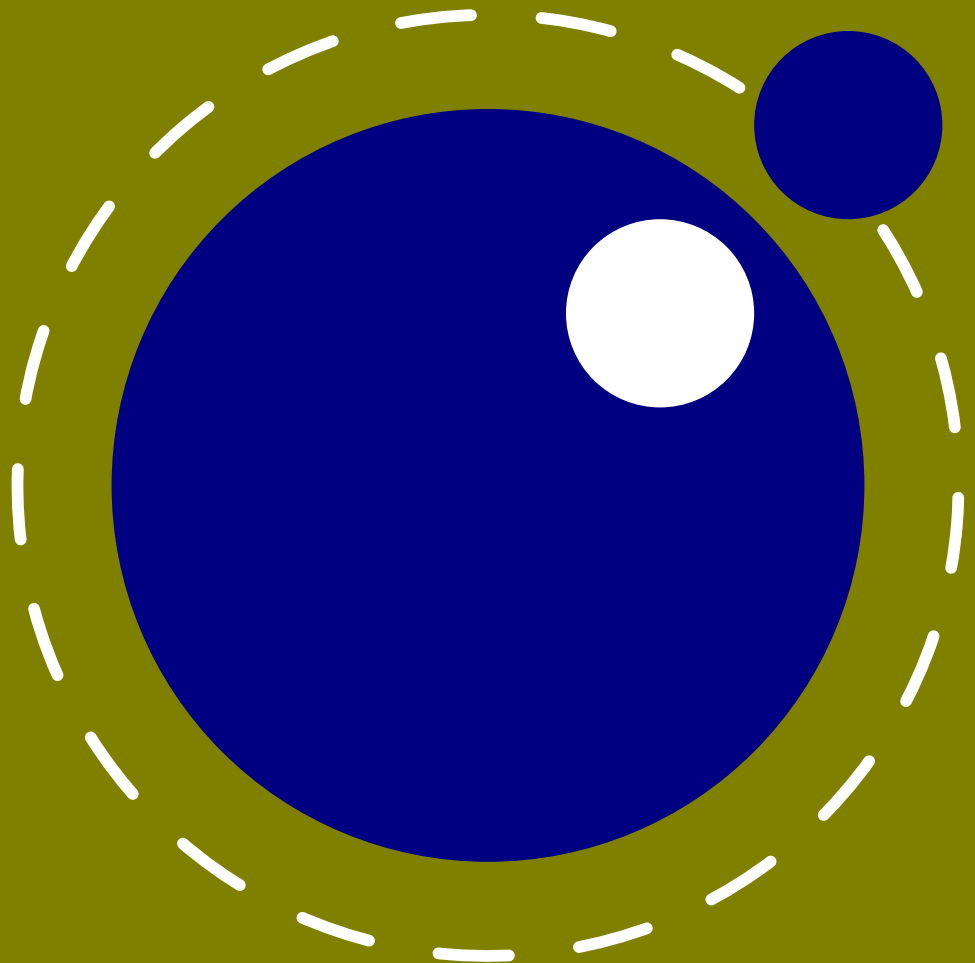
LuaTEX Reference

beta 0.82.0



LuaTEX Reference Manual

more info : www.luatex.org
version : October 18, 2015

Contents

Introduction

This book will eventually become the reference manual of LuaTEX. At the moment, it simply reports the behavior of the executable matching the snapshot or beta release date in the title page.

Features may come and go. The current version of LuaTEX can be used for production (in fact it is used in production by the authors) but users cannot depend on complete stability, nor on functionality staying the same. This means that when you update your binary, you also need to check if something fundamental has changed. Normally this is communicated in articles or messages to a mailing list. We're still not at version 1 but when we reach that state the interface will be stable. Of course we then can decide to move towards version 2 with different properties.

Don't expect LuaTEX to behave the same as pdfTEX! Although the core functionality of that 8 bit engine is present, LuaTEX can behave different due to not only its 32 bit character: there is native utf input, support for wide fonts, and the math machinery is tuned for OpenType math. Also, the log output can differ (and will likely differ more as we move forward).

LuaTEX consists of a number of interrelated but (still) distinguishable parts. The organization of the source code is adapted so that it cna glue all these components together. We continue cleaning up side effects of the accumulated code in TEX engines (especially code that is not needed any longer).

- Most of pdfTEX version 1.40.9, converted to C (with patches from later releases). Some experimental features have been removed and some utility macros are not inherited as their functionality can be done in Lua. We still use the `\pdf*` primitive namespace.
- The direction model and some other bits from Aleph RC4 (derived from Omega) is included. The related primitives are part of core LuaTEX.
- We currently use Lua 5.2.*. At some point we might decide to move to 5.3.* but that is yet to be decided.
- There are few Lua libraries that we consider part of the core Lua machinery.
- There are additional Lua libraries that interface to the internals of TEX.
- There are various TEX extensions but only those that cannot be done using the Lua interfaces.
- The fontloader uses parts of FontForge 2008.11.17 combined with additionaL code specific for usage in a TEX engine.
- the MetaPost library

Neither Aleph's I/O translation processes, nor tcx files, nor encTEX can be used, these encoding-related functions are superseded by a Lua-based solution (reader callbacks).

The yearly TEXLive version is the stable version, any version between them is considered beta. Keep in mind that new (or changed) features also need to be reflected in the macro package that you use.

LuaTEX : Version 82.0
ConTEXt : 2015.10.18 18:24
timestamp : October 18, 2015

1 LUA general

1.1 Initialization

1.1.1 LUATEX as a LUA interpreter

There are some situations that make LuaTEX behave like a standalone Lua interpreter:

- if a `--luaonly` option is given on the commandline, or
- if the executable is named `texlua` or `luatexlua`, or
- if the only non-option argument (file) on the commandline has the extension `lua` or `luc`.

In this mode, it will set Lua's `arg[0]` to the found script name, pushing preceding options in negative values and the rest of the commandline in the positive values, just like the Lua interpreter.

LuaTEX will exit immediately after executing the specified Lua script and is, in effect, a somewhat bulky standalone Lua interpreter with a bunch of extra preloaded libraries.

1.1.2 LUATEX as a LUA byte compiler

There are two situations that make LuaTEX behave like the Lua byte compiler:

- if a `--luaconly` option is given on the commandline, or
- if the executable is named `texluac`

In this mode, LuaTEX is exactly like `luac` from the standalone Lua distribution, except that it does not have the `-l` switch, and that it accepts (but ignores) the `--luaconly` switch.

1.1.3 Other commandline processing

When the LuaTEX executable starts, it looks for the `--lua` commandline option. If there is no `--lua` option, the commandline is interpreted in a similar fashion as in traditional pdfTEX and Aleph. Some options are accepted but have no consequence. The following command-line options are understood:

`--fmt=FORMAT`	load the format file `FORMAT`
`--lua=FILE`	load and execute a Lua initialization script
`--safer`	disable easily exploitable Lua commands
`--nosocket`	disable the Lua socket library
`--help`	display help and exit
`--ini`	be iniluatex, for dumping formats
`--interaction=STRING`	set interaction mode: `batchmode`, `nonstopmode` `scrollmode` or `errorstopmode`
`--halt-on-error`	stop processing at the first error

`--kpathsea-debug=NUMBER`	set path searching debugging flags according to the bits of `NUMBER`
`--progname=STRING`	set the program name to `STRING`
`--version`	display version and exit
`--credits`	display credits and exit
`--recorder`	enable filename recorder
`--etex`	ignored
`--output-comment=STRING`	use `STRING` for dvi file comment instead of date (no effect for pdf)
`--output-directory=DIR`	use `DIR` as the directory to write files to
`--draftmode`	switch on draft mode i.e. generate no output in pdf mode
`--output-format=FORMAT`	use `FORMAT` for job output; `FORMAT` is `dvi` or `pdf`
`--[no-]shell-escape`	disable/enable `\write 18{SHELL COMMAND}`
`--enable-write18`	enable `\write 18{SHELL COMMAND}`
`--disable-write18`	disable `\write 18{SHELL COMMAND}`
`--shell-restricted`	restrict `\write 18` to a list of commands given in `texmf.cnf`
`--debug-format`	enable format debugging
`--[no-]file-line-error`	disable/enable `file:line:error` style messages
`--[no-]file-line-error-style`	aliases of `--[no-]file-line-error`
`--jobname=STRING`	set the job name to `STRING`
`--[no-]parse-first-line`	ignored
`--translate-file=`	ignored
`--default-translate-file=`	ignored
`--8bit`	ignored
`--[no-]mktex=FMT`	disable/enable `mktexFMT` generation with `FMT` is `tex` or `tfm`
`--synctex=NUMBER`	enable `synctex`

A note on the creation of the various temporary files and the `\jobname`. The value to use for `\jobname` is decided as follows:

- If `--jobname` is given on the command line, its argument will be the value for `\jobname`, without any changes. The argument will not be used for actual input so it need not exist. The `--jobname` switch only controls the `\jobname` setting.
- Otherwise, `\jobname` will be the name of the first file that is read from the file system, with any path components and the last extension (the part following the last `.`) stripped off.
- An exception to the previous point: if the command line goes into interactive mode (by starting with a command) and there are no files input via `\everyjob` either, then the `\jobname` is set to `texput` as a last resort.

The file names for output files that are generated automatically are created by attaching the proper extension (`.log`, `.pdf`, etc.) to the found `\jobname`. These files are created in the directory pointed to by `--output-directory`, or in the current directory, if that switch is not present.

Without the `--lua` option, command line processing works like it does in any other web2c-based typesetting engine, except that LuaTEX has a few extra switches.

If the `--lua` option is present, LuaTEX will enter an alternative mode of commandline processing in comparison to the standard web2c programs.

In this mode, a small series of actions is taken in order. First, it will parse the commandline as usual, but it will only interpret a small subset of the options immediately: `--safer`, `--nosocket`, `--[no-]shell-escape`, `--enable-write18`, `--disable-write18`, `--shell-restricted`, `--help`, `--version`, and `--credits`.

Now it searches for the requested Lua initialization script. If it cannot be found using the actual name given on the commandline, a second attempt is made by prepending the value of the environment variable `LUATEXDIR`, if that variable is defined in the environment.

Then it checks the various safety switches. You can use those to disable some Lua commands that can easily be abused by a malicious document. At the moment, `--safer` `nil`s the following functions:

library	**functions**
`os`	`execute exec setenv rename remove tmpdir`
`io`	`popen output tmpfile`
`lfs`	`rmdir mkdir chdir lock touch`

Furthermore, it disables loading of compiled Lua libraries and it makes `io.open()` fail on files that are opened for anything besides reading.

`--nosocket` makes the socket library unavailable, so that Lua cannot use networking.

The switches `--[no-]shell-escape`, `--[enable|disable]-write18`, and `--shell-restricted` have the same effects as in pdfTEX, and additionally make `io.popen()`, `os.execute`, `os.exec` and `os.spawn` adhere to the requested option.

Next the initialization script is loaded and executed. From within the script, the entire commandline is available in the Lua table `arg`, beginning with `arg[0]`, containing the name of the executable. As consequence, the warning about unrecognized option is suppressed.

Commandline processing happens very early on. So early, in fact, that none of TEX's initializations have taken place yet. For that reason, the tables that deal with typesetting, like `tex`, `token`, `node` and `pdf`, are off-limits during the execution of the startup file (they are nilled). Special care is taken that `texio.write` and `texio.write_nl` function properly, so that you can at least report your actions to the log file when (and if) it eventually becomes opened (note that TEX does not even know its `\jobname` yet at this point). See chapter ?? for more information about the LuaTEX-specific Lua extension tables.

Everything you do in the Lua initialization script will remain visible during the rest of the run, with the exception of the aforementioned `tex`, `token`, `node` and `pdf` tables: those will be initialized to their documented state after the execution of the script. You should not store anything in variables or within tables with these four global names, as they will be overwritten completely.

We recommend you use the startup file only for your own TEX-independent initializations (if you need any), to parse the commandline, set values in the `texconfig` table, and register the callbacks you need.

LuaTEX allows some of the commandline options to be overridden by reading values from the `texconfig` table at the end of script execution (see the description of the `texconfig` table later on in this document for more details on which ones exactly).

Unless the `texconfig` table tells LuaTEX not to initialize kpathsea at all (set `texconfig.kpse_init` to `false` for that), LuaTEX acts on some more commandline options after the

initialization script is finished: in order to initialize the built-in kpathsea library properly, LuaTEX needs to know the correct program name to use, and for that it needs to check `--progname`, or `--ini` and `--fmt`, if `--progname` is missing.

1.2 LUA behaviour

Luas `tonumber` function may return values in scientific notation, thereby confusing the TEX end of things when it is used as the right-hand side of an assignment to a `\dimen` or `\count`.

Loading dynamic Lua libraries will fail if there are two Lua libraries loaded at the same time (which will typically happen on `win32`, because there is one Lua 5.2 inside LuaTEX, and another will likely be linked to the dll file of the module itself). We plan to fix that later by switching LuaTEX itself to using de dll version of Lua 5.2 inside LuaTEX instead of including a static version in the binary.

LuaTEX is able to use the kpathsea library to find `require()`d modules. For this purpose, `package.searchers[2]` is replaced by a different loader function, that decides at runtime whether to use kpathsea or the built-in core Lua function. It uses kpathsea when that is already initialized at that point in time, otherwise it reverts to using the normal `package.path` loader.

Initialization of kpathsea can happen either implicitly (when LuaTEX starts up and the startup script has not set `texconfig.kpse_init` to false), or explicitly by calling the Lua function `kpse.set_program_name()`.

LuaTEX is able to use dynamically loadable Lua libraries, unless `--safer` was given as an option on the command line. For this purpose, `package.searchers[3]` is replaced by a different loader function, that decides at runtime whether to use kpathsea or the built-in core Lua function. It uses kpathsea when that is already initialized at that point in time, otherwise it reverts to using the normal `package.cpath` loader.

This functionality required an extension to kpathsea:

> There is a new kpathsea file format: `kpse_clua_format` that searches for files with extension `.dll` and `.so`. The `texmf.cnf` setting for this variable is `CLUAINPUTS`, and by default it has this value:
>
> ```
> CLUAINPUTS=.:$SELFAUTOLOC/lib/{$progname,$engine,}/lua//
> ```
>
> This path is imperfect (it requires a tds subtree below the binaries directory), but the architecture has to be in the path somewhere, and the currently simplest way to do that is to search below the binaries directory only. Of course it no big deal to write an alternative loader and use that in a macro package.
>
> One level up (a `lib` directory parallel to `bin`) would have been nicer, but that is not doable because TEXLive uses a `bin/<arch>` structure.

In keeping with the other TEX-like programs in TEXLive, the two Lua functions `os.execute` and `io.popen`, as well as the two new functions `os.exec` and `os.spawn` that are explained below, take the value of `shell_escape` and/or `shell_escape_commands` in account. Whenever LuaTEX is run with the assumed intention to typeset a document (and by that we mean that it is called as `luatex`, as opposed to `texlua`, and that the commandline option `--luaonly` was not given), it will only run the four functions above if the matching `texmf.cnf` variable(s) or their `texconfig`

(see section ??) counterparts allow execution of the requested system command. In 'script interpreter' runs of LuaTEX, these settings have no effect, and all four functions function as normal.

The `f:read("*line")` and `f:lines()` functions from the io library have been adjusted so that they are line-ending neutral: any of `LF`, `CR` or `CR+LF` are acceptable line endings.

`luafilesystem` has been extended: there are two extra boolean functions (`lfs.isdir(filename)` and `lfs.isfile(filename)`) and one extra string field in its attributes table (`permissions`). There is an additional function `lfs.shortname()` which takes a file name and returns its short name on `win32` platforms. On other platforms, it just returns the given argument. The file name is not tested for existence. Finally, for non-`win32` platforms only, there is the new function `lfs.readlink()` hat takes an existing symbolic link as argument and returns its content. It returns an error on `win32`.

The `string` library has an extra function: `string.explode(s[,m])`. This function returns an array containing the string argument `s` split into sub-strings based on the value of the string argument `m`. The second argument is a string that is either empty (this splits the string into characters), a single character (this splits on each occurrence of that character, possibly introducing empty strings), or a single character followed by the plus sign `+` (this special version does not create empty sub-strings). The default value for `m` is ' +' (multiple spaces). Note: `m` is not hidden by surrounding braces as it would be if this function was written in TEX macros.

The `string` library also has six extra iterators that return strings piecemeal:

- `string.utfvalues(s)`: an integer value in the Unicode range
- `string.utfcharacters(s)`: a string with a single utf-8 token in it
- `string.characters(s)` a string containing one byte
- `string.characterpairs(s)` two strings each containing one byte or an empty second string if the string length was odd
- `string.bytes(s)` a single byte value
- `string.bytepairs(s)` two byte values or nil instead of a number as its second return value if the string length was odd

The `string.characterpairs()` and `string.bytepairs()` iterators are useful especially in the conversion of utf-16 encoded data into utf-8.

There is also a two-argument form of `string.dump()`. The second argument is a boolean which, if true, strips the symbols from the dumped data. This matches an extension made in `luajit`.

The `string` library functions `len`, `lower`, `sub` etc. are not Unicode-aware. For strings in the utf8 encoding, i.e., strings containing characters above code point 127, the corresponding functions from the `slnunicode` library can be used, e.g., `unicode.utf8.len`, `unicode.utf8.lower` etc. The exceptions are `unicode.utf8.find`, that always returns byte positions in a string, and `unicode.utf8.match` and `unicode.utf8.gmatch`. While the latter two functions in general *are* Unicode-aware, they fall-back to non-Unicode-aware behavior when using the empty capture `()` but other captures work as expected. For the interpretation of character classes in `unicode.utf8` functions refer to the library sources at http://luaforge.net/projects/sln. Version 5.3 of Lua will provide some native utf8 support.

The `os` library has a few extra functions and variables:

- `os.selfdir` is a variable that holds the directory path of the actual executable. For example: `\directlua{tex.sprint(os.selfdir)}`.
- `os.exec(commandline)` is a variation on `os.execute`. Here `commandline` can be either a single string or a single table.
  If the argument is a table: LuaTEX first checks if there is a value at integer index zero. If there is, this is the command to be executed. Otherwise, it will use the value at integer index one. (if neither are present, nothing at all happens).
  The set of consecutive values starting at integer 1 in the table are the arguments that are passed on to the command (the value at index 1 becomes `arg[0]`). The command is searched for in the execution path, so there is normally no need to pass on a fully qualified pathname.
  If the argument is a string, then it is automatically converted into a table by splitting on whitespace. In this case, it is impossible for the command and first argument to differ from each other.
  In the string argument format, whitespace can be protected by putting (part of) an argument inside single or double quotes. One layer of quotes is interpreted by LuaTEX, and all occurrences of `\"`, `\'` or `\\` within the quoted text are unescaped. In the table format, there is no string handling taking place.
  This function normally does not return control back to the Lua script: the command will replace the current process. However, it will return the two values `nil` and `'error'` if there was a problem while attempting to execute the command.
  On MS Windows, the current process is actually kept in memory until after the execution of the command has finished. This prevents crashes in situations where TEXLua scripts are run inside integrated TEX environments.
  The original reason for this command is that it cleans out the current process before starting the new one, making it especially useful for use in TEXLua.
- `os.spawn(commandline)` is a returning version of `os.exec`, with otherwise identical calling conventions.
  If the command ran ok, then the return value is the exit status of the command. Otherwise, it will return the two values `nil` and `'error'`.
- `os.setenv('key','value')` sets a variable in the environment. Passing `nil` instead of a value string will remove the variable.
- `os.env` is a hash table containing a dump of the variables and values in the process environment at the start of the run. It is writeable, but the actual environment is *not* updated automatically.
- `os.gettimeofday()` returns the current 'Unix time', but as a float. This function is not available on the SunOS platforms, so do not use this function for portable documents.
- `os.times()`returns the current process times according to the Unix C library function 'times'. This function is not available on the MS Windows and SunOS platforms, so do not use this function for portable documents.
- `os.tmpdir()` creates a directory in the 'current directory' with the name `luatex.XXXXXX` where the `X`-es are replaced by a unique string. The function also returns this string, so you can `lfs.chdir()` into it, or `nil` if it failed to create the directory. The user is responsible for cleaning up at the end of the run, it does not happen automatically.
- `os.type` is a string that gives a global indication of the class of operating system. The possible values are currently `windows`, `unix`, and `msdos` (you are unlikely to find this value 'in the wild').
- `os.name` is a string that gives a more precise indication of the operating system. These pos-

sible values are not yet fixed, and for `os.type` values `windows` and `msdos`, the `os.name` values are simply `windows` and `msdos`
The list for the type `unix` is more precise: `linux`, `freebsd`, `kfreebsd`, `cygwin`, `openbsd`, `solaris`, `sunos` (pre-solaris), `hpux`, `irix`, `macosx`, `gnu` (hurd), `bsd` (unknown, but bsd-like), `sysv` (unknown, but sysv-like), `generic` (unknown).
- `os.version` is planned as a future extension.
- `os.uname()` returns a table with specific operating system information acquired at runtime. The keys in the returned table are all string valued, and their names are: `sysname`, `machine`, `release`, `version`, and `nodename`.

In stock Lua, many things depend on the current locale. In LuaTEX, we can't do that, because it makes documents unportable. While LuaTEX is running if forces the following locale settings:

```
LC_CTYPE=C
LC_COLLATE=C
LC_NUMERIC=C
```

1.3 LUA modules

The implied use of the built-in Lua modules in this section is deprecated. If you want to use one of these libraries, please start your source file with a proper `require` line. At some point LuaTEX will switch to loading these modules on demand.

Some modules that are normally external to Lua are statically linked in with LuaTEX, because they offer useful functionality:

- `slnunicode`, from the `Selene` libraries, http://luaforge.net/projects/sln. (version 1.1) This library has been slightly extended so that the `unicode.utf8.*` functions also accept the first 256 values of plane 18. This is the range LuaTEX uses for raw binary output, as explained above.
- `luazip`, from the kepler project, http://www.keplerproject.org/luazip/. (version 1.2.1, but patched for compilation with Lua 5.2)
- `luafilesystem`, also from the kepler project, http://www.keplerproject.org/luafilesystem/. (version 1.5.0)
- `lpeg`, by Roberto Ierusalimschy, http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html. (version 0.10.2) This library is not Unicode-aware, but interprets strings on a byte-per-byte basis. This mainly means that `lpeg.S` cannot be used with utf characters encoded in more than two bytes, and thus `lpeg.S` will look for one of those two bytes when matching, not the combination of the two. The same is true for `lpeg.R`, although the latter will display an error message if used with multibyte characters. Therefore `lpeg.R('aä')` results in the message `bad argument #1 to 'R' (range must have two characters)`, since to `lpeg`, `ä` is two 'characters' (bytes), so `aä` totals three. In practice this is no real issue.
- `lzlib`, by Tiago Dionizio, http://luaforge.net/projects/lzlib/. (version 0.2)
- `md5`, by Roberto Ierusalimschy http://www.inf.puc-rio.br/~roberto/md5/md5-5/md5.html.
- `luasocket`, by Diego Nehab http://w3.impa.br/~diego/software/luasocket/ (version 2.0.2). The `.lua` support modules from `luasocket` are also preloaded inside the executable, there are no external file dependencies.