Tebas Script

Documentation for Tebas Script, by Siljam.
https://github.com/siljamdev/Tebas

Scripts

Scripts are files, usually with the *.tbscr* extension.

Each file is divided by lines and each line is a **sentence**.

Tables and strings

Before explaining normal sentences, we first have to understand the central concept of tables. Tables are global variables that do not need to be initialized. They are dynamic arrays of strings that can contain any values.

The syntax used to access tables is the following:

```
table.index

deck.0
deck.1
deck.-1
deck.random
deck.length
```

Tables are 0 indexed, so the first element is element 0. We can also access things starting from the end instead of the start. Element -1 is the last element, element -2 is the previous and so forth.

You can also access *length*, *random* and *center* properties. *length* is the length of the table, *random* returns an element chosen at random and *center* returns the element in the middle.

Arguments

Sentences are divided into command and arguments. The parts are divided by whitespace, but whitespace is allowed inside of quotes or square brackets.
There are 3 types of arguments:

Strings

These can be thought of as text. They can be values from a table or literals.

```
“Hello World!”
deck.0
```

Literals are written between double quotes ‘”’, to escape one of them use the backslash \. Also, \n will be replaced by a newline.

There are also special values, that are preceded by %.

```
%pn //Project name
%tn //Template name
%wd //Working directory
%td //Template directory
%pl //Current plugin name
%d //Current date and time
```

String references

This arguments are used mostly in sentences that return values, they symbolized where to put the output value. Note that you cannot use *length* here, as it is read-only.

```
deck.1
show.0
```

Tables

The tables themselves, you can either reference a global table or write a literal.

```
[“value 1”, “value 2”]

deck
args
```

Literals are written between square brackets [] and they include string literals separated by commas.

Table references

This arguments are used mostly in sentences that return values, they symbolized where to put the output value.

```
deck
show
```

Special tables

There are some special tables that have some functionality.

args

This is the command line args that the script was called with.

true and false

These two are initialized with 1 and 0 respectively and are used for all Boolean operations or logic handling. You can set them to something custom or add extra values.

error

When an error occurs, this table will be erased and a single entry will be added, a description of the error. Also, information will appear on the console.

Flow control sentences

Flow control happens in blocks, encased with curly brackets {}. It is important to leave whitespace before brackets.

Now, a list of all sentences will be provided along with what they do and their format. For the arguments, *s* will be used to symbolize a string argument, **s* for a string reference, *t* for a table and **t* for a table reference.

exit

Used to instantly exit the script.

```
exit
```

function

Used to define a function, with a name. It can be called at any moment. Functions do not have arguments, but all tables are global.

```
function name {
      body
}
```

return

Used inside a function, to return immediately.

```
return
```

call

Used to invoke functions.

```
call name
```

scope

This sentence only makes sense inside a function. It is used to have local data that makes no sense outside of the function. Only one table can be declared the local scope per function. The value of that table is saved when entering the function and retrieved when passing control to other functions or finishing, so that table can be also used outside.

```
scope *t
```

while

Used for while loops, common in many programming languages. The condition is a string, that will be evaluated as a boolean. You can also use *while!*, and the condition will be reversed (false needed).

```
while s {
      body
}
```

do

Used for do-while loops. However, the condition is at the start of the loop, but is evaluated at the end. *do!* Also exists.

```
do s {
      body
}
```

continue

This sentence only makes sense inside a while or do loop. It continues to the next iteration of the loop, skipping the remaining code.

```
continue
```

break

This sentence only makes sense inside a while or do loop. It breaks out of the loop, skipping the remaining code and iterations.

```
break
```

if-elseif-else

Sentences used for if statements. In the same sentence as the closing bracket of the main if statement, you can include the *elseif* (or *elseif!*) or *else*.

```
if s {
      body
} elseif s {
      body
} elseif! s {
      body
} else {
      body
}
```

Action sentences

Now, a list of all sentences will be provided, grouped together in groups, along with what they do and their format. For the arguments, *s* will be used to symbolize a string argument, **s* for a string reference, *t* for a table and **t* for a table reference.

console

```
console.print s
```

Prints a string into the console. This string will be expanded (string expansion is explained later)

```
console.printNoExpand s
```

Prints a string into the console as it is.

```
console.pause s
```

Will output a string into the console and then wait for any key to be pressed to continue.

```
console.ask *s s
```

Will output a string(2nd arg) to the console and then let the user type an answer, which is outputted in the first arg.

process

```
process.cmd s
```

Starts cmd process with commands as arguments.

```
process.run s s
```

Starts process(1st arg) with cli arguments(2nd arg).

```
process.runOutput s s *t *t
```

Starts process(1st arg) with cli arguments(2nd arg), and outputs the process output into the table in 3rd arg and the process errors into the table in 4th.

time

```
time.wait s
```

Reads the string as a number and waits that number of milliseconds.

string

```
string.set *s s
```

Sets a string

```
string.expand *s s
```

Sets a string to another string expanded.

```
string.append *s s s
```

Sets a string to two strings concatenated.

```
string.split *t s s
```

Sets the table to the string in 2nd arg splitted, using 3rd arg as separator.

```
string.substring *s s s s
```

Sets a string to another string(2nd arg) substring. 3rd arg is start index(0 based) and 4th is length.

```
string.replace *s s s s
```

Sets a string to another string where the 3rd string has been replaced by the 4th string.

```
string.equal *s s s
```

Sets a string to true if two strings are equal, or true if they are not.

```
string.lower *s s
```

Sets a string to lowercase of another string

```
string.upper *s s
```

Sets a string to uppercase of another string

```
string.contains *s s s
```

Sets a string to true if a string contains a substring, and false otherwise.

```
string.trim *s s
```

Sets a string to another string where the start and end whitespace has been removed.

```
string.removeQuotes *s s
```

Sets a string to another string has been attempted to remove double quotes “” if it was encased in them.

```
string.count *s s
```

Sets a string to the length of another string.

self

```
self.expand *s
```

A string is set to itself expanded.

```
self.append *s s
```

A string is set to itself concatenated with another string.

```
self.substring *s s s
```

A string is set to a substring of itself. 2nd arg is start index(0 based) and 3rd is length.

```
self.replace *s s s
```

A string is set to itself where it got a string(2nd arg) replaced by another string(4th arg)

```
self.upper *s
```

A string is set to itself in uppercase.

```
self.lower *s
```

A string is set to itself in lowercase.

```
self.trim *s
```

A string is set to itself where the start and end whitespace has been removed.

```
self.removeQuotes *s
```

A string is set to itself where it has been attempted to remove double quotes “” if it was encased in them.

table

```
table.access *s *t s
```

Sets a string(1st arg) to an string from the table(2nd arg) using the 3rd string as the index. You can still use random and center here.

```
table.setAt *t s s
```

Sets a string from the table in the first arg, using the 2nd arg as index, and tr¡he 3rd arg as the value to set it to.

```
table.delete *s
```

Delete a string from its table. Tables are dynamic, so the next item will occupy its place.

```
table.deleteAt *t s
```

Deleting a string from a table. 2nd arg is used as the index. Tables are dynamic, so the next item will occupy its place.

```
table.insert *s s
```

Inserting a value(2nd arg) in a string in the table. Tables are dynamic, so the next item will move down.

```
table.insertAt *t s s
```

Inserting a value(3rd arg) into a table(1st arg), using 2nd arg as index.

```
table.contains *s t s
```

Sets a string(1st arg) to true if the table(2nd arg) contains the string(3rd arg), false otherwise.

```
table.clear *t
```

Clears a table to its empty state

```
table.add *t s
```

Adds a string to the end of a table.

```
table.append *t t
```

Adds multiple strings(a table, 2nd arg) to the end of a table.

```
table.set *t t
```

Sets a table.

```
table.shuffle *t t
```

Sets a table to another table(2nd arg) randomly shuffled.

```
table.join *s t s
```

Sets a string to a table joined together, using the 3rd arg as separator.

bool

```
bool.negate *s s
```

Sets a string to the negated bool of another(uses true and false tables, described earlier).

```
bool.and *s s s
```

Sets a string to true if both strings are true.

```
bool.or *s s s
```

Sets a string to true if at least one string is true.

math

```
math.isNumber *s s
```

Sets a string to true if the string can be parsed to a number(int). Numbers are represented as parsable strings.

```
math.isNegative *s s
```

Sets a string to true if the string is a negative number.

```
math.equal *s s s
```

Sets a string to true if the strings represent the same number.

```
math.getRandom *s s s
```

Sets a string to a random number between the 2nd arg(inclusive) and the 3rd(exclusive).

```
math.sumUp *s s
```

Sets a string to the result of adding 1 to another(2nd arg).

```
math.sumDown *s s
```

Sets a string to the result of subtracting 1 to another(2nd arg).

```
math.sum *s s s
```

Sets a string to the result of adding two numbers together (2nd and 3rd args).

```
math.substract *s s s
```

Sets a string to the result of subtracting the 3rd arg from the 2nd.

```
math.multiply *s s s
```

Sets a string to the result of multiplying two numbers together (2nd and 3rd args).

```
math.divide *s s s
```

Sets a string to the result of dividing the 2nd arg by the 3rd. This is integer division, no floating point.

```
math.modulus *s s s
```

Sets a string to the result of performing modulus operation(equivalent to % operator in c).

```
math.abs *s s
```

Sets a string to the absolute value of the number as arg.

```
math.greater *s s s
```

Sets a string to true if the 2nd arg is greater than the 3rd, false otherwise.

```
math.greaterEqual *s s s
```

Sets a string to true if the 2nd arg is greater or equal than the 3rd, false otherwise.

```
math.less *s s s
```

Sets a string to true if the 2nd arg is less than the 3rd, false otherwise.

```
math.lessEqual *s s s
```

Sets a string to true if the 2nd arg is less or equal than the 3rd, false otherwise.

path

```
path.extension *s s
```

Sets a string to the extension (including the dot .) of the file, providing the path in the 2nd arg. More on paths later.

```
path.filename *s s
```

Sets a string to the file name and extension of a file proving its path. For example *folder/folder/hello.txt => hello.txt*

```
path.filenameNoExtension *s s
```

Sets a string to the file name of a file proving its path. For example *folder/folder/hello.txt => hello*

```
path.directory *s s
```

Sets a string to the directory containing a file.

file

```
file.create s
```

Creates a file if it doesn't exist already, provided its path.

```
file.read *s s
```

Sets a string to the full contents of a file, provided its path(2nd arg).

```
file.delete s
```

Deletes a file based on its path

```
file.rename s s
```

Renames a file from old name(2nd arg) to new name(3rd arg).

```
file.write s s
```

Writes content to a file, overriding it. 2nd arg is the path, and 3rd is the content.

```
file.append s s
```

Writes content to a file, adding it to the end. 2nd arg is the path, and 3rd is the content.

```
file.exists *s s
```

Sets a string to true if a file exists, provided its path(2nd arg). False otherwise.

```
file.size *s s
```

Sets a string to the size of a file, in bytes, provided its path(2nd arg).

folder

```
folder.create s
```

Creates a directory, provided its path. More on paths later.

```
folder.delete s
```

Deletes a directory, provided its path.

```
folder.rename s s
```

Renames a folder, provided the old name(2nd arg) and its new name(3rd arg).

```
folder.exists *s s
```

Sets a string to true if a directoty exists, provided its path(2nd arg). False otherwise.

```
folder.list *t s s
```

Sets a table to the list of existing files inside a directory, provided its path(2nd arg) and a format(3rd arg). This format can contain wildcards, for example '*' or '*.*txt*'.

```
folder.listChild *t s s
```

Sets a table to the list of existing files inside a directory and its subdirectories, provided its path(2nd arg) and a format(3rd arg). This format can contain wildcards, for example '*' or '*.*txt*'.

template

```
template.read *s s
```

For scripts that work inside a template, gets a saved resource from the template, using the 2nd arg as name.

```
template.write s s
```

Writes a resource into the current template. Its value will be overwritten. 2nd arg is the resource name and 3rd is the content.

```
template.append s s
```

Writes a resource into the current template. Its value will be added to the end. 2nd arg is the resource name and 3rd is the content.

```
template.run s s
```

Runs a script from the current template. 2nd arg is the script name, and 3rd arg is cli arguments. These are a continuous string, but will be split like cli args do.

```
template.installed *s s
```

Sets a string to true if a template with 2nd arg as name is installed, false otherwise.

```
template.list *t
```

Returns a table with the names of all installed templates.

```
template.create s
```

Create a template using the creator utility. It is the equivalent of doing the cli command. The arg is the path.

plugin

```
plugin.read *s s s
```

Sets a string(1st arg) to the result of reading a resource from a plugin. The 2nd arg is the name of the plugin, and the 3rd arg is the name of the resource.

```
plugin.write s s s
```

Saves a plugin resource. 1st arg is the name of the plugin, 2nd arg is the name of the resource, and 3rd arg is the content being written.

```
pugin.append s s s
```

Saves a plugin resource, this value will be appended to the end. 1st arg is the name of the plugin, 2nd arg is the name of the resource, and 3rd arg is the content being written.

```
plugin.run s s s
```

Runs a script from a plugin. 1st arg is the name of the plugin, 2nd arg the name of the script, and 3rd arg is cli commands.

```
plugin.installed *s s
```

Sets a string to true if a string(string name 3nd arg) is installed, false otherwise.

```
plugin.list *t
```

Gets a list of installed pugins.

```
plugin.create s
```

Create a plugin using the creator utility. It is the equivalent of doing the cli command. The arg is the path.

project

```
project.read *s s
```

If there is currently a local project active, it reads a resource from the project file. 2nd arg is the resource name.

```
project.write s s
```

It writes a resource to the local project. 2nd arg is the resource name, and 3rd is the content.

```
project.append s s
```

It appends to the end of a resource. 2nd arg is the resource name, and 3rd is the content.

tebas

```
tebas.commit s
```

Equivalent of doing the cli command. Commits to git. The argument is the commit message.

```
tebas.push s
```

Equivalent of doing the cli command. Commits to git. The argument is the remote.

```
tebas.pull s
```

Equivalent of doing the cli command. Commits to git. The argument is the remote.

```
tebas.add
```

Equivalent of doing the cli command.

```
tebas.remoteSet s s
```

Equivalent of doing the cli command, 2nd arg is the remote name, and 3rd arg is the remote url.

```
tebas.remoteDelete s
```

Equivalent of doing the cli command, 2nd arg is the remote name.

```
tebas.remoteList *t
```

Sets a table of all the remote names of the current local project.

```
tebas.remoteExists *s s
```

Sets a string to true if a remote exists(2nd arg as remote name), false otherwise.

Notes

There is a couple important things to say about the sentences.

Comments

Comments are started with '//', and they can be placed is any place of the sentence, except inside quotes or brackets.
For example:

```
//This is a valid comment
command arg1 arg2 //This is a valid comment
command "//This comment will be absorbed into arg1" //This is a valid comment
```

File paths

When file path appear(this does not apply to the application path in the run sentence) they usually do inside of a literal.

Only local files are accessible, that means only working directory (most of the time project folder) or template directory.

You can access them with the prefixes W and T(working directory and template directory, respectively)

So you can do:

```
"W/hello.txt" //This refers to a text file in the project folder
"T/folder1/folder2/res.dll" //Dll file in template folder
```

Only these local files can be accessed. However, there is an exception. If a path file is not started by neither W or T in the file.exists sentence, it will check the normal system file. Also, wildcards(* and ?) are supported in file.exists and folder.list(and folder.listChild).

String expansion

String expansion is a useful tool. It works with the format of a string. For example:

```
“Hello! Welcome to the project {%pn}”
```

If this string is expanded, the value between the curly brackets {} will be replaced. This is only valid with table references or % values.

```
“Value: {value.0}”
```

To stop this from happening and actually use the curly brackets, we can use a backslash \.

```
“You can read \{this}”
```