and pushes it in the stack

parameters:

`HSQUIRRELVM v`

the target VM

sq_newtableex

void **sq_newtableex**(*HSQUIRRELVM v*, *SQInteger initialcapacity*);

creates a new table and pushes it in the stack. This function allows to specify the initial capacity of the table to prevent unnecessary rehashing when the number of slots required is known at creation-time.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger initialcapacity`

number of key/value pairs to preallocate

sq_newuserdata

SQUserPointer **sq_newuserdata**(*HSQUIRRELVM v*, *SQUnsignedInteger size*);

creates a new userdata and pushes it in the stack

parameters:

`HSQUIRRELVM v`

the target VM

`SQUnsignedInteger size`

the size of the userdata that as to be created in bytes

sq_pushbool

void **sq_pushbool**(*HSQUIRRELVM v*, *SQBool b*);

pushes a bool into the stack

parameters:

`HSQUIRRELVM v`

the target VM

`SQBool b`

the bool that has to be pushed(SQTrue or SQFalse)

sq_pushfloat

void **sq_pushfloat**(*HSQUIRRELVM v*, *SQFloat f*);

pushes a float into the stack

parameters:

`HSQUIRRELVM v`

the target VM

`SQFloat f`

the float that has to be pushed

sq_pushinteger

void **sq_pushinteger**(*HSQUIRRELVM v*, *SQInteger n*);

pushes a integer into the stack

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger n`

the integer that has to be pushed

sq_pushnull

void **sq_pushnull**(*HSQUIRRELVM v*);

pushes a null value into the stack

parameters:

HSQUIRRELVM v

the target VM

sq_pushstring

void **sq_pushstring**(*HSQUIRRELVM v*, *const SQChar * s*, *SQInteger len*);

pushes a string in the stack

parameters:

`HSQUIRRELVM v`

the target VM

`const SQChar * s`

pointer to the string that has to be pushed

`SQInteger len`

lenght of the string pointed by s

remarks:

if the parameter len is less than 0 the VM will calculate the length using strlen(s)

sq_pushuserpointer

void **sq_pushuserpointer**(*HSQUIRRELVM v*, *SQUserPointer p*);

pushes a userpointer into the stack

parameters:

`HSQUIRRELVM v`

the target VM

`SQUserPointer p`

the pointer that as to be pushed

sq_setbyhandle

SQRESULT **sq_setbyhandle**(*HSQUIRRELVM v*, *SQInteger idx*, *HSQMEMBERHANDLE* handle*);

pops a value from the stack and sets it to a class or instance member using a member handle (see sq_getmemberhandle)

parameters:

HSQUIRRELVM v

the target VM

SQInteger idx

an index in the stack pointing to the class

HSQMEMBERHANDLE handle*

a pointer the member handle

return:

a SQRESULT

sq_setclassudsize

SQRESULT **sq_setclassudsize**(*HSQUIRRELVM v*, *SQInteger idx*, *SQInteger udsize*);

Sets the user data size of a class. If a class 'user data size' is greater than 0. When an instance of the class is created additional space will is reserved at the end of the memory chunk where the instance is stored. The userpointer of the instance will also be automatically set to this memory area. This allows to minimize allocations in applications that have to carry data along with the class instance.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack pointing to the class

`SQInteger udsize`

size in bytes reserved for user data

return:

a SQRESULT

sq_setinstanceup

SQRESULT **sq_setinstanceup**(*HSQUIRRELVM v*, *SQInteger idx*, *SQUserPointer up*);

sets the userpointer of the class instance at position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQUserPointer up`

an arbitrary user pointer

return:

a SQRESULT

sq_setnativeclosurename

SQRESULT **sq_setnativeclosurename**(*HSQUIRRELVM v*, *SQInteger idx*, *const SQChar * name*);

sets the name of the native closure at the position idx in the stack. the name of a native closure is purely for debug pourposes. The name is retieved trough the function sq_stackinfos() while the closure is in the call stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target native closure

`const SQChar * name`

the name that has to be set

return:

an SQRESULT

sq_setparamscheck

SQRESULT **sq_setparamscheck**(*HSQUIRRELVM v*, *SQInteger nparamscheck*, *const SQChar * typemask*);

Sets the parameters validation scheme for the native closure at the top position in the stack. Allows to validate the number of paramters accepted by the function and optionally their types. If the function call do not comply with the parameter schema set by sq_setparamscheck, an exception is thrown.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger nparamscheck`

defines the parameters number check policy(0 disable the param checking). if nparamscheck is greater than 0 the VM ensures that the number of parameters is exactly the number specified in nparamscheck(eg. if nparamscheck == 3 the function can only be called with 3 parameters). if nparamscheck is less than 0 the VM ensures that the closure is called with at least the absolute value of the number specified in nparamcheck(eg. nparamscheck == -3 will check that the function is called with at least 3 parameters). the hidden paramater 'this' is included in this number free variables aren't. If SQ_MATCHTYPEMASKSTRING is passed instead of the number of parameters, the function will automatically extrapolate the number of parameters to check from the typemask(eg. if the typemask is ".sn" is like passing 3).

`const SQChar * typemask`

defines a mask to validate the parametes types passed to the function. if the parameter is NULL no typechecking is applyed(default).

remarks:

The typemask consists in a zero teminated string that represent the expected parameter type. The types are expressed as follows: 'o' null, 'i' integer, 'f' float, 'n' integer or float, 's' string, 't' table, 'a' array, 'u' userdata, 'c' closure and nativeclosure, 'g' generator, 'p' userpointer, 'v' thread, 'x' instance(class instance), 'y' class, 'b' bool. and '.' any type. The symbol '|' can be used as 'or' to accept multiple types on the same parameter. There isn't any limit on the number of 'or' that can be used. Spaces are ignored so can be inserted between types to increase readbility. For instance to check a function that espect a table as 'this' a string as first parameter and a number or a userpointer as second parameter, the string would be "tsn|p" (table,string,number or userpointer). If the parameters mask is contains less parameters than 'nparamscheck' the remaining parameters will not be typechecked.

eg.

```
//example
SQInteger testy(HSQUIRRELVM v)
{
	SQUserPointer p;
	const SQChar *s;
	SQInteger i;
	//no type checking, if the call comply to the mask
	//surely the functions will succeed.
	sq_getuserdata(v,1,&p,NULL);
	sq_getstring(v,2,&s);
	sq_getinteger(v,3,&i);
	//... do something
	return 0;
}

//the reg code

//....stuff
sq_newclosure(v,testy,0);
//expects exactly 3 parameters(userdata,string,number)
sq_setparamscheck(v,3,_SC("usn"));
//....stuff
```

sq_setreleasehook

void **sq_setreleasehook**(*HSQUIRRELVM v*, *SQInteger idx*, *SQRELEASEHOOK hook*);

sets the release hook of the userdata, class instance or class at position idx in the stack.

parameters:

HSQUIRRELVM v

the target VM

SQInteger idx

an index in the stack

SQRELEASEHOOK hook

a function pointer to the hook(see sample below)

remarks:

the function hook is called by the VM before the userdata memory is deleted.

eg.

```
/* tyedef SQInteger (*SQRELEASEHOOK)(SQUserPointer,SQInteger size);

SQInteger my_release_hook(SQUserPointer p,SQInteger size)
{
	/* do something here */
	return 1;
}
```

sq_settypetag

SQRESULT **sq_settypetag**(*HSQUIRRELVM v*, *SQInteger idx*, *SQUserPointer typetag*);

sets the typetag of the object(userdata or class) at position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQUserPointer typetag`

an arbitrary SQUserPointer

return:

a SQRESULT

sq_tobool

void **sq_tobool**(*HSQUIRRELVM v*, *SQInteger idx*, *SQBool * b*);

gets the value at position idx in the stack as bool.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

`SQBool * b`

A pointer to the bool that will store the value

remarks:

if the object is not a bool the function converts the value too bool according to squirrel's rules. For instance the number 1 will result in true, and the number 0 in false.

sq_tostring

void **sq_tostring**(*HSQUIRRELVM v*, *SQInteger idx*);

converts the object at position idx in the stack to string and pushes the resulting string in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

sq_typeof

SQObjectType **sq_typeof**(*HSQUIRRELVM v*, *SQInteger idx*);

pushes the type name of the value at the position idx in the stack, it also invokes the _typeof metamethod for tables and class instances that implement it; in that case the pushed object could be something other than a string (is up to the _typeof implementation).

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

an index in the stack

return:

a SQRESULT

Calls

sq_call

SQRESULT **sq_call**(*HSQUIRRELVM v*, *SQInteger params*, *SQBool retval*, *SQBool raiseerror*);

calls a closure or a native closure.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger params`

number of parameters of the function

`SQBool retval`

if true the function will push the return value in the stack

`SQBool raiseerror`

if true, if a runtime error occurs during the execution of the call, the vm will invoke the error handler.

return:

a SQRESULT

remarks:

the function pops all the parameters and leave the closure in the stack; if retval is true the return value of the closure is pushed. If the execution of the function is suspended through sq_suspendvm(), the closure and the arguments will not be automatically popped from the stack.

sq_getcallee

SQRESULT **sq_getcallee**(*HSQUIRRELVM v*);

push in the stack the currently running closure.

parameters:

`HSQUIRRELVM v`

the target VM

return:

a SQRESULT

sq_getlasterror

SQRESULT **sq_getlasterror**(*HSQUIRRELVM v*);

pushes the last error in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

return:

a SQRESULT

remarks:

the pushed error descriptor can be any valid squirrel type.

sq_getlocal

const SQChar * **sq_getlocal**(*HSQUIRRELVM v*, *SQUnsignedInteger level*, *SQUnsignedInteger nseq*);

Returns the name of a local variable given stackframe and sequence in the stack and pushes is current value. Free variables are treated as local variables, by sq_getlocal(), and will be returned as they would be at the base of the stack, just before the real local variables.

parameters:

`HSQUIRRELVM v`

the target VM

`SQUnsignedInteger level`

the function index in the calls stack, 0 is the current function

`SQUnsignedInteger nseq`

the index of the local variable in the stack frame (0 is ‘this’)

return:

the name of the local variable if a variable exists at the given level/seq otherwise NULL.

sq_reseterror

void **sq_reseterror**(*HSQUIRRELVM v*);

reset the last error in the virtual machine to null

parameters:

`HSQUIRRELVM v`

the target VM

sq_resume

SQRESULT **sq_resume**(*HSQUIRRELVM v*, *SQBool retval*, *SQBool raiseerror*);

resumes the generator at the top position of the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQBool retval`

if true the function will push the return value in the stack

`SQBool raiseerror`

if true, if a runtime error occurs during the execution of the call, the vm will invoke the error handler.

return:

a SQRESULT

remarks:

if retval != 0 the return value of the generator is pushed.

sq_throwerror

SQRESULT **sq_throwerror**(*HSQUIRRELVM v*, *const SQChar * err*);

sets the last error in the virtual machine and returns the value that has to be returned by a native closure in order to trigger an exception in the virtual machine.

parameters:

`HSQUIRRELVM v`

the target VM

`const SQChar * err`

the description of the error that has to be thrown

return:

the value that has to be returned by a native closure in order to throw an exception in the virtual machine.

sq_throwobject

SQRESULT **sq_throwobject**(*HSQUIRRELVM v*);

pops a value from the stack sets it as the last error in the virtual machine. Returns the value that has to be returned by a native closure in order to trigger an exception in the virtual machine (aka SQ_ERROR).

parameters:

`HSQUIRRELVM v`

the target VM

return:

the value that has to be returned by a native closure in order to throw an exception in the virtual machine.

Objects manipulation

sq_arrayappend

SQRESULT **sq_arrayappend**(*HSQUIRRELVM v*, *SQInteger idx*);

pops a value from the stack and pushes it in the back of the array at the position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target array in the stack

return:

a SQRESULT

remarks:

Only works on arrays.

sq_arrayinsert

SQRESULT **sq_arrayinsert**(*HSQUIRRELVM v*, *SQInteger idx*, *SQInteger destpos*);

pops a value from the stack and inserts it in an array at the specified position

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target array in the stack

`SQInteger destpos`

the postion in the array where the item has to be inserted

return:

a SQRESULT

remarks:

Only works on arrays.

sq_arraypop

SQRESULT **sq_arraypop**(*HSQUIRRELVM v*, *SQInteger idx*);

pops a value from the back of the array at the position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target array in the stack

return:

a SQRESULT

remarks:

Only works on arrays.

sq_arrayremove

SQRESULT **sq_arrayremove**(*HSQUIRRELVM v*, *SQInteger idx*, *SQInteger itemidx*);

removes an item from an array

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target array in the stack

`SQInteger itemidx`

the index of the item in the array that has to be removed

return:

a SQRESULT

remarks:

Only works on arrays.

sq_arrayresize

SQRESULT **sq_arrayresize**(*HSQUIRRELVM v*, *SQInteger idx*, *SQInteger newsize*);

resizes the array at the position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target array in the stack

`SQInteger newsize`

requested size of the array

return:

a SQRESULT

remarks:

Only works on arrays.if newsize if greater than the current size the new array slots will be filled with nulls.

sq_arrayreverse

SQRESULT **sq_arrayreverse**(*HSQUIRRELVM v*, *SQInteger idx*);

reverse an array in place.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target array in the stack

return:

a SQRESULT

remarks:

Only works on arrays.

sq_clear

SQRESULT **sq_clear**(*HSQUIRRELVM v*, *SQInteger idx*);

clears all the element of the table/array at position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

return:

a SQRESULT

remarks:

Only works on tables and arrays.

sq_clone

SQRESULT **sq_clone**(*HSQUIRRELVM v, SQInteger idx*);

Clones the table, array or class instance at the position idx, clones it and pushes the new object in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

return:

a SQRESULT

sq_createslot

SQRESULT **sq_createslot**(*HSQUIRRELVM v*, *SQInteger idx*);

pops a key and a value from the stack and performs a set operation on the table or class that is at position idx in the stack, if the slot does not exits it will be created.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target table in the stack

return:

a SQRESULT

remarks:

invoke the _newslot metamethod in the table delegate. it only works on tables. [this function is deperecated since version 2.0.5 use sq_newslot() instead]

sq_deleteslot

SQRESULT **sq_deleteslot**(*HSQUIRRELVM v*, *SQInteger idx*, *SQBool pushval*);

pops a key from the stack and delete the slot indexed by it from the table at position idx in the stack, if the slot does not exits nothing happens.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target table in the stack

`SQBool pushval`

if this param is true the function will push the value of the deleted slot.

return:

a SQRESULT

remarks:

invoke the _delslot metamethod in the table delegate. it only works on tables.

sq_get

SQRESULT **sq_get**(*HSQUIRRELVM v*, *SQInteger idx*);

pops a key from the stack and performs a get operation on the object at the position idx in the stack, and pushes the result in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

return:

a SQRESULT

remarks:

this call will invokes the delegation system like a normal dereference it only works on tables, arrays and userdata. if the function fails nothing will be pushed in the stack.

sq_getattributes

SQRESULT **sq_getattributes**(*HSQUIRRELVM v*, *SQInteger idx*);

Gets the attribute of a class mameber. The function pops a key from the stack and pushes the attribute of the class member indexed by they key from class at position idx in the stack. If key is null the function gets the class level attribute.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target class in the stack

return:

a SQRESULT

sq_getclass

SQRESULT **sq_getclass**(*HSQUIRRELVM v*, *SQInteger idx*);

pushes the class of the 'class instance' at stored position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target class instance in the stack

return:

a SQRESULT

sq_getdelegate

SQRESULT **sq_getdelegate**(*HSQUIRRELVM v*, *SQInteger idx*);

pushes the current delegate of the object at the position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

return:

a SQRESULT

sq_getfreevariable

const SQChar * **sq_getfreevariable**(*HSQUIRRELVM v*, *SQInteger idx*, *SQInteger nval*);

gets the value of the free variable of the closure at the position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack(closure)

`SQInteger nval`

0 based index of the free variable(relative to the closure).

return:

the name of the free variable for pure squirrel closures. NULL in case of error or if the index of the variable is out of range. In case the target closure is a native closure, the return name is always "@NATIVE".

remarks:

The function works for both squirrel closure and native closure.

sq_getweakrefval

SQRESULT **sq_getweakrefval**(*HSQUIRRELVM v*, *SQInteger idx*);

pushes the object pointed by the weak reference at position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target weak reference

return:

a SQRESULT

remarks:

if the function fails, nothing is pushed in the stack.

sq_instanceof

SQBool **sq_instanceof**(*HSQUIRRELVM v*);

Determintes if an object is an instance of a certain class. Expects an istance and a class in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

return:

SQTrue if the instance at position -2 in the stack is an instance of the class object at position -1 in the stack.

remarks:

The function doesn't pop any object from the stack.

sq_newmember

SQRESULT **sq_newmember**(*HSQUIRRELVM v*, *SQInteger idx*, *SQBool bstatic*);

pops a key, a value and an object(that will be set as attribute of the member) from the stack and performs a new slot operation on the class that is at position idx in the stack, if the slot does not exits it will be created.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target table in the stack

`SQBool bstatic`

if SQTrue creates a static member.

return:

a SQRESULT

remarks:

Invokes the _newmember metamethod in the class. it only works on classes.

sq_newslot

SQRESULT **sq_newslot**(*HSQUIRRELVM v*, *SQInteger idx*, *SQBool bstatic*);

pops a key and a value from the stack and performs a set operation on the table or class that is at position idx in the stack, if the slot does not exits it will be created.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target table in the stack

`SQBool bstatic`

if SQTrue creates a static member. This parameter is only used if the target object is a class.

return:

a SQRESULT

remarks:

Invokes the _newslot metamethod in the table delegate. it only works on tables and classes.

sq_next

SQRESULT **sq_next**(*HSQUIRRELVM v*, *SQInteger idx*);

Pushes in the stack the next key and value of an array, table or class slot. To start the iteration this function expects a null value on top of the stack; at every call the function will substitute the null value with an iterator and push key and value of the container slot. Every iteration the application has to pop the previous key and value but leave the iterator(that is used as reference point for the next iteration). The function will fail when all slots have been iterated(see Tables and arrays manipulation).

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

return:

a SQRESULT

sq_rawdeleteslot

SQRESULT **sq_rawdeleteslot**(*HSQUIRRELVM v*, *SQInteger idx*, *SQBool pushval*);

Deletes a slot from a table without employing the _delslot metamethod. pops a key from the stack and delete the slot indexed by it from the table at position idx in the stack, if the slot does not exits nothing happens.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target table in the stack

`SQBool pushval`

if this param is true the function will push the value of the deleted slot.

return:

a SQRESULT

sq_rawget

SQRESULT **sq_rawget**(*HSQUIRRELVM v*, *SQInteger idx*);

pops a key from the stack and performs a get operation on the object at position idx in the stack, without employing delegation or metamethods.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

return:

a SQRESULT

remarks:

Only works on tables and arrays.

sq_rawnewmember

SQRESULT **sq_rawnewmember**(*HSQUIRRELVM v*, *SQInteger idx*, *SQBool bstatic*);

pops a key, a value and an object(that will be set as attribute of the member) from the stack and performs a new slot operation on the class that is at position idx in the stack, if the slot does not exits it will be created.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target table in the stack

`SQBool bstatic`

if SQTrue creates a static member.

return:

a SQRESULT

remarks:

it only works on classes.

sq_rawset

SQRESULT **sq_rawset**(*HSQUIRRELVM v*, *SQInteger idx*);

pops a key and a value from the stack and performs a set operation on the object at position idx in the stack, without employing delegation or metamethods.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

return:

a SQRESULT

remarks:

it only works on tables and arrays. if the function fails nothing will be pushed in the stack.

sq_set

SQRESULT **sq_set**(*HSQUIRRELVM v*, *SQInteger idx*);

pops a key and a value from the stack and performs a set operation on the object at position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

return:

a SQRESULT

remarks:

this call will invoke the delegation system like a normal assignment, it only works on tables, arrays and userdata.

sq_setattributes

SQRESULT **sq_setattributes**(*HSQUIRRELVM v*, *SQInteger idx*);

Sets the attribute of a class mameber. The function pops a key and a value from the stack and sets the attribute (indexed by they key) on the class at position idx in the stack. If key is null the function sets the class level attribute. If the function succeed, the old attribute value is pushed in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target class in the stack.

return:

a SQRESULT

sq_setdelegate

SQRESULT **sq_setdelegate**(*HSQUIRRELVM v*, *SQInteger idx*);

pops a table from the stack and sets it as delegate of the object at the position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

return:

a SQRESULT

remarks:

to remove the delgate from an object is necessary to use null as delegate instead of a table.

sq_setfreevariable

SQRESULT **sq_setfreevariable**(*HSQUIRRELVM v*, *SQInteger idx*, *SQInteger nval*);

pops a value from the stack and sets it as free variable of the closure at the position idx in the stack.

parameters:

HSQUIRRELVM v

the target VM

SQInteger idx

index of the target object in the stack

SQInteger nval

0 based index of the free variable(relative to the closure).

return:

a SQRESULT

sq_weakref

void **sq_weakref**(*HSQUIRRELVM v*, *SQInteger idx*);

pushes a weak reference to the object at position idx in the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index to the target object in the stack

return:

a SQRESULT

remarks:

if the object at idx position is an integer,float,bool or null the object itself is pushed instead of a weak ref.

Bytecode serialization

sq_readclosure

SQRESULT **sq_readclosure**(*HSQUIRRELVM v*, *SQREADFUNC readf*, *SQUserPointer up*);

serialize (read) a closure and pushes it on top of the stack, the source is user defined through a read callback.

parameters:

`HSQUIRRELVM v`

the target VM

`SQREADFUNC readf`

pointer to a read function that will be invoked by the vm during the serialization.

`SQUserPointer up`

pointer that will be passed to each call to the read function

return:

a SQRESULT

sq_writeclosure

SQRESULT **sq_writeclosure**(*HSQUIRRELVM v*, *SQWRITEFUNC writef*, *SQUserPointer up*);

serialize(write) the closure on top of the stack, the desination is user defined through a write callback.

parameters:

`HSQUIRRELVM v`

the target VM

`SQWRITEFUNC writef`

pointer to a write function that will be invoked by the vm during the serialization.

`SQUserPointer up`

pointer that will be passed to each call to the write function

return:

a SQRESULT

remarks:

closures with free variables cannot be serialized

Raw object handling

sq_addref

void **sq_addref**(*HSQUIRRELVM v*, *HSQOBJECT * po*);

adds a reference to an object handler.

parameters:

`HSQUIRRELVM v`

the target VM

`HSQOBJECT * po`

pointer to an object handler

sq_getobjtypetag

SQRESULT **sq_getobjtypetag**(*HSQOBJECT * o*, *SQUserPointer * typetag*);

gets the typetag of a raw object reference(userdata or class).

parameters:

`HSQOBJECT * o`

pointer to an object handler

`SQUserPointer * typetag`

a pointer to the variable that will store the tag

return:

a SQRESULT

remarks:

the function works also with instances. if the taget object is an instance, the typetag of it's base class is fetched.

sq_getrefcount

SQUnsignedInteger **sq_getrefcount**(*HSQUIRRELVM v*, *HSQOBJECT *po*);

returns the number of references of a given object.

parameters:

`HSQUIRRELVM v`

the target VM

`HSQOBJECT *po`

object handler

sq_getstackobj

SQRESULT **sq_getstackobj**(*HSQUIRRELVM v*, *SQInteger idx*, *HSQOBJECT * po*);

gets an object from the stack and stores it in a object handler.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger idx`

index of the target object in the stack

`HSQOBJECT * po`

pointer to an object handler

return:

a SQRESULT

sq_objtobool

SQBool **sq_objtobool**(*HSQOBJECT * po*);

return the bool value of a raw object reference.

parameters:

`HSQOBJECT * po`

pointer to an object handler

remarks:

If the object is not a bool will always return false.

sq_objtofloat

SQFloat **sq_objtofloat**(*HSQOBJECT * po*);

return the float value of a raw object reference.

parameters:

`HSQOBJECT * po`

pointer to an object handler

remarks:

If the object is an integer will convert it to float. If the object is not a number will always return 0.

sq_objtointeger

SQInteger **sq_objtointeger**(*HSQOBJECT * po*);

return the integer value of a raw object reference.

parameters:

*HSQOBJECT * po*

pointer to an object handler

remarks:

If the object is a float will convert it to integer. If the object is not a number will always return 0.

sq_objtostring

const SQChar * **sq_objtostring**(*HSQOBJECT * po*);

return the string value of a raw object reference.

parameters:

`HSQOBJECT * po`

pointer to an object handler

remarks:

If the object doesn't reference a string it returns NULL.

sq_objtouserpointer

SQUserPointer **sq_objtouserpointer**(*HSQOBJECT * po*);

return the userpointer value of a raw object reference.

parameters:

`HSQOBJECT * po`

pointer to an object handler

remarks:

If the object doesn't reference a userpointer it returns NULL.

sq_pushobject

void **sq_pushobject**(*HSQUIRRELVM v*, *HSQOBJECT obj*);

push an object referenced by an object handler into the stack.

parameters:

`HSQUIRRELVM v`

the target VM

`HSQOBJECT obj`

object handler

sq_release

SQBool **sq_release**(*HSQUIRRELVM v*, *HSQOBJECT * po*);

remove a reference from an object handler.

parameters:

`HSQUIRRELVM v`

the target VM

`HSQOBJECT * po`

pointer to an object handler

return:

SQTrue if the object handler released has lost all is references(the ones added with sq_addref). SQFalse otherwise.

remarks:

the function will reset the object handler to null when it losts all references.

sq_resetobject

void **sq_resetobject**(*HSQOBJECT * po*);

resets(initialize) an object handler.

parameters:

*HSQOBJECT * po*

pointer to an object handler

remarks:

Every object handler has to be initialized with this function.

Garbage Collector

sq_collectgarbage

SQInteger **sq_collectgarbage**(*HSQUIRRELVM v*);

runs the garbage collector and returns the number of reference cycles found(and deleted)

parameters:

`HSQUIRRELVM v`

the target VM

remarks:

this api only works with gabage collector builds (NO_GARBAGE_COLLECTOR is not defined)

sq_resurrectunreachable

SQRESULT **sq_resurrectunreachable**(*HSQUIRRELVM v*);

runs the garbage collector and pushes an array in the stack containing all unreachable object found. If no unreachable object is found, null is pushed instead. This function is meant to help debugging reference cycles.

parameters:

`HSQUIRRELVM v`

the target VM

remarks:

this api only works with gabage collector builds (NO_GARBAGE_COLLECTOR is not defined)

Debug interface

sq_getfunctioninfo

SQRESULT **sq_getfunctioninfo**(*HSQUIRRELVM v*, *SQInteger level*, *SQFunctionInfo * fi*);

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger level`

calls stack level

`SQFunctionInfo * fi`

pointer to the SQFunctionInfo structure that will store the closure informations

return:

a SQRESULT.

remarks:

the member 'funcid' of the returned SQFunctionInfo structure is a unique identifier of the function; this can be useful to identify a specific piece of squirrel code in an application like for instance a profiler. this method will fail if the closure in the stack is a native C closure.

eg.

```
typedef struct tagSQFunctionInfo {
	SQUserPointer funcid; //unique idetifier for a function (all it's closures w
	const SQChar *name; //function name
	const SQChar *source; //function source file name
}SQFunctionInfo;
```

sq_setdebughook

void **sq_setdebughook**(*HSQUIRRELVM v*);

pops a closure from the stack an sets it as debug hook. When a debug hook is set it overrides any previously set native or non native hooks. if the hook is null the debug hook will be disabled.

parameters:

`HSQUIRRELVM v`

the target VM

remarks:

In order to receive a 'per line' callback, is necessary to compile the scripts with the line informations. Without line informations activated, only the 'call/return' callbacks will be invoked.

sq_setnativedebughook

void **sq_setnativedebughook**(*HSQUIRRELVM v*, *SQDEBUGHOOK hook*);

sets the native debug hook. When a native hook is set it overrides any previously set native or non native hooks. if the hook is NULL the debug hook will be disabled.

parameters:

`HSQUIRRELVM v`

the target VM

`SQDEBUGHOOK hook`

the native hook function

remarks:

In order to receive a 'per line' callback, is necessary to compile the scripts with the line informations. Without line informations activated, only the 'call/return' callbacks will be invoked.

sq_stackinfos

SQRESULT **sq_stackinfos**(*HSQUIRRELVM v*, *SQInteger level*, *SQStackInfos * si*);

retrieve the calls stack informations of a ceratain level in the calls stack.

parameters:

`HSQUIRRELVM v`

the target VM

`SQInteger level`

calls stack level

`SQStackInfos * si`

pointer to the SQStackInfos structure that will store the stack informations

return:

a SQRESULT.