Squirrel 1.0 Reference Manual

Alberto Demichelis

Squirrel 1.0 Reference Manual

Alberto Demichelis
Extensive review: Wouter Van Oortmersern

Table of Contents

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:

```
break    case     catch    clone    continue default  delegate delete
else     for      if       in       local    null     resume   return
switch   this     throw    try      typeof   while    yield    parent
```

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 stings literals.

`34`	Integer number
`0xFF00A120`	Integer number

`'a'`	Integer number
`1.52`	Floating point number
`"I'm a string"`	String
`@"I'm a verbatim string"`	String
`@" I'm a multiline verbatim string "`	String

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

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, thread and userdata.

Integer

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

```
local a = 123 //decimal
local b = 0x0012 //hexadecimal
local c = '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).

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
```

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. In squirrel it is also used to represent a false Boolean value.

```
local a=null
```

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).

Generator

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

ors).

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.

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'.

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

if/else

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

Conditionally execute a statement depending on the result of an expression. exp is considered 'false' when is value is null and true for any other value.

Warning

In Squirrel also the number 0 is considered 'true'.

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

while

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

Executes a statement until the condition is false(null).

```
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 null.

```
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 null.

```
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, 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.

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.

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 2 compact operators (+= and -=) and increment and decrement operators(++ and --);

```
a+=2;
//is the same as write
a=a+2;
x++
//is the same as write
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.

Relational

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

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

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

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 a value different than null 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();
```

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.

delegate

exp:= 'delegate' parentexp : exp

Sets the parent of a table. The result of parentexp is set as parent of the result of exp, the result of the expression is exp (see Delegation).

clone

exp:= 'clone' exp

Clone performs shallow copy of a table, (copies all slots in the new table 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 table is ready the "_cloned" meta method is called (see Metamethods).

Array contructor

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[1]) //prints "string"
print(typeof a[0]) //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)
{
    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;
        }
}
```

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).

Free variables

Free variables are variables referenced by a function that are not visible in the function scope. In the following example the function foo() declares x, y and testy as free variables.

```
local x=10,y=20
testy <- “I’m testy”

function foo(a,b):(x,y,testy)
```

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

The value of a free variable is frozen and bound to the function when the function is created; the value is passed to the function as implicit parameter every time is called.

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);
```

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
```

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 has 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");
```

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=delegate Entity : {
    _name="I'm the new entity"
}

newentity.DoStuff(); //prints "I'm the new entity"
```

The parent of a table can be retreived through keyword `parent`. `parent` is a pseudo slot that and works only on tables. The `parent` slot cannot be set, the `delegete` statement has to be used instead.

```
local thedelegate = newentity.parent;
```

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); Is possible to change many aspect of a table behavior by just defining a metamethod for this parent. For instance 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=delegate comparable : { name="Alberto" };
local b=delegate comparable : { name="Wouter" };

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

_set

invoked when the index idx is not present in the table or in its delegate chain

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

_get

invoked when the index idx is not present in the table or in its delegate chain

```
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 and userdata

```
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 tables or userdatas are called

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

_cloned

invoked when a table is cloned(in the cloned table)

```
function cloned(original)
```

_nexti

invoked when a userdata 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.

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

`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.

`assert(exp)`
throws an exception if exp is null

`print(x)`
prints x in the standard 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()`
calls the garbage collector and returns the number of reference cycles found(and deleted)

`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

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.

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.

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 + start. 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.

Table

`len()`
returns the number of slots contained in a table

`rawget(key)`
tries to get a value from the slot ‘key’ without employ delegation

`rawset(key,val)`
sets the slot ‘key’ with the value ‘val’ without employing delegation. If the slot do 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.

Array

`len()`
returns the length of the array

`append(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;
}
```

`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 + start. If end is omitted end is equal to the array length.

Function

`call(_this,args…)`
calls the function with the specified environment object(’this’) and parameters

`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.

Generator

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

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")

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 '_UNICODE' is defined the VM, compiler and API will use 16-bits characters.

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 cre-

ate 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,int idx);
```

To pop an arbitrary number of elements

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

To remove an element from the stack

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

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

```
int sq_gettop(HSQUIRRELVM v);
```

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

```
void sq_settop(HSQUIRRELVM v,int 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,int len);
void sq_pushfloat(HSQUIRRELVM v,SQFloat f);
void sq_pushinteger(HSQUIRRELVM v,SQInteger n);
void sq_pushuserpointer(HSQUIRRELVM v,SQUserPointer p);
```

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,int idx);
```

the result can be one of the following values:

```
OT_NULL,OT_INTEGER,OT_FLOAT,OT_STRING,OT_TABLE,OT_ARRAY,OT_USERDATA,
OT_CLOSURE,OT_NATIVECLOSURE,OT_GENERATOR,OT_USERPOINTER
```

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

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

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

```
int 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,SQUserPointer p,
        const SQChar *sourcename,int 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,file,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 is must be declared as follow

```
typedef void (*SQCOMPILERERROR)(const SQChar * /*desc*/,const SQChar *
/*source*/,int /*line*/,int /*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,4,0);
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 int (*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.

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

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

```
int register_global_func(HSQUIRRELVM v,SQFUNCTION f,const char *fname)
{
    sq_pushroottable(v);
    sq_pushstring(v,fname,-1);
    sq_newclosure(v,f,0,0); //create a new function
    sq_createslot(v,-3);
    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_createslot(HSQUIRRELVM v,int idx);
```

To set or get the table delegate

```
SQRESULT sq_setdelegate(HSQUIRRELVM v,int idx);
SQRESULT sq_getdelegate(HSQUIRRELVM v,int 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,int size);
```

To append a value to the back of the array

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

To remove a value from the back of the array

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

To resize the array

```
SQRESULT sq_arrayresize(HSQUIRRELVM v,int idx,int newsize);
```

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

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

To set a value in an array or table

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

To get a value from an array or table

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

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

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

To iterate a table or an array

```
SQRESULT sq_next(HSQUIRRELVM v,int 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,unsigned int 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 int (*SQUSERDATARELEASE)(SQUserPointer,int size);

void sq_setreleasehook(HSQUIRRELVM v,int idx,SQUSERDATARELEASE hook);
```

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);
```

Keeping object references from C

Squirrel allows to keep objects references from C; the function sq_getstackobject() gets a handle to a squirrel object(any type), this object can be pushed later in the stack.

```
HSQOBJECT obj;

sq_resetobject(v,&obj) //initialize the handle
sq_geststackobject(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 function sq_setdebughook 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);
```

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()

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

While the local variables info through sq_getlocal()

```
int sq_getlocal(HSQUIRRELVM v,unsigned int level,unsigned int nseq);
```

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, int debuginfo);
```

Chapter 4. API Reference

Virtual Machine

sq_open

```
HSQUIRRELVM sq_open(int initialstacksize);
```

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

parameters:

int 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_close

```
void sq_close(HSQUIRRELVM v);
```

release a squirrel VM and all related friend VMs

parameters:

HSQUIRRELVM v the target VM

sq_newthread

```
HSQUIRRELVM sq_newthread(HSQUIRRELVM friendvm, int 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

int 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_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.

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

sq_wakeupvm

```
HRESULT sq_wakeupvm(HSQUIRRELVM v, int resumedret, int retval);
```

Wake up the execution a previously suspended virtual machine.

parameters:

HSQUIRRELVM v	the target VM
int resumedret	if > 0 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.
int retval	if > 0 the function will push the return value of the function that suspend the excution or the main function one.

return: an HRESULT.

sq_move

```
void sq_move(HSQUIRRELVM dest, HSQUIRRELVM src, int 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
int idx	the index in the source stack of the value that has to be moved

sq_getvmstate

```
int 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_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_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_pushroottable

```
SQRESULT sq_pushroottable(HSQUIRRELVM v);
```

pushes the current root table in the stack

parameters:

HSQUIRRELVM v the target VM

sq_pushregistrytable

```
SQRESULT sq_pushregistrytable(HSQUIRRELVM v);
```

pushes the registry table in the stack

parameters:

HSQUIRRELVM v the target VM

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_setprintfunc

```
void sq_setprintfunc(HSQUIRRELVM v, SQPRINTFUNCTION printfunc);
```

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.

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

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.

Compiler

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.

sq_compile

```
SQRESULT sq_compile(HSQUIRRELVM v, HSQLEXREADFUNC read, SQUserPointer p, const SQChar * sourcename, int 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)
int raiseerror	if this value is different than 0 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, int size,
const SQChar * sourcename, int 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.
int 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)
int raiseerror	if this value is different than 0 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, int debuginfo);
```

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

parameters:

HSQUIRRELVM v	the target VM
int debuginfo	if != 0 enables the debug info generation, if == 0 disables it.

remarks: The function affects all threads as well.

Stack Operations

sq_push

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

pushes in the stack the value at the index idx

parameters:

HSQUIRRELVM v	the target VM
int idx	the index in the stack of the value that has to be pushed

sq_pop

```
void sq_pop(HSQUIRRELVM v, int nelementstopop);
```

pops n elements from the stack

parameters:

HSQUIRRELVM v	the target VM
int nelementstopop	the number of elements to pop

sq_reservestack

```
void sq_reservestack(HSQUIRRELVM v, int nsize);
```

ensure that the stack space left is at least of a specified size.If the stack is smaller it will automatically

grow.

parameters:

HSQUIRRELVM v	the target VM
int nsize	required stack size

sq_remove

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

removes an element from an arbitrary position in the stack

parameters:

HSQUIRRELVM v	the target VM
int idx	index of the element that has to be removed

sq_gettop

```
int 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_settop

```
void sq_settop(HSQUIRRELVM v, int v);
```

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

parameters:

HSQUIRRELVM v	the target VM
int v	the new top index

sq_cmp

```
int sq_cmp(HSQUIRRELVM v);
```

pops 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

Object creation and handling

sq_newuserdata

```
SQUserPointer sq_newuserdata(HSQUIRRELVM v, unsigned int size);
```

creates a new userdata and pushes it in the stack

parameters:

HSQUIRRELVM v	the target VM
unsigned int size	the size of the userdata that as to be created in bytes

sq_newtable

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

creates a new table and pushes it in the stack

parameters:

HSQUIRRELVM v	the target VM

sq_newarray

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

creates a new array and pushes it in the stack

parameters:

HSQUIRRELVM v	the target VM
int size	the size of the array that as to be created

sq_newclosure

```
void sq_newclosure(HSQUIRRELVM v, HSQFUNCTION func, int 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
int nfreevars	number of free variables(can be 0)

sq_setparamscheck

```
SQRESULT sq_setparamscheck(HSQUIRRELVM v, int 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
int 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.
*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: '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 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. 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
int 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_setnativeclosurename

```
SQRESULT sq_setnativeclosurename(HSQUIRRELVM v, int 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
int idx	index of the target native closure
*const SQChar * name*	the name that has to be set

return: an SQRESULT

sq_getclosureinfo

```
SQRESULT sq_getclosureinfo(HSQUIRRELVM v, int idx, unsigned int * nparams, unsigned int * nfreevars);
```

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

parameters:

HSQUIRRELVM v	the target VM
int idx	index of the target closure
*unsigned int * nparams*	a pointer to an unsigned integer that will store the number of parameters
*unsigned int * nfreevars*	a pointer to an unsigned integer that will store the number of free variables

return: an SQRESULT

sq_pushstring

```
void sq_pushstring(HSQUIRRELVM v, const SQChar * s, int 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
int 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_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_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_pushnull

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

pushes a null value into the stack

parameters:

HSQUIRRELVM v	the target VM

sq_gettype

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

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

parameters:

HSQUIRRELVM v	the target VM
int idx	an index in the stack

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

sq_getsize

```
SQObjectType sq_getsize(HSQUIRRELVM v, int idx);
```

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

parameters:

HSQUIRRELVM v	the target VM
int 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 and userdata if the value is not one of those types the function will return –1

sq_getstring

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

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

parameters:

HSQUIRRELVM v	the target VM
int idx	an index in the stack
*const SQChar ** c*	a pointer to the pointer that will point to the string

return: a SQRESULT

sq_getinteger

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

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

parameters:

HSQUIRRELVM v	the target VM
int idx	an index in the stack
*SQInteger * i*	A pointer to the integer that will store the value

return: a SQRESULT

sq_getfloat

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

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

parameters:

HSQUIRRELVM v	the target VM
int idx	an index in the stack
*SQFloat * f*	A pointer to the float that will store the value

return: a SQRESULT

sq_getthread

```
SQRESULT sq_getthread(HSQUIRRELVM v, int idx, HSQUIRRELVM* v);
```

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

parameters:

HSQUIRRELVM v	the target VM
int idx	an index in the stack
HSQUIRRELVM v*	A pointer to the variable that will store the thread pointer

return: a SQRESULT

sq_getuserpointer

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

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

parameters:

HSQUIRRELVM v	the target VM
int idx	an index in the stack
*SQUserPointer * p*	A pointer to the userpointer that will store the value

return: a SQRESULT

sq_getuserdata

```
SQRESULT sq_getuserdata(HSQUIRRELVM v, int idx, SQUserPointer * p, unsigned int * typetag);
```

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

parameters:

HSQUIRRELVM v	the target VM
int idx	an index in the stack
*SQUserPointer * p*	A pointer to the userpointer that will point to the userdata's payload
*unsigned int * typetag*	A pointer to an usigned int that will store the userdata tag(see sq_settypetag). The parameter can be NULL.

return: a SQRESULT

sq_settypetag

```
SQRESULT sq_settypetag(HSQUIRRELVM v, int idx, unsigned int typetag);
```

sets the typetag of the userdata at position idx in the stack.

parameters:

HSQUIRRELVM v	the target VM
int idx	an index in the stack
unsigned int typetag	an arbitrary unsigned int

return: a SQRESULT

sq_setreleasehook

```
void sq_setreleasehook(HSQUIRRELVM v, int idx, SQUSERDATARELEASE hook);
```

sets the release hook of the userdata at position idx in the stack.

parameters:

HSQUIRRELVM v	the target VM
int idx	an index in the stack
SQUSERDATARELEASE 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 int (*SQUSERDATARELEASE)(SQUserPointer,int size); */

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

sq_getscratchpad

```
SQChar * sq_getscratchpad(HSQUIRRELVM v, int minsize);
```

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

parameters:

HSQUIRRELVM v	the target VM
int minsize	the requested size for the scratchpad buffer

remarks: the buffer is valid until the next call to sq_getscratchpad

Calls

sq_call

```
SQRESULT sq_call(HSQUIRRELVM v, int params, int retval);
```

calls a closure or a native closure.

parameters:

HSQUIRRELVM v	the target VM
int params	number of parameters of the function
int retval	if >0 the function will push the return value in the stack

return: a SQRESULT

remarks: the function pops all the parameters and leave the closure in the stack; if retval != 0 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_resume

```
SQRESULT sq_resume(HSQUIRRELVM v, int retval);
```

resumes the generator at the top position of the stack.

parameters:

HSQUIRRELVM v	the target VM
int retval	if >0 the function will push the return value in the stack

return: a SQRESULT

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

sq_getlocal

```
const SQChar * sq_getlocal(HSQUIRRELVM v, unsigned int level, unsigned int nseq);
```

returns the name of a local variable given stackframe and sequence in the stack and pushes is current value.

parameters:

HSQUIRRELVM v	the target VM
unsigned int level	the function index in the calls stack, 0 is the current function
unsigned int 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_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_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.

Objects manipulation

`sq_createslot`

```
SQRESULT sq_createslot(HSQUIRRELVM v, int idx);
```

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

parameters:

HSQUIRRELVM v	the target VM
int 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.

`sq_deleteslot`

```
SQRESULT sq_deleteslot(HSQUIRRELVM v, int idx, int 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
int idx	index of the target table in the stack
int pushval	if this param is different than 0 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_rawdeleteslot

```
SQRESULT sq_rawdeleteslot(HSQUIRRELVM v, int idx, int 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
int idx	index of the target table in the stack
int pushval	if this param is different than 0 the function will push the value of the deleted slot.

return: a SQRESULT

sq_set

```
SQRESULT sq_set(HSQUIRRELVM v, int 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
int 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_get

```
SQRESULT sq_get(HSQUIRRELVM v, int 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
int 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_rawset

```
SQRESULT sq_rawset(HSQUIRRELVM v, int 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
int 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_rawget

```
SQRESULT sq_rawget(HSQUIRRELVM v, int 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

int idx index of the target object in the stack

return: a SQRESULT

remarks: Only works on tables and arrays.

sq_arrayappend

```
SQRESULT sq_arrayappend(HSQUIRRELVM v, int 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

int idx index of the target array in the stack

return: a SQRESULT

remarks: Only works on arrays.

sq_arraypop

```
SQRESULT sq_arraypop(HSQUIRRELVM v, int idx);
```

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

parameters:

HSQUIRRELVM v the target VM

int idx index of the target array in the stack

return: a SQRESULT

remarks: Only works on arrays.

sq_arrayreverse

```
SQRESULT sq_arrayreverse(HSQUIRRELVM v, int idx);
```

reverse an array in place.

parameters:

HSQUIRRELVM v the target VM

int idx	index of the target array in the stack

return: a SQRESULT

remarks: Only works on arrays.

sq_arrayresize

```
SQRESULT sq_arrayresize(HSQUIRRELVM v, int idx, int newsize);
```

resizes the array at the position idx in the stack.

parameters:

HSQUIRRELVM v	the target VM
int idx	index of the target array in the stack
int 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_setdelegate

```
SQRESULT sq_setdelegate(HSQUIRRELVM v, int 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
int 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_getdelegate

```
SQRESULT sq_getdelegate(HSQUIRRELVM v, int idx);
```

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

parameters:

HSQUIRRELVM v	the target VM
int idx	index of the target object in the stack

return: a SQRESULT

sq_clone

```
SQRESULT sq_clone(HSQUIRRELVM v, int idx);
```

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

parameters:

HSQUIRRELVM v	the target VM
int idx	index of the target object in the stack

return: a SQRESULT

sq_setfreevariable

```
SQRESULT sq_setfreevariable(HSQUIRRELVM v, int idx, int 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
int idx	index of the target object in the stack
int nval	0 based index of the free variable(relative to the closure).

return: a SQRESULT

sq_next

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

Pushes in the stack the next key and value of an array or table 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
int idx	index of the target object in the stack

return: a SQRESULT

Bytecode serialization

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

sq_readclosure

```
SQRESULT sq_readclosure(HSQUIRRELVM v, SQREADFUNC readf, SQUserPointer up);
```

serialize(read) the closure on top of the stack, the source is user defined through a write 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

Raw object handling

sq_getstackobj

```
SQRESULT sq_getstackobj(HSQUIRRELVM v, int idx, HSQOBJECT * po);
```

gets an object from the stack and stores it in a object handler.

parameters:

HSQUIRRELVM v	the target VM
int idx	index of the target object in the stack
*HSQOBJECT * po*	pointer to an object handler

return: a SQRESULT

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_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_release

```
void 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

sq_resetobject

```
void sq_resetobject(HSQUIRRELVM v, HSQOBJECT * po);
```

resets(initialize) an object handler.

parameters:

HSQUIRRELVM v	the target VM
*HSQOBJECT * po*	pointer to an object handler

remarks: Every object handler has to be initialized with this function.

Debug interface

sq_setdebughook

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

pops a closure from the stack an sets it as debug hook. on.

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_stackinfos

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

retrieve the calls stack informations of a ceratain level in the calls stack.

parameters:

HSQUIRRELVM v	the target VM
int level	calls stack level
*SQStackInfos * si*	pointer to the SQStackInfos structure that will store the stack informations

return: a SQRESULT.

Index

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

U

V

W

Y