Grid Cartographer 4
Scripting Manual

Version 4.0.6

Table of Contents

Chapter 1 Introduction

1.1 Scripting Overview

Grid Cartographer Pro Edition incorporates powerful scripting functionality to allow exporting of map data into user defined formats.

The scripting capabilities of Grid Cartographer are not available in the Gamer Edition.

1.2 Built-In Scripts

Comma-Separated Values

Exports the current tile map floor as a comma-separated values file. Each value in the output is a number that represents a custom tile index. Empty tiles appear as missing values in the row.

Instructions: Go to the *File* menu and *Export Data* page. Select the *Comma-Separated Values* option from under the *Export Scripts* heading. A file selector dialog box will appear asking you where to save the file (extension `.csv`). Click *OK* and the file will be written. A message box prompt will appear when the operation is complete indicating success or if any error occurred.

1.3 Installing User Scripts

New scripts can be installed into Grid Cartographer by copying them into the User Scripts folder. On start-up the root of this folder is scanned (not sub-directories) for XML 'hook' files and registered with the software. See the next chapter *Creating Scripts* for more information on the hook file format.

You can open this folder from within Grid Cartographer by going to the *File* menu, selecting the *Scripts* page and clicking the *User Scripts* button.

Grid Cartographer must be restarted after installing new scripts.

1.4 Stopping a Faulty Script

If a script spends too long without interacting with the Grid Cartographer API, it is presumed to have malfunctioned (e.g. it has become stuck in a loop). This is indicated by a *Not Responding* message shown on the console window caption.

In the event of the script entering this state it may be aborted by opening the console window and pressing `CTRL+C` (`CMD+C` on mac os). The script will then stop immediately. Depending on what the script was doing some data may be lost. It is recommended to backup your work and restart Grid Cartographer in the event of a faulty script.

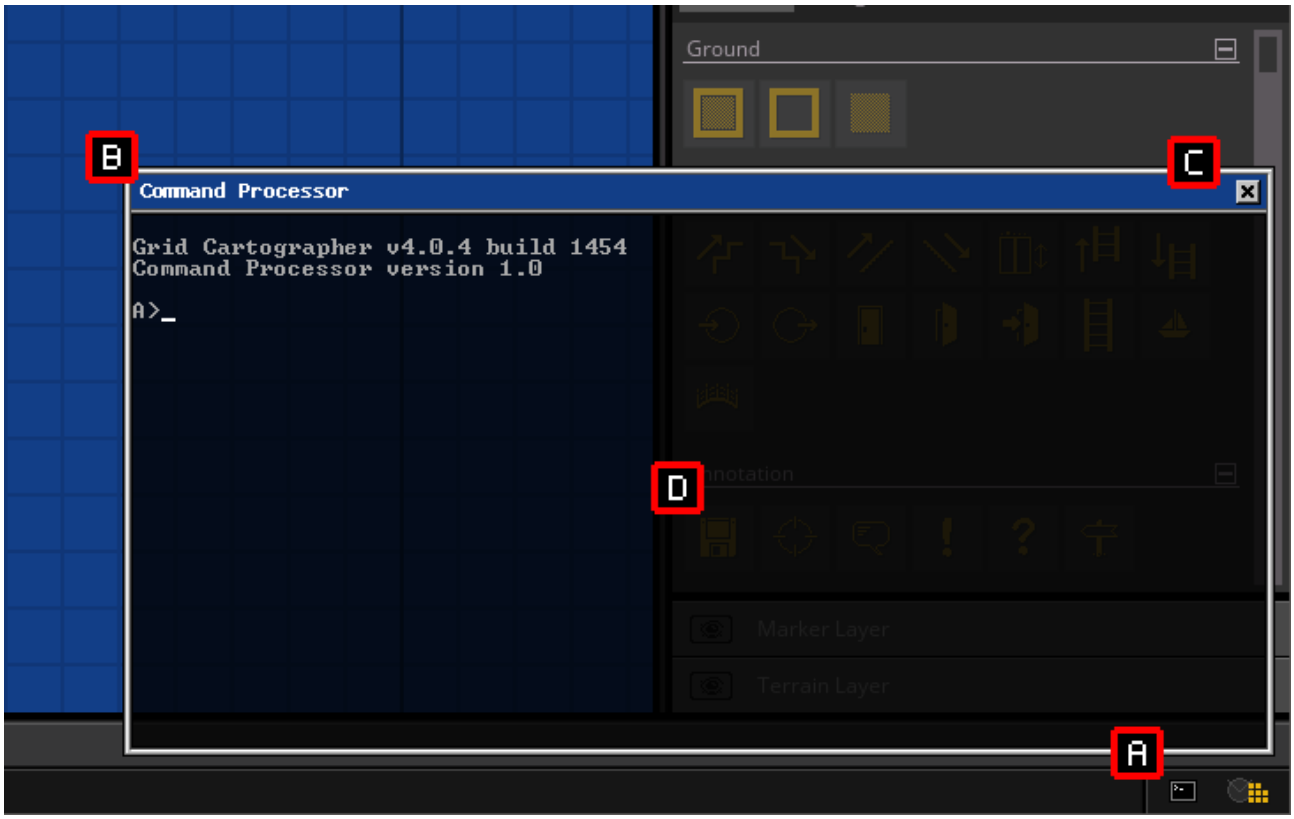
Chapter 2 Console

Introduction

Grid Cartographer scripts can make use of the interactive console window. The console provides a simple text based output and keyboard input "terminal" which can be used to print messages, give prompts and listen for typed commands.

2.1 Console Window

To open the console window click the icon on the status bar in the bottom-right of the user interface `[A]`, or press the 'console key' on your keyboard (by default this is located directly above the TAB key and marked with a tilde (~) character on US keyboards).



The console window will open in the lower-right of the main interface by default. The window is divided into two main parts, the window caption `[B]` and content area `[D]`. To close the console window at any time, press the X button on the top-right hand corner `[C]`.

Moving the Console

You can move the console window around the interface by clicking on and dragging in the window caption area. If you move the console near to one of the four corners it will automatically snap into position and remain aligned to that corner if the application window is resized. This snapping behavior can be disabled by holding the `CTRL` key on Windows and Linux, or the `CMD` key on macOS.

Scrolling

When text is written into the console that reaches the bottom of the window, it is scrolled upwards and the top line is moved out of view. This text is not discarded immediately however and a limited buffer is used to preserve the last 300 lines. In order to scroll this older text back into view use the mouse wheel while positioning the pointer over the console window. Note that clearing the console using the `CLS` command will also clear the history buffer.

Appearance

The console uses a semi-transparent background by default. The opacity can be adjusted using the slider in the *Options* menu on the *Interface* page.

2.2 Virtual File Store

Grid Cartographer provides a basic file system facility known as the *Virtual File Store* for persistent storage of data and script files, should they be needed.

The location of the Virtual File Store on your operating system can be found from within the application by using the *File* > *Scripts* > *File Store* menu option. However it is recommended to use the console command processor (see below) to manipulate file data, so as to avoid any incompatibly named files.

Within the file store is a folder called A, this is the primary virtual "drive" of the file system and contains all the files associated with that drive. Other folders can be created by the user to add 25 additional virtual drives (`B - Z`), as needed.

Files within virtual drives have more restrictive naming than any of the underlying operating systems that Grid Cartographer runs on. A valid file name contains only upper case letters `A` to `Z`, numbers `0 - 9` or underscore characters. A file extension is always required, separating the name and extension is a '**.**' (period) character. Non-conforming file names (which can occur if you manually edit the operating system) are ignored by all functions.

2.3 Console Command Processor

By default the console is automatically loaded with a simple utility service called the *Console Command Processor* or CCP.

The CCP provides a command line interface to assist with management of files and other tasks. The full list of built-in commands can be found below.

In addition, the CCP can execute Squirrel scripts that are stored in files with the NUT file extension. See the following chapters for more information on writing scripts.

Built-in Commands

The CCP has a number of built-in commands described below. Many of these commands are related to manipulating files in the virtual file store. Additional commands can be added through user scripts.

Change Drive

Syntax: `<DRIVE>:`

Select the default drive for subsequent commands. If the drive does not exist you will not be permitted to select it. Use the `MKDRIVE` command to create a new drive in the virtual store.

Close Console

Syntax: `CLOSE`

Close the console window.

Clear Screen

Syntax: `CLS`

Clear the console window, including any scroll history.

Command Sub-process

Syntax: `COMMAND`

Launch a new command processor as a child process. Use EXIT to return.

Copy File

Syntax: `COPY <SRCFILEPATH> <DSTFILEPATH>`

Copy a file from one location to another, possibly to another virtual drive. During the copy operation the file name may also be changed. If the destination file already exists the copy will fail and print an error message.

Delete File

Syntax: `DEL <FILEPATH>`

Delete a file from the virtual store immediately. The `filepath` parameter must be a specific file (wildcards are not supported). If the path is not qualified with a virtual drive letter the current drive will be used.

Examples:

```
A>DEL FILE.TXT        - delete FILE.TXT from drive A (current)
A>DEL B:FILE.TXT      - delete FILE.TXT from drive B
```

Directory Listing

Syntax: `DIR [<FILTER>]`

Print a directory listing to the console. This command accepts an optional filter. Filters use characters `*` to match any sequence of characters and `?` to match a single character.

Examples:

```
A>DIR                 - list all files on the current drive.
A>DIR B:XA*.*         - list files beginning with XA on drive B.
A>DIR *.TXT           - list files with the TXT extension in drive A.
A>DIR C:FILE?.TXT     - list FILE.TXT, FILE1.TXT, FILEF.TXT, etc.
```

Exit

Syntax: `EXIT`

Exit the Command Processor. If it is running as a child process, control will return to the parent. If this is the top level process the Command Processor will restart automatically.

Export File

Syntax: `EXPORT <FILEPATH>`

Copy a file from the virtual file store to an anywhere on your computer. When this command is used a standard file selector will open and allow you to chose the destination of the copy. This is the recommended method of moving files out of the file store.

Help

Syntax: `HELP`

Print a list of all supported commands, including those added by installing user scripts.

Import File

Syntax: `IMPORT <FILEPATH>`

Copy a file from anywhere on your computer to the virtual file store. When this command is used a standard file selector will open and allow you to chose the source file of the copy. This is the recommended method of moving files into the file store.

Move File

Syntax: `MOVE <SRCFILEPATH> <DSTFILEPATH>`

Move a file from one location to another, possibly to another virtual drive. During the move operation the file name may also be changed. If the destination file already exists the move will fail and print an error message.

Rename File

Syntax: `REN <OLDFILE> <NEWFILE>`

Change the name of a file. A rename operation cannot be used to overwrite an existing file nor move the file to a new virtual drive.

File Size

Syntax: `SIZE <FILEPATH>`

Print the size, in bytes, of the specified file.

Type File Contents

Syntax: `TYPE <FILEPATH>`

Open the specified file and print its contents to the console.

Version Information

Syntax: `VER`

Print the version number of the command processor and Grid Cartographer to the console.

Chapter 3 Creating Scripts

Introduction

Grid Cartographer scripts are intended to be created by the users and community to meet their own requirements.

This chapter describes how to create a script that hooks into the user interface and how to install it. In order to do this requires no special tools, only the use of a text editor (such as Notepad or TextEdit).

This chapter is not a guide for how to program. You will some find help for that in the friendly community of the Grid Cartographer forum, and on the wider Internet.

3.1 Components of a Script

A complete script is formed from two parts - a hook and some code. A hook is specification that Grid Cartographer uses to attach the script code to part of the user interface.

A hook is written in XML and looks something like this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<script>
     <hook location="file _ export _ tilemap">
          <button>BTN _ CAPTION</button>
          <vm language="squirrel">
               <load src="export _ csv.nut" />
               <call function="main" />
          </vm>
          <locale id="en-us">
               <string id="BTN _ CAPTION">
                    <![CDATA[COMMA-SEPARATED VALUES]]>
               </string>
          </locale>
     </hook>
</script>
```

This hook requests that when editing a tile map a button should be added to the *Export Data* page of the *File* menu with the caption “Comma-separated Values”. When clicking the button a script (written in the Squirrel programming language) called `export_csv.nut` should be loaded into memory. The hook states that once loaded, Grid Cartographer should execute the function called `main`.

The script code itself is written using one of the supported programming languages (currently only Squirrel). You can find example scripts on the Grid Cartographer forum.

3.2 Hook File Specification

Overview

Hook files are written in XML and use a simple container format. The root element should be `<script>` and each hook is described under a `<hook>` element. Multiple hooks can be included in a single file and will be processed in the order they appear in the document.

Each `<hook>` element requires a `location` attribute that specifies where it should be attached. The following sections describe the formats depending on this attribute. Since many of the child elements required for each hook type are common amongst multiple types, they are documented separately in the section following this one.

Command Hooks

Hooks can attach scripts to the console as user commands. These hooks require the `command` location attribute. The format of the hook definition is:

```
<hook>
|--- <name>
|--- <vm>
|    |--- <load>
|    \--- <call>
|         \--- <arg>
\--- <locale>
     \--- <string>
```

Note: Command hooks installed in the *User Scripts* folder will override and replace built-in commands with the same id.

Export Data Hooks

Hooks attached to the *Export Data* page of the *File* menu use either the `file_export_standard` or `file_export_tilemap` location attributes. The former applying when currently editing a region with either a square or hexagonal grid shape. The latter is only available when editing a tile map region.

The format of the hook definition is the same for either of these locations:

```
<hook>
|--- <button>
|--- <vm>
|    |--- <load>
|    \--- <call>
```

```
|          \--- <arg>
\--- <locale>
     \--- <string>
```

3.3 Hook Element Descriptions

Since many elements are common to multiple hook types, this section contains a complete reference for all of them.

<hook>

This is the top level element for the hook

Attribute	Meaning
`location`	The location code of the hook. One of: • `command` • `file_export_standard` • `file_export_tilemap`

<button>

The caption for a menu button is stored in a child CDATA element of this element. The text isn't used literally, rather it is looked up in the localization table to get the correct string for the current interface language.

<name>

The name of a command is stored in the id attribute of this element.

<vm>

This is the container element for the virtual machine to create to compile and execute the export script.

Attribute	Meaning
`language`	Specifies the scripting language to use. Only `squirrel` is supported at this time. This attribute must be used.

<load>

This element specified a script file to load into the virtual machine. Multiple load elements are supported and there must be at least one.

Attribute	Meaning
`src`	Specifies the file name of the script to load. The file is searched for in the following places, in order: • Relative to the path the hook XML file is in. • The *User Scripts* folder. • The internal scripts folder (`base0.zip/scripts/`)

<call>

This element specifies the function to call automatically after all scripts files have been loaded and compiled - the entrypoint. Only one `<call>` element can be used. This call is optional (can put your code at the top level of the script) - it is recommended for more complex scripts with multiple source files.

Attribute	Meaning
`function`	The function name identifier without decoration, paramters, etc.. e.g. `"main"`

<arg>

Arguments can be added to the entrypoint function call using `<arg>` elements, if required. Each element is added in the order specified in the hook file.

Attribute	Meaning
`type`	Data type of the argument. Pick one from: `bool`, `int`, `float` or `string`. An additional type `user` is also available and it will add an array of strings when spawned using `GCKernel.ExecHookParam`.
`value`	The value of the data. Bool values support `true` or `1` and `false` or `0`. String values allow anything including an empty value (which is passed in as `null`)

<locale>

Container for localization string tables. Multiple `<locale>` elements are supported and are added to the table in order, with later strings replacing older ones.

Localization strings are accessible to the child script and should be used where possible in place of writing string text directly into the script, to allow for future localization.

Attribute	Meaning
`id`	Specify the locale id for the strings within this container (e.g. '`en-us`'). If the attribute is missing then the strings will apply to all locales.

<string>

Holds a mapping from an id to a string of text.

Attribute	Meaning
`id`	The string id. Duplicates replace existing strings with new content.

The text of the string should be contained within a child `<![CDATA[]]>` element.

Chapter 4 Squirrel Programming Language

Introduction

Grid Cartographer scripts can be written using the Squirrel programming language created by Alberto Demichelis. It is currently the only supported language for scripting.

The official website for Squirrel is: http://squirrel-lang.org/

"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."

4.1 Grid Cartographer Changes

The version of Squirrel included in Grid Cartographer is a modified version of the v3.0.6 release. This section lists the changes that have been made to the language specification and runtime operation.

• `var` is now a reserved word. It is a synonym for `local`.

• # line comments have been removed. Use `//` comments instead.

• `#"` prefix applied to string literals computes the CRC32 hash of the string at compile time and replaces the string with this hash as an integer value.

• Allow setting default `class` values after declaration.

• Renamed the variadic argument array parameter to `vargs`.

• Renamed the function environment binding method to `Bind`.

• Renamed the class attribute get/set functions to `GetAttributes` and `SetAttribute`.

• Replaced all built-in functions and standard libraries with the Grid Cartographer API (see the *API Reference* chapter).

• Removed `userdata` and `thread` (co-routines) data types.

• Removed 'const table' access for runtime `const` / `enum` modification.

• Renamed the modulo meta-method to `_mod`.

Remainder of this Chapter

The rest of this chapter is an adapted copy of the official *Squirrel Reference Manual* with the changes above applied to describe the final version of the language as implemented within Grid Cartographer.

4.2 Lexical Structure

Identifiers

Identifiers start with a alphabetic character or '_' (underscore) followed by any number of alphabetic characters, '_' (underscore) 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 is a list of words reserved by the language that cannot be used as identifiers:

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

Keywords are covered in more detail below.

Operators

Squirrel recognizes the following operators:

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

Other Tokens

Other language tokens are:

```
{    }    [    ]    .    :    ::    '    ;    "    @"    #"
</   />
```

Literals

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

`34`	Integer number (base 10)
`0xFF00A120`	Hexadecimal number (base 16)
`0753`	Octal number (base 8)
`0b10101`	Binary number (base 2)
`'a'`	Integer number
`1.52`	Floating point number
`1.e2`	Floating point number
`1.e-2`	Floating point number
`"I'm a string"`	String
`@"I'm a verbatim string"`	String
`@"I'm a multiline verbatim string "`	String
`#"Convert me to a hash"`	Integer number

```
IntegerLiteral := [1-9][0-9]*
                | 0[0-7]+
                | '0x' [0-9A-Fa-f]+
                | ''' [.]+ '''

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

FloatLiteral := [0-9]+ '.' 'e'|'E' '+'|'-' [0-9]+

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

VerbatimStringLiteral:= '@"' [.]* '"'

HashStringLiteral:= '#"' [.]* '"'
```

Comments

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

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

```
/*
     This is
     a multiline comment.
     All of these lines will be ignored by the compiler
*/
```

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

```
// This is a single line comment. Ignored by the compiler
```

4.3 Values and Data Types

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

Integer

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

```
var a = 123; // decimal
var b = 0x0012; // hexadecimal
var c = 075; // octal
var d = 0b1001; // binary
var e = 'w'; // char code
```

Float

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

```
var a = 1.0
var b = 0.234
```

String

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

Standard strings are delimited by quotation marks (") and can contain escape sequences (\t, \a, \b, \r, \n, \v, \f, \\, \", \', \0, \xhhhh).

```
var a = "I'm a string\n";

// The \n is converted into a newline at the end of the string
```

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:

```
var x = @"I'm a verbatim string\n";

// The \n is literally copied into the string
// To do the same in a standard string use \\n
```

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:

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

Null

The null value is a primitive value that represents a null, empty, or nonexistent reference. The type `null` has exactly one value, called `null`.

```
var a = null;
```

Bool

The `bool` data type has only two values, the literals `true` and `false`. A bool value expresses the validity of a condition (tells whether the condition is true or false).

```
var a = true;
```

Table

Tables are dynamic associative containers implemented as a key/value pair (called a slot).

```
var t = {};

var table2 = {
     a = 10,
     b = function( a ) { return a+1; }
}
```

Array

Arrays are linear sequences of objects, their size is dynamic and their index starts from `0` (zero).

```
var a = [ "I'm", "an", "array" ];

var b = [ null ]

b[0] = a[2];
```

Function

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

Class

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

Class Instance

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

Generator

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

Weak References

Weak References are objects that point to another (non value type) object but do not own a strong reference to it. (See *Weak References* section below).

4.4 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 execution, the body of a function can only transparently refer to the 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 table/array 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 )
{
	var a = 10;
	print( a );
	return arg;
}
```

Will access local variable `'a'` and output `10` to the console.

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

In this example `foo` will be equivalent to `this.foo` or `this["foo"]`.

Global variables are stored in a table called the `root` table. If a variable is not local and is not found in the 'this' object Squirrel will search it in the root table.

To explicitly access the global table from another scope, the slot name must be prefixed with `::` (e.g. `::foo`).

```
exp := '::' id
```

For instance:

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

Accesses the global variable '`foo`' in the root table and ignores the '`this`' object.

For example:

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

Is the equivalent to writing:

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

4.5 Statements

A Squirrel program is a sequence of statements.

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

Statements in Squirrel are comparable to the 'C' family of languages (C/C++, Java, C# etc.) and include assignment, function calls, program flow control structures plus some new statements like table and array constructors, and yield. These are covered below.

Statements can be separated with a new line or '`;`'. Additionally the keywords `case` or `default` can be used inside a switch/case statement. Neither symbol is required if the statement is followed by '}'.

Block

```
stat := '{' stats '}'
```

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

Control Flow Statements

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

true and false

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

if / else

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

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

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


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

while

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

Executes a statement while the condition is true. If the expression evaluates to `false` on the first iteration, the statement is not executed at all.

```
function testy( n )
{
	var 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 the condition expression evaluates to `false`.

```
var 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 that directs control to one of the case statements within its body based on the value of an expression. Control is transferred to the case label whose `case_exp` matches with `exp`. If no match is found, control will jump to the default label (if present).

Once control has jumped all following statements will be executed including those following further case statements (execution will 'fall through' these other case instances). Use a

`break` statement to stop this behaviour and jump again to the end of the switch block.

A switch statement can contain any number of case instances, if two cases have the same case expression only the first one will be used. The `default` label is permitted once and must appear after all `case` instances.

Loop Statements

for

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

Executes a statement as long as a condition expression isn't false.

```
for ( var a = 0; a < 10; ++a )
     println( a );

// or

glob <- null

for ( glob = 0; glob < 10; glob += 2 ) {
     print ( glob + "\n" );
}

// or

for ( ; ; ) {
     print( "loop forever" );
}
```

foreach

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

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

```
var a = [ 10, 23, 33, 41, 589, 56 ];

foreach( val in a )
     println( "value = " + val );

// or

foreach ( idx, val in a )
     print( "index = " + idx + " 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 any following statements.

yield

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

See the *Generators* section below.

Other 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 cease to be resumable.

Local Variable Declaration

```
init := id [= exp][',' initz]

stat := 'var' init
      | 'local' init
```

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. A local declaration statement is also allowed as the first expression in a `for` loop.

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

Function Declaration

```
funcname := id ['::' id]

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

Creates a new function.

Class Declaration

```
memberdecl := id '=' exp [';']
            | '[' exp ']' '=' exp [';']
            | function-stat
            | 'constructor'

stat := 'class' derefexp ['extends' derefexp] '{'
          [memberdecl]
'}'
```

Creates a new class.

try / catch

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

The try statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a throw statement. The catch clause provides the exception handling code. When it catches an exception, the id of the exception is passed to the catch statement like a function argument.

throw

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

Throws an exception. The value of the expression can be of any type. If the throw occurs during a try/catch statement, the result of the thrown expression will be passed to the catch clause. If outside of a try/catch a runtime error will occur and script execution will stop.

const

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

Declares a constant (see *Constants & Enumerations*).

enum

```
enums := ( 'id' '=' Integer | Float | StringLiteral ) [',']

stat:= 'enum' id '{' enumerations '}'
```

Declares an enumeration (see *Constants & Enumerations*).

Expression Statement

```
stat := exp
```

In Squirrel every expression is also a valid statement. The value of the expression is thrown away (but side effects may be useful).

4.6 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 permits creation of a new slot in a table (see Tables). If the slot already exists in the table it behaves like a normal assignment. You cannot create new table slots with normal assignment.

Operators

?: Operator

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

Conditionally evaluate an expression depending on the result of a control expression.

Arithmetic Operators

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

Squirrel supports the standard arithmetic operators (+, -, *, / and %). It also supports compact assignment operators (+=, -=, *=, /= and %=) and increment and decrement operators (++ and --)

```
a += 2;
```

```
// is the same as writing
```

```
a = a + 2;
```

```
x++
```

```
// is the same as writing
```

```
x = x + 1
```

All operators work normally with integers and floats; if one operand is an integer and one is a float the result of the expression will be float.

The + operator has a special behavior with strings; if one of the operands is a string the operator + will try to convert the other operand to string as well and concatenate both together. For instances and tables, `ToString` is invoked.

Relational Operators

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

Squirrel supports the standard relational operators: ==, <, <=, >, >= and !=

The result of each of these operators is true if the relation is `true` (e.g. 3 < 5 or 3 != 5) and `false` if the relation is false (e.g. 5 <= 3 or 3 == 4).

3-way Compare

```
exp:= exp '<=>' exp
```

The 3-way compare operator <=> compares 2 evaluated expression values A and B and returns an integer less than 0 if A < B, 0 if A == B and an integer greater than 0 if A > B.

Logical Operators

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

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

Logical operators in Squirrel are: &&, || and !

The operator && (logical and) returns `false` if its first argument is `false`, otherwise returns its second argument.

The operator || (logical or) returns its first argument if it is different than `false`, otherwise returns the second argument.

The ! (logical not) operator will return `false` if the given value to negate was true, or `true` if the given value was false.

in Operator

```
exp:= keyexp 'in' tableexp
```

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

```
var t =
{
      foo = "I'm foo",
```

```
    [123] = "I'm bar"
}

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

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

instanceof Operator

```
exp := instanceexp 'instanceof' classexp
```

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

typeof Operator

```
exp := 'typeof' exp
```

Returns the type name of a value as a string.

```
var 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.

```
var j = 0, k = 0;

for ( var i = 0; i < 10; i++, j++ )
{
    k = i + j;
}

var a, k;

a = ( k=1, k + 2 ); // a becomes 3
```

Bitwise Operators

```
exp := exp op exp

exp := '~' exp
```

Squirrel supports the standard C-like bitwise operators &, |, ^, ~, << and >>.

Also included is the unsigned right shift operator >>>. This operator 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.

Bitwise operators only work on integer values, passing of any other operand type will cause an exception.

Operator Precedence

Operators are evaluated in the order of precedence below from highest to lowest. Operators on the same line are evaluated from left to right.

`-`, `~`, `!`, `typeof`, `++`, `--`
`/`, `*`, `%`
`+`, `-`
`<<`, `>>`, `>>>`
`<`, `<=`, `>`, `>=`
`==`, `!=`, `<=>`
`&`
`^`
`|`
`&&`, `in`
`||`
`?:`
`+=`, `=`, `-=`
`,` (comma operator)

Table Constructor

```
tslots := ( id '=' exp | '[' exp ']' '=' exp ) [',']

exp := '{' [tslots] '}'
```

Creates a new table.

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

A table constructor can also contain slot declarations; With the syntax:

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

A new slot with `id` as key and `exp` as value is created.

```
var 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.

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

Both syntaxes can be mixed.

```
var table =
{
	a = 10,
	b = "string",
	[10] = {},

	function bau( a, b )
	{
		return a+b;
	}
}
```

The comma between slots is optional.

Table with JSON syntax

It is also possible to declare a table using JSON syntax.

The following JSON snippet:

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

Is equivalent to the following standard Squirrel table:

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

Array Constructor

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

Creates a new array.

```
a <- [] // creates an empty array and assigns to a new slot.
```

Arrays can be initialized with values during construction.

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

clone

```
exp:= ‘clone’ exp
```

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

After the new object is ready the “_cloned” meta-method is called (see Meta-methods).

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

4.7 Tables

Tables are associative containers implemented as key/value pairs (called slots); 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 the *root table*).

Construction

Tables are created using the table constructor (see Table Constructor above).

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 exist it will be created.

```
var a = {}
```

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

```
a.newslot = 1234
```

However, this statement will succeed:

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

As will

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

Which creates a new key `1` (integer) in the table 'a'.

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 ); // will print the string "now"
```

4.8 Arrays

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

```
var a = [ “I’m a string”, 123 ]

print( typeof a[0] ) // prints "string"
print( typeof a[1] ) // prints "integer"
```

Resizing, inserting and deleting of arrays and array elements is done through a set of standard functions (see the *Squirrel API* chapter).

4.9 Functions

Functions are first class values like integers or strings and can be stored in table slots, local variables, arrays and passed as function parameters.

Function Declaration

Functions are declared through the function expression:

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

or with this 'syntactic sugar':

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

which is equivalent to:

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

Local Functions

A local function can be declared with this syntactic sugar

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

which is equivalent to:

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

Table Functions

It is also possible to append a function to a table, after its initial declaration using the following syntax:

```
T <- {}

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

which is equivalent to writing:

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

// or

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

Default Parameters

Squirrel's functions can have default parameters. A function with default parameters is declared as follows:

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

When the function `test` is called and the parameter `c` or `d` aren't specified, the virtual machine automatically assigns the default value to the unspecified parameter. A default parameter can be any valid Squirrel expression (the expression is evaluated at runtime).

Variadic Functions

Squirrel's functions can have variable number of parameters. A variadic function is declared by adding three dots ('`...`') at the end of its parameter list.

When the function is called any extra parameters will be accessible through the array called `vargs` which is passed as an implicit parameter.

`vargs` is a regular Squirrel array and can be used accordingly.

```
function test( a, b, ... )
{
```

```
        for ( var i = 0; i < vargs.Length(); ++i )
        {
            ::print( "varg[" + i + "=" + vargs[ i ] + "\n" );
        }

        foreach( i, val in vargs )
        {
            ::print( "varg[" + i + " = " + val + "\n" );
        }
    }

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

Function Calls

```
exp := derefexp '(' explist ')'
```

Every function call in Squirrel passes the environment object `this` as an implicit parameter to the called function. The '`this`' parameter is the object (table, class, etc.) that the function was indexed from.

If we call a function with this syntax:

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

The environment object `this` passed to `foo` will be '`table`'

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

The environment object will be '`this`' (the same of the caller function).

Binding an Environment to a Call

By default a Squirrel function call passes as environment object '`this`', equal to the object the function was indexed from. However, it is also possible to statically bind an environment to a closure using the built-in method `closure.Bind( env_obj )`. This method `Bind` returns a new instance of `closure` with the given environment bound to it.

When an environment object is bound to a function, every time the function is invoked, its 'this' parameter will always be the previously bound environment. This mechanism is useful to implement callback systems similar to C# delegates.

Note: The closure keeps a weak reference to the bound environment object, because of this if the object is deleted, the next call to the closure will result in a `null` environment object.

Lambda Expressions

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

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

Here is an example lambda expression:

```
var mylambda = @( a, b ) a + b;
```

This is equivalent to:

```
var mylambda = function( a, b ) { return a + b; }
```

A more useful usage could be:

```
var sequence = [ 2, 3, 5, 8, 3, 5, 1, 2, 6 ];

sequence.Sort( @(a,b) a <=> b );
sequence.Sort( @(a,b) -(a <=> b) );
```

That example could have been written as:

```
var sequence = [ 2, 3, 5, 8, 3, 5, 1, 2, 6 ];

sequence.Sort( function( a, b ) { return a <=> b; } );
sequence.Sort( function( a, b ) { return -(a <=> b); } );
```

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

Free Variables

Free variables are simply local variables referenced from an outer scope. In the following example the variables `testy`, `x` and `y` are accessible by the function `foo`. Note: variables `a` and `b` are not accessible inside `foo` as they become shadowed by the function parameters.

```
var x = 10, y = 20, testy = “I’m testy”;

var a = 10, b = 15;

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

A function can both read from and write to free variables.

Tail Recursion

Tail recursion is a method for partially transforming a recursion in a program into an iteration: it applies when the recursive call in a function is the last executed statement in that function (just before the `return`).

If this happens the Squirrel interpreter collapses the caller stack frame before the recursive call; this feature allows for very deep recursions without the risk of a stack overflow.

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

loopy( 1000 );
```

4.10 Classes

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

Class Declaration

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

For example:

```
class Foo
{
	// constructor
	constructor( a )
	{
		testy = [ "stuff", 1, 2, 3, a ];
	};

	// member function
	function PrintTesty()
	{
		foreach( i, val in testy ) {
			::print( "idx = " + i + " = " + val + " \n" );
		}
```

```
        };

        // property
        testy = null;

    }
```

The previous code example is syntactic sugar for:

```
    Foo <- class
    {
        // constructor
        constructor( a )
        {
            testy = [ "stuff", 1, 2, 3, a ];
        };

        // member function
        function PrintTesty()
        {
            foreach( i, val in testy ) {
                ::print( "idx = " + i + " = " + val + " \n" );
            }
        };

        // property
        testy = null;

    };
```

Namespace Emulation

In order to emulate namespaces, it is also possible to declare something like this:

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

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

        function DoSomething()
        {
            ::print( "DoSomething()" );
        };
```

```
};

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

var testy = FakeNamespace.Utils.SuperClass();

testy.DoSomething();

testy.DoSomethingElse();
```

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

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

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

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

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

Static Member Variables

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

```
class Foo
{
	constructor() {
		//..stuff
	}

	name = "normal variable";

	// static variable
	static classname = "The class name is foo";

};
```

Class Attributes

Classes allow 'attributes' to be associated to it's members. Attributes are a form of arbitrary meta-data that can be used to store application specific information. Attributes can be used to store documentation strings, properties for IDEs, code generators, etc.

Class attributes are declared in the class body by preceding the member declaration and are delimited by the symbols </ and />. Here is an example:

```
class Foo </ test = "I'm a class level attribute" />
{

	</ atr = "I'm an attribute, baby!" /> // attributes of PrintTesty
	function PrintTesty()
	{
		foreach( i, val in testy ) {
			::print( "idx = " + i + " = " + val + " \n" );
		}
	}

	</ flippy = 10, second = [1,2,3] /> // attributes of testy
	testy = null;

}
```

Attributes are declared and stored as a table. Squirrel uses </ /> syntax instead of curly brackets { } for the attribute declaration to increase readability. This means that all rules that apply to tables apply to attributes.

Attributes can be retrieved through the built-in function `classobj.GetAttributes( membername )` (see the *Squirrel API* chapter for more information) and individual attributes can be modified / added through the built-in function `classobj.SetAttribute( membername, value )`.

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

```
foreach( member, val in Foo )
{
	::print( member + "\n" );

	var attr;

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

```
        }
}
```

Class Instances

Class objects inherit several features from tables but also differ in they allow the creation of multiple independent instances of the class to be created. A class instance is an object that shares the same structure of the table that created it but holds is own values.

Class instantiation uses function notation. A class instance is created by calling a class object. It can be useful to imagine a class like a function that returns a class instance.

```
// Create a new instance of Foo
var fooInst = Foo();
```

When a class instance is created its members are initialized using the same values specified in the class declaration. Values are copied verbatim, no cloning is performed even if the value is a container (i.e. an array or table) or another class instance.

It is important to note that Squirrel doesn't clone a member's default values nor execute the member declaration for each instace (as in C# or Java). Consider this example:

```
class Foo {

        myarray = [1,2,3];
        mytable = {};

};

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

In the snippet above both instances will refer to the same array and table object. Any modification to one instance will appear as a change in the other. To achieve what a C# or Java programmer would expect, the following approach should be taken.

```
class Foo {

        myarray = null;
        mytable = null;

        constructor() {
                myarray = [1,2,3];
                mytable = {};
        };
};

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

Constructors

When a class defines a method called '`constructor`', the class instantiation operation will automatically call this function for each newly created instance.

The constructor method can have parameters, this will impact on the number of parameters that the instantiation operation will require. Constructors, as normal functions, can have variable number of parameters (using the variadic parameter '`...`').

```
class Rect {

	constructor( w, h ) {
		width = w;
		height = h;
	};

	x = 0;
	y = 0;

	width = null;
	height = null;
}

// Rect's constructor has 2 parameters so the class has to be
// 'called' with 2 parameters.

var rc = Rect( 100, 100 );
```

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

The class object that created an instance can be retrieved through the built-in function `instance.GetClass()` (see the Squirrel API chapter for more information).

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

```
var rc = Rect( 100, 100 );

if ( rc instanceof ::Rect ) {
	::print( "It's a Rect" );
} else {
	::print("It's not a Rect");
}
```

Inheritance

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

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

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

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

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

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

Since the constructor is a regular function (apart from being automatically invoked on instantiation of the class), the same rules apply.

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

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

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

```
var theFooClass = SuperFoo.GetBase();
```

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

Note that methods do not have a special protection policy when calling methods of the same object, a method of a base class that calls a method of the same class can end up calling a method overridden by the derived class instead.

```
class Foo {

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

	function DoIt() {
		DoSomething();
	}
};

class SuperFoo extends Foo {

	// overridden method
	function DoSomething() {
		::print("I'm the derived class");
	}

	function DoIt() {
		base.DoIt();
	}
};

// Create a new instance of Foo
var fooInst = Foo();

fooInst.DoIt(); // prints "I'm the base class"

// Create a new instance of SuperFoo
var superInst = SuperFoo();

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

Instance Meta-methods

Squirrel allows the customization of certain aspects of class semantics through the use of meta-methods. Meta-methods are the Squirrel equivalent of operator overloading features found in other languages. See the *Meta-methods* section below for more information.

The meta-methods supported by class instances are:

```
_add            _sub            _mul            _div

_unm            _modulo         _set            _get

_typeof         _nexti          _cmp            _call

_delslot        _tostring
```

The following example show how to create a class that implements the meta-method `_add`.

```
class Vector3 {

     constructor( ... ) {
          if ( vargs.Length() >= 3 ) {
               x = vargs[ 0 ];
               y = vargs[ 1 ];
               z = vargv[ 2 ];
          }
     }

     function _add( other ) {
          return ::Vector3( x + other.x, y + other.y, z + other.z );
     }

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

var v0 = Vector3( 1, 2, 3 );
var v1 = Vector3( 11, 12, 13 );
var v2 = v0 + v1;

::print( v2.x + "," + v2.y + "," + v2.z + "\n" );
```

Class Meta-methods

Class objects support two meta-methods:

```
_newmember      _inherited
```

`_inherited` is invoked when a class inherits from the one that implements `_inherited`.

`_newmember` is invoked for each member that is added to the class (at declaration time).

4.11 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 execution of a generator and optionally returns the result of an expression to the function that resumed it.

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.

```
function geny( n )
{
      for ( var i = 0; i < n; ++i ) {
            yield i;
      }

      return null;
}

var gtor = geny( 10 );
var x;

while ( x = result gtor ) {
      ::print( x + ”\n” );
}
```

The output of this program will be:

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

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

Note: A generator holds strong references to all the values stored in it's local variables except the `this` object which is held as a weak reference. A running generator also holds a strong reference to the `this` object.

4.12 Constants

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

Constant values can only be integers, floats or string literals. No expressions are allowed. They are declared with the following syntax:

```
const foobar = 100;

const floatbar = 1.2;

const stringbar = "I'm a constant string";
```

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

```
var x = foobar * 2;
```

4.13 Enumerations

As with constants, enumerations bind a specific value to an identifier. Enumerations are also evaluated at compile time and their value cannot be changed.

An `enum` declaration introduces a new named enumeration into the program.

```
eslots := id [ '=' IntegerLiteral|FloatLiteral| ] [',']

exp := 'enum' id '{' [eslots] '}'
```

Enumeration values can only be integers, floats or string literals. No expressions are allowed.

If a value is not provided an integer will be generated automatically either starting from `0` (zero) or equal to the previous automatically generated value + 1. Manually specified values have no effect on automatic values.

```
enum FirstExample
{
      a, // this will be 0
      b = 3, // this will be 3
      c // this will be 1
};

enum SecondExample
{
      first = 10,
      second = "string",
      third = 1.2
};
```

An enumeration value is accessed using both of the name of the enum and the required identifier. This is similar to accessing a static class member. e.g. `FirstExample.b`.

Enumerations are always globally scoped, from the moment they are declared, any following code can reference them. Enumerations will shadow any global slot with the same name (the global slot remains available by using the `::` prefix syntax).

4.14 Weak References

A weak reference allows the programmer to a create reference to an objects without influencing the lifetime of the object itself.

In Squirrel weak references are first-class objects created through the built-in method `obj.WeakRef()`. All types except `null` implement the `WeakRef()` method; however in the value types bool, integer and float the method simply returns the object itself.

When a weak reference is assigned to a container slot (table slot, array element, class or instance member) it is treated differently from other objects. When a container slot that holds a weak reference is fetched, it always returns the value pointed by the weak reference instead of the weak reference object. This behaviour allow the programmer to ignore the fact that the value is wrapped inside a weak reference.

When the object pointed by weak reference is destroyed, the weak reference is automatically set to null.

```
var t = {}

var a = [ "first", "second", "third" ];

// Create a weak array reference and assign it to a table slot.

t.thearray <- a.WeakRef();
```

Table slot `thearray` now contains a weak reference to an array. The following line prints "first", because tables (as with other container types) always return the object pointed by a weak reference.

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

The only strong reference to the array is owned by the local variable `a`. Changing the value of a (for example assigning an integer to it) removes the strong reference to the array and causes it to be destroyed.

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

```
a = 123;

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

Explicitly Handling Weak References

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

```
local t = {}

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

The following line prints "`weakref`".

```
print( typeof( weakobj ) );
```

The object pointed by the weak reference can be obtained through the built-in method `Ref()`. For example, the following line prints "`table`".

```
print( typeof( weakobj.Ref() ) )
```

4.15 Delegation

Every table can have a parent table or delegate. A parent table is a normal table that allows the definition of special behaviors for its child. When a table 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 );
}

var newentity = {
      _name="I'm the new entity"
};

newentity.SetDelegate( Entity )

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

The delegate of a table can be retrieved through built-in method `table.GetDelegate()`.

```
var thedelegate = newentity.GetDelegate();
```

4.16 Meta-methods

Meta-methods are a mechanism that allows the customization of certain language semantics. They are similar to the operator overloading features found in other languages.

Meta-methods are functions typically placed in a table delegate or class declaration. It is possible to change many behaviors using the different meta-methods.

Use Case Example

When using a relational operator (except '==') on two tables, the VM will check if the table has a method (or one in its delegate parent) called `_cmp`. If so it calls it to determine the relation between the tables.

```
var comparable = {

	_cmp = function( other )
	{
		if ( name < other.name ) {
			return -1;
		} else if ( name > other.name ) {
			return 1;
		} else {
			return 0;
		}
	}
};

var a = { name = "Alberto" }.SetDelegate( comparable );
var b = { name = "Wouter" }.SetDelegate( comparable );

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

For classes the previous code becomes:

```
class Comparable {

	constructor( n ) {
		name = n;
	}

	function _cmp( other )
	{
		if ( name < other.name ) {
			return -1;
		} else if ( name > other.name ) {
```

```
                return 1;
            } else {
                return 0;
            }
        }

        name = null;

    }

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

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

Supported Meta-methods

_set

```
function _set( key, val ) // return val
```

Invoked when a key is not present in the object or in its delegate chain.

_set should '`throw null;`' to notify that a key wasn't found but there were no runtime errors (clean failure). This allows the program to differentiate between a runtime error and an 'index not found' condition.

_get

```
function _get( key ) // return fetched values
```

Invoked when the key is not present in the object or in its delegate chain.

_get should '`throw null;`' to notify that a key wasn't found but there were no runtime errors (clean failure). This allows the program to differentiate between a runtime error and an 'index not found' condition.

_newslot

```
function _newslot( key, value ) // return value
```

Invoked when a script tries to add a new slot to a table. If the slot already exists in the target table the method will not be invoked.

_delslot

```
function _delslot( key )
```

Invoked when a script deletes a slot from a table. if the method is invoked Squirrel will not automatically delete the slot itself, you must instead use the `delete` operator manually.

_add

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

Overloads the add ('+') operator with user code.

_sub

```
function _sub( op ) // return this - op
```

Overloads the subtract ('-') operator with user code.

_mul

```
function _mul( op ) // return this * op
```

Overloads the multiply ('*') operator with user code.

_div

```
function _div( op ) // return this / op
```

Overloads the multiply ('/') operator with user code.

_mod

```
function _mod( op ) // return this % op
```

Overloads the modulo ('%') operator with user code.

_unm

```
function _unm( op ) // return -this
```

Overloads the unary minus operator ('-') with user code.

_typeof

```
function _typeof() // return the type of this as a string.
```

Invoked by the typeof operator on tables and class instances.

_cmp

```
function _cmp( other ) // returns an integer:

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

Invoked to emulate <, >, <= and >= operators.

_call

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

Invoked when a table or class instance is "called" like a function.

_cloned

```
function _cloned( original )
```

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

_nexti

```
function _nexti( previdx )
```

Invoked when a class instance is iterated by a `foreach` loop. If previdx == `null` it means that it is the first iteration. The function must return the index of the ‘next’ value.

_tostring

```
function _tostring() // return a string representation of this
```

Invoked during string concatenation or when the `print` function prints a table or class instance.

_inherited

```
function _inherited( attributes )
```

Invoked when a class object inherits from a class implementing _inherited. The value of `this` contains the new class. The return value of this meta-method is ignored.

_newmember

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

Invoked for each member declared in a class body (at declaration time). if the function is implemented, members will not be added to the class.

Chapter 5 API Reference

Introduction

Grid Cartographer scripts interface with the main application through a set of API classes and global functions. This chapter describes the interface and includes information on what each function does and its parameters when calling from Squirrel.

Chapter Structure

This chapter is divided into three sections.

1. Built-in Squirrel data-type methods such as `array.Length()`

2. Top level global functions such as `print`.

3. The full set of system classes such as `Math` or `GCConsole`.

Chapter Conventions

Functions are documented in a standard format: Name, function specification and a description of how the function works.

The function specification includes a fully qualified name and arguments with expected data types and a return type (or `void` if the function does not return a value). Multiple argument data types separated by a vertical bar means any of the given types are supported and an asterisk means all types are supported.

5.1 Squirrel Data-type Methods

The built-in data types of the Squirrel language have supplementary methods which can be called on variables of that type. See below for the methods available for each type.

Boolean Values

ToFloat

`b.ToFloat(): float`

Convert to a float. Returns `1.0` for `true` and `0.0` for `false`.

ToInt

`b.ToInt(): int`

Convert to a integer. Returns `1` for `true` and `0` for `false`.

ToString

`b.ToString(): string`

Convert to a string. Returns `"true"` for `true` and `"false"` for `false`.

Integer Values

ToChar

`i.ToChar(): string`

Convert the integer into a single character string, interpreting the value as an ASCII code.

ToFloat

`i.ToFloat(): float`

Convert the integer into a float and return it.

ToString

`i.ToString(): string`

Convert the integer into a string representation of the number.

Float Values

ToChar

`f.ToChar(): string`

Convert the integer part of the value into a single character string, interpreting it as an ASCII code.

ToInt

`f.ToInt(): int`

Convert to an integer and returns it. Any fractional part to the value is truncated.

ToString

`f.ToString(): string`

Convert to a string and return it.

String Values

ChatAt

`str.CharAt( index: int ): int`

Returns the character in the string at the given `index` as an integer. Use `ToChar` to turn into a string value.

Find

`str.Find( substr: string, [start: int] ): int|null`

Search through the string for the given `substr` value. If the `start` index is provided then search begins at this index. If the `substr` value is found, the offset into the string is returned. If the sub-string is not found, the call will return `null`.

Length

`str.Length(): int`

Returns the length of the string in characters.

Slice

`str.Slice( start: int, [end: int] ): string`

Returns a section of the string starting at `start` and continuing until the character before `end`. If `end` is omitted the slice will contain the whole of the remainder of the string. If `start` is negative the index is calculated as `length + start`. If `end` is negative the index is calculated as `length + end`.

Split

str.Split(delim: string): array

Splits the string into an array of sub-strings whenever the given delimiting string found in the original. The delimiter and resulting empty sub-strings are not included in the output.

For example:

```
var message = "HELLO WORLD";
var arr = message.Split( "L" );

printobj( arr );

// output is:
//    [
//          0 : "HE"
//          1 : "O WOR"
//          2 : "D"
//    ]
```

ToFloat

`str.ToFloat(): float`

Convert the string to a float. Stops at the first non-digit. If no digits are found, returns `0` (zero).

ToInt

`str.ToInt(): int`

Convert the string to an integer. Stops at the first non-digit. If no digits are found, returns `0` (zero).

ToLower

`str.ToLower(): string`

Returns a lower-case copy of the string. The original string is not changed.

ToUpper

`str.ToUpper(): string`

Returns an upper-case copy of the string. The original string is not changed.

Trim

`str.Trim(): string`

Returns a copy of the string with all of the white-space characters removed from the beginning and the end. Trimming stops as soon as the first non-whitespace character is found.

White-space characters are:

Character	Hex Value	Name
' '	0x20	Space
\t	0x09	Horizontal Tab
\n	0x0a	New-line
\v	0x0b	Vertical Tab
\f	0x0c	Form-feed
\r	0x0d	Carriage return

TrimLeft

`str.TrimLeft(): string`

Returns a copy of the string with all white-space characters (see above) removed from the beginning of the string only.

TrimRight

`str.TrimRight(): string`

Returns a copy of the string with all white-space characters (see above) removed from the end of the string only.

Array Values

Add

```
arr.Add( val: * ): void
```

Appends the given value to the end of the array.

Apply

```
arr.Apply( fn: function ): void
```

For each element of the array, call `fn(element)` and replace the original element with the return value of the function.

Clear

```
arr.Clear(): void
```

Remove all elements from the array.

Filter

```
arr.Filter( passFunc: function ): array
```

Creates a new array including only those elements which are permitted by the `passFunc` filter function. For each element `passFunc(index,value)` is called, if it returns true the value is included in the new array. Otherwise the element is omitted.

Find

```
arr.Find( value ): int|null
```

Performs a linear search through the array for the given value. If found, returns the index of the element, otherwise it returns null.

InsertAt

```
arr.InsertAt( index: int, val: * ): void
```

Insert a value at the given index in the array.

Join

```
arr.Join( other: array ): void
```

Add all elements from another array onto the end of this one.

Length

`arr.Length(): int`

Returns the number of elements in the array.

Map

`arr.map( fn: function ): array`

Creates a new array of the same length as this array. Each value in this array is passed as a parameter to function fn and the returns value is assigned to the element in the new array.

Pop

`arr.Pop(): *`

Remove an element from the end of the array and return it.

Push

`arr.Push( val: * ): void`

Appends the given value to the end of the array. Just like `Add` does.

Reduce

`arr.Reduce( fn: function ): *`

Reduce an array down to a single value. For each element in the array, invoke the function `fn(previous, current)` where `previous` is the result of the previous call and `current` is the current element being operated on. For a zero length array, `Reduce` returns null. For length 1, it simply returns the first element.

For arrays of length 2 or more, the function is called with the first two elements as initial parameters. The next iteration then uses the result of that function call, plus the value of the third element as the parameters to `fn`, and so on. The final result is returned to the caller of `Reduce`.

RemoveAt

`arr.RemoveAt( index: int ): void`

Remove the value in the array at the given index.

Reverse

`arr.Reverse(): void`

Reverse all elements of the array in-place.

Slice

`arr.Slice( start: int, [end: int] ): string`

Returns a sub-section of the rray starting at `start` and continuing until the element before `end`. If `end` is omitted the slice will contain the whole of the remainder of the array. If `start` is negative the index is calculated as `length + start`. If `end` is negative the index is calculated as `length + end`.

Sort

`arr.Sort( [compare: function] ): void`

Sort the array. If using the optional compare function it must confirm to the prototype:

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

Alternatively a more compact implementation would be to use the <=> operator and a lambda expression:

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

Table Values

Clear

`tbl.Clear(): void`

Removes all slots from the table.

Length

`tbl.Length(): int`

Returns the number of slots in (the top level of) the table.

GetDelegate

`tbl.GetDelegate(): table`

Returns the table's delegate, or `null` if no delegate was set.

SetDelegate

`tbl.SetDelegate( deleg: table ): void`

Assigns a delegate to the table. Pass in `null` to remove the delegate. See the *Table* section in the language specification for more information.

5.2 Global Functions

These global functions are provided as convenience and to aid script readability.

GCLocale

```
GCLocale( string_id: string ): string
```

Perform a lookup into the localization table using the given string id.

getchar

```
getchar(): int
```

Try and read a character from the console. This is equivalent to `GCConsole.GetChar`. If no input is available, the call will wait until there is (or the user presses Ctrl+C to abort).

hash

```
hash( data: string ): void
```

Computes the CRC-32 hash of the given string. For static strings this function can be replaced by a compile-time `#"` prefix string literal for improved runtime performance.

include

```
include( file: string ): void
```

Load, compile and execute a script file into the virtual machine. This is a similar function to the `<load>` element in a hook file but can be called selectively.

The file is searched for, in order:

- Relative to the hook file of the script (if present)
- Relative to the script itself if it's a command launched from the *File Store* folder.
- In the *User Scripts* folder.
- In the internal scripts folder (`base0.zip/scripts/`).

Warning: If the file is not found, the script will exit immediately.

If the script causes a timeout in the Virtual Machine (for example getting caught in an infinite loop) the script will exit automatically rather than giving the user the choice. If your script triggers a false positive then consider wrapping the code in a function and calling it from the parent script after the `include` call has completed.

print

`print( message: string|int|float ): void`

Prints a text string or number to the console from the current cursor position. Special 'escape characters' found in the string are processed. See `GCConsole.Print` for more information.

println

`println( message: string|int|float ): void`

Prints a text string or number to the console from the current cursor position. This variation of `print` also automatically outputs a newline character immediately after the text.

printobj

`printobj( object: * ): void`

Print a complex object such as an array or table to the console from the current cursor position. See `GCConsole.PrintObj` for more information.

putchar

`putchar( ch: int ): void`

Write a single character to the console at the current cursor position. The `ch` parameter supports values from `0` to `255`. Special character processing is also performed, see `GCConsole.Print` for more information.

5.3 GCBuffer

A set of functions to create and manage generic in-memory data buffers. Buffers can be used to cache data when reading from or writing to disk to improve performance.

Create

`GCBuffer.Create(): int`

Create a new empty buffer and return a handle to it. Returns the handle or `0` (zero) if there was a failure (too many buffers, internal error, etc.)

Destroy

`GCBuffer.Destroy( handle: int ): void`

Destroy a buffer referenced by the given handle. The handle is internally marked as invalid and cannot be used for future operations.

Length

`GCBuffer.Length( handle: int ): int`

Return the size in bytes of the data in the buffer referenced by `handle`. If the handle is invalid the function returns `GCERR_INVARG`.

Prepare

`GCBuffer.Prepare( handle: int, size: int ): void`

Prepare the buffer referenced by `handle` to receive `size` bytes of data. This is an optimization function that affects the storage used to hold the data.

When data is written to the buffer it automatically expands into space provided by the 'backing storage' a new size. up to the size of the backing storage. by allocating a new larger block of memory. This expansion involves copying the current data to the new location which may reduce performance. Prepare instructs the buffer to expand

If the size given is insufficient to hold the data in the buffer, the call will do nothing without error.

Print

`GCBuffer.Print( handle: int, message: string|int|float ): int`

Write a string, integer or float value as plain ASCII text to the buffer. Returns `GCOK` on success and automatically advances the read/write cursor to the end of the message. If the handle is invalid it returns `GCERR_INVARG`.

Read8

`GCBuffer.Read8( handle: int ): int|null`

Read an unsigned 8-bit binary value from the file buffer at the current cursor position. If the handle is invalid, or there is insufficient data in the buffer, returns `null`.

Read16

`GCBuffer.Read16( handle: int ): int|null`

Read an unsigned 16-bit binary value from the file buffer at the current cursor position. If the handle is invalid, or there is insufficient data in the buffer, returns `null`.

Read32

`GCBuffer.Read32( handle: int ): int|null`

Read a signed 32-bit value from the file buffer at the current cursor position. If the handle is invalid, or there is insufficient data in the buffer, returns `null`.

Reset

`GCBuffer.Reset( handle: int ): void`

Reset the contents of the buffer referenced by `handle` to `0` (zero) length. Also resets the read/write cursor to `0` (zero). This call will preserve the backing storage allocated so future writes to the buffer will be a faster.

Seek

`GCBuffer.Seek( handle: int, where: int ): int`

Change the position of the read/write cursor to the byte offset specified by `where`. If the handle is invalid the function returns `GCERR_INVARG`.

If the position is beyond the start or end of the data in the buffer, the cursor will be set to the first or last position respectively and the function will return `GCERR_EOF`. Otherwise the cursor is updated normally and the function returns `GCOK`.

Where

`GCBuffer.Where( handle: int ): int`

Return the position of the read/write cursor for the buffer referenced by `handle`. If the handle is invalid the function will return `GCERR_INVARG`.

Write8

`GCBuffer.Write8( handle: int, data: int ): int`

Write an unsigned 8-bit binary value (1 byte) to the given buffer at the read/write cursor (the `data` parameter is converted to an unsigned integer and then the bottom 8 bits are used). If `handle` is invalid the function will return `GCERR_INVARG`. Otherwise `GCOK`.

Write16

`GCBuffer.Write16( handle: int, data: int ): int`

Write an unsigned 16-bit binary value (2 bytes) to the given buffer at the read/write cursor (the `data` parameter is converted to an unsigned integer and then the bottom 16 bits are used). If `handle` is invalid the function will return `GCERR_INVARG`. Otherwise `GCOK`.

Write32

`GCBuffer.Write32( handle: int, data: int ): int`

Write a 32-bit binary value (4 bytes) to the given buffer at the read/write cursor. If `handle` is invalid the function will return `GCERR_INVARG`. Otherwise `GCOK`.

5.4 GCConsole

These functions allow control of the Grid Cartographer console window.

Clear

`GCConsole.Clear(): void`

Clears the entire console window and resets the cursor to the top-left `(0,0)`.

Close

`GCConsole.Close(): void`

Close the console window. Don't use this unless the user has specifically requested the action.

Column

`GCConsole.Column(): int`

Returns the current column of the text cursor.

GetChar

`GCConsole.GetChar(): int`

Try to read a single key press from the console input buffer. If no key has been pressed, this call will block until one is (see `PeekChar` for a non-blocking variant). Once an input has been read, it is automatically echoed back to the console window.

Special function keys are also detected and also returned as values mapped to entries in the *Key Constant* table (defined below).

Locate

`GCConsole.Locate( x: int, y: int ): void`

Adjust the position of the cursor to the given `(x,y)` co-ordinates. Position `(0,0)` is the top-left of the console display. If the given position is outside of the size of the console, the call is ignored and the cursor will remain where it was.

Open

`GCConsole.Open(): void`

Opens the console window. Don't use this unless the output is important.

PeekChar

`GCConsole.PeekChar(): int`

Try to read a single character from the console input buffer. If no input has been made, this call will return `0` (zero) and continue immediately (unlike `GetChar`). The input isn't echoed to the console (again, unlike `GetChar`). Special keys are also detected and returned as codes. See `GetChar` for more information.

Print

`GCConsole.Print( text: string|int|float ): void|int`

Writes the given text string or number to the console from the current cursor position. Returns **`-1`** if `CTRL+C` has been pressed.

If one of the following escape sequences is encountered in the string, then a special action will be taken:

Character	Hex Value	Meaning
`\b`	`0x08`	**Nondestructive Backspace**. Move the cursor left by one character. If the cursor is in the left-most column (and not on the top row) it will wrap around to the end of the previous line.
`\t`	`0x09`	**Horizontal Tab**. Moves the cursor to the next multiple of 8 characters. If this is already the case, the cursor will move a full 8 characters. If the cursor moves beyond the edge of the console width the cursor will advance to the next line, possibly scrolling the screen up.
`\n`	`0x0A`	**New Line**. Return the cursor to the left-most column and move down by one. If the cursor moves beyond the bottom of the console, all text will be moved up to accommodate the new line.
`\r`	`0x0D`	**Carriage Return**. Return the cursor to the left-most column of the line.

PrintObj

`GCConsole.PrintObj( object: * ): void`

Write a generic object to the console from the current cursor position, followed by a new-line character. This function handles complex data types such as arrays, tables and classes.

PutChar

`GCConsole.PutChar( ch: int ): void|int`

Write a single character to the console at the current cursor position. The `ch` parameter supports values from `0` to `255`. Special character processing is also performed, see `Print` for more information. Returns **`-1`** if `CTRL+C` is pressed.

Row

`GCConsole.Row(): int`

Returns the row position of the cursor. Use `Locate` to move it to a new position.

SetCaption

`GCConsole.SetCaption( text: string ): void`

Sets the title caption of the console window (displayed at the top during execution of the script) to the given text string. By default the caption is set to "`Squirrel Script`".

SetCursorHeight

`GCConsole.SetCursorHeight( height: int ): void`

Sets the height of the blinking text cursor. Valid `height` values range from `0` (off) to `12` (full height). By default the cursor height is `2`. The default value is restored when a script ends.

5.5 GCDOS

Grid Cartographer scripts have access to a virtual file store for permanent storage use. Refer to the *File Store* chapter for more information.

Copy

`GCDOS.Copy( srcpath: string: dstpath: string ): int`

Make a copy of a file. Returns `GCOK` if the operation is successful. On failure returns one of: `GCERR_INVARG`, `GCERR_NODRIVE`, `GCERR_NOENT`, `GCERR_EXISTS` or `GCERR_FAIL`.

Delete

`GCDOS.Delete( filepath: string ): int`

Deletes a file from the given path. Returns `GCOK` on success or a standard error code. One of: `GCERR_INVARG`, `GCERR_NODRIVE`, `GCERR_NOENT` or `GCERR_FAIL`.

Dir

`GCDOS.Dir( filterpath: string ): table|int`

Parses the given filter path and returns the corresponding directory listing. Returns a standard error code on failure. One of: `GCERR_INVARG` or `GCERR_NODRIVE`.

Otherwise returns the following table:

Key	Value
`drive: int`	The drive that this directory is listing represented by an ASCII value from 'A' (65) to 'Z' (90). Use `drive.ToChar()` to convert to a string.
`entries: array<table>`	An array of file entries. See below for the entry table structure.

File entry values consist of the following table slots:

Key	Value
`name: string`	The name of the file, including extension.
`size: int`	The size of the file in bytes, up to 2GB.

GetDrive

```
GCDOS.GetDrive(): int
```

Returns the current drive letter as an integer character. Use `ToChar` to convert to a string.

Move

```
GCDOS.Move( oldpath: string: newpath: string ): int
```

Move a file from one drive to another and/or rename a file. Returns `GCOK` if the operation is successful. On failure returns one of: `GCERR_INVARG`, `GCERR_NODRIVE`, `GCERR_NOENT`, `GCERR_EXISTS` or `GCERR_FAIL`.

Rename

```
GCDOS.Rename( oldpath: string: newname: string ): int
```

Rename a file without permitting moving it. Returns `GCOK` if the operation is successful. On failure returns one of: `GCERR_INVARG`, `GCERR_INVDST`, `GCERR_NODRIVE`, `GCERR_NOENT`, `GCERR_EXISTS` or `GCERR_FAIL`.

SetDrive

```
GCDOS.SetDrive( letter: int ): int
```

Change the GCDOS default drive to any capital letter `A - Z`. If successful returns `GCOK`, otherwise `GCERR_INVARG` or `GCERR_NODRIVE`. Note: do not assume that the lack of a `GCERR_NODRIVE` response means that the drive will continue to exist in the future.

Size

```
GCDOS.Size( filepath: string ): int
```

Gets the size of a file in bytes. If the file does not exist, returns a standard error code. One of: `GCERR_INVARG`, `GCERR_NODRIVE` or `GCERR_NOENT`.

5.6 GCExport

In order to write data outside of the file store, to an arbitrary location on the users' computer, requires the use of an export function. The intention is to restrict the ability without explicit user consent, typically by displaying a file selector.

BufferAs

```
GCExport.BufferAs( bufferHandle: int,
                   caption: string|null,
                   defaultName: string|null,
                   fileExt: string|null,
                   fileExtDesc: string|null ): int
```

Attempts to export the buffer referenced by `bufferHandle` to an arbitrary location selected by the user using a standard file selector.

The selector title is configured using the `caption` parameter. A default file name can be given as a string (including extension), or omitted by passing `null`. A file type filter for the export can be defined using `fileExt` (e.g. `"TXT"`) and `fileExtDesc` (e.g. `"Text Files"`). To offer only a generic 'all files' option, set either of these values to `null`.

If the file is saved correctly it returns `GCOK`. If the user chooses to cancel the file selector the function returns `GCERR_ABORT`. If there was an error writing the file returns `GCERR_ACCESS`. If the buffer handle is invalid it returns `GCERR_INVARG`.

5.7 GCFile

This class provides functions for reading and writing individual files. See the *File Store* chapter for information about path structure and the file naming requirements.

Close

`GCFile.Close( handle: int ): void`

Close the given file handle. File handles are created by `OpenRead`, `Open` and `Append`.

GetChar

`GCFile.GetChar( handle: int ): int`

Read a character from the given file handle. If the end of file has been reached returns `GCERR_EOF`, or the handle is invalid, the call will return `GCERR_INVARG`.

Length

`GCFile.Length( handle: int ): int`

Return the byte length of the file. If the handle is invalid, returns `GCERR_INVARG`.

Name

`GCFile.Name( handle: int ): string|null`

Returns the file name (including extension) of the file. If the handle is invalid, the call will return `null`.

OpenRead

`GCFile.OpenRead( filepath: string ): int`

Opens an existing file for reading only. If successful a non-zero handle is returned. If not the call returns `0` (zero). Failure can occur when trying to open a file that does not exist, or to open one that has already been opened (either writing mode).

When finished reading use the `Close` function. Any remaining open files will be automatically closed when the script ends.

OpenCreate

`GCFile.OpenCreate( filepath: string ): int`

Create a new file for reading and writing. If the file already exists then it will be truncated. If the function is successful a non-zero handle is returned. If not the call returns `0` (zero).

When finished reading use the `Close` function. Any remaining open files will be automatically closed when the script ends.

OpenAppend

```
GCFile.OpenAppend( filepath: string ): int
```

Opens an existing file for writing additional data. If the file doesn't exist then it will be created. If it does exist then it will be opened and the write cursor will be positioned at the end of the file.

Print

```
GCFile.Print( handle: int, message: string|int|float ): int
```

Write a string of text, and integer or a float to the given file handle. If successful returns `GCOK`. If the handle is invalid returns `GCERR_INVARG`. If the file has been opened in read-only mode it returns `GCERR_ACCESS`.

PutChar

```
GCFile.PutChar( handle: int, ch: int ): int
```

Writes a character to the given handle. Returns `-1` on failure. This can happen if the handle is invalid, or the file has been opened in read-only mode.

ReadBuffer

```
GCFile.ReadBuffer( fileHandle: int, bytes: int, bufferHandle: int ): int
```

Read the specified number of bytes from the given file handle and copy them into the buffer object referenced by `bufferHandle`. If the read was successful and all bytes were copied, the function returns `GCOK`. If there are insufficient bytes available, fewer bytes are read and the function returns `GCERR_EOF` and the read cursor will positioned at the end of the file.

If the `fileHandle`, bytes or `bufferHandle` values are invalid it returns `GCERR_INVARG`. If the file wasn't opened for reading, it returns `GCERR_ACCESS`.

Seek

```
GCFile.Seek( fileHandle: int, position: int ): int
```

Move the read/write cursor for the file specified by fileHandle to a new position. If the function is successful it returns `GCOK`. If the file handle is invalid the function returns `GCERR_INVARG`. If the `position` value is beyond the extents of the file the position is clamped to the beginning/end and the function returns `GCERR_EOF`.

WriteBuffer

`GCFile.WriteBuffer( fileHandle: int, bufferHandle: int ): int`

Write an entire buffer referenced by `bufferHandle` into the file referenced by `fileHandle`. If successful the function returns `GCOK`. If either handle is invalid the function returns `GCERR_INVARG` and if the file is opened for reading only, the function returns `GCERR_ACCESS`.

Where

`GCFile.Where( fileHandle: int ): int`

Return the current byte position of the read/write cursor for the file referenced by `filehandle`. If the file handle is invalid, returns `GCERR_INVARG`.

5.8 GCFloor

These functions allow interaction with individual floors in the active region. These functions use a single "active floor" as the implicit context for operations. Use `Select` and `Move` functions to change the index of the active floor. Note that when dealing with tile-maps a 'floor' is equivalent to a 'plane'. Note there is only one active floor for all regions.

WARNING: Unless the script is running in blocking mode there is the potential for the user to interact with the map and cause inconsistent results. Take care to guard your script from unwanted user interaction. See `GCKernel.SetBlockMode`.

Select

```
GCFloor.Select( index: int ): void
```

Select the index of the active floor. Use positive values for floors, negative values for basements and `0` (zero) for the ground floor. The default cursor position is the floor index of the current region when the script starts. Note: If you select a floor which does not exist the function will succeed but functions that need to access the floor will fail.

Active

```
GCFloor.Active(): int
```

Returns the current value of the floor cursor. This defaults to the floor index of the currently selected region when the script starts.

Exists

```
GCFloor.Exists( index: int ): bool
```

Returns `true` if the given floor index exists in the current active region, otherwise `false`.

FindBound

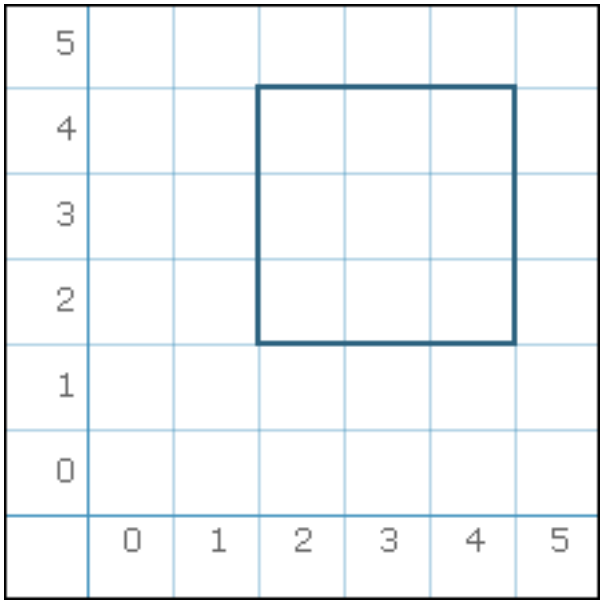
`GCFloor.FindBound(): table|null`

Compute the bounding box of tiles on the currently selected active floor. Note this is a potentially slow operation and may impact performance if it is called frequently.

If the active floor or active region don't exist, or the floor is empty, the function returns `null`. Otherwise it returns a table containing the following fields:

Key	Value
`min_x: int`	The minimum X position of the bounding rectangle.
`max_x: int`	The maximum X position.
`min_y: int`	The minimum Y position. This value is correct for both top-left and bottom-left origins.
`max_y: int`	The maximum Y position.
`edge_only_left: bool`	True if the left column of the bounding box contains only edges drawn on the right side of the tile. This allows the min_x value to be incremented to disregard these tiles.
`edge_only_top: bool`	True if the top row of the bounding box contains only edges drawn on the bottom side of the tile. This allows max_y to be decremented to disregard these tiles.

FindBound example:



This shape will return the table:

```
{
	min_x = 1
	max_x = 4
	min_y = 2
	max_y = 5
	width = 4
	height = 4
	edge_only_left = true
	edge_only_top = true
}
```

Note as both the left and top sides (column 1 and row 5) of the box are technically in use but contain only edges. The 'edge only' flags provides a way to detect this.

5.9 GCImport

Helper functions to copy files from outside into the file store. See also `GCExport` to get data back out again.

BufferAs

```
GCImport.BufferAs( bufferHandle: int,
                   caption: string,
                   ext: string,
                   extDesc: string ): int
```

Open a standard file selector to select an external file for import into the given buffer. The buffer is referenced by handle `bufferHandle` and the file will be written at the position of the current write cursor.

The selector title is configured using the `caption` parameter. A file type filter can be defined using `fileExt` (e.g. `"TXT"`) and `fileExtDesc` (e.g. `"Text Files"`). To offer only a generic 'all files' option, set either of these values to `null`.

If the file is saved correctly it returns `GCOK`. If the user chooses to cancel the file selector the function returns `GCERR_ABORT`. If there was an error writing the file returns `GCERR_ACCESS`. If the buffer handle is invalid it returns `GCERR_INVARG`.

5.10 GCKernel

The kernel functions provide some low level operating system features.

CommandHooks

`GCKernel.CommandHooks(): array<string>`

Return an array containing the id values of all script hooks using the "`command`" location.

Exec

`GCKernel.Exec( filepath: string, args: array<string> ): int|null`

Attempt to execute the given program file from the virtual store path. If the file-name is missing an extension it will be retried with one automatically appended.

If successful the full argument list (including element zero, which should be the program name) is passed to the program's entry-point. For Squirrel scripts this is the `main` function.

Once the child process has finished executing, this function will return `null`. If execution fails the function will return an error code.

ExecHook

`GCKernel.ExecHook( id: string ): int|null`

Spawns a new child process using the given id value. If the process was spawned this function will return null. If execution fails, the function returns false.

ExecHookParam

`GCKernel.ExecHookParam( id: string, args: array<string> ): int|null`

Spawns a new child process using the given id value. If the process was spawned this function will return null. If execution fails, the function returns false.

The `args` parameter is made available to the hook call and is included when the user argument type is used. See the section on command hooks for additional information.

Exit

`GCKernel.Exit(): void`

Causes the currently running process to stop and revert back to the caller.

QueryAppInfo

`GCKernel.QueryAppInfo( select: string ): string|int|null`

Query general information about Grid Cartographer. Select the information returned using the `select` parameter. If the selection is not understood, returns `null`.

Select	Returns
`"version"`	Version information in string format "#.#.#", e.g. "4.0.4"
`"build"`	Build number as an integer.

SetBlockingMode

`GCKernel.SetBlockingMode( enable: bool ): void`

Sets whether the current script should run in blocking mode.

By default, scripts run in co-operative mode which allows the editor to function fully and receive user inputs, map editing commands, etc. If the script requires access to the map and you wish to prevent conflicts or inconsistent data, then use this function to enable blocking mode.

When blocking mode enabled, only the toggle console key (and user interface button) will function, the rest of the interface will freeze. It is recommended to ensure that blocking mode is not used in situations where the script might fail to end as this may cause a loss of user data.

As a precaution against faulty usage of blocking mode, the shortcut `CTRL+C` will abort the current script when this mode is enabled.

SubProcDepth

GCKernel.SubProcDepth(): int

Returns the number of parent processes this process has. Returns `0` (zero) a top-level process (UI button triggered script, console command line, etc.), `1` for programs launched from it, `2` for child processes launched from those, and so on.

Wait

`GCKernel.Wait( time_ms: int ): void`

Stop execution of the script for approximately time_ms milliseconds. Grid Cartographer will remain responsive during this time (although the user may be blocked from editing) but the script will not continue until time has elapsed.

5.11 GCRegion

Functions relating to the regions of the map.

Count

`GCRegion.Count(): int`

Returns the number of regions in the map.

Shape

`GCRegion.Shape(): string`

Returns the shape of the active region. One of `square`, `hexh`, `hexv` or `tilemap`. If the active region is not valid, returns `null`.

Origin

`GCRegion.Origin(): string|null`

Returns the origin of the active region. One of `"tl"` (top-left) or `"bl"` (bottom-left). If the active region is not valid, returns `null`.

5.12 GCStatus

These functions control the Grid Cartographer status bar. Use it to show low priority messages and an operation progress bar.

Message

```
GCStatus.Message( text: string ): void
```

Set a new message on the status bar. The existing message will be replaced.

Progress

```
GCStatus.Progress( percent: int|null ): void
```

Attach a progress bar to the end of the status bar message. Set the value of the bar using the `percent` parameter - a value from 0 to 100. Pass in `null` to remove the progress bar.

5.13 GCTile

These functions allow interaction with individual tiles on the active region / floor / tile-map. These functions use a single "active tile" as the implicit context for operations. Use `Select` and `Move` functions to change the position of the active tile. Note there is only one active tile cursor for all floors / regions.

WARNING: Unless the script is running in blocking mode there is the potential for the user to interact with the map and cause inconsistent results. Take care to guard your script from unwanted user interaction. See `GCKernel.SetBlockMode`.

Select

```
GCTile.Select( x: int, y: int ): void
```

Change the "active tile" to the specified `(x,y)` co-ordinate value. All subsequent `GCTile` functions will use this tile as the context for their operation. The default value when the script starts is co-ordinate `(0,0)`.

Note that the y value of the co-ordinate will automatically respect the current grid origin such that in the default bottom-left mode, increasing values will move up the screen and in the top-left mode the value will move down the screen. There is no need to compensate for this manually.

MoveNorth

```
GCTile.MoveNorth( steps: int ): void
```

Advances the active tile by `steps` tiles to the north. Negative values are supported and will move to the south instead.

Note that when this function is called the current active region is checked to determine the grid origin so that a move north will always be equivalent to moving up the display in the editor.

MoveSouth

```
GCTile.MoveSouth( steps: int ): void
```

Advances the active tile by `steps` tiles to the south. Negative values are supported and will move to the north instead.

Note that when this function is called the current active region is checked to determine the grid origin so that a move south will always be equivalent to moving down the display in the editor.

MoveEast

`GCTile.MoveEast( steps: int ): void`

Advances the active tile by `steps` tiles to the east. Negative values are supported and will move to the west instead.

MoveWest

`GCTile.MoveWest( steps: int ): void`

Advances the active tile by `steps` tiles to the west. Negative values are supported and will move to the east instead.

Move

`GCTile.Move( eastSteps: int, northSteps: int ): void`

Move the active tile cursor in two directions using one function call, relative to its current position. This is equivalent to calling `GCTile.MoveEast( eastSteps )` followed immediately by `GCTile.MoveNorth( northSteps )`. To move south or west, use negative values.

ActiveX

`GCTile.ActiveX(): int`

Returns the x co-ordinate of the active tile cursor.

ActiveY

`GCTile.ActiveY(): int`

Returns the y co-ordinate of the active tile cursor.

Get

`GCTile.Get(): table`

Returns the full set of information about the active tile. This function always succeeds even if a nonexistent floor or region has been selected (it will return a default 'empty' tile).

For standard grid shapes (square, hex 'H', hex 'V'), the slots of the table are as follows:

Standard Grid Tile Table

Key	Value
`Visible: bool`	True if the tile is visible. False if hidden by fog-of-war.
`Marker: table`	Sub-table containing marker layer information.
`Terrain: table`	Sub-table containing terrain layer information.
`EdgeR: table`	Sub-table containing information about the 'R' edge of a tile. Refer to the *Tile Data Models* section of the User Manual for more information.
`EdgeI: table`	Sub-table containing information about the 'I' edge of a tile. Refer to the *Tile Data Models* section of the User Manual. This table is omitted for square grid regions.
`EdgeB: table`	Sub-table containing information about the 'B' edge of a tile. Refer to the *Tile Data Models* section of the User Manual for more information.
`FX: table`	Sub-table containing tile effects.
`Ceiling: bool`	True if the tile has a ceiling.
`Print: bool`	True if the tile has a footprint on it.

Sub-tables are described below.

FX Table

Key	Value
`Dark: bool`	True if darkness is applied
`R: bool`	True if red field applied
`G: bool`	True if green field applied
`B: bool`	True if blue field applied

Marker Table

Key	Value
`Type: int`	Type of this marker. 0 (zero) means the tile is empty. Values used are listed in the *Data Tables* section of the User Manual.
`Sub: int`	Sub-channel data for the marker. Several marker types (including custom tiles) use this to provide additional type information.
`Color: int`	Marker tint color. Value from `0` (default) to `255`.
`Switch: bool`	The switch state of the marker (e.g. book open/closed). Value is `false` by default and `true` if the alternate appearance has been selected.

Terrain Table

Key	Value
`Type: int`	Type of this terrain. 0 (zero) means the tile is empty. Values used are listed in the *Data Tables* section of the User Manual.
`Sub: int`	Sub-channel data for the terrain. Custom terrain types use this to provide additional type information.
`Color: int`	Terrain tint color. Value from `0` (default) to `255`.

Edge Tables ('R', 'I' or 'B')

Key	Value
`Type: int`	Type of this edge. 0 (zero) means the edge is empty. Values used are listed in the *Data Tables* section of the User Manual.
`Color: int`	Edge tint color. Value from `0` (default) to `255`.
`Switch: bool`	The switch state of the edge (e.g. torch unlit/lit). Value is `false` by default and `true` if the alternate appearance has been selected.

Get (tile maps)

For tile maps the return value is significantly different.

Use the `GCRegion.Shape` function to determine the return type used

Tile Map Table

Key	Value
`Index: int`	The index of the custom tile (from zero), or `-1` if no tile is present.
`FlipH: bool`	True if the tile is flipped horizontally.
`FlipV: bool`	True if the tile is flipped vertically.

5.14 GCUI

A set of functions to interact with Grid Cartographer user interface services.

MessageBox

`GCUI.MessageBox( message: string, type: int ): string|null`

Opens a standard message box dialog box. The text of the message is given by the message parameter and settings are provided using `type` to specify one (or a combination) of these `GCMB` prefix constant flags:

Type Flag	Description
`GCMB_OK`	The message box will have one option "OK". This is the default if no option is selected.
`GCMB_OKCANCEL`	Two options "OK" and "Cancel".
`GCMB_YESNO`	Two options "Yes" and "No".
`GCMB_YESNOCANCEL`	Three options "Yes", "No" and "Cancel".
`GCMB_INFO`	Adds an "information" background image to the message.
`GCMB_STOP`	Adds a "stop" icon to the message.
`GCMB_ASK`	Adds a question mark icon to the message background.

If no icon flag is used the message box will display the message text only.

The return values of the message box depend on the settings and user input:

Result	Description
`null`	There was an error showing the message box, the flags used were invalid or the given message was `null`.
`"ok"`	The user clicked "OK".
`"cancel"`	The user clicked "Cancel".
`"yes"`	The user clicked "Yes".
`"no"`	The user clicked "No".

5.15 Math Constants

A collection of helpful mathematical constants.

PI

`Math.PI : float`

An approximation of the mathematical constant pi.

TWOPI

`Math.TWOPI: float`

An approximation of 2 x pi.

RADTODEG

`Math.RADTODEG: float`

An approximation of 180/pi. Multiply this by a value given in radians to convert to degrees.

DEGTORAD

`Math.DEGTORAD: float`

An approximation of pi/180. Multiply this by a value given in degrees to convert to radians.

5.16 Math Functions

A collection of standard mathematical functions.

abs

```
Math.abs( value: int|float ): int
```

Returns the absolute value of a number as an integer.

acos

```
Math.acos( value: float ): float
```

Returns the arc cosine of value (in radians).

asin

```
Math.asin( value: float ): float
```

Returns the arc sine of value (in radians).

atan

```
Math.atan( value: float ): float
```

Returns the arc tangent of value (in radians).

atan2

```
Math.atan2( x: float, y: float ): float
```

Returns the arc tangent of `x,y` (in radians).

ceil

```
Math.ceil( value: float ): float
```

Returns value rounded up to the next integer value. If `value` is already a whole number then it is returned unmodified.

cos

```
Math.cos( value: float ): float
```

Returns the cosine of value (given in radians).

exp

`Math.exp( value: float ): float`

Returns *e* raised to the power of `value`.

fabs

`Math.fabs( value: int|float ): float`

Returns the absolute value of a floating point number. i.e. a positive value with equal magnitude to `value` (which may already have been positive.)

floor

`Math.floor( value: float ): float`

Returns `value` rounded down to the nearest integer value (truncating the fractional part). If `value` is already a whole number then it is returned unmodified.

log

`Math.log( value: float ): float`

Returns the natural logarithm of value.

log10

`Math.log10( value: float ): float`

Returns the base10 logarithm of value.

mod

`Math.mod( a: int, m: int ): int`

Returns `a mod m`, a value between 0 and m - 1 using modular arithmetic. Unlike the % operator this function also handles negative values of `a` properly.

pow

`Math.pow( a: float, b: float ): float`

Returns `a` raised to the power of `b`.

rand

Math.rand(limit: int): int

Returns a pseudo-random number from `0` to `limit - 1`.

sin

`Math.sin( value: float ): float`

Returns the sine of `value` (given in radians).

sqrt

`Math.sqrt( value: float ): float`

Returns the square root of `value`.

srand

`Math.srand( seed: int ): void`

Seed the random number generator with a new value.

tan

`Math.tan( value: float ): float`

Returns the tangent of value (given in radians).

5.17 Error Constants

The API defines a set of standard error codes for all functions. These constant values are declared as integer type values. Always check for error codes!

Constant	Value	Description
GCOK	0	Function was successful. No error.
GCERR_FAIL	-1	General unspecified error.
GCERR_INVARG	-2	Invalid argument. The parameter to a function (e.g. a file name or drive letter) was not in the correct format.
GCERR_EOF	-3	End of file reached when reading.
GCERR_NODRIVE	-4	Attempt to access a virtual drive in the File Store which does not exist.
GCERR_NOENT	-5	Attempt to access a file which does not exist.
GCERR_EXISTS	-6	Trying to rename or move a file where the new name already exists.
GCERR_INVDST	-7	Invalid destination argument, e.g. when renaming.
GCERR_ACCESS	-8	Access is not permitted. e.g. trying to write to a file opened for reading.
GCERR_ABORT	-9	Function was aborted, typically by the user. e.g. cancelling a file selector.

5.18 Keyboard Constants

Special function keys are defined using the `GCKey` constant table. Use these identifiers instead of literal values to ensure that any future changes to Grid Cartographer will not break compatibility with the script.

Constant	Key Press
`GCKey.Break`	`CTRL+C` (`CMD+C` on macOS)
`GCKey.Back`	`Backspace`
`GCKey.Tab`	`Tab`
`GCKey.Enter`	`Return` / `Enter`
`GCKey.Esc`	`Escape`
`GCKey.Up`	`Cursor Up`
`GCKey.Right`	`Cursor Right`
`GCKey.Down`	`Cursor Down`
`GCKey.Left`	`Cursor Left`
`GCKey.Ins`	`Insert`
`GCKey.Del`	`Delete`
`GCKey.Home`	`Home`
`GCKey.End`	`End`
`GCKey.PageUp`	`Page Up`
`GCKey.PageDown`	`Page Down`
`GCKey.F1`	`F1`
`GCKey.F2`	`F2`
`GCKey.F3`	`F3`
`GCKey.F4`	`F4`

Constant	Key Press
GCKey.F5	F5
GCKey.F6	F6
GCKey.F7	F7
GCKey.F8	F8
GCKey.F9	F9
GCKey.F10	F10
GCKey.F11	F11
GCKey.F12	F12

5.19 Standard Script Library

Grid Cartographer is supplied with a number of standard library functions that perform common tasks. These are available to all Squirrel scripts by adding `<load>` entries into the hook definition. Note these references should appear *before* the main script so that the function names, etc. are registered correctly prior to being used.

stdio

getline

Script Path: `stdio/getline.nut`

```
getline( maxlength ): string
```

A simple line input buffer, up to `maxlength` characters. It supports backspace, delete, cursor left/right, home and end. Entered lines are stored in a history and can be accessed using the up/down cursor keys. F3 will also restore the last typed line.

The line editor continues until the enter key is pressed. If CTRL+C is pressed, the function returns an empty string. ESC will clear the current line to let you start again.