Basic Functions

The basic library provides some core functions to Lua. If you do not include this library in your application, you should check carefully whether you need to provide implementations for some of its facilities.

`ipairs (t)`

Returns three values: an iterator function, the table `t`, and 0, so that the construction

```
for i,v in ipairs(t) do body end
```

will iterate over the pairs `(1,t[1])`, `(2,t[2])`, ···, up to the first integer key absent from the table.

`next (table [, index])`

Allows a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. `next` returns the next index of the table and its associated value. When called with **nil** as its second argument, `next` returns an initial index and its associated value. When called with the last index, or with **nil** in an empty table, `next` returns **nil**. If the second argument is absent, then it is interpreted as **nil**. In particular, you can use `next(t)` to check whether a table is empty.

The order in which the indices are enumerated is not specified, *even for numeric indices*. (To traverse a table in numeric order, use a numerical **for** or the ipairs function.)

The behavior of `next` is *undefined* if, during the traversal, you assign any value to a non-existent field in the table. You may however modify existing fields. In particular, you may clear existing fields.

`pairs (t)`

Returns three values: the next function, the table `t`, and **nil**, so that the construction

```
for k,v in pairs(t) do body end
```

will iterate over all key–value pairs of table `t`.

`tonumber (e [, base])`

Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then `tonumber` returns this number; otherwise, it returns **nil**.

An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter 'A' (in either upper or lower case) represents 10, 'B' represents 11, and so forth, with 'Z' representing 35. In base 10 (the default), the number can have a decimal part, as well as an optional exponent part. In other bases, only unsigned integers are accepted.

`tostring (e)`

Receives an argument of any type and converts it to a string in a reasonable format. For complete control of how numbers are converted, use string.format.

If the metatable of `e` has a `"__tostring"` field, then `tostring` calls the corresponding value with `e` as argument, and uses the result of the call as its result.

`type (v)`

Returns the type of its only argument, coded as a string. The possible results of this function are `"nil"` (a string, not the value **nil**), `"number"`, `"string"`, `"boolean"`, `"table"`, `"function"`, `"thread"`, and `"userdata"`.

String Manipulation

This library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. When indexing a string in Lua, the first character is at position 1 (not at 0, as in C). Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position -1, and so on.

The string library provides all its functions inside the table `string`. It also sets a metatable for strings where the `__index` field points to the `string` table. Therefore, you can use the string functions in object-oriented style. For instance, `string.byte(s, i)` can be written as `s:byte(i)`.

The string library assumes one-byte character encodings.

`string.byte (s [, i [, j]])`

Returns the internal numerical codes of the characters `s[i]`, `s[i+1]`, ···, `s[j]`. The default value for `i` is 1; the default value for `j` is `i`.

Note that numerical codes are not necessarily portable across platforms.

`string.char (···)`

Receives zero or more integers. Returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its corresponding argument.

Note that numerical codes are not necessarily portable across platforms.

`string.find (s, pattern [, init [, plain]])`

Looks for the first match of `pattern` in the string `s`. If it finds a match, then `find` returns the indices of `s` where this occurrence starts and ends; otherwise, it returns **nil**. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and can be negative. A value of **true** as a fourth, optional argument `plain` turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in `pattern` being considered "magic". Note that if `plain` is given, then `init` must be given as well.

If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

`string.format (formatstring, ···)`

Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string follows the same rules as the `printf` family of standard C functions. The only differences are that the options/modifiers `*`, `l`, `L`, `n`, `p`, and `h` are not supported and that there is an extra option, `q`. The `q` option formats a string in a form suitable to be safely read back by the Lua interpreter: the string is written between double quotes, and all double quotes, newlines, embedded zeros, and backslashes in the string are correctly escaped when written. For instance, the call

```
string.format('%q', 'a string with "quotes" and \n new line')
```

will produce the string:

```
"a string with \"quotes\" and \
 new line"
```

The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument, whereas `q` and `s` expect a string. This function does not accept string values containing embedded zeros, except as arguments to the `q` option.

`string.gmatch (s, pattern)`

Returns an iterator function that, each time it is called, returns the next captures from `pattern` over string `s`. If `pattern` specifies no captures, then the whole match is produced in each call.

As an example, the following loop

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
  print(w)
end
```

will iterate over all the words from string `s`, printing one per line. The next example `key=value` from the given string into a table:

```
t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
  t[k] = v
end
```

For this function, a '^' at the start of a pattern does not work as an anchor, as this would prevent the iteration.

---

string.gsub (s, pattern, repl [, n])

Returns a copy of `s` in which all (or the first `n`, if given) occurrences of the `pattern` have been replaced by a replacement string specified by `repl`, which can be a string, a table, or a function. `gsub` also returns, as its second value, the total number of matches that occurred.

If `repl` is a string, then its value is used for replacement. The character `%` works as an escape character: any sequence in `repl` of the form *%n*, with *n* between 1 and 9, stands for the value of the *n*-th captured substring (see below). The sequence `%0` stands for the whole match. The sequence `%%` stands for a single `%`.

If `repl` is a table, then the table is queried for every match, using the first capture as the key; if the pattern specifies no captures, then the whole match is used as the key.

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the pattern specifies no captures, then the whole match is passed as a sole argument.

If the value returned by the table query or by the function call is a string or a number, then it is used as the replacement string; otherwise, if it is **false** or **nil**, then there is no replacement (that is, the original match is kept in the string).

Here are some examples:

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0", 1)
--> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return loadstring(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.1"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.1.tar.gz"
```

---

string.len (s)

Receives a string and returns its length. The empty string `""` has length 0. Embedded zeros are counted, so `"a\000bc\000"` has length 5.

---

string.lower (s)

Receives a string and returns a copy of this string with all uppercase letters changed to lowercase. All other characters are left unchanged. The definition of what an uppercase letter is depends on the current locale.

`string.match (s, pattern [, init])`

Looks for the first *match* of `pattern` in the string `s`. If it finds one, then `match` returns the captures from the pattern; otherwise it returns **nil**. If `pattern` specifies no captures, then the whole match is returned. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and can be negative.

`string.rep (s, n)`

Returns a string that is the concatenation of `n` copies of the string `s`.

`string.reverse (s)`

Returns a string that is the string `s` reversed.

`string.sub (s, i [, j])`

Returns the substring of `s` that starts at `i` and continues until `j`; `i` and `j` can be negative. If `j` is absent, then it is assumed to be equal to -1 (which is the same as the string length). In particular, the call `string.sub(s,1,j)` returns a prefix of `s` with length `j`, and `string.sub(s, -i)` returns a suffix of `s` with length `i`.

`string.upper (s)`

Receives a string and returns a copy of this string with all lowercase letters changed to uppercase. All other characters are left unchanged. The definition of what a lowercase letter is depends on the current locale.

Patterns

Character Class:

A *character class* is used to represent a set of characters. The following combinations are allowed in describing a character class:

- ***x*:** (where *x* is not one of the *magic characters* `^$()%.[]*+-?`) represents the character *x* itself.
- **`.`:** (a dot) represents all characters.
- **`%a`:** represents all letters.
- **`%c`:** represents all control characters.
- **`%d`:** represents all digits.
- **`%l`:** represents all lowercase letters.
- **`%p`:** represents all punctuation characters.
- **`%s`:** represents all space characters.
- **`%u`:** represents all uppercase letters.
- **`%w`:** represents all alphanumeric characters.
- **`%x`:** represents all hexadecimal digits.
- **`%z`:** represents the character with representation 0.
- **`%`*x*:** (where *x* is any non-alphanumeric character) represents the character *x*. This is the standard way to escape the magic characters. Any punctuation character (even the non magic) can be preceded by a '%' when used to represent itself in a pattern.
- **`[`*set*`]`:** represents the class which is the union of all characters in *set*. A range of characters can be specified by separating the end characters of the range with a '-'. All classes `%`*x* described above can also be used as components in *set*. All other characters in *set* represent themselves. For example, `[%w_]` (or `[_%w]`) represents all alphanumeric characters plus the underscore, `[0-7]` represents the octal digits, and `[0-7%l%-]` represents the octal digits plus the lowercase letters plus the '-' character.
  - The interaction between ranges and classes is not defined. Therefore, patterns like `[%a-z]` or `[a-%%]` have no meaning.
- **`[^`*set*`]`:** represents the complement of *set*, where *set* is interpreted as above.

For all classes represented by single letters (`%a`, `%c`, etc.), the corresponding uppercase letter represents the complement of the class. For instance, `%S` represents all non-space characters.

The definitions of letter, space, and other character groups depend on the current locale. In particular, the class `[a-z]` may not be equivalent to `%l`.

Pattern Item:

A *pattern item* can be

- a single character class, which matches any single character in the class;
- a single character class followed by '*', which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '+', which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '-', which also matches 0 or more repetitions of characters in the class. Unlike '*', these repetition items will always match the *shortest* possible sequence;
- a single character class followed by '?', which matches 0 or 1 occurrence of a character in the class;
- `%`*n*, for *n* between 1 and 9; such item matches a substring equal to the *n*-th captured string (see below);
- `%b`*xy*, where *x* and *y* are two distinct characters; such item matches strings that start with *x*, end with *y*, and where the *x* and *y* are *balanced*. This means that, if one reads the string from left to right, counting *+1* for an *x* and *-1* for a *y*, the ending *y* is the first *y* where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.

Pattern:

A *pattern* is a sequence of pattern items. A '^' at the beginning of a pattern anchors the match at the beginning of the subject string. A '$' at the end of a pattern anchors the match at the end of the subject string. At other positions, '^' and '$' have no special meaning and represent themselves.

Captures:

A pattern can contain sub-patterns enclosed in parentheses; they describe *captures*. When a match succeeds, the substrings of the subject string that match captures are stored (*captured*) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern `"(a*(.)%w(%s*))"`, the part of the string matching `"a*(.)%w(%s*)"` is stored as the first capture (and therefore has number 1); the character matching `"."` is captured with number 2, and the part matching "`%s*`" has number 3.

As a special case, the empty capture `()` captures the current string position (a number). For instance, if we apply the pattern `"()aa()"` on the string `"flaaap"`, there will be two captures: 3 and 5.

A pattern cannot contain embedded zeros. Use `%z` instead.

5 - Table Manipulation

This library provides generic functions for table manipulation. It provides all its functions inside the table `table`.

Most functions in the table library assume that the table represents an array or a list. For these functions, when we talk about the "length" of a table we mean the result of the length operator.

`table.concat (table [, sep [, i [, j]]])`

Given an array where all elements are strings or numbers, returns `table[i]..sep..table[i+1] ··· sep..table[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is the length of the table. If `i` is greater than `j`, returns the empty string.

`table.insert (table, [pos,] value)`

Inserts element `value` at position `pos` in `table`, shifting up other elements to open space, if necessary. The default value for `pos` is `n+1`, where `n` is the length of the table (see §2.5.5), so that a call `table.insert(t,x)` inserts `x` at the end of table `t`.

`table.maxn (table)`

Returns the largest positive numerical index of the given table, or zero if the table has no positive numerical indices. (To do its job this function does a linear traversal of the whole table.)

`table.remove (table [, pos])`

Removes from `table` the element at position `pos`, shifting down other elements to close the space, if necessary. Returns the value of the removed element. The default value for `pos` is `n`, where `n` is the length of the table, so that a call `table.remove(t)` removes the last element of table `t`.

`table.sort (table [, comp])`

Sorts table elements in a given order, *in-place*, from `table[1]` to `table[n]`, where `n` is the length of the table. If `comp` is given, then it must be a function that receives two table elements, and returns true when the first is less than the second (so that `not comp(a[i+1],a[i])` will be true after the sort). If `comp` is not given, then the standard Lua operator `<` is used instead.

The sort algorithm is not stable; that is, elements considered equal by the given order may have their relative positions changed by the sort.

6 - Mathematical Functions

This library is an interface to the standard C math library. It provides all its functions inside the table `math`.

`math.abs (x)`

Returns the absolute value of `x`.

`math.acos (x)`

Returns the arc cosine of `x` (in radians).

`math.asin (x)`

Returns the arc sine of `x` (in radians).

`math.atan (x)`

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

`math.atan2 (y, x)`

Returns the arc tangent of `y/x` (in radians), but uses the signs of both parameters to find the quadrant of the result. (It also handles correctly the case of `x` being zero.)

`math.ceil (x)`

Returns the smallest integer larger than or equal to `x`.

`math.cos (x)`

Returns the cosine of `x` (assumed to be in radians).

`math.cosh (x)`

Returns the hyperbolic cosine of `x`.

`math.deg (x)`

Returns the angle `x` (given in radians) in degrees.

`math.exp (x)`

Returns the value e^x.

`math.floor (x)`

Returns the largest integer smaller than or equal to `x`.

`math.fmod (x, y)`

Returns the remainder of the division of `x` by `y` that rounds the quotient towards zero.

`math.frexp (x)`

Returns `m` and `e` such that $x = m2^e$, `e` is an integer and the absolute value of `m` is in the range $[0.5, 1)$ (or zero when `x` is zero).

`math.huge`

The value `HUGE_VAL`, a value larger than or equal to any other numerical value.

`math.ldexp (m, e)`

Returns $m2^e$ (`e` should be an integer).

`math.log (x)`

Returns the natural logarithm of `x`.

`math.log10 (x)`

Returns the base-10 logarithm of `x`.

`math.max (x, ···)`

Returns the maximum value among its arguments.

`math.min (x, ···)`

Returns the minimum value among its arguments.

`math.modf (x)`

Returns two numbers, the integral part of `x` and the fractional part of `x`.

`math.pi`

The value of *pi*.

`math.pow (x, y)`

Returns x^y. (You can also use the expression `x^y` to compute this value.)

`math.rad (x)`

Returns the angle `x` (given in degrees) in radians.

`math.random ([m [, n]])`

This function is an interface to the simple pseudo-random generator function `rand` provided by ANSI C. (No guarantees can be given for its statistical properties.)

When called without arguments, returns a uniform pseudo-random real number in the range *[0,1)*. When called with an integer number `m`, `math.random` returns a uniform pseudo-random integer in the range *[1, m]*. When called with two integer numbers `m` and `n`, `math.random` returns a uniform pseudo-random integer in the range *[m, n]*.

`math.randomseed (x)`

Sets `x` as the "seed" for the pseudo-random generator: equal seeds produce equal sequences of numbers.

`math.sin (x)`

Returns the sine of `x` (assumed to be in radians).

`math.sinh (x)`

Returns the hyperbolic sine of `x`.

`math.sqrt (x)`

Returns the square root of `x`. (You can also use the expression `x^0.5` to compute this value.)

`math.tan (x)`

Returns the tangent of `x` (assumed to be in radians).

`math.tanh (x)`

Returns the hyperbolic tangent of `x`.

7 - Input and Output Facilities

The I/O library provides two different styles for file manipulation. The first one uses implicit file descriptors; that is, there are operations to set a default input file and a default output file, and all input/output operations are over these default files. The second style uses explicit file descriptors.

When using implicit file descriptors, all operations are supplied by table `io`. When using explicit file descriptors, the operation io.open returns a file descriptor and then all operations are supplied as methods of the file descriptor.

The table `io` also provides three predefined file descriptors with their usual meanings from C: `io.stdin`, `io.stdout`, and `io.stderr`. The I/O library never closes these files.

Unless otherwise stated, all I/O functions return **nil** on failure (plus an error message as a second result and a system-dependent error code as a third result) and some value different from **nil** on success.

`io.close ([file])`

Equivalent to `file:close()`. Without a `file`, closes the default output file.

`io.flush ()`

Equivalent to `file:flush` over the default output file.

`io.input ([file])`

When called with a file name, it opens the named file (in text mode), and sets its handle as the default input file. When called with a file handle, it simply sets this file handle as the default input file. When called without parameters, it returns the current default input file.

In case of errors this function raises the error, instead of returning an error code.

`io.lines ([filename])`

Opens the given file name in read mode and returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
for line in io.lines(filename) do body end
```

will iterate over all lines of the file. When the iterator function detects the end of file, it returns **nil** (to finish the loop) and automatically closes the file.

The call `io.lines()` (with no file name) is equivalent to `io.input():lines()`; that is, it iterates over the lines of the default input file. In this case it does not close the file when the loop ends.

`io.open (filename [, mode])`

This function opens a file, in the mode specified in the string `mode`. It returns a new file handle, or, in case of errors, **nil** plus an error message.

The `mode` string can be any of the following:

- **"r":** read mode (the default);
- **"w":** write mode;
- **"a":** append mode;
- **"r+":** update mode, all previous data is preserved;
- **"w+":** update mode, all previous data is erased;
- **"a+":** append update mode, previous data is preserved, writing is only allowed at the end of file.

The `mode` string can also have a 'b' at the end, which is needed in some systems to open the file in binary mode. This string is exactly what is used in the standard C function `fopen`.

`io.output ([file])`

Similar to io.input, but operates over the default output file.

`io.popen (prog [, mode])`

Starts program `prog` in a separated process and returns a file handle that you can use to read data from this program (if `mode` is `"r"`, the default) or to write data to this program (if `mode` is `"w"`).

This function is system dependent and is not available on all platforms.

`io.read (···)`

Equivalent to `io.input():read`.

`io.tmpfile ()`

Returns a handle for a temporary file. This file is opened in update mode and it is automatically removed when the program ends.

`io.type (obj)`

Checks whether `obj` is a valid file handle. Returns the string `"file"` if `obj` is an open file handle, `"closed file"` if `obj` is a closed file handle, or **nil** if `obj` is not a file handle.

`io.write (···)`

Equivalent to `io.output():write`.

`file:close ()`

Closes `file`. Note that files are automatically closed when their handles are garbage collected, but that takes an unpredictable amount of time to happen.

`file:flush ()`

Saves any written data to `file`.

`file:lines ()`

Returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
for line in file:lines() do body end
```

will iterate over all lines of the file. (Unlike io.lines, this function does not close the file when the loop ends.)

`file:read (···)`

Reads the file `file`, according to the given formats, which specify what to read. For each format, the function returns a string (or a number) with the characters read, or **nil** if it cannot read data with the specified format. When called without formats, it uses a default format that reads the entire next line (see below).

The available formats are

- **"*n":** reads a number; this is the only format that returns a number instead of a string.
- **"*a":** reads the whole file, starting at the current position. On end of file, it returns the empty string.
- **"*l":** reads the next line (skipping the end of line), returning **nil** on end of file. This is the default format.
- ***number*:** reads a string with up to this number of characters, returning **nil** on end of file. If number is zero, it reads nothing and returns an empty string, or **nil** on end of file.

`file:seek ([whence] [, offset])`

Sets and gets the file position, measured from the beginning of the file, to the position given by `offset` plus a base specified by the string `whence`, as follows:

- **"set":** base is position 0 (beginning of the file);
- **"cur":** base is current position;
- **"end":** base is end of file;

In case of success, function `seek` returns the final file position, measured in bytes from the beginning of the file. If this function fails, it returns **nil**, plus a string describing the error.

The default value for `whence` is `"cur"`, and for `offset` is 0. Therefore, the call `file:seek()` returns the current file position, without changing it; the call `file:seek("set")` sets the position to the beginning of the file (and returns 0); and the call `file:seek("end")` sets the position to the end of the file, and returns its size.

`file:setvbuf (mode [, size])`

Sets the buffering mode for an output file. There are three available modes:

- **"no":** no buffering; the result of any output operation appears immediately.
- **"full":** full buffering; output operation is performed only when the buffer is full (or when you explicitly `flush` the file (see io.flush)).
- **"line":** line buffering; output is buffered until a newline is output or there is any input from some special files (such as a terminal device).

For the last two cases, `size` specifies the size of the buffer, in bytes. The default is an appropriate size.

`file:write (···)`

Writes the value of each of its arguments to the `file`. The arguments must be strings or numbers. To write other values, use tostring or string.format before `write`.

Operating System Facilities

os.date ([format [, time]])

Returns a string or a table containing date and time, formatted according to the given string `format`.

If the `time` argument is present, this is the time to be formatted (see the os.time function for a description of this value). Otherwise, `date` formats the current time.

If `format` starts with '!', then the date is formatted in Coordinated Universal Time. After this optional character, if `format` is the string "*t", then `date` returns a table with the following fields: `year` (four digits), `month` (1--12), `day` (1--31), `hour` (0--23), `min` (0--59), `sec` (0--61), `wday` (weekday, Sunday is 1), `yday` (day of the year), and `isdst` (daylight saving flag, a boolean).

If `format` is not "*t", then `date` returns the date as a string, formatted according to the same rules as the C function `strftime`.

When called without arguments, `date` returns a reasonable date and time representation that depends on the host system and on the current locale (that is, `os.date()` is equivalent to `os.date("%c")`).

---

os.difftime (t2, t1)

Returns the number of seconds from time `t1` to time `t2`. In POSIX, Windows, and some other systems, this value is exactly `t2-t1`.

---

os.execute ([command])

This function is equivalent to the C function `system`. It passes `command` to be executed by an operating system shell. It returns a status code, which is system-dependent. If `command` is absent, then it returns nonzero if a shell is available and zero otherwise.

---

os.exit ([code])

Calls the C function `exit`, with an optional `code`, to terminate the host program. The default value for `code` is the success code.

---

os.getenv (varname)

Returns the value of the process environment variable `varname`, or **nil** if the variable is not defined.

---

os.remove (filename)

Deletes the file or directory with the given name. Directories must be empty to be removed. If this function fails, it returns **nil**, plus a string describing the error.

---

os.rename (oldname, newname)

Renames file or directory named `oldname` to `newname`. If this function fails, it returns **nil**, plus a string describing the error.

---

os.time ([table])

Returns the current time when called without arguments, or a time representing the date and time specified by the given table. This table must have fields `year`, `month`, and `day`, and may have fields `hour`, `min`, `sec`, and `isdst` (for a description of these fields, see the os.date function).

The returned value is a number, whose meaning depends on your system. In POSIX, Windows, and some other systems, this number counts the number of seconds since some given start time (the "epoch"). In other systems, the meaning is not specified, and the number returned by `time` can be used only as an argument to `date` and `difftime`.

---

os.tmpname ()

Returns a string with a file name that can be used for a temporary file. The file must be explicitly opened before its use and explicitly removed when no longer needed.

On some systems (POSIX), this function also creates a file with that name, to avoid security risks. (Someone else might create the file with wrong permissions in the time between getting the name and creating the file.) You still have to open the file to use it and to remove it (even if you do not use it).

When possible, you may prefer to use io.tmpfile, which automatically removes the file when the program ends.

`toboolean (value)`

Converts given value to boolean using following rules: **nil**, **false**, 0, empty string, '0' string are treated as **false**, everything else as **true**.

`sleep (delay)`

Delays script execution by delay seconds

`alert (fmt, ...)`

Adds a message to Alert database. This function behaves exactly as string.format.

`log (...)`

Logs any number of variables in human-readable format.

Bit operators

`bit.bnot (value)`

Binary not

`bit.band (x1 [, x2...])`

Binary and between any number of variables

`bit.bor (x1 [, x2...])`

Binary and between any number of variables

`bit.bxor (x1 [, x2...])`

Binary and between any number of variables

`bit.lshift (value, shift)`

Left binary shift

`bit.rshift (value, shift)`

Right binary shift

Conversion

Compatibility layer: lmcore is an alias of cnv.

`cnv.strtohex (str)`

Converts given binary string to a hex-encoded string.

`cnv.hextostr (hex [, keepnulls])`

Converts given hex-encoded string to a binary string. NULL characters are ignored by default, but can be included by setting second parameter to true.

`cnv.tonumber (value)`

Converts given value to a number using following rules: **true** is 1, **false** is 0, numbers and numeric strings are treated as is, everything else is **nil**.

Data type functions

Compatibility layer: knxdatatype is an alias of dpt.

`dpt.decode(value, datatype)`

Converts hex-encoded value to Lua variable according to datatype passed.

Data types

1. 1 bit (boolean) - **dt.bool** — boolean
2. 2 bit (1 bit controlled) - **dt.bit2** — number
3. 4 bit (3 bit controlled) - **dt.bit4** — number
4. 1 byte ASCII character - **dt.char** — string
5. 1 byte unsigned integer - **dt.uint8** — number
   - 1 byte scaling - **dt.scale** — number
6. 1 byte signed integer - **dt.int8** — number
7. 2 byte unsigned integer - **dt.uint16** — number
8. 2 byte signed integer - **dt.int16** — number
9. 2 byte floating point - **dt.float16** — number
10. 3 byte time / day - **dt.time** — table with the following items:
    - day — number (0-7)
    - hour — number (0-23)
    - minute — number (0-59)
    - second — number (0-59)
11. 11. 3 byte date - **dt.date** — table with the following items:
    - day — number (1-31)
    - month — number (1-12)
    - year — number (1990-2089)
12. 4 byte unsigned integer - **dt.uint32** — number
13. 4 byte signed integer - **dt.int32** — number
14. 4 byte floating point - **dt.float32** — number
15. 4 byte access control - **dt.access** — number, currently not fully supported
16. 14 byte ASCII string - **dt.string** — string, null characters ('\0') are discarded during decoding

Bus object access

grp provides simplified access to the objects stored in the database and group address request helpers.

Most functions use alias parameter — object group address or unique object name. (e.g. '1/1/1' or 'My object')

`grp.getvalue ()`

Returns value for given alias or **nil** when object cannot be found.

`grp.find (alias)`

Returns single object with the for the given alias. Object value will be decoded automatically only if the data type has been specified in the 'Objects' module.

Returns **nil** when object cannot be found, otherwise it returns table with the following items:

- address — object group address
- updatetime — latest update time in UNIX timestamp format. Use os.date() to convert to readable date formats

When object data type has been specified in the 'Objects' module the following fields are available:

- name — unique object name
- datatype — object data type as specified by user
- decoded — set to **true** when decoded value is available
- value — decoded object value

---

grp.tag (tags [, mode])

Returns table containing objects with given tag. Tags parameter can be either table or a string. Mode parameter can be either 'all' (return objects that have all of the given tags) or 'any' (default — returns objects that have any of the given tags). You can use object functions on the returned table.

---

grp.alias ()

Converts group address to object name or name to address. Returns **nil** when object cannot be found.

---

grp.write (alias, value [, datatype])

Sends group write request to the given alias. Data type is taken from the database if not specified as third parameter. Returns boolean as the result.

---

grp.response (alias, value [, datatype])

Similar to grp.write. Sends group response request to the given alias.

---

grp.read (alias)

Sends group read request to the given alias. Note: this function returns immediately and cannot be used to return the result of read request. Use event-based script instead.

---

grp.update (alias, value [, datatype])

Similar to grp.write, but does not send new value to the bus. Useful for objects that are used only in visualization.

Objects received by using grp.find (alias) or grp.tag (tags, mode) have the following functions.

Note: Always check that the returned object was found otherwise calling these functions will result in an error. See the example below.

---

object:write (value, datatype)

Sends group write request to object's group address. Data type is taken from the database if not specified as second parameter. Returns boolean as the result.

---

object:response (value [, datatype])

Similar to object:write. Sends group response request to object's group address.

---

object:read ()

Sends group read request to object's group address. Note: this function returns immediately and cannot be used to return the result of read request. Use event-based script instead.

---

object:update (value [, datatype])

Similar to object:write, but does not send new value to the bus. Useful for objects that are used only in visualization.

Time functions

---

os.microtime ()

Returns two values: current timestamp in seconds and timestamp fraction in nanoseconds

---

os.udifftime (sec, usec)

Returns time difference as floating point value between now and timestamp components passed to this function (seconds, nanoseconds)

---

os.sleep (delay)

Delays script execution by delay seconds.

Script data storage

Storage object provides persistent key-value data storage for user scripts. Only the following Lua data types are supported:

- boolean
- number
- string
- table

`storage.set (key, value)`

Sets new value for the given key. Old value is overwritten. Returns boolean as the result and an optional error string.

`storage.get (key [, default])`

Gets value for the given key or returns default value (**nil** if not specified) if key is not found in the data storage.

Data serialization

`serialize.encode (value)`

Generates a storable representation of a value.

`serialize.decode (value)`

Creates a Lua value from a stored representation.

String functions

`string.trim (str)`

Trims the leading and trailing spaces off a given string.

`string.split (str, sep)`

Splits string by given separator string. Returns Lua table.

Input and output functions

`io.exists (path)`

Checks if given path (file or directory) exists. Return boolean.

`io.readfile (file)`

Reads whole file at once. Return file contents as a string on success or **nil** on error.

`io.writefile (file, data)`

Writes given data to a file. Data can be either a value convertible to string or a table of such values. When data is a table then each table item is terminated by a new line character. Return boolean as write result when file can be open for writing or **nil** when file cannot be accessed.

JSON library

Note: json is not loaded by default, use `require('json')` before calling any functions from this library.

`json.encode (value)`

Converts Lua variable to JSON string. Script execution is stopped in case of an error.

`json.pencode (value)`

Converts Lua variable to JSON string in protected mode, returns **nil** on error.

`json.decode (value)`

Converts JSON string to Lua variable. Script execution is stopped in case of an error.

`json.pdecode (value)`

Converts JSON string to Lua variable in protected mode, returns **nil** on error.