

Squirrel 0.2pre-alpha

Reference manual

© 2003.Alberto Demichelis. All rights reserved.

You are allowed to copy and to print this document as long as you don't exploit the information commercially.

The author of this document makes no warranty of any kind, expressed or implied, with regard to the programs or the documentation contained in this document.

1 Contents

1	<u>CONTENTS.....</u>	<u>2</u>
2	<u>INTRODUCTION.....</u>	<u>5</u>
3	<u>THE LANGUAGE.....</u>	<u>5</u>
3.1	LEXICAL STRUCTURE	5
3.1.1	IDENTIFIERS	5
3.1.2	KEYWORDS	5
3.1.2.1	Operators	5
3.1.3	OTHER TOKENS	5
3.1.4	LITERALS	5
3.2	VALUES AND DATA TYPES	6
3.2.1	INTEGERS	6
3.2.2	FLOAT	6
3.2.3	STRING.....	6
3.2.4	NULL.....	6
3.2.5	TABLE	6
3.2.6	ARRAY	6
3.2.7	FUNCTION	7
3.2.8	GENERATOR.....	7
3.2.9	USERDATA	7
3.3	EXECUTION CONTEXT.....	7
3.3.1	VARIABLES	7
3.4	STATEMENTS.....	8
3.4.1	BLOCKS.....	8
3.4.2	CONTROL FLOW STATEMENTS	9
3.4.2.1	if/else	9
3.4.2.2	while	9
3.4.2.3	do/while.....	9
3.4.2.4	switch	10
3.4.3	LOOPS	10
3.4.3.1	for.....	10
3.4.3.2	foreach.....	10
3.4.4	BREAK	11
3.4.5	CONTINUE	11
3.4.6	RETURN	11
3.4.7	YIELD	11
3.4.8	LOCAL VARIABLE DECLARATION	11
3.4.9	FUNCTION DECLARATION	11
3.4.10	TRY/CATCH	11
3.4.11	THROW	12
3.4.12	EXPRESSION STATEMENT.....	12
3.5	EXPRESSIONS	12
3.5.1	ASSIGNMENT(=) & NEW SLOT(<-).....	12
3.5.2	OPERATORS.....	12
3.5.2.1	?: Operator.....	12
3.5.2.2	Arithmetic.....	12
3.5.2.3	Relational	13
3.5.2.4	Logical.....	13
3.5.2.5	in operator	13

3.5.2.6	typeof operator	14
3.5.2.7	Bitwise Operators	14
3.5.2.8	Operators precedence	14
3.5.3	TABLE CONSTRUCTOR	14
3.5.4	DELEGATE	15
3.5.5	CLONE	15
3.5.6	ARRAY CONSTRUCTOR	15
3.6	TABLES.....	16
3.6.1	CONSTRUCTION.....	16
3.6.2	SLOT CREATION	16
3.6.3	SLOT DELETION	16
3.7	ARRAYS.....	17
3.8	FUNCTIONS	17
3.8.1	FUNCTION DECLARATION	17
3.8.2	FUNCTION CALLS	18
3.8.3	FREE VARIABLES	18
3.8.4	TAIL RECURSION	18
3.9	GENERATORS.....	19
3.10	DELEGATION.....	19
3.11	METAMETHODS	20
3.11.1	_SET	20
3.11.2	_GET.....	20
3.11.3	_NEWSLOT.....	20
3.11.4	_ADD.....	20
3.11.5	_SUB.....	21
3.11.6	_MUL.....	21
3.11.7	_DIV	21
3.11.8	_MODULO.....	21
3.11.9	_UNM	21
3.11.10	_TYPEOF.....	21
3.11.11	_CMP	21
3.11.12	_CALL	21
3.11.13	_CLONE	21
3.11.14	_NEXTI	21
3.12	BUILT-IN FUNCTIONS.....	22
3.12.1	GLOBAL FUNCTIONS.....	22
3.12.2	DEFAULT DELEGATES.....	22
3.12.2.1	Integer	22
3.12.2.2	Float.....	22
3.12.2.3	String	22
3.12.2.4	Table.....	23
3.12.2.5	Array	23
3.12.2.6	Function.....	23
3.12.2.7	Generator.....	24

4 EMBEDDING SQUIRREL..... 24

4.1	MEMORY MANAGEMENT	24
4.2	ERROR CONVENTIONS	24
4.3	INITIALIZING SQUIRREL.....	25
4.4	THE STACK	25
4.4.1	STACK INDEXES	25
4.4.2	STACK MANIPULATION.....	25
4.5	RUNTIME ERRORS HANDLING	26
4.6	COMPILING A SCRIPT.....	27
4.7	CALLING A FUNCTION	27
4.8	CREATE A C FUNCTION	28
4.9	TABLES AND ARRAYS MANIPULATION.....	29
4.10	USERDATA AND USERPOINTERS	30

4.11	KEEPING OBJECT REFERENCE FROM C	31
4.12	DEBUG INTERFACE	31
<u>5</u>	<u>API REFERENCE MANUAL</u>	<u>32</u>
5.1	VIRTUAL MACHINE	32
5.2	COMPILER.....	33
5.3	STACK OPERATIONS.....	34
5.4	OBJECT CREATION AND HANDLING.....	35
5.5	CALLS	39
5.6	OBJECT MANIPULATION	40
5.7	RAW OBJECT HANDLING.....	44
5.8	FUNCTION PROTOTYPES	45
<u>6</u>	<u>INDEX.....</u>	<u>46</u>

2 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.

3 The language

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

3.1 Lexical structure

3.1.1 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]*

3.1.2 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	long	null	resume
return	switch	this	throw	try	typeof	while	yield

Keywords are covered in detail later in this document.

3.1.2.1 Operators

Squirrel recognizes the following operators:

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

3.1.3 Other tokens

Other used tokens are:

{ } [] . : :: ' ; "

3.1.4 Literals

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

34	Integer number
0xFF00A120	Integer number
'a' or 'FORM'	String literal

1.52	Floating point number
"I'm a string"	String

IntegerLiteral := [0-9]+ / '0x' [0-9A-Fa-f]+ / ''' [.]+ '''

FloatLiteral := [0-9]+ '.' [0-9]+

StringLiteral := '''[.]* '''

3.2 Values and Data types

Squirrel is a dynamically typed language so variables do not have a type, although they contain a value that does have a type.

Squirrel basic types are integer, float, string, null, table, array, function, generator and userdata.

3.2.1 Integers

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

```
local a = 123 //decimal
local b = 0x0012 //hexadecimal
```

3.2.2 Float

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

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

3.2.3 String

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

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

3.2.4 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
```

3.2.5 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; }
}
```

3.2.6 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];
```

3.2.7 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).

3.2.8 Generator

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

3.2.9 Userdata

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

3.3 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 its execution context. This means 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`.

3.3.1 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;

quad brackets are used to index tables or arrays

derefexp := exp '[' exp '']'

```
_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'.

3.4 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 & 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 '}'.

3.4.1 Blocks

stat := '{' stats '}'

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

3.4.2 Control Flow Statements

3.4.2.1 *if/else*

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

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

NOTE: 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;
}
```

3.4.2.2 *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;
    }
}
```

3.4.2.3 *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)
```

3.4.2.4 *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 of 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.

3.4.3 Loops

3.4.3.1 *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");
}
```

3.4.3.2 *foreach*

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

Executes a statement for every element contained in an array, table 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");
```

3.4.4 break

```
stat := 'break'
```

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

3.4.5 continue

```
stat := 'continue'
```

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

3.4.6 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.

3.4.7 yield

```
stat := yield [exp]
```

(see [Generators](#)).

3.4.8 Local variable 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);
```

3.4.9 Function declaration

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

creates a new function.

3.4.10 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 exception-handling code. When a catch clause catches an exception, its *id* is bound to that exception.

3.4.11 **throw**

stat := *'throw' exp*

Throws an exception. Any value can be thrown.

3.4.12 **expression statement**

stat := *exp*

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

3.5 *Expressions*

3.5.1 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.

3.5.2 Operators

3.5.2.1 **? : Operator**

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

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

3.5.2.2 **Arithmetic**

exp := *'exp' op 'exp'*

Squirrel supports the standard arithmetic operators +, -, * and /. Other than that it also supports 2 compact operators += and -=;

```
a+=1;
//is the same as writing
a=a+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.

3.5.2.3 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.

3.5.2.4 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 it 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.

3.5.2.5 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*

Eg.

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

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

if(123 in t) dostuff();
```

3.5.2.6 *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"
```

3.5.2.7 *Bitwise Operators*

exp := 'exp' op 'exp'
exp := '~' **exp**

Squirrel supports the standard c-like bit wise operators &,|,^,~,<<,>>. Those operators only work on integers values, passing of any other operand type to these operators will cause an exception.

3.5.2.8 *Operators precedence*

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

3.5.3 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
 ex.

```
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
ex.

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

both syntaxes can be mixed

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

The comma between slots is optional.

3.5.4 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](#)).

3.5.5 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 “_clone” meta method is called ([see Metamethods](#)).

3.5.6 Array constructor

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

Creates a new array.

eg.

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

arrays can be initialized with values during the construction

eg.

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

3.6 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).

3.6.1 Construction

Tables are created through the table constructor (see [Table constructor](#))

3.6.2 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.

eg.

```
local a={}
```

The following line will cause an exception because the slot named 'news1ot' does not exist in the table 'a'

```
a.news1ot = 1234
```

this will succeed:

```
a.news1ot <- 1234;
```

or

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

3.6.3 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"
```


3.7 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.

eg.

```
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](#)).

3.8 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.

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

3.8.2 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).

3.8.3 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.

3.8.4 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 happens 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);
```

3.9 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
```

3.10 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"

//
<<FIXME>>

```

3.11 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)`; It is possible to change many aspects of a table's 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 its 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");

```

3.11.1 `_set`

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

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

3.11.2 `_get`

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

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

3.11.3 `_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.

3.11.4 `_add`

the `+` operator

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

3.11.5 **_sub**

the – operator (like _add)

3.11.6 **_mul**

the * operator (like _add)

3.11.7 **_div**

the / operator (like _add)

3.11.8 **_modulo**

the % operator (like _add)

3.11.9 **_unm**

the unary minus operator

```
function _unm()
```

3.11.10 **_typeof**

invoked by the typeof operator on tables and userdata

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

3.11.11 **_cmp**

invoked to emulate the < > <= >= operators

```
function _cmp(other)
```

returns an integer

>0 if this>other

==0 if this==other

<0 if this<other

3.11.12 **_call**

invoked when a tables or userdatas are called

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

3.11.13 **_clone**

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

```
function clone()
```

3.11.14 **_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.

3.12 Built-in functions

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

3.12.1 Global functions

seterrorhandler(func)
sets the runtime error handler

setdebughook(hook_func)
sets the debug hook

getroottable()
returns the root table of the VM.

setroottable(table)
sets the root table of the VM and returns the previous one

assert(exp)
throw an exception if exp is null

print(x)
print x in the standard output

ascii2string(code)
converts an integer representing a ASCII code to a string

collect_garbage()
calls the garbage collector and returns the number of reference cycles found(and deleted)

//<<FIXME>>

3.12.2 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.

3.12.2.1 Integer

tofloat()
convert the number to float and returns it

tostring()
convert the number to string and returns it

3.12.2.2 Float

tointeger()
convert the number to integer and returns it

tostring()
convert the number to string and returns it

3.12.2.3 String

len()
returns the string length

tointeger()

convert the string to integer and returns it

tofloat()

convert the string to float and returns it

slice(start,[end])

3.12.2.4 *Table*

len()

returns the number of slots contained in a table

rawget(key)

try 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.

getdelegate()

returns the table delegate or null if no delegate is assigned.

3.12.2.5 *Array*

len()

returns the length of the array

append(val)

append the value 'val' at the end of the array

pop()

remove a value from the back of the array and returns it

top()

return the value of the array with the higher index

insert(idx,val)

insert the value 'val' at the position 'idx' in the array

remove(idx)

remove the value at the position 'idx' in the array

resize(size)

resize the array

sort()

sorts the array

slice(start,[end])

3.12.2.6 *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.

3.12.2.7 *Generator*

getstatus()

return the status of the generator : "running", "dead" or "suspended".

4 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'.

4.1 *Memory Management*

Squirrel uses reference counting (RC) as primary system for memory management; however, is possible to conditionally compile the virtual machine (VM) with an auxiliary mark and sweep garbage collector.

There are 3 possible compile time options:

First a situation where Squirrel is compiled with RC only; in this case it is impossible for the VM to detect reference cycles, so is the programmer that has to remove the links to avoid memory leaks (no special define is needed).

The second option is compiling with 'CYCLIC_REF_SAFE' defined; in this situation the VM keeps a linked list of all objects that can cause a reference cycle (tables, arrays etc..) and ensures that, when the VM is deleted, all cycles are detected and removed.

The third configuration consists in a mark and sweep garbage collector (option 'GARBAGE_COLLECTOR'). This option behaves like CYCLIC_REF_SAFE but in addition the host program can call the function sq_collectgarbage() and perform a garbage collection during the program execution. The garbage collector isn't invoked by the VM and has to be explicitly called by the host program.

The only disadvantage introduced by the second and third option is that 2 additional pointers have to be stored for each object (8 bytes for 32 bits systems).

The types involved are: tables, arrays, functions, userdata and generators; all other types are untouched. These options do not affect execution speed.

4.2 *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");
```


4.3 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_newvm()`.

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

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

    //do some stuff with squirrel here

    sq_releasevm(v);
}
```

4.4 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.

4.4.1 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)

	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;

4.4.2 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(HSQUIRRELV v);
```

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

```
void sq_settop(HSQUIRRELV v,int newtop);
```

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

The following function pushes a C value into the stack

```
void sq_pushstring(HSQUIRRELV v,const SQChar *s,int len);
void sq_pushfloat(HSQUIRRELV v,SQFloat f);
void sq_pushinteger(HSQUIRRELV v,SQInteger n);
void sq_pushuserpointer(HSQUIRRELV v,SQUserPointer p);
```

this function pushes a null into the stack

```
void sq_pushnull(HSQUIRRELV v);
```

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

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

4.5 Runtime errors 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(HSQUIRRELV v);
```

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

4.6 Compiling a script

You can compile a Squirrel script with the function `sq_compile`.

```
typedef int (*SQREADFUNC)(SQUserPointer userdata, SQUserPointer destbuf, int
size);
```

```
SQRESULT sq_compile(HSQUIRRELV v, SQREADFUNC read, SQUserPointer p, const
SQChar *sourcename, int raiseerror, int lineinfo);
```

In order to compile a script is necessary for the host application to implement a reader function(`SQREADFUNC`); this function is used to feed the compiler with the script data. The function is called every time the compiler needs some data; It has to fill the buffer pointed by `destbuf` with the number of bytes specified in `size` and return the number of bytes read if successful and `SQ_ERROR` if failed or if the data is finished.

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

Here an example of a 'read' function that read from a file:

```
int read_func(SQUserPointer file, SQUserPointer buf, int size)
{
    int ret;
    if( ( ret=fread(buf,1,size,(FILE *)file )!=0) )return ret;
    return -1;
}

int compile_file(HSQUIRRELV v, const char *filename)
{
    FILE *f=fopen(filename, "rb");
    if(f)
    {
        sq_compile(v, read_func, file, filename, 1, 0);
        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(HSQUIRRELV v, SQCOMPILERERROR f);
```

4.7 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 an exception is thrown) during the squirrel code execution the `sq_call` will fail.

4.8 Create a C function

A native C function must have the following prototype:

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

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

- Bigger than 0 if the function returns a value
- 0 if the function does not return a value
- Less than 0 if a runtime error is thrown

To obtain a new callable function 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 a value bigger than 0.

Here an example, the following function prints the value of each argument and returns 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"); //throw 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); //create a new function
    sq_set(v,-3);
    sq_pop(v,1); //pops the root table
}

```

4.9 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);
```

4.10 Userdata and UserPointers

Squirrel allows the host application put arbitrary data 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 anything it wants with this memory chunk, the VM will automatically take care 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, but default Squirrel cannot manipulate directly it. However it 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 gets deleted, it 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);
```

```
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 incur any memory allocation.

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

4.11 Keeping object reference from C

Squirrel allows to keep object's 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
in the stack

sq_addrref(v,&obj); //adds a reference to the object

... //do stuff

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

sq_release(v,&obj); //release the object
```

4.12 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. The callback will pass as argument the current line the current source and the current function name(if any).

```
SQUIRREL_API void sq_setdebughook(HSQUIRRELV 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(type,line_or_func,file)
{
    switch(type){
    case 'line':
        ::print("LINE line [" +line_or_func+"]);
        ::print("file [" +(file?file:"no source")+"]\n");
        break;
    case 'call':
        ::print("CALL func [" +line_or_func+"]);
        ::print("file [" +(file?file:"no source")+"]\n");
    }
}
```

The parameter `type` can be 'line' or 'call'; the first if a new line was executed the second if a function call occurred.

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

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

While the local variables info through `sq_getlocal()`

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

5 API Reference Manual

5.1 Virtual Machine

sq_newvm

HSQIRRELVM sq_newvm(HSQIRRELVM friendvm,int initialstacksize)

creates a new instance of a squirrel VM that consists in a new stack, new environment table etc..

parameters:

HSQIRRELVM friendvm

A vm that can share objects with the newone, this parameter can be NULL.

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_releasevm

void sq_releasevm(HSQIRRELVM v)

release a squirrel VM

parameters:

HSQIRRELVM v

a handle to a squirrel VM

sq_seterrorhandler

void sq_seterrorhandler(HSQIRRELVM v)

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

parameters:

HSQIRRELVM v

the target VM

sq_setforeignptr

void sq_setforeignptr(HSQIRRELVM 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:

HSQIRRELVM 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 VM's foreign pointer.

5.2 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

sq_compile

SQRESULT **sq_compile**(*HSQUIRRELVM v, HSQREADFUNC read, SQUserPointer p, const SQChar *sourcename, int raiseerror, int lineinfo*)

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

parameters:

HSQUIRRELVM v
the target VM
HSQREADFUNC 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
int lineinfo
if this value is different than 0 the compiled code will contain debug information (to use with the runtime debug interface)

return:

an *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()**.

5.3 Stack Operations

sq_push

void sq_push(HSQIRRELVM v, int idx)
pushes in the stack the value at the index idx

parameters:

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

sq_pop

void sq_pop(HSQIRRELVM v, int nelemstopop)
pops n elements from the stack

parameters:

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

sq_remove

void sq_remove(HSQIRRELVM v, int idx)
removes an element from an arbitrary position in the stack

parameters:

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

sq_gettop

int sq_gettop(HSQIRRELVM v)
returns the index of the top of the stack

parameters:

HSQIRRELVM v
the target VM

return:

an integer representing the index of the top of the stack

sq_settop

void sq_settop(HSQIRRELVM v, int newtop)
resize the stack, if new top is bigger then the current top the function will push nulls.

parameters:

HSQIRRELVM v
the target VM
int newtop
the new top index

sq_cmp

int sq_cmp(HSQIRRELVM v)

pops 2 object from the stack and compares them.

parameters:

HSQIRRELVM v
the target VM

return:

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

5.4 Object creation and handling

sq_newuserdata

SQUserPointer sq_newuserdata (HSQIRRELVM v,unsigned int size);

creates a new userdata and pushes it in the stack

parameters:

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

return:

a pointer to the userdata buffer

sq_newtable

void sq_newtable (HSQIRRELVM v);

creates a new table and pushes it in the stack

parameters:

HSQIRRELVM v
the target VM

sq_newarray

void sq_newarray (HSQIRRELVM v,int size);

creates a new array and pushes it in the stack

parameters:

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

sq_newclosure

void sq_newclosure(HSQIRRELVM v,HSQFUNCTION func,unsigned 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:

HSQIRRELVM v
the target VM
HSQFUNCTION func
a pointer to a native-function
int nfreevars
number of free variables(can be 0)

sq_pushstring

`void sq_pushstring(HSQUIRRELVM v, const SQChar *s, int len);`
 pushes a string in the stack

parameters:

HSQFUNCTION 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:

HSQFUNCTION 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)`

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

sq_getsize

SQInteger 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 and tables 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:

an 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:

an SQRESULT

sq_getfloat

SQRESULT sq_getfloat(HSQIRRELVM v,int idx,SQFloat *f)
 gets the value of the float at the idx position in the stack.

parameters:

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

return:

an SQRESULT

sq_getuserpointer

SQRESULT sq_getuserpointer(HSQIRRELVM v,int idx,SQUserPointer *p)
 gets the value of the userpointer at the idx position in the stack.

parameters:

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

return:

an SQRESULT

sq_getuserdata

SQRESULT sq_getuserdata(HSQIRRELVM v,int idx,SQUserPointer *p)
 gets a pointer to the value of the userdata at the idx position in the stack.

parameters:

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

return:

an SQRESULT

sq_setreleasehook

void sq_setreleasehook(HSQIRRELVM v,int idx,HSQUSERDATARELEASE hook)
 sets the release hook of the userdata at position idx in the stack.

parameters:

HSQIRRELVM v
 the target VM
 int idx
 an index in the stack
 HSQUSERDATARELEASE hook
 a function pointer(see HSQUSERDATARELEASE)

remarks:

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

sq_getscratchpad

SQChar *sq_getscratchpad(HSQIRRELVM v,int minsize);

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

parameters:

HSQUIRRELM 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

5.5 Calls

sq_call

SQRESULT sq_call(HSQUIRRELM v, int params, int retval)

calls a closure or a native closure.

parameters:

HSQUIRRELM 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:

an SQRESULT

remarks:

pops n values(params) and a closure and calls it; if retval != 0 the return value of the closure is pushed.

sq_resume

SQRESULT sq_resume(HSQUIRRELM v, int retval)

resumes the generator at the top position of the stack.

parameters:

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

return:

an SQRESULT

remarks:

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

sq_getlocal

const SQChar *sq_getlocal(HSQUIRRELM v, unsigned int level, unsigned int nseq)

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

parameters:

HSQUIRRELM 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.

5.6 Object manipulation

sq_pushroottable

void sq_pushroottable(HSQUIRRELVM v);

pushes the current root 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_createslot

void sq_createslot(HSQUIRRELVM v, int idx)

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

parameters:

HSQUIRRELV `v`
the target VM
`int idx`
index of the target object in the stack

return:

a value >0 if succeeded <0 if failed

remarks:

no delegation is involved in the operation. it only works on tables.

sq_set

SQRESULT sq_set(HSQUIRRELV `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:

HSQUIRRELV `v`
the target VM
`int idx`
index of the target object in the stack

return:

an 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(HSQUIRRELV `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:

HSQUIRRELV `v`
the target VM
`int idx`
index of the target object in the stack

return:

an SQRESULT

remarks:

this call will invoke 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_rawget

SQRESULT sq_rawget(HSQUIRRELV `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:

HSQUIRRELV `v`
the target VM
`int idx`
index of the target object in the stack

return:

an SQRESULT

remarks:

Only works on tables, arrays and userdata.

sq_rawset

SQRESULT sq_rawset(HSQIRRELVM 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:

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

return:

an SQRESULT

remarks:

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

sq_arrayappend

SQRESULT sq_arrayappend(HSQIRRELVM 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:

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

return:

an SQRESULT

remarks:

Only works on arrays.

sq_arraypop

SQRESULT sq_arraypop(HSQIRRELVM v,int idx);

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

parameters:

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

return:

an SQRESULT

remarks:

it only works on arrays.

sq_arrayresize

SQRESULT sq_arrayresize(HSQIRRELVM v,int idx,int newsize);

resizes the array at the position idx in the stack.

parameters:

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

int newsize
 requested size of the array
return:
 an SQRESULT
remarks:
 it only works on arrays.

sq_setdelegate

SQRESULT sq_setdelegate(HSQIRRELVM 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:
 HSQIRRELVM v
 the target VM
 int idx
 index of the target object in the stack
return:
 an SQRESULT

sq_getdelegate

SQRESULT sq_getdelegate(HSQIRRELVM v,int idx)
 pushes the current delegate of the object at the position idx in the stack.

parameters:
 HSQIRRELVM v
 the target VM
 int idx
 index of the target object in the stack
return:
 an SQRESULT

sq_setfreevariable

SQRESULT sq_setfreevariable(HSQIRRELVM v,int idx,unsigned 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:
 HSQIRRELVM 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:
 an SQRESULT

sq_next

SQRESULT sq_next(HSQIRRELVM v,int idx)
 Iterate a table or array.<<FIXME>>

parameters:
 HSQIRRELVM v
 the target VM
 int idx
 index of the target object in the stack
return:

an SQRESULT

5.7 Raw object handling

sq_getstackobj

SQRESULT *sq_getstackobj*(*HSQUIRRELM* *v*, *int* *idx*, *HSQOBJECT* **po*)
gets an object from the stack and stores it in a object handler.

parameters:

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

return:

an SQRESULT

sq_pushobject

void *sq_pushobject*(*HSQUIRRELM* *v*, *HSQOBJECT* *obj*)

push an object referenced by an object handler into the stack.

parameters:

HSQUIRRELM *v*
the target VM
int *idx*
index of the target object in the stack
HSQOBJECT *obj*
object handler

sq_addref

void *sq_addref*(*HSQUIRRELM* *v*, *HSQOBJECT* **po*);
add a reference to an object handler.

parameters:

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

sq_release

void *sq_release*(*HSQUIRRELM* *v*, *HSQOBJECT* **po*);

remove a reference to an object handler.

parameters:

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

sq_resetobject

void *sq_resetobject*(*HSQUIRRELM* *v*, *HSQOBJECT* **obj*);

reset(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.

5.8 Function prototypes

SQFUNCTION

```
typedef int (*SQFUNCTION)(HSQUIRRELVm);
```

SQUSERDATARELEASE

```
typedef int (*SQUSERDATARELEASE)(SQUserPointer);
```

SQCOMPILERERROR

```
typedef void (*SQCOMPILERERROR)(HSQUIRRELVm, const SQChar * /*desc*/, const  
SQChar * /*source*/, int /*line*/, int /*column*/);
```

SQWRITEFUNC

```
typedef int (*SQWRITEFUNC)(SQUserPointer, SQUserPointer, int);
```

SQREADFUNC

```
typedef int (*SQREADFUNC)(SQUserPointer, SQUserPointer, int);
```

SQUIRREL_MALLOC

```
typedef void *(*SQUIRREL_MALLOC)(unsigned int);
```

SQUIRREL_REALLOC

```
typedef void *(*SQUIRREL_REALLOC)(void*, unsigned int, unsigned int);
```

SQUIRREL_FREE

```
typedef void (*SQUIRREL_FREE)(void*, unsigned int);
```

6 Index

- sq_addrf, 44
- sq_arrayappend, 42
- sq_arraypop, 42
- sq_arrayresize, 42
- sq_call, 39
- sq_cmp, 34
- sq_compile, 33
- sq_createslot, 40
- sq_get, 41
- sq_getdelegate, 43
- sq_getfloat, 37
- sq_getforeignptr, 33
- sq_getinteger, 37
- sq_getlasterror, 40
- sq_getlocal, 39
- sq_getscratchpad, 38
- sq_getsize, 37
- sq_getstackobj, 44
- sq_getstring, 37
- sq_gettop, 34
- sq_gettype, 36
- sq_getuserdata, 38
- sq_getuserpointer, 38
- sq_newarray, 35
- sq_newclosure, 35
- sq_newtable, 35
- sq_newuserdata, 35
- sq_newvm, 32
- sq_next, 43
- sq_pop, 34
- sq_push, 34
- sq_pushfloat, 36
- sq_pushinteger, 36
- sq_pushnull, 36
- sq_pushobject, 44
- sq_pushroottable, 40
- sq_pushstring, 35
- sq_pushuserpointer, 36
- sq_rawget, 41
- sq_rawset, 42
- sq_release, 44
- sq_releasevm, 32
- sq_remove, 34
- sq_resetobject, 44
- sq_resume, 39
- sq_set, 41
- sq_setcompilererrorhandler, 33
- sq_setdelegate, 43
- sq_seterrorhandler, 32
- sq_setforeignptr, 32
- sq_setfreevariable, 43
- sq_setreleasehook, 38
- sq_setroottable, 40
- sq_settop, 34
- sq_throwerror, 40
- SQCOMPILERERROR, 45
- SQFUNCTION, 45
- SQREADFUNC, 45
- SQUIRREL_FREE, 45
- SQUIRREL_MALLOC, 45
- SQUIRREL_REALLOC, 45
- SQUSERDATARELEASE, 45
- SQWRITEFUNC, 45
- Table, 6, 14, 16, 23
 - delegation, 15, 19
- Userdata
 - userdata, 7