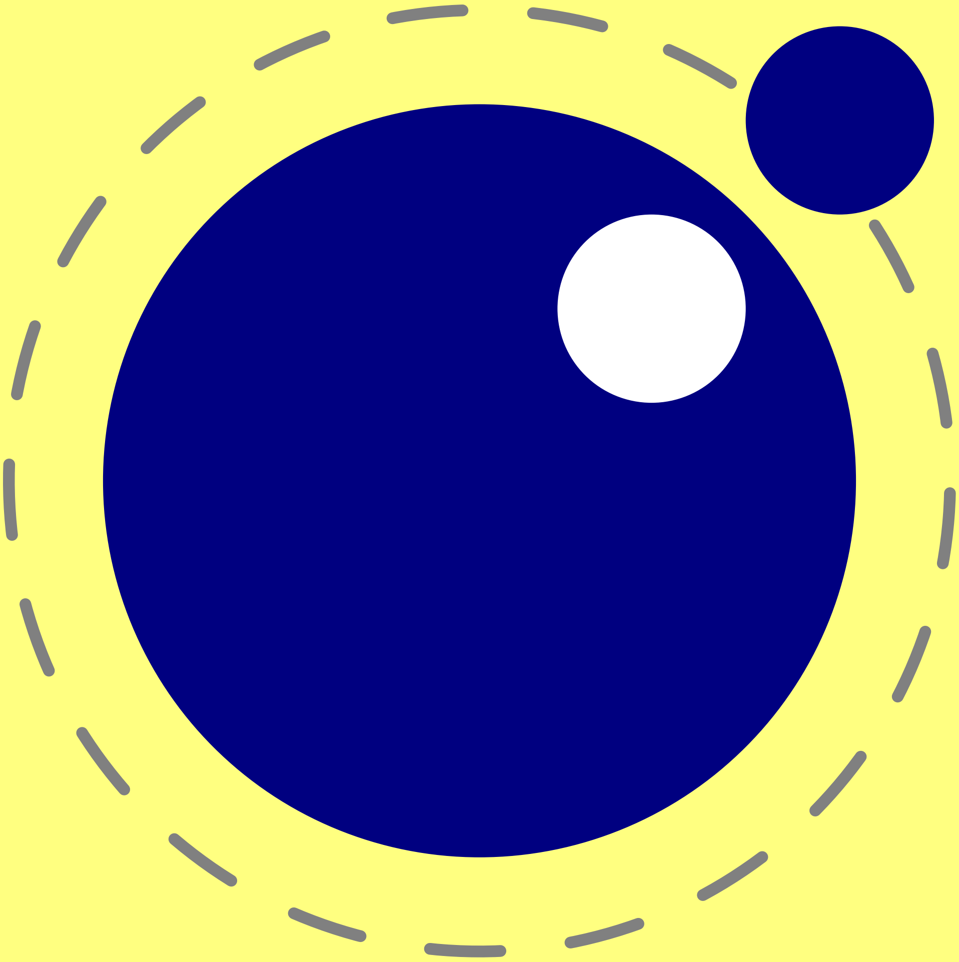
LuaTEX Reference

Snapshot 2007-03-09



Contents

1 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 date in the title page.

Features may come and go. The current version of LuaTeX is not meant for production and users cannot depend on functionality staying the same.

Nothing in the API is considered stable just yet. This manual therefore simply reflects the current state of the executable. ***Absolutely nothing*** on the following pages is set in stone. When the need arises, anything can (and will) be changed without prior notice.

If you are unhappy with this situation, wait for the public betas.

LuaTeX consists of a number of interrelated but (still) distinguishable parts:

- pdfTeX version 1.40.3
- Aleph RC4 (from the TeXLive repository)
- Functionality of ε-TeX 2.2
- Lua 5.1.1
- Dedicated lua libraries
- Various TeX extensions
- The (OpenType) Font Parser from FontForge 2006.12.20
- Compiled source code to glue it all together

LuaTeX has two separate identities:

1. When `\pdfoutput` is set to one, LuaTeX behaves like pdfTeX, with the addition of OTP processing and Aleph directionality commands.
2. When `\pdfoutput` is zero, LuaTeX behaves like Aleph with the addition of the micro-typography features. The pdfTeX commands that are not specific to the PDF output format should work.

In either mode, neither I/O translation processes, nor tcx files, nor enctex can be used. All these encoding-related functions are superseded by a Lua-based solution (`reader` callbacks).

2 Basic TEX enhancements

2.1 Unicode support

Text input and output is now considered to be Unicode text, so characters can use the full range of Unicode ($2^{20} + 2^{16}$ = "10FFFF = 1114111).

For now, it only makes sense to use values above the base plane ("FFFF) for \mathcode and \catcode assignments, since the hyphenation patterns are still limited to at the most 16-bit values, so the other commands will not know what to do with those high values.

Many primitives are affected by this. For instance, \char now accepts values between 0 and 1114111. This should not be a problem for well-behaved input files, but it could create incompatibilities for input that would have generated an error when processed by older TEX-based engines.

Primitive	Bits	Hex	Range
\char	21	"10FFFF	$(2^{20} + 2^{16})$
\chardef	21=21	"10FFFF="10FFFF	$(2^{20} + 2^{16}) = (2^{20} + 2^{16})$
\lccode	21=21	"10FFFF="10FFFF	$(2^{20} + 2^{16}) = (2^{20} + 2^{16})$
\uccode	21=21	"10FFFF="10FFFF	$(2^{20} + 2^{16}) = (2^{20} + 2^{16})$
\sfcode	21=15	"10FFFF="7FFF	$(2^{20} + 2^{16}) = (2^{15})$
\catcode	21=4	"10FFFF="F	$(2^{20} + 2^{16}) = (2^{4})$
\mathchardef	21=15	"10FFFF="8000	$(2^{20} + 2^{16}) = (2^{3} * 2^{8} * 2^{4})$
\mathcode	21=15	"10FFFF="8000	$(2^{20} + 2^{16}) = (2^{3} * 2^{8} * 2^{4})$
\delcode	21=27	"10FFFF="7FFFFFF	$(2^{20} + 2^{16}) = (2^{3} * 2^{4} * 2^{8} * 2^{4} * 2^{8})$

As far as the core engine is concerned, all input and output to text files is UTF-8 encoded. Input files can be preprocessed using the reader callback. This will be explained in a later chapter.

Output in byte-sized chunks can be achieved by using characters in the private use block that starts at index 1.113.856 ("10FF00). When the times comes to print a character $c >= 1.113.856$, LuaTEX will actually print the single byte corresponding to $c - 1.113.856$.

Output to the terminal uses ^^ notation for the lower control range ($c < 32$), with the exception of ^^I, ^^J and ^^M. These are considered 'safe' and therefore printed as-is.

Normalization of the Unicode input can be handled by a macro package during callback processing (will be explained below).

2.2 Wide math characters

Text is now extended up to the full Unicode range, but math mode deals mostly with glyphs in fonts directly, and fonts tend to be 16-bit at maximum.

Therefore, the math primitives from Aleph are kept mostly as-is, except for the ones that convert from input to math commands. The extended commands (with the 'o' prefix) accept 16-bit glyph indices in one of 256 possible families. The traditional TEX primitives are unchanged, their arguments are upscaled internally.

Primitive	Bits	Hex	Range
\mathchar	15	"7FFF	$(2^3 * 2^8 * 2^4)$
\delimiter	27	"7FFFFFF	$(2^3 * 2^4 * 2^8 * 2^4 * 2^8)$
\omathchar	27	"7FFFFFF	$(2^3 * 2^{16} * 2^8)$
\odelimiter	27+24	"7FFFFFF+"FFFFFF	$(2^3 * 2^8 * 2^{16})$+$(2^8 * 2^{16})$
\omathchardef	21=27	"10FFFF="8000000	$(2^{20} + 2^{16}) = (2^3 * 2^{16} * 2^8)$
\omathcode	21=27	"10FFFF="8000000	$(2^{20} + 2^{16}) = (2^3 * 2^{16} * 2^8)$
\odelcode	21=27+24	"10FFFF="7FFFFFF+ "FFFFFF	$(2^{20} + 2^{16}) = (2^3 * 2^8 * 2^{16})+ (2^8 * 2^{16})$

2.3 Extended register tables

All registers can be `<16-bit number>`, as in ALEPH. The affected commands are:

\count
\dimen
\skip
\muskip
\marks
\toks
\countdef
\dimendef
\skipdef
\muskipdef
\toksdef
\box
\unhbox
\unvbox
\copy
\unhcopy
\unvcopy
\wd
\ht
\dp
\setbox
\vsplit

2.4 Lua related primitives

In order to merge lua code with TEX input, a few new primitives are needed. LUATEX has support for 65536 separate lua interpreter states. States are automatically created based on the integer argument to the primitives `\directlua` and `\latelua`.

2.4.1 \directlua

The primitive `\directlua` is used to execute lua code. The syntax is

```
\directlua <16-bit number> <general text>
```

The <general text> is fed into the lua interpreter state indicated by the <16-bit number>. If the state does not exist yet, then it will be initialized automatically.

This command is expandable.

2.4.2 \latelua

`\latelua` stores lua code in a whatsit that will be processed inside the output routine. It's intended use is is very similar to `\pdfliteral`.

Within the lua code, you should use `pdf.print` to print stuff directly to the pdffile..

```
\latelua <16-bit number> <general text>
```

2.4.3 \luaescapestring

This primitive converts a TEX token string so that it can be safely used as the contents of a LUA string: embedded backslashes, double quotes and single quotes are escaped by prepending an extra token consisting of a backslash with catcode 12.

```
\luaescapestring <general text>
```

2.4.4 \luaclose

This primitive allows you to close a lua state, freeing all of its used memory.

```
\luaclose <16-bit number>
```

You cannot close lua state zero (0), any attempt to do so will be silently ignored.

States are only closed automatically when a fatal (out of memory) error occurs, but at that point LUATEX will exit anyway.

States are not closed immediately, but only when the output routine comes into play next (because there may be pending `\latelua` calls)

2.5 New ε-TEX primitives

2.5.1 \clearmarks

This primitive clears a marks class completely, resetting all three connected mark texts to empty.

```
    \clearmarks <16-bit number>
```

2.5.2 \formatname

`\formatname`'s syntax is identical to `\jobname`.

In initex, the expansion is empty. Otherwise, the expansion is the value that \jobname had during the initex run that dumped the currently loaded format.

2.5.3 \scantextokens

The syntax of \scantextokens is identical to \scantokens.

This is a slightly adapted version of ε-TEX's \scantokens. The differences are:

- The last (and usually only) line does not have a \endlinechar appended
- \scantextokens never raises an EOF error, and it does not execute \everyeof tokens.
- The 'while end of file' tests are not executed, allowing the expansion to end on a different grouping level or while a conditional is still incomplete

2.5.4 Catcode tables

Catcode tables are a new feature that allows you to switch to a predefined catcode regime in a single statement. You can have a practically unlimited number of different tables (at this moment up to 268,435,456. The limit depends on an array allocation).

The subsystem is backward compatible: if you never use the following commands, your document will not notice any difference in behavior compared to traditional TEX.

The contents of each catcode table is independent of any other catcode tables, and their contents is stored and retrieved from the format file.

2.5.4.1 \catcodetable

```
\catcodetable <28-bit number>
```

The \catcodetable switches to a different catcode table. Such a table has to be previously created using one of the two primitives below, or it has to be zero (table zero is initialized by initex)

2.5.4.2 \initcatcodetable

```
\initcatcodetable <28-bit number>
```

The \initcatcodetable creates a new table with catcodes identical to those defined by initex:

^^M (<return>)	car_ret	5
(space)	spacer	10
\\	escape	0
%	comment	14
^^? (<delete>)	invalid_char	15
^^@ (<null>)	ignore	9
a--z	letter	11
A--Z	letter	11
everything else	other	12

The new catcode table is allocated globally: it will not go away after the current group has ended. If the supplied number is the currently active table, an error is raised.

2.5.4.3 \savecatcodetable

```
\savecatcodetable <28-bit number>
```

`\savecatcodetable` copies the current set of catcodes to a new table with the requested number. The definitions in this new table are all treated as if they were made in the outermost level.

The new table is allocated globally: it will not go away after the current group has ended. If the supplied number is the currently active table, an error is raised.

2.5.5 Font syntax

LuaTeX will accept a braced argument as a font name:

```
\font\myfont = {cmr10}
```

This allows for embedded spaces, without the need for double quotes. Macro expansion takes place in the argument.

3 Lua general

3.1 Initialization

3.1.1 Luatex as a lua interpreter

In a number of cases, luatex behaves like it is a lua interpreter only.

- If a `--luaonly` option is given
- If the executable is named `luatexlua`
- if the non-option (file) on the command-line has the extension `lua` or `luc`.

On 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 somehwat bulky standalone lua interpreter.

3.1.2 Other command-line processing

Whenever the LuaTEX executable starts, it looks for a `--lua` command—line option. If such an option is present, it will enter an alternative mode of command—line parsing.

In this mode, it will only interpret a very small subset of the command—line directly:

--luaonly	execute a lua script, then exit
--lua=s	load and execute a lua init script
--safer	disable easily exploitable lua commands
--help	display help and exit
--version	display version and exit

If a requested lua script can not be found using the actual name given on the command—line, a second attempt is made by prepending the value of the environment variable `LUATEXDIR`, if that variable is defined.

Then the script is loaded and executed. It will find the entire commandline in the table `arg`, beginning with `arg[0]`, that is the name of the executable.

LuaTEX will fetch some of the other commandline options from the `texconfig` table at the end of script execution (see the description of the `texconfig` table later on in this document).

Commandline processing happens very early on. So early, in fact, that none of TEX's initializations have taken place yet. For that reason, the `tex` and `pdf` tables 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 it's `\jobname` yet at this point).

The file is loaded into Lua state 0, and everything you do will remain visible during the rest of the run, with the exception of the `tex` and `pdf` tables: those will be restored to their normal meaning right after the execution of the script.

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

You can use the `--safer` switch to disable some commands that can easily be abused by a malicious document. At the moment, this switch `nil`s the following functions:

```
os.execute()
os.exec()
os.setenv()
os.rename()
os.remove()
io.popen()
io.output()
io.tmpfile()
lfs.rmdir()
lfs.mkdir()
lfs.chdir()
lfs.lock()
lfs.touch()
```

And it makes `io.open()` fail on files that are opened for anything besides reading.

Unless the `texconfig` table tells it not to start kpathsea at all (set `texconfig.kpse_init` to `false` for that), it also acts on three other command—line options:

--fmt=s	set the format name
--progname=s	set the progname (only for kpathsea)
--ini	enable initex mode

In order to initialize the built-in kpathsea library properly, LuaTEX needs to know the correct 'progname' to use, and for that it needs to check `-progname` (and `-ini` and `-fmt`, if `-progname` is missing).

If there is no `--lua` option, the commandline is interpreted in a similar fashion as in traditional pdfTEX and Aleph.

3.2 Lua changes

Five modules that are normally external are statically linked in with LuaTEX: `slnunicode`, `luazip`, `luafilesystem`, `lpeg` (version 0.4), and `md5`.

The `read("*line")` function from the io library has been adjusted so that it is line-ending neutral: any of `LF`, `CR` or typeCR+LF are accepted.

The `tostring()` printer for numbers has been changed so that it returns '0' instead of something like '2e-5' (which confused TEX enormously) when the value is so small that TEX cannot distinguish it from zero.

The (currently three) known bugs in Lua 5.1.1 have been patched.

Dynamic loading of `.so` and `.dll` files is disabled on all platforms.

`luafilesystem` has been extended with two extra boolean functions (`isdir(filename)` and `isfile(filename)`) and one extra string field in the attributes table (`permissions`).

The `string` library has six extra iterators that return strings piecemeal: "utfvalues" (returns an integer value in the unicode range), "utfcharacters" (returns a string with a single UTF-8 token in it), "characters" (a string of length one), "characterpairs" (two strings of length one), "bytes" (a single byte value), and "bytepairs" (two byte values). The "bytepairs" will produce nil instead of a number as its second return value if the string length was odd. "characterpairs" will produce an empty second string in that case.

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

- `os.exec('command')` is a non-returning version of `os.execute`. The advantage of this command is that it cleans out the current process before starting the new one, making it especially useful for use in `luatexlua`.
- `os.setenv('key','value')` This sets a variable in the environment. Passing 'nil' instead of a value string will remove the variable.
- `os.environ` This is a read-only hash table containing all of the variables and values in the process environment.

4 Lua Libraries

The interfacing between TEX and Lua is facilitated by a set of Lua modules.

4.1 The tex library

The tex table contains a large list of virtual internal TEX parameters that are partially writable.

The designation 'virtual' means that these items are not properly defined in Lua, but are only frontends that are handled by a metatable that operates on the actual TEX values. As a result, most of the lua table operators (like `pairs` and `#`) do not work on such items.

At the moment, it is possible to access almost every parameter that has these characteristics:

- You can use it after `\the`
- It is a single token.

This excludes parameters that need extra arguments, like `\the\scriptfont`.

The subset comprising simple integer and dimension registers are writable as well as readable (stuff like `\tracingcommands` and `\parindent`).

4.1.1 Integer parameters

The integer parameters accept and return lua numbers.

Read-write:

```
tex.adjdemerits
tex.binoppenalty
tex.brokenpenalty
tex.catcodetable
tex.clubpenalty
tex.day
tex.defaulthyphenchar
tex.defaultskewchar
tex.delimiterfactor
tex.displaywidowpenalty
tex.doublehyphendemerits
tex.endlinechar
tex.errorcontextlines
tex.escapechar
tex.exhyphenpenalty
tex.fam
tex.finalhyphendemerits
tex.floatingpenalty
tex.globaldefs
tex.hangafter
tex.hbadness
tex.holdinginserts
tex.hyphenpenalty
tex.interlinepenalty
tex.language
tex.lastlinefit
tex.lefthyphenmin
tex.linepenalty
tex.localbrokenpenalty
tex.localinterlinepenalty
tex.looseness
tex.mag
tex.maxdeadcycles
tex.month
tex.newlinechar
tex.outputpenalty
tex.pausing
tex.pdfadjustinterwordglue
tex.pdfadjustspacing
tex.pdfappendkern
tex.pdfcompresslevel
tex.pdfdecimaldigits
tex.pdfforcepagebox
tex.pdfgamma
tex.pdfgentounicode
tex.pdfimageapplygamma
tex.pdfimagegamma
tex.pdfimagehicolor
tex.pdfimageresolution
tex.pdfinclusionerrorlevel
tex.pdfminorversion
tex.pdfmovechars
tex.pdfobjcompresslevel
tex.pdfoptionalwaysusepdfpagebox
tex.pdfoptionpdfinclusionerrorlevel
tex.pdfoptionpdfminorversion
tex.pdfoutput
tex.pdfpagebox
tex.pdfpkresolution
tex.pdfprependkern
tex.pdfprotrudechars
tex.pdftracingfonts
tex.pdfuniqueresname
tex.postdisplaypenalty
tex.predisplaydirection
tex.predisplaypenalty
tex.pretolerance
tex.relpenalty
tex.righthyphenmin
tex.savinghyphcodes
tex.savingvdiscards
tex.showboxbreadth
tex.showboxdepth
tex.time
tex.tolerance
tex.tracingassigns
tex.tracingcommands
tex.tracinggroups
tex.tracingifs
tex.tracinglostchars
tex.tracingmacros
tex.tracingnesting
tex.tracingonline
tex.tracingoutput
tex.tracingpages
tex.tracingparagraphs
tex.tracingrestores
tex.tracingscantokens
tex.tracingstats
tex.uchyph
tex.vbadness
tex.widowpenalty
tex.year
```

Read-only:

`tex.deadcycles`
`tex.insertpenalties`
`tex.parshape`
`tex.prevgraf`
`tex.spacefactor`

4.1.2 Dimension parameters

The dimension parameters accept lua numbers (signifying scaled points) or strings (with included dimension). The result is always a string.

Read-write:

`tex.boxmaxdepth`
`tex.delimitershortfall`
`tex.displayindent`
`tex.displaywidth`
`tex.emergencystretch`
`tex.hangindent`
`tex.hfuzz`
`tex.hoffset`
`tex.hsize`
`tex.lineskiplimit`
`tex.mathsurround`
`tex.maxdepth`
`tex.nulldelimiterspace`
`tex.overfullrule`
`tex.pagebottomoffset`
`tex.pageheight`
`tex.pagerightoffset`
`tex.pagewidth`
`tex.parindent`
`tex.pdfdestmargin`
`tex.pdfeachlinedepth`
`tex.pdfeachlineheight`
`tex.pdffirstlineheight`
`tex.pdfhorigin`
`tex.pdflastlinedepth`
`tex.pdflinkmargin`
`tex.pdfpageheight`
`tex.pdfpagewidth`
`tex.pdfpxdimen`
`tex.pdfthreadmargin`
`tex.pdfvorigin`
`tex.predisplaysize`
`tex.scriptspace`
`tex.splitmaxdepth`
`tex.vfuzz`
`tex.voffset`
`tex.vsize`

Read-only:

`tex.pagedepth`
`tex.pagefilllstretch`
`tex.pagefillstretch`
`tex.pagefilstretch`
`tex.pagegoal`
`tex.pageshrink`
`tex.pagestretch`
`tex.pagetotal`
`tex.prevdepth`

4.1.3 Direction parameters

All direction parameters are read-only and return a lua string

`tex.bodydir`
`tex.mathdir`
`tex.pagedir`
`tex.pardir`
`tex.textdir`

4.1.4 Glue parameters

All glue parameters are read-only and return a lua string

```
tex.abovedisplayshortskip
tex.abovedisplayskip
tex.baselineskip
tex.belowdisplayshortskip
tex.belowdisplayskip
tex.leftskip
tex.lineskip
tex.parfillskip
tex.parskip
tex.rightskip
tex.spaceskip
tex.splittopskip
tex.tabskip
tex.topskip
tex.xspaceskip
```

4.1.5 Muglue parameters

All muglue parameters are read-only and return a lua string

```
tex.medmuskip
tex.thickmuskip
tex.thinmuskip
```

4.1.6 Tokenlist parameters

All tokenlist parameters are read-only and return a lua string

```
tex.errhelp
tex.everycr
tex.everydisplay
tex.everyeof
tex.everyhbox
tex.everyjob
tex.everymath
tex.everypar
tex.everyvbox
tex.output
tex.pdfpageattr
tex.pdfpageresources
tex.pdfpagesattr
tex.pdfpkmode
```

4.1.7 Convert commands

The supported commands at this moment are:

```
tex.AlephVersion
tex.Alephrevision
tex.OmegaVersion
tex.Omegarevision
tex.eTeXVersion
tex.eTeXrevision
tex.formatname
tex.jobname
tex.pdfnormaldeviate
tex.pdftexbanner
tex.pdftexrevision
```

All ‘convert’ commands are read-only and return a lua string

This list looks haphazard, but it really is not. These are all the cases of the 'convert' internal command that do not require an argument.

4.1.8 Count, dimension and token registers

TEX's counters (`\count`), dimensions (`\dimen`) and token (`\toks`) registers can be accessed and written to using three virtual sub-tables of the `tex` table:

```
tex.count
tex.dimen
tex.toks
```

It is possible to use the names of relevant `\countdef`, `\dimendef`, or `\toksdef` control sequences as indices to these tables:

```
tex.count.scratchcounter = 0
enormous = tex.dimen["maxdimen"]
```

In this case, luatex looks up the value for you on the fly. You have to use a valid `\countdef` (or `\dimendef`, or `\toksdef`), anything else will generate an error (the goal is to eventually also allow `<chardef tokens>` and even macros that expand into a number)

The count registers accept and return lua numbers.

The dimension registers accept lua numbers (in scaled points) or strings (with an included absolute dimension. "em" and "ex" and "px" are forbidden). The result is always a number in scaled points.

The token registers accept and return lua strings. Lua strings are converted to token lists using `\the\toks` style expansion.

As an alternative to array addressing, there are also accessor functions defined:

```
tex.setdimen(number n, string s)
tex.setdimen(string s, string s)
tex.setdimen(number n, number n)
tex.setdimen(string s, number n)
number n = tex.getdimen(number n)
number n = tex.getdimen(string s)

tex.setcount(number n, number n)
tex.setcount(string s, number n)
number n = tex.getcount(number n)
number n = tex.getcount(string s)

tex.settoks (number n, string s)
tex.settoks (string s, string s)
string s = tex.gettoks (number n)
string s = tex.gettoks (string s)
```

4.1.9 Box register size information

The current dimensions of \box registers can be read and altered using three other virtual sub-tables :

```
tex.wd
tex.ht
tex.dp
```

These are indexed strictly by number.

The box size registers accept lua numbers (in scaled points) or strings (with included dimension). The result is always a number in scaled points.

As an alternative to array addressing, there are also accessor functions defined:

```
tex.setboxwd(number n, string s)
tex.setboxwd(number n, number n)
number n = tex.getboxwd(number n)

tex.setboxht(number n, string s)
tex.setboxht(number n, number n)
number n = tex.getboxht(number n)

tex.setboxdp(number n, string s)
tex.setboxdp(number n, number n)
number n = tex.getboxdp(number n)
```

4.1.10 Print functions

The tex table also contains the three print functions that are the major interface from lua scripting to TEX.

The arguments to these three functions are all stored in an in-memory virtual file that is fed to the TEX scanner as the result of the expansion of \directlua.

The total amount of returnable text from a \directlua command is only limited by available system RAM. However, each separate printed string has to fit completely in TEX's input buffer.

4.1.10.1 tex.print

```
tex.print(<string s>, ...)
tex.print(<number n>, <string s>, ...)
```

Each string argument is treated by TEX as a separate input line.

The optional parameter can be used to print the strings using the catcode regime defined by `\catcodetable` *n*. If *n* is not a valid catcode table, then it is ignored, and the currently active catcode regime is used instead.

The very last string of the very last `tex.print()` command in a `\directlua` will not have the `\endlinechar` appended, all others do.

4.1.10.2 `tex.sprint`

```
tex.sprint(<string s>, ...)
tex.sprint(<number n>, <string s>, ...)
```

Each string argument is treated by TEX as a special kind of input line that makes it suitable for use as a partial line input mechanism:

- TEX does not switch to the 'new line' state, so that leading spaces are not ignored
- no `\endlinechar` is inserted
- trailing spaces are not removed

4.1.10.3 `tex.write`

```
tex.write(<string s>, ...)
```

Each string argument is treated by TEX as a special kind of input line that makes is suitable for use as a quick way to dump information:

- all catcodes on that line are either 'space' (for " ") or 'character' (for all others).
- there is no `\endlinechar` appended.

4.2 The texio library

This library takes care of the low-level I/O interface.

4.2.1 Printing functions

4.2.1.1 `texio.write`

```
texio.write(string target, tring s)
texio.write(string s)
```

Without the `target` argument, Writes the string to the same location(s) TEX writes messages to at this moment. If `\batchmode` is in effect, it writes only to the log, otherwise it writes to the log and the terminal.

The optional `target` can be one of three possibilities: ‘term’, ‘log’ or ‘term and log’.

4.2.1.2 tex.write_nl

```
texio.write_nl(string target, tring s)
texio.write_nl(string s)
```

Like `texio.write`, but make sure that the string s will appear at the beginning of a line. You can use an empty string if you only want to move to the next line.

4.3 The pdf library

This table contains the current `h` en `v` values that define the location on the output page. The values can be queried and set using scaled points as units.

```
pdf.v
pdf.h
```

The associated function calls are

```
pdf.setv(number n)
number n = pdf.getv()
pdf.seth(number n)
number n = pdf.geth()
```

It also holds a print function to write stuff to the pdf document, to be used from within a `\latelua` argument.

1 pdf.print

```
pdf.print(<string s>)
pdf.print(<string type>, <string s>)
```

The optional parameter can be used to mimic the behaviour of pdfliteral: the `type` is `"direct"` or `"page"`.

4.4 The callback library

This library has functions that register, find and list callbacks.

The callback library is only available in lua state zero (0).

```
callback.register(string <callback name>,function <callback_func>)
callback.register(string <callback name>,nil)
```

where the <callback name> is a predefined callback name, see below.

LuaTEX internalizes the callback function in such a way that it does not matter if you redefine a function accidentally.

Callback assignments are always global. You can use the special value 'nil' instead of a function for clearing the callback.

```
table <info> = callback.list()
```

The keys in the table are the known callback names, the value is a boolean where `true` means that the callback is currently set (active).

```
function <f> = callback.find(<callback name>)
```

If the callback is not set, `callback.find` returns `nil`.

4.4.1 File discovery callbacks

4.4.1.1 `find_read_file` and `find_write_file`

You callback function should have the following conventions:

```
string <actual_name> = function (number <id_number>, string <asked_name>)
```

Arguments:

id_number
: zero for the log or `\input` files, or TeX's `\read` or `\write` number incremented by one (`\read0` becomes 1).

asked_name
: the user—supplied filename, as found by `\input`, or `\openin`, or `\openout`.

Return value:

actual_name
: the filename used. For the very first file that is read in by TEX, you have to make sure you return an `actual_name` that has an extension and that is suitable for use as `jobname`. If you don't, you will have to manually fix the name for the log file and output file, and an eventual format filename will become mangled, since these depend on the jobname.
Return `nil` if the file cannot be found.

4.4.1.2 `find_font_file`

You callback function should have the following conventions:

```
string <actual_name> = function (string <asked_name>)
```

The asked_name is an OTF or TFM font metrics file.

Return nil if the file cannot be found.

4.4.1.3 find_output_file

You callback function should have the following conventions:

```
string <actual_name> = function (string <asked_name>)
```

The asked_name is the PDF or DVI file for writing.

4.4.1.4 find_format_file

You callback function should have the following conventions:

```
string <actual_name> = function (string <asked_name>)
```

The asked_name is a format file for reading (the format file for writing is always opened in the current directory).

4.4.1.5 find_vf_file

Like find_font_file, but for virtual fonts. This applies to both Aleph's ovf files and traditional Knuthian vf files.

4.4.1.6 find_ocp_file

Like find_font_file, but for ocp files.

4.4.1.7 find_map_file

Like find_font_file, but for map files.

4.4.1.8 find_enc_file

Like find_font_file, but for enc files.

4.4.1.9 find_sfd_file

Like find_font_file, but for subfont definition files.

4.4.1.10 find_pk_file

Like find_font_file, but for pk bitmap files. The argument <name> is a bit special in this case. It's form is

```
<base res>dpi/<fontname>.<actual res>pk
```

So you may be asked for 600dpi/manfnt.720pk. It is up to you to find a 'reasonable' bitmap file to go with that specification.

4.4.1.11 find_data_file

Like find_font_file, but for embedded files (\pdfobj file "...").

4.4.1.12 find_opentype_file

Like find_font_file, but for opentype font files.

4.4.1.13 find_truetype_file and find_type1_file

You callback function should have the following conventions:

```
string <actual_name> = function (string <asked_name>)
```

The asked_name is a font file. This callback is called while LuaTEX is building its internal list of needed font files, so the actual timing may surprise you. Your return value is later fed back into the matching read__file callback.

Strangely enough, find_type1_file is also used for OpenType (otf) fonts.

4.4.1.14 find_image_file

You callback function should have the following conventions:

```
string <actual_name> = function (string <asked_name>)
```

The asked_name is an image file. Your return value is used to open a file from the harddisk, so make sure you return something that is considered the name of a valid file by your operating system.

4.4.2 File reading callbacks

4.4.2.1 open_read_file

You callback function should have the following conventions:

```
table <env> = function (string <file_name>)
```

Argument:

file_name
: the filename returned by a previous `find_read_file` or the return value of `kpse_find_file()` if there was no such callback defined.

Return value:

env
: this is a table containing at least one required and one optional callback functions for this file. The required field is '`reader`' and the associated function will be called once for each new line to be read, the optional one is '`close`' that will be called once when LuaTEX is done with the file. LuaTEX never looks at the rest of the table, so you can use it to store your private per-file data. Both the callback functions will receive the table as their only argument.

4.4.2.1.1 reader

LuaTEX will run this function whenever it needs a new input line from the file.

```
function (table <env>)
  return string <line>
end
```

Your function should return either a string or 'nil'. The value 'nil' signals that the end of file has occurred, and will make TEX call the optional '`close`' function next.

4.4.2.1.2 close

LuaTEX will optionally run this function when it needs to close the file.

```
function (table <env>)
  return
end
```

Your function should not return any value.

4.4.2.2 read_font_file

This function is called when TEX needs to read a `ofm` or `tfm` file.

```
function (string <name>)
   return boolean <success>, string <data>, number <data_size>
end
```

success
: return false when a fatal error occured (e.g. when the file cannot be found, after all).

data
: the bytes comprising the file.

data_size
: the length of the `data`, in bytes.

return an empty string and zero if the file was found but there was a reading problem.

4.4.2.3 `read_vf_file`

Like `read_font_file`, but for virtual fonts.

4.4.2.4 `read_ocp_file`

Like `read_font_file`, but for ocp files.

4.4.2.5 `read_map_file`

Like `read_font_file`, but for map files.

4.4.2.6 `read_enc_file`

Like `read_font_file`, but for enc files.

4.4.2.7 `read_sfd_file`

Like `read_font_file`, but for subfont definition files.

4.4.2.8 `read_pk_file`

Like `read_font_file`, but for pk bitmap files.

4.4.2.9 `read_data_file`

Like `read_font_file`, but for embedded files (`\pdfobj file "..."`).

4.4.2.10 `read_truetype_file`

Like `read_font_file`, but for truetype font files. The `name` is a path name as returned by `find_truetype_file` or `kpse_find_file`.

4.4.2.11 read_type1_file

Like read_font_file, but for type1 font files. The name is a path name as returned by find_type1_file or kpse_find_file.

4.4.2.12 read_opentype_file

Like read_font_file, but for opentype font files. The name is a path name as returned by find_type1_file or kpse_find_file.

4.4.3 Data processing callbacks

4.4.3.1 process_input_buffer

This callback allows you to change the contents of the line input buffer just before LuaTEX actually starts looking at it.

```
function (string <buffer>)
   return string <adjusted_buffer>
end
```

If you return nil, LuaTEX will pretend like your callback never happened. You can gain a small amount of processing time from that.

4.4.3.2 token_filter

This callback allows you to change the modify any lexical token that enters the main_control function before LuaTEX executes the associated command.

Note: not all tokens can be intercepted, only those that are 'seen' by LuaTEX's main control function. Supplemental tokens like the bodies of macro definitions and the right-hand side of register assignments are not seen. For now, this is intentional.

```
function (table <token>)
   return table <token>
end
```

Calling convention for this callback is bit more complicated then for most other callbacks. Initially, lua function will be called with the next token from get_next() as argument, represented as a small lua table. The function should either return a lua table representing a valid to-be-processed token, or something else like nil or an empty table.

If your lua function does not return a table representing a valid token, it will be immediately called again with yet another token from get_next() as argument, until it eventually does return a useful token.

But if the function does return a usable token, then that token will be processed by LuaTEX. Afterwards, the function will be called again, but now without an argument. This is repeated until it stops returning tokens. Then processing reverts back to the other branch.

The point behind that roundabout calling convention is that it allows the lua function to buffer tokens for various uses. That in turn makes it possible to do some really advanced things like replacing OTPs.

Now about that table. The table that the function will receive contains four fields:

Key	type	Explanation
cmd	string	A representation of LuaTEX's internal command code
chr	number	The command code modifier
cs	string	If the token came from a csname, this is that csname
mod	character	A single character string representing the current processing mode. One of vertical, **h**orizontal, display **m**ath, **n**o, internal **V**ertical, restricted **H**orizontal, or inline **M**ath mode.

If you modify the table before returning it, then it is wise to return either a (`cmd`, `chr`) pair, or a `cs` string. That is because if both options are present, the pair has precedence and the string is ignored. On the return table, **mod** is ignored always.

4.4.4 Information reporting callbacks

4.4.4.1 start_run

```
function ()
```

Replaces the code that prints LuaTEX's banner

4.4.4.2 stop_run

```
function ()
```

Replaces the code that prints LuaTEX's statistics and 'Output written to' messages.

4.4.4.3 start_page_number

```
function ()
```

Replaces the code that prints the `[` and the page number at the begin of `\shipout`. This callback will also override the printing of box information that normally takes place when `\tracingoutput` is positive.

4.4.4.4 stop_page_number

```
function ()
```

Replaces the code that prints the] at the end of \shipout

4.4.4.5 show_error_hook

```
function ()
  return
end
```

This callback is run from inside the TEX error function, and the idea is to allow you to do some extra reporting on top of what TEX already does (none of the normal actions are removed). You may find some of the values in the statistics table useful.

message
    is the formal error message TEX has given to the user (the line after the "! ")

indicator
    is either a filename (when it is a string) or a location indicator (a number) that can means lots of different things like a token list id or a \read number.

lineno
    is the current line number

This is an investigative item only, only for 'testing the water'.

The final goal is the total replacement of TEX's error handling routines, but that needs lots of adjustments in the web source because TEX deals with errors in a somewhat haphazard fashion.

4.4.5 Font-related callbacks

4.4.5.1 define_font

```
function (string <name>, string <area>, number <size>)
  return table <font>
end
```

The string <name> is the filename part of the font specification, as given by the user.

The string <area> is the areaname part of the font specification, as given by the user.

The number <size> is a bit special:

- if it is positive, it specifies an 'at size' in scaled points.
- if it is negative, its absolute value represents a 'scaled' setting relative to the designsize of the font.

The internal structure of the `<font>` table that is to be returned is explained in chapter 5. That table is saved internally, so you can put extra fields in the table for your later lua code to use.

4.5 The lua library

This library contains two read-only items:

4.5.1 Variables

```
number n = lua.id
```

the id number of the instance

```
string s = lua.version
```

a luatex version identifier string (currently `"0.1"`)

4.5.2 Lua bytecode registers

Lua registers can be used to communicate lua functions across lua states. The accepted values for assignments are functions and nil. Likewise, the retrieved value is either a function or nil.

```
lua.bytecode[n] = function () .. end
lua.bytecode[n]()
```

The contents of the `lua.bytecode` array is stored inside the format file as actual lua bytecode, so it can also be used to preload lua code.

The associated function calls are

```
function f = lua.getbytecode(number n)
lua.setbytecode(number n, function f)
```

4.6 The kpse library

4.6.1 `kpse.find_file`

The most important function in the library is find_file:

```
string f = kpse.find_file(string filename)
string f = kpse.find_file(string filename, string ftype)
string f = kpse.find_file(string filename, boolean mustexist)
string f = kpse.find_file(string filename, string ftype, boolean mustexist)
```

Arguments:

filename
: the name of the file you want to find, with or without extension.

type
: maps to the '-format' argument of `kpsewhich`. The supported values are:

```
"gf"
"pk"
"bitmap font"
"tfm"
"afm"
"base"
"bib"
"bst"
"cnf"
"ls-R"
"fmt"
"map"
"mem"
"mf"
"mfpool"
"mft"
"mp"
"mppool"
"MetaPost support"
"ocp"
"ofm"
"opl"
"otp"
"ovf"
"ovp"
"graphic/figure"
"tex"
"TeX system documentation"
"texpool"
"TeX system sources"
"PostScript header"
"Troff fonts"
"type1 fonts"
"vf"
"dvips config"
"ist"
"truetype fonts"
"type42 fonts"
"web2c files"
"other text files"
"other binary files"
"misc fonts"
"web"
"cweb"
"enc files"
"cmap files"
"subfont definition files"
"opentype fonts"
"pdftex config"
"lig files"
"texmfscripts"
```

The default type is "tex".

mustexist

is similar to kpsewhich's '-must-exist', and the default is 'false'. If you specify 'true' (or a non-zero integer), then the kpse library will search the disk as well as the ls-R databases.

4.6.2 kpse.expand_path

Like kpsewhich's '-expand-path':

```
string r = kpse.expand_path(string s)
```

4.6.3 kpse.expand_var

Like kpsewhich's '-expand-var':

```
string r = kpse.expand_var(string s)
```

4.6.4 kpse.expand_braces

Like kpsewhich's '-expand-braces':

```
string r = kpse.expand_braces(string s)
```

4.7 The statistics library

This contains a number of run–time configuration items that you may find useful in message reporting, as well as an iterator function that gets all of the names and values as a table.

```
table <info> = statistics.list()
```

The keys in the table are the known items, the value is the current value.

Almost all of the values in statistics are fetched through a metatable at run–time whenever they are accessed, so you cannot use pairs onstatistics, but you *can* use pairs on <info>, of course.

If you do not need the full list, you can also ask for a single item by using it's name as an index into statistics.

The current list is:

Key	Explanation
pdf_gone	written pdf bytes
pdf_ptr	not yet written pdf bytes
dvi_gone	written dvi bytes
dvi_ptr	not yet written dvi bytes

total_pages	number of written pages
output_file_name	name of the pdf or dvi file
log_name	name of the log file
banner	terminal display banner
pdftex_banner	--
var_used	variable (one-word) memory in use
dyn_used	token (multi-word) memory in use
str_ptr	number of strings
init_str_ptr	number of initex strings
max_strings	maximum allowed strings
pool_ptr	string pool index
init_pool_ptr	initex string pool index
pool_size	maximum allowed string characters
lo_mem_max	current top of multi-word memory
mem_min	bottom index of memory array
mem_end	top index of memory array
hi_mem_min	current bottom of one-word memory
cs_count	number of control sequences
hash_size	size of hash
hash_extra	extra allowed hash
font_ptr	number of active fonts
hyph_count	hyphenation exceptions
hyph_size	max used hyphenation exceptions
max_in_stack	max used input stack entries
max_nest_stack	max used nesting stack entries
max_param_stack	max used parameter stack entries
max_buf_stack	max used buffer position
max_save_stack	max used save stack entries
stack_size	input stack size
nest_size	nesting stack size
param_size	parameter stack size
buf_size	line buffer size
save_size	save stack size
obj_ptr	max pdf object pointer
obj_tab_size	pdf object table size
pdf_os_cntr	max pdf object stream pointer
pdf_os_objidx	pdf object stream index
pdf_dest_names_ptr	max pdf destination pointer
dest_names_size	pdf destination table size
pdf_mem_ptr	max pdf memory used
pdf_mem_size	pdf memory size
largest_used_mark	max referenced marks class
filename	name of the current input file
inputid	numeric id of the current input

linenumber	location in the current input file
lasterrorstring	last error string
luabytecodes	number of active luabytecode registers
luabytecode_bytes	number of bytes in luabytecode registers
luastates	number of active lua interpreters
luastate_bytes	number of bytes in use by lua interpreters

4.8 The texconfig table

This is a table that is created empty. A startup lua script could fill this table with a number of settings that are read out by the executable after loading and executing the startup file.

key	type	default	explanation
mem_bot	number	0	cf. web2c docs
main_memory	number	250000	cf. web2c docs
extra_mem_top	number	0	cf. web2c docs
extra_mem_bot	number	0	cf. web2c docs
pool_size	number	100000	cf. web2c docs
string_vacancies	number	75000	cf. web2c docs
pool_free	number	5000	cf. web2c docs
max_strings	number	15000	cf. web2c docs
strings_free	number	100	cf. web2c docs
trie_size	number	20000	cf. web2c docs
hyph_size	number	659	cf. web2c docs
buf_size	number	3000	cf. web2c docs
nest_size	number	50	cf. web2c docs
max_in_open	number	15	cf. web2c docs
param_size	number	60	cf. web2c docs
save_size	number	4000	cf. web2c docs
stack_size	number	300	cf. web2c docs
dvi_buf_size	number	16384	cf. web2c docs
error_line	number	79	cf. web2c docs
half_error_line	number	50	cf. web2c docs
max_print_line	number	79	cf. web2c docs
ocp_list_size	number	1000	cf. web2c docs
ocp_buf_size	number	1000	cf. web2c docs
ocp_stack_size	number	1000	cf. web2c docs
hash_extra	number	0	cf. web2c docs
pk_dpi	number	72	cf. web2c docs
kpse_init	boolean	true	`false` totally disables Kpathsea initialisation (only ever unset this if you implement *all* file find callbacks!)
trace_file_names	boolean	true	`false` disables TeX's normal file open–close feedback (the assumption is that callbacks will take care of that).

src_special_auto	boolean	false	Source specials sub-item
src_special_everypar	boolean	false	Source specials sub-item
src_special_everyparend	boolean	false	Source specials sub-item
src_special_everycr	boolean	false	Source specials sub-item
src_special_everymath	boolean	false	Source specials sub-item
src_special_everyhbox	boolean	false	Source specials sub-item
src_special_everyvbox	boolean	false	Source specials sub-item
src_special_everydisplay	boolean	false	Source specials sub-item
file_line_error	boolean	false	Do `file:line` style error messages
halt_on_error	boolean	false	Abort run on the first encountered error
formatname	string	--	If no format name was given on the command--line, this key will be tested first instead of simply quitting
jobname	string	--	If no input file name was given on the command--line, this key will be tested first instead of simply giving up

4.9 The font library

The font library will provide the interface into the internals of the font system, as well as contain some binary font loaders.

4.9.1 Loading a tfm file

```
table fnt = font.read_tfm(string name, number s)
```

The number is a bit special:

- if it is positive, it specifies an 'at size' in scaled points.
- if it is negative, its absolute value represents a 'scaled' setting relative to the designsize of the font.

The internal structure of the virtual font table that is returned is explained in chapter 5.

4.9.2 Loading a vf file

```
table vf_fnt = font.read_vf(string name, number s)
```

The number is a bit special:

- if it is positive, it specifies an 'at size' in scaled points.
- if it is negative, its absolute value represents a 'scaled' setting relative to the designsize of the font.

4.9.3 Loading an opentype or truetype file

If you want to use an OpenType font, you have to get the metric information from somewhere. The next two functions provide a way of doing that.

```
table ttf_metrics = font.read_otf(string filename)
table ttf_metrics = font.read_ttf(string filename)
```

The result is identical in both cases, but you have to use the 'read_otf' for loading of information from PostScript-based OpenType and 'read_ttf' for loading of TrueType-based OpenType (or simply a TrueType font). Bitmap-only OpenType fonts are not supported.

At the moment, the `filename` font file is actually parsed and even partially interpreted by the OpenType/TrueType loading routines from FontForge. There are a few reasons for this:

- The font is automatically re-encoded, so that the `ttf_metrics` table is using unicode for the character indices.
- Many features are pre-processed into a format that is easier to handle than just the bare tables would be.
- PostScript-based OpenType fonts do not store the character height and depth in the font file, so the actual character boundingbox has to be calculated.
- In the future, it may be interesting to allow Lua scripts access to the actual font programs.

The top—level keys in the returned table are (this documentation is not yet finished):

key	type	explanation
fontname	string	
fullname	string	
familyname	string	
weight	string	
copyright	string	
filename	string	
defbasefilename	string	
version	string	
italicangle	float	
upos	float	
uwidth	float	
ascent	number	
descent	number	
vertical_origin	number	
uniqueid	number	
glyphcnt	number	
glyphmax	number	
glyphs	array	
changed	number	
hasvmetrics	number	

order2	number	
strokedfont	number	
weight_width_slope_only	number	
head_optimized_for_cleartype	number	
uni_interp	enum	Possible values: "unset", "none", "adobe", "greek", "japanese", "trad_chinese", "simp_chinese", "korean", "ams"
map	table	
private	table	
xuid	string	
pfminfo	table	
names	table	
cidinfo	table	
subfonts	array	
cidmaster	array	
commments	string	
anchor	table	
orders	table	
ttf_tables	table	
script_lang	table	
kerns	table	
vkerns	table	
texdata	table	
tt_cur	number	
tt_max	number	
gentags	table	
possub	table	
features	table	
chosenname	string	
macstyle	number	
sli_cnt	number	
fondname	string	
design_size	number	
fontstyle_id	number	
fontstyle_name	table	
design_range_bottom	number	
design_range_top	number	
strokewidth	float	
mark_class_cnt	number	
mark_classes	array	
mark_class_names	array	
creationtime	number	
modificationtime	number	
os2_version	number	
gasp_version	number	

gasp_cnt	number
gasp	table

4.9.3.1 Glyph items

The glyphs is an array containing the per-character information.

key	type	explanation
name	string	
unicodeenc	number	
boundingbox	array	Array of four numbers
orig_pos	number	
width	number	
vwidth	number	
lsidebearing	number	
ticked	number	
widthset	number	
glyph_class	number	
kerns	array	
vkerns	array	
dependents	array	Linear array of glyph name strings
possub	table	
ligofme	table	
comment	string	
color	number	
tex_height	number	
tex_depth	number	
tex_sub_pos	number	
tex_super_pos	number	

The kerns and vkerns are linear arrays of small hashes:

key	type	explanation
char	string	
off	number	
sli	number	
flags	number	

The possub is a linear array of small hashes:

key	type	explanation
type	enum	"position", "pair", "substitution", "alternate", "multiple", "ligature", "lcaret", "kerning", "vkerning", "anchors", "contextpos", "contextsub", "chainpos", "chainsub", "reversesub", "max", "kernback", "vkernback"
macfeature	number	

flags	number
tag	string
script_lang_index	number

For the first seven values of `type`, there can be additional sub-information:

value	key	type	explanation
position	pos	table	‘vr’ table
pair	pair	table	one string: ’paired’, and a ’vr’ (sub)table
substitution	subs	table	one string: ’variant’
alternate	alt	table	one string: ’components’
multiple	mult	table	one string: ’components’
ligature	lig	table	two strings: ’components’, ’char’
lcaret	lcaret	array	linear array of numbers

The ‘vr’ table contains for number-valued fields: `xoff`, `yoff`, `h_adv_off` and `v_adv_off`.

The other values of `type` could probably use some extra information as well, but I do not know which case of the union is supposed to be selected.

The `ligofme` is a linear array of small hashes:

key	type	explanation
lig	table	uses the same substructure as a single ‘possub’ item
char	string	
components	array	linear array of named components
ccnt	number	

4.9.3.2 map table

The top-level map is a list of encoding mappings. Each of those is a table itself.

key	type	explanation
enccount	number	
encmax	number	
backmax	number	
ticked	number	
remap	table	
map	array	non-linear array of mappings
backmap	array	non-linear array of backward mappings
enc	table	

The ‘remap’ table is very small:

key	type	explanation
firstenc	number	

lastenc	number
infont	number

The ‘enc’ table is a bit more verbose:

key	type	explanation
enc_name	string	
char_cnt	number	
char_max	number	
unicode	array	of unicode position numbers
psnames	array	of postscript glyph names
builtin	number	
hidden	number	
only_1byte	number	
has_1byte	number	
has_2byte	number	
is_unicodebmp	number	
is_unicodefull	number	
is_custom	number	
is_original	number	
is_compact	number	
is_japanese	number	
is_korean	number	
is_tradchinese	number	
is_simplechinese	number	
low_page	number	
high_page	number	
is_temporary	number	
iconv_name	string	
iso_2022_escape	string	

4.9.3.3 private table

This is the font’s private PostScript dictionary, if any. Keys and values are both strings.

4.9.3.4 cidinfo table

registry	string
ordering	string
supplement	number
version	number

4.9.3.5 pfminfo table

The ‘pfminfo’ table contains most of the OS/2 information:

key	type	explanation
pfmset	number	
winascent_add	number	
windescent_add	number	
hheadascent_add	number	
hheaddescent_add	number	
typoascent_add	number	
typodescent_add	number	
subsuper_set	number	
panose_set	number	
hheadset	number	
vheadset	number	
pfmfamily	number	
weight	number	
width	number	
avgwidth	number	
firstchar	number	
lastchar	number	
fstype	number	
linegap	number	
vlinegap	number	
hhead_ascent	number	
hhead_descent	number	
hhead_descent	number	
os2_typoascent	number	
os2_typodescent	number	
os2_typolinegap	number	
os2_winascent	number	
os2_windescent	number	
os2_subxsize	number	
os2_subysize	number	
os2_subxoff	number	
os2_subyoff	number	
os2_supxsize	number	
os2_supysize	number	
os2_supxoff	number	
os2_supyoff	number	
os2_strikeysize	number	
os2_strikeypos	number	
os2_family_class	number	
os2_xheight	number	

os2_capheight	number
os2_defaultchar	number
os2_breakchar	number
os2_vendor	string
panose	table

The `panose` subtable has exactly 10 string keys:

	key	type
familytype	string	Values as in the OpenType font specification: "Any", "No Fit", "Text and Display", "Script", "D
serifstyle	string	See the OpenType font specification for values
weight	string	id.
proportion	string	id.
contrast	string	id.
strokevariation	string	id.
armstyle	string	id.
letterform	string	id.
midline	string	id.
xheight	string	id.

4.9.3.6 names table

Each item has two top-level keys:

	key	type	explanation
lang	number	language for this entry	
names	table		

The `names` keys are the actual TrueType name strings. The possible keys are:

key	explanation
copyright	
family	
subfamily	
uniqueid	
fullname	
version	
postscriptname	
trademark	
manufacturer	
designer	
descriptor	
venderurl	
designerurl	
license	

licenseurl
idontknow
preffamilyname
prefmodifiers
compatfull
sampletext
cidfindfontname

4.9.3.7 anchor table

The anchor classes:

key	type	explanation
name	string	
feature_tag	string	
script_lang_index	number	
flags	number	
merge_with	number	
type	number	
processed	number	
has_mark	number	
matches	number	
ac_num	number	

4.9.3.8 orders table

key	type	explanation
table_tag	string	
ordered_features	array	list of tag strings

4.9.3.9 ttf_tables table

key	type	explanation
tag	string	
len	number	
maxlen	number	
data	number	

4.9.3.10 script_lang table

key	type	explanation
script	string	
langs	array	list of language tags

4.9.3.11 kerns table

Substructure is identical to the per-glyph subtable.

4.9.3.12 vkerns table

Substructure is identical to the per-glyph subtable.

4.9.3.13 texdata table

key	type	explanation
type	string	possible values: "unset", "text", "math", "mathext"
params	array	22 font numeric parameters

4.9.3.14 gentags table

key	type	explanation
tagtype	array	

The array items are mini-hashes:

key	type	explanation
type	enum	allowed values: "null", "position", "pair", "substitution", "alternate", "multiple", "ligature", "lcaret", "kerning", "vkerning", "anchors", "contextpos", "contextsub", "chainpos", "chainsub", "reversesub", "max", "kernback", "vkernback"
tag	string	

4.9.3.15 possub table

Top-level `possub` is quite different from the ones at character level.

key	type	explanation
type	number	
format	enum	Possible values: "glyphs", "class","coverage","reversecoverage"
script_lang_index	number	
flags	number	
tag	string	
nccnt	number	
bccnt	number	
fccnt	number	
rule_cnt	number	
nclass	array	
bclass	array	

fclass	array	
rules	array	an array of rule items
ticked	number	

Rule items have one common item and one specialized item:

key	type	explanation
lookups	array	A list of ‘lookup items’
glyph	array	Only if the parent’s format is ’glyph’
class	array	Only if the parent’s format is ’glyph’
coverage	array	Only if the parent’s format is ’glyph’
rcoverage	array	Only if the parent’s format is ’glyph’

Each of the lookup item is:

key	type	explanation
seq	number	
lookup_tag	string	

glyph:

key	type	explanation
names	string	
back	string	
fore	string	

class:

key	type	explanation
nclasses	array	of numbers
bclasses	array	of numbers
fclasses	array	of numbers

coverage:

key	type	explanation
ncovers	array	of strings
bcovers	array	of strings
fcovers	array	of strings

rcoverage:

key	type	explanation
ncovers	array	of strings
bcovers	array	of strings
fcovers	array	of strings
replacements	string	

4.9.3.16 features table

These are Apple features.

key	type	explanation
feature	number	
ismutex	number	
default_setting	number	
strid	number	
featname	array	array of 'macname' items
settings	array	

The `settings` items are hashes:

key	type	explanation
setting	number	
strid	number	
initially_enabled	number	
setname	array	array of 'macname' items

The 'macname' hashes:

key	type	explanation
enc	number	
lang	number	
name	string	

4.9.4 Loading opentype or truetype name information

```
table ttf_info    = font.read_otf_info(string name)
table ttf_info    = font.read_ttf_info(string name)
```

These two functions are very similar to the two commands from previous section, but they only return a small subset of the information. The returned table only has four keys: `fontname`, `fullname`, `familyname` and `weight`.

4.9.5 The fonts array

```
font.fonts[n] = { ... }
table f = font.fonts[n]
```

See chapter 5 for the structure of the tables.

The associated function calls are

```
table f = font.getfont(number n)
font.setfont(number n, table f)
```

Note the following: Assignments can only be made to fonts that have already be defined in TEX, but have not been accessed *at all* since that definition. This limits the usability of the write access to font.fonts quite a lot, a less stringent ruleset will be implemented later.

4.9.6 Checking a font's status

You can test for the status of a font by calling this function:

```
boolean f = font.frozen(number n)
```

The return value is one of true (unassignable), false (can be changed) or nil (not a valid font at all).

4.9.7 Defining a font directly

You can define your own font into `font.fonts`

```
number i = font.define(table f)
```

The return value is the internal id number of the defined font (the index into `font.fonts`). If the font creation fails, an error is raised. The table is a font structure, as explained in chapter 5.

The value of this function is debatable, because there is no direct way of accessing the newly defined font, except from Lua code.

5 Font structure

All TEX fonts are represented to Lua code as tables, an internally as C structures. All keys in the table below are saved in the internal font structure if they are present in the table returned by the 'define_font' callback, or if they result from the normal tfm/vf reading routines if there is no 'define_font' callback defined.

The column 'from VF' means that this key will be created by the 'font.read_vf()' routine, 'from TFM' means that the key will be created by the 'font.read_tfm()' routine, and 'used' means whether or not the luatex engine itself will do something with the key.

The top-level keys in the table are as follows:

key	from VF	from TFM	used	value type	description
name	yes	yes	yes	string	metric (file) name
area	no	yes	yes	string	(directory)location, typically empty
used	no	yes	yes	boolean	used already? (initial: false)
characters	yes	yes	yes	table	the defined glyphs of this font
checksum	yes	yes	no	number	default: 0
designsize	no	yes	yes	number	expected size (default: 655360 == 10pt)
direction	no	yes	yes	number	default: 0 (LTR)
encodingname	no	no	yes	string	encoding name
fonts	yes	no	yes	table	locally used fonts
fullname	no	no	yes	string	actual (PostScript) name
header	yes	no	no	string	header comments, if any
hyphenchar	no	no	yes	number	default: TeX's \hyphenchar
parameters	no	yes	yes	hash	default: 7 parameters, all zero
size	no	yes	yes	number	loaded (at) size. (default: same as designsize)
skewchar	no	no	yes	number	default: TeX's \skewchar
type	yes	no	yes	string	basic type of this font
format	no	no	yes	string	disk format type
embedding	no	no	yes	string	PDF inclusion
filename	no	no	yes	string	disk file name

The key name is always required.

The key used is set by the engine when a font is actively in use, this makes sure that the font's definition is written to the output file (DVI or PDF). The TFM reader sets it to false.

The direction is a number signalling the 'normal' direction for this font. There are sixteen possibilities:

number	meaning
0	LT
1	LL

number	meaning
8	TT
9	TL

2	LB	10	TB
3	LR	11	TR
4	RT	12	BT
5	RL	13	BL
6	RB	14	BB
7	RR	15	BR

These are Omega-style direction abbreviations: the first character indicates the 'first' edge of the character glyphs (the edge that is seen first in the writing direction), the second the 'top' side.

The `parameters` is a hash with mixed key types. There are seven possible string keys, as well as a number of integer indices (these start from 8 up). The seven strings are actually used instead of the bottom seven indices, because that gives a nicer user interface.

The names and their internal remapping:

name	internal remapped number
slant	1
space	2
space_stretch	3
space_shrink	4
x_height	5
quad	6
extra_space	7

The keys `type`, `format`, `embedding`, `fullname` and `filename` are used to embed OpenType fonts in the result PDF.

The `characters` table is a list of character hashes indexed by integer number. The number is the 'internal code' TeX knows this character by.

Two very special string indexes can be used also: `left_boundary` is a virtual character whose 'ligatures and 'kerns' are used to handle word boundary processing. `right_boundary` is similar but not actually used for anything (yet!).

Other index keys are ignored.

Each character hash itself is a hash. For example, here is the character 'f' (decimal 102) in the font cmr10 at 10 points:

```
[102] = {
  ["kerns"] = {
    [63] = 50973,
    [93] = 50973,
    [39] = 50973,
    [33] = 50973,
    [41] = 50973
   },
  ["italic"] = 50973,
 ["height"] = 455111,
 ["depth"] = 0,
 ["ligatures"] = {
   [102] = {
     ["char"] = 11,
     ["type"] = 0
    },
   [108] = {
     ["char"] = 13,
     ["type"] = 0
    },
   [105] = {
     ["char"] = 12,
     ["type"] = 0
    }
  },
 ["width"] = 200250
}
```

The following top-level keys can be present inside a character hash:

key	from VF	from TFM	used	value type	description
width	yes	yes	yes	number	character's width, in sp (default 0)
height	no	yes	yes	number	character's height, in sp (default 0)
depth	no	yes	yes	number	character's depth, in sp (default 0)
italic	no	yes	yes	number	character's italic correction, in sp (default zero)
next	no	yes	yes	number	the 'next larger' character index
extensible	no	yes	yes	table	the constituent bits of an extensible recipe
kerns	no	yes	yes	table	kerning information
ligatures	no	yes	yes	table	ligaturing information
commands	yes	no	yes	array	virtual font commands
name	no	no	no	string	the character (PostScript) name
index	no	no	yes	number	the (opentype or truetyoe) font glyph index
used	no	yes	yes	boolean	typeset already (default: false)?

The presence of `extensible` will overrule `next`, if that is also present.

The `extensible` table is very simple:

key	value type	description
top	number	'top' character index
mid	number	'middle' character index
bot	number	'bottom' character index
rep	number	'repeatable' character index

The kerns table is a hash indexed by character index (and 'character index' is defined as either a non-negative integer or the string value 'right_boundary'), with the values the kerning to be appled, in scaled points.

The ligatures table is a hash indexed by character index (and 'character index' is defined as either a non-negative integer or the string value 'right_boundary'), with the values being yet another small hash, with two fields:

key	value type	description
type	number	the type of this ligature command, default 0
char	number	the character index of the resultant ligature

The char field in a ligature is required.

The type field inside a ligature is the numerical or string value of one of the eight possible ligature types supported by TEX. When TEX inserts a new ligature, it puts the new glyph in the middle of the left and right glyphs. The original left and right glyphs can optionally be retained, and when at least one of them is kept, it is also possible to move the new 'insertion point' forward one or two places. The glyph that ends up to the right of the insertion point will become the next 'left'.

<table>
<tr><th>textual (Knuth)</th><th>number</th><th>string</th><th>result (| = final 'insertion point')</th></tr>
<tr><td>l + r =: n</td><td>0</td><td>=:</td><td>|n</td></tr>
<tr><td>l + r =:| n</td><td>1</td><td>=:|</td><td>|nr</td></tr>
<tr><td>l + r |=: n</td><td>2</td><td>|=:</td><td>|ln</td></tr>
<tr><td>l + r |=:| n</td><td>3</td><td>|=:|</td><td>|lnr</td></tr>
<tr><td>l + r =:|> n</td><td>5</td><td>=:|></td><td>n|r</td></tr>
<tr><td>l + r |=:> n</td><td>6</td><td>|=:></td><td>l|n</td></tr>
<tr><td>l + r |=:|> n</td><td>7</td><td>|=:|></td><td>l|nr</td></tr>
<tr><td>l + r |=:|>> n</td><td>11</td><td>|=:|</td><td>ln|r</td></tr>
</table>

The default value is 0, and can be left out. That signifies a 'normal' ligature where the ligature replaces both original glyphs.

The commands array is explained below.

5.1 Real fonts

Whether or not a TEX font is a 'real' font that should be written to the PDF document is decided by the type value in the top-level font structure. If the value is real, then this is a proper font, and the inclusion mechanism will attempt to add the needed font object definitions to the PDF.

Values for type:

value	description
real	This is a base font
virtual	This is a virtual font

The actions to be taken depend on a number of different variables:

- Whether the used font fits in an 8-bit encoding scheme or not
- The type of the disk font file
- The level of embedding requested

A font that uses anything other than an 8-bit encoding vector has to be written to the PDF in a different way.

The test that decides if this is the case is fairly simple in the current version of LuaTEX: If a 'real' font has a 'cidinfo' structure, then it is assumed to be a wide font, in all other cases it isn't. A more flexible approach is often possible, and will perhaps be implemented later.

If no special care is needed, LuaTEX currently falls back to the mapfile—based solution used by pdfTEX and DVIPS. This behaviour will be removed in the future, when the existing code becomes integrated in the new subsystem.

But if this is a 'wide' font, then the new subsystem kicks in, and some extra fields have to be present in the font structure. In this case, LuaTEX does not use a map file at all.

The extra fields are: `format`, `embedding`, `fullname`, `cidinfo` (as explained above), `filename`, and the `index` key in the separate characters.

Values for `format`:

value	description
type1	This is a PostScript Type1 font
type3	This is a bitmapped (PK) font
truetype	This is a TrueType or TrueType-based OpenType font
opentype	This is a PostScript-based OpenType font

Curerntly, only `truetype` and `opentype` fonts can be 'wide' fonts (Type0 PostScript fonts are not supported).

Values for `embedding`:

value	description
no	Don't embed the font at all
subset	Include and atttempt to subset the font
full	Include this font in it's entirety

At the moment, `full` is the only implemented form.

It is not possible to artificially modify the transformation matrix for the font at the moment.

The other fields are used as follows: The `fullname` will be the PostScript/PDF font name. The `cidinfo` will be used as the character set (the CID `/Ordering` and `/Registry` keys). The `filename` points to the actual font file. If you include the full path in the `filename` or if the file is in the local directory, LuaTEX will run a little bit more efficient because it will not have to re-run the `find_xxx_file` callback in that case.

Be careful: when mixing old and new fonts in one document, it is possible to create name PostScript name clashes that can result in printing errors. When this happens, you have to change the `fullname` of the font.

Typeset strings are written out in a wide format using 2 bytes per glyph, using the index key in the character information as value. The overall effect is like having an encoding based on numbers instead of traditional (PostScript) name-based reencoding.

This type of reencoding means that there is no longer a clear connection between the text in your input file and the strings in the output PDF file; I have not found a convenient away around that yet.

5.2 Virtual fonts

You have to take the following steps if you want LuaTEX to treat the returned table from 'define_font' as a virtual font:

- Set the top-level key 'type' to 'virtual'.
- Make sure there is at least one valid entry in 'fonts' (see below)
- Give a 'commands' array to every character (see below)

The presence of the toplevel 'type' key with the specific value 'virtual' will trigger handling of the rest of the special virtual font fields in the table, but the mere existance of 'type' is enough to prevent luatex from looking for a virtual font on its own.

Therefore, this also works 'in reverse': if you are absolutely certain that a font is not a virtual font, assigning the value 'base' or 'real' to 'type' will inhibit LuaTEX from looking for a virtual font file, thereby saving you a disk search.

The fonts is another Lua array. The values are one- or two-key hashes themselves, each entry indicating one of the base fonts in a virtual font. An example makes this easy to understand

```
"fonts" = { { name = "ptmr8a", size = 655360},
            { name = "psyr", size = 600000},
            { id = 38 } }
```

says that the first referenced font (index 1) in this virtual font is ptrmr8a loaded at 10pt, and the second is psyr loaded at a little over 9pt. The third one is previously defined font that is known to luatex as fontid '38'.

The array index numbers are used by the character command definitions that are part of each character.

The commands array is a hash here each item is another small array, with first entry representing a command and the extra items the parameters to that command. The allowed commands and their arguments are:

command name	arguments	arg type	description
font	1	number	select a new font from the local 'fonts' table
char	1	number	typeset this character number from the current font, and move right
slot	2	number	a shortcut for a font, char set
push	0	--	save current position
nop	0	--	do nothing

pop	0	--	pop position
rule	2	2 numbers	output a rule $w * h$, and move right
down	1	number	move down on the page
right	1	number	move right on the page
special	1	string	output a `\special` command
comment	any	any	the rest of the command is ignored

Here is a rather elaborate example:

```
...
"commands" = {
   {"push"},                     -- remember where we are
   {"right", 5000},              -- move right about 0.08pt
   {"font", 1},                  -- select the fonts[1] entry
   {"char", 97},                 -- place character 97 'a'
   {"pop"},                      -- go all the way back
   {"down", -200000},            -- move *up* about 3pt
   {"special", "pdf: 1 0 0 rg"}  -- switch to red color
   {"rule", 500000, 20000}       -- draw a bar
   {'special',"pdf: 0 g"}        -- back to black
}
...
```

The default value for ‘font’ is always 1, for each character anew. If the virtual font is essentially only a re-encoding, then you do usually do not have create an explicit ‘font’ entry.

Regardless of the amount of movement you create within the ‘commands’, the output pointer will always move by exactly the width as given in the ‘width’ key of the character hash, after running the ‘commands.

Even in a ‘real’ font, there can be virtual characters: When LuaTEX encounters a ‘commands’ field inside a character when it becomes time to typeset the character, it will interpret the commands, just like for a true virtual character. In this case, if you have created no ‘fonts’ array, then the default and only ‘base’ font is taken to be the current font itself. In practise, this means that you can create virtual duplicates of existing characters.

Note: this feature does *not* work the other way around. There can not be ‘real’ characters in a virtual font!

Finally, here is a plain TEX input file with a demonstration:

```
% start of virtual-demo.tex

\pdfoutput=1
\directlua0 {
   callback.register("define_font",
     function (name,area,size)
        if name == 'cmr10-red' then
```

```
            f = font.read_tfm('cmr10',size)
            f.name = 'cmr10-red'
            f.type = 'virtual'
            f.fonts = {{'cmr10', size}}
            for i,v in pairs(f.characters) do
                if (string.char(i)):find("[tacohanshartmut]") then
                    v.commands = {
                        {'special','pdf: 1 0 0 rg'},
                        {'char',i},
                        {'special','pdf: 0 g'},
                    }
                else
                    v.commands = {{'char',i}}
                end
            end
        else
          f = font.read_tfm(name,size)
        end
        return f
        end )
     }

\font\myfont = cmr10-red \myfont  This is a line of text \par
\font\myfontx= cmr10 \myfontx Here is another line of text \par

\bye
% end of virtual-demo.tex
```

6 Modifications

Besides the expected changes caused by new functionality, there are a number of not-so-expected changes. These are sometimes a side-effect of a new (conflicting) feature, or, more often than not, a change necessary to clean up the internal interfaces.

6.1 Changes from TEX 3.141592

- There is no pool file, all strings are embedded during compilation.
- "plus 1 fillll" does not generate an error. The extra 'l' is simply typeset.

6.2 Changes from ε-TEX 2.2

- The ε-TEX functionality is always present and enabled (but see below about TEXXET), so the prepended asterisk or `-etex` switch for initex is not needed.
- TEXXET is not present, so the primitives

  ```
  \TeXXeTstate
  \beginR
  \beginL
  \endR
  \endL
  ```

  are missing

6.3 Changes from PDFTEX 1.40

- A number of 'utility functions' is removed:

  ```
  \pdfelapsedtime
  \pdfescapehex
  \pdfescapename
  \pdfescapestring
  \pdffiledump
  \pdffilemoddate
  \pdffilesize
  \pdflastmatch
  \pdfmatch
  \pdfmdfivesum
  \pdfresettimer
  \pdfshellescape
  ```

```
\pdfstrcmp
\pdfunescapehex
```

- A few other experimental primitives are provided without the extra 'pdf' prefix, so they are simply called:

```
\primitive
\ifprimitive
\ifabsnum
\ifabsdim
```

6.4 Changes from ALEPH RC4

- The input translations from Aleph are not implemented, the related primitives are not available

```
\DefaultInputMode
\noDefaultInputMode
\noInputMode
\InputMode
\DefaultOutputMode
\noDefaultOutputMode
\noOutputMode
\OutputMode
\DefaultInputTranslation
\noDefaultInputTranslation
\noInputTranslation
\InputTranslation
\DefaultOutputTranslation
\noDefaultOutputTranslation
\noOutputTranslation
\OutputTranslation
```

- A small series of bounds checking fixes to `\ocp` and `\ocplist` has been added to prevent the system from crashing due to array indexes running out of bounds.
- The `\hoffset` bug when `\pagedir TRT` is fixed, removing the need for an explicit fix to `\hoffset`
- A bug causing `\fam` to fail for family numbers above 15 is fixed.
- Some bits of Aleph assumed `0` and `null` were identical. This resulted for instance in a bug that sometimes caused an eternal loop when trying to `\show` a box.
- A fair amount of minor bugs are fixed as well, most of these related to `\tracingcommands` output.
- The number of possible fonts, ocps and ocplists is smaller than their maximum Aleph value (around 5000 fonts and 30000 ocps / ocplists).
- The internal function `scan_dir()` has been renamed to `scan_direction()` to prevent a naming clash.

6.5 Changes from standard WEB2C

- There is no mltex
- There is no enctex
- The following command-line switches are silently ignored, even in non—lua mode:

  ```
  -8bit
  -translate-file=TCXNAME
  -mltex
  -enc
  -etex
  ```

- `\openout` whatsits are not written to the log file.
- Some of the so—called web2c extensions are hard to set up in non-kpse mode because texmf.cnf is not read: `shell-escape` is off (but that is not a problem because of Lua's `os.execute`), and the paranoia checks on `openin` and `openout` do not happen (however, it is easy for a Lua script to do this itself by overloading `io.open`).

7 Implementation notes

1 Primitives overlap

The primitives

```
\pdfpagewidth and \pagewidth,
\pdfpageheight and \pageheight,
\fontcharwd and \charwd,
\fontcharht and \charht,
\fontchardp and \chardp,
\fontcharic and \charic,
```

are all aliases of each other.

2 Sparse arrays

The `\mathcode`, `\delcode`, `\catcode`, `\sfcode`, `\lccode` and `\uccode` tables are now sparse arrays that are implemented in C. They are no longer part of the TEX "equivalence table" and because each had 1.1 million entries with a few memory words each, this makes a major difference in memory usage.

These assignments do not yet show up when using the etex tracing routines `\tracingassigns` and `\tracingrestores` (code simply not written yet)

A side-effect of the current implementation is that `\global` is now more expensive in terms of processing than non-global assignments.

See `mathcodes.c` and `textcodes.c` if you are interested in the gory details.

Also, the glyph ids within a font are now managed by means of a sparse array and glyph ids can go up to index $2^21 - 1$.

3 Simple single-character csnames

Single-character commands are no longer treated aspecially in the internals, they are stored in the hash just like the multiletter csnames.

The code that displays control sequences explicitly checks if the length is one when it has to decide whether or not to add a trailing space.

4 Compressed format

The format is passed through zlib, allowing it to shrink to roughly a third of the size it would have had in uncompressed form. This takes a bit more CPU cycles but much less disk I/O, so it should still be faster.

The chosen compression factor is fairly low, equivalent to `gzip -3`.

5 Binary file reading

All of the internal code is changed in such a way that if one of the `read_xxx_file` callbacks is not set, then the file is read by a C function using basically the same convention as the callback: a single read into a buffer big enough to hold the entire file contents. While this uses more memory than the previous code (that mostly used `getc` calls), it can be quite a bit faster (depending on your I/O subsystem).

8 Known bugs

The bugs below are going to be fixed eventually.

The top ones will be fixed soon, but in the later items either the actual problem is hard to find, or the code that causes the bug is going to be replaced by a new subsystem soon anyway.

- Not all of Aleph's direction commands are handled properly in PDF mode yet: this affects all the Top-Bottom and Bottom-Top writing directions.
- Letter spacing (`\letterspacefont`) is currently non-functional due to massive changes in the virtual font handling. This functionality may actually be removed completely in the future, because it is straightforward to set up letterspacing using the Lua 'define_font' interface.
- Attempting hyphenation in initex (sometimes) creates segfaults.
- Hyphenation can only deal with the Base Multilingual Plane (BMP)
- `tex.print()` and `tex.sprint()` do not work if `\directlua` is used in an OTP file (in the output of an `expression` rule).

9 TODO

On top of the 'normal' extensions that are planned, there are some more specific small feature requests .

- Implement the TEX primitive `\dimension`, cf. `\number`
- Change the lua table typetex.dimen to accept and return float values instead of strings
- Do something about `\withoutpt` and/or a new register type `\real`?
- Implement the TEX primitive `\htdp`?
- Do boxes with dual baselines.
- A way to (re?)calculate the width of a `\vbox`, taking only the natural width of the included items into account.
- Make the number of the output box configurable.