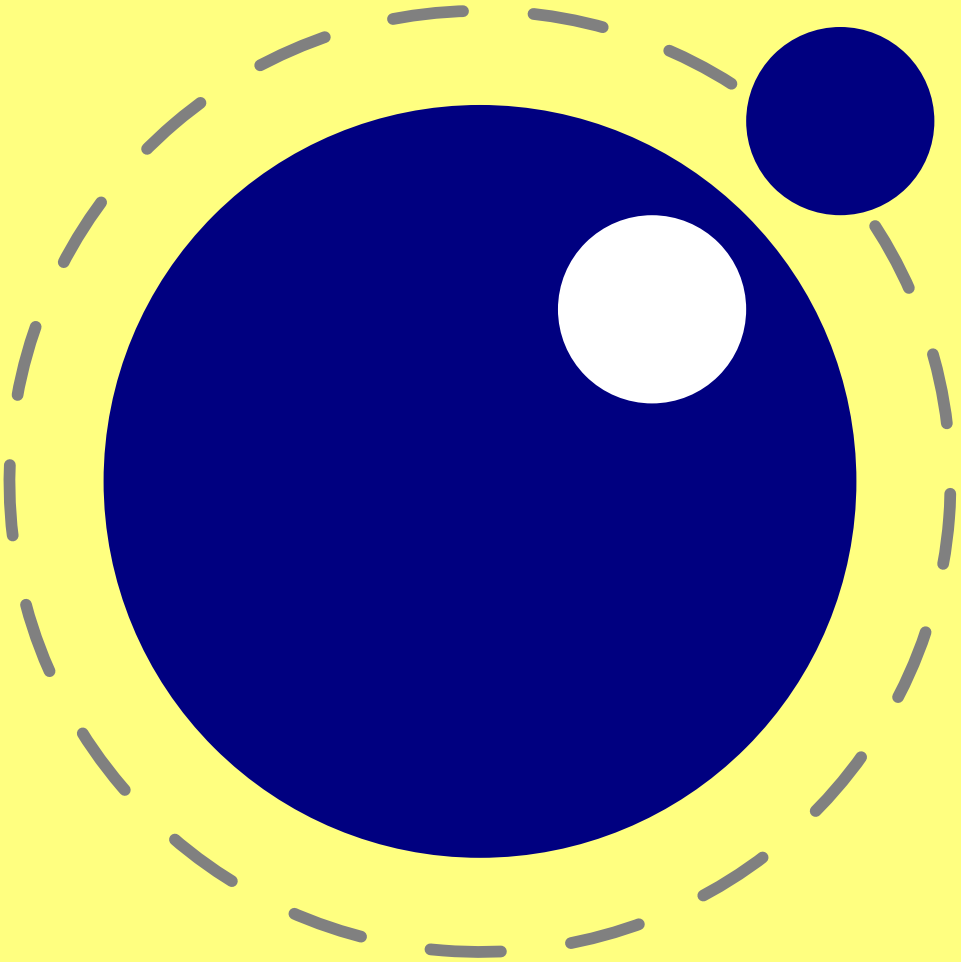
LuaTEX Reference

snapshot 2007-06-19



LuaTEX Reference Manual

more info: www.luatex.org
version: June 20, 2007(snapshot 2007-06-19)

Contents

1 Introduction

This book will eventually become the reference manual of LuaTeX. At the moment, it simply reports the behaviour 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 5.1.2
- Dedicated Lua libraries
- Various TeX extensions
- The (OpenType) Font Parser from FontForge 2006.12.20
- Compiled source code to glue it all together

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 All these encoding-related functions are superseded by a Lua-based solution (reader callbacks).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. The maximum number of allocations is `"10FFFF` or $2^{20} + 2^{16}$ (21 bits). The maximum value that can be assigned are:

primitive	bits	hex	numeric
`\char`	21	10FFFF	$2^{20} + 2^{16}$
`\chardef`	21	10FFFF	$2^{20} + 2^{16}$
`\lccode`	21	10FFFF	$2^{20} + 2^{16}$
`\uccode`	21	10FFFF	$2^{20} + 2^{16}$
`\sfcode`	15	7FFF	2^{15}
`\catcode`	4	F	2^4
`\mathchardef`	15	8000	$2^3 \times 2^8 \times 2^4$
`\mathcode`	15	8000	$2^3 \times 2^8 \times 2^4$
`\delcode`	27	7FFFFFF	$2^3 \times 2^4 \times 2^8 \times 2^4 \times 2^8$

As far as the core engine is concerned, all input and output to text files is UTF-8 encoded. Input files can be pre-processed 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	max index/bits	**hex**	**numeric**
\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	$2^{20} + 2^{16} = 2^3 * 2^8 * 2^{16}$
		+ FFFFFF	$+2^8 * 2^{16}$

2.3 Extended register tables

All registers can be 16 bit numbers as in Aleph. The affected commands are:

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

2.4 Attribute registers

Attributes are a completely new concept to LuaTeX. Syntactically, they behave a lot like counters: attributes obey TeX's nesting stack and can be used after \the etc. just like the normal \count registers.

> \attribute ⟨16-bit number⟩ ⟨optional equals⟩ ⟨31-bit number⟩\crlf \attributedef ⟨csname⟩ ⟨optional equals⟩ ⟨16-bit number⟩

However, there are some differences as well. Conceptually, an attribute is either set or unset. Set attributes can only have values of 0 or more, otherwise they are considered unset and automatically remapped to −1. All attributes start out in the unset state (in iniTeX).

Attributes can be used as extra counter values, but their usefulness comes from the fact that the numbers and values of all set attributes are attached to all nodes created in their scope. These can then be queried from any Lua code that deals with node processing. Further information is given in the chapter on 6.

2.5 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`.

2.5.1 `\directlua`

The primitive `\directlua` is used to execute LUA code immediately. 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.

2.5.2 `\latelua`

`\latelua` stores LUA code in a whatsit that will be processed inside the output routine. It's intended use 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⟩

2.5.3 `\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⟩

2.5.4 `\closelua`

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

`\closelua` ⟨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)

2.6 New ε-TEX primitives

2.6.1 \clearmarks

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

\clearmarks ⟨16-bit number⟩

2.6.2 \noligs and \nokerns

These primitives prohibit ligature and kerning insertion at the time when the initial node list is built by LuaTEX's main control loop (It is a temporary trick that will be removed soon).

\noligs ⟨integer⟩ \nokerns ⟨integer⟩

2.6.3 \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.

2.6.4 \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

2.6.5 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.

2.6.5.1 \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).

2.6.5.2 \initcatcodetable

\initcatcodetable ⟨28-bit number⟩

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

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

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.

2.6.5.3 \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.

2.6.6 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 `texlua` (or `luatexlua`)
- if the non-option (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 somehwat bulky standalone LUA interpreter.

3.1.2 Other commandline processing

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

In this mode, it will only interpret a very small subset of the commandline 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 commandline, 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`, `token`, `node` 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`, `token`, `node` 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 commandline, 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:

library	functions
os	execute exec setenv rename remove
io	popen output tmpfile
lfs	rmdir mkdir chdir lock 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 commandline options:

flag	meaning
`--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 0.6), 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 `CR+LF` are accepted.

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

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:

- `string.utfvalues(s)` (returns an integer value in the UNICODE range)
- `string.utfcharacters(s)` (returns a string with a single UTF-8 token in it)
- `string.characters(s)` (a string of length one)
- `string.characterpairs(s)` (two strings of length one) will produce an empty second string in the string length was odd.
- `string.bytes(s)` (a single byte value)
- `string.bytepairs(s)` (two byte values) Will produce nil instead of a number as its second return value if the string length was odd.

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 TEXLUA.
- `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 attribute, count, dimension and token registers

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

tex.attribute
tex.count
tex.dimen
tex.toks

It is possible to use the names of relevant \attributedef, \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 `\attributedef`, or `\dimendef`, or `\toksdef`), anything else will generate an error (the intent is to eventually also allow `<chardef tokens>` and even macros that expand into a number)

The attribute and 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 registers

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, number n)
number n = tex.getboxwd(number n)

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

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

It is also possible to set and query actual boxes, using the node interface as defined in the `node` library:

```
tex.box
```

for array access, or

```
tex.setbox(number n, node s)
node n = tex.getbox(number n)
```

for function-based access

Be warned that an assignment like

```
tex.box[0] = tex.box[2]
```

does not copy the node list, it just duplicates a node pointer. If \box2 will be cleared by TEX commands later on, the contents of \box0 becomes invalid as well. To prevent this from happening, always use `node.copy_list()` unless you are assigning to a temporary variable:

```
tex.box[0] = node.copy_list(tex.box[2])
```

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 token library

The `token` table contains interface functions to TEX's handling of tokens. These functions are most useful when combined with the `token_filter` callback, but they could be used standalone as well.

A token is represented in Lua as a small table. For the moment, this table consists of three numeric entries:

nr	meaning	description
1	command code	this is a value between 0 and 130 (approximately)
2	command modifier	this is a value between 0 and 2^{21}
3	control sequence id	for commands that are not te result of control sequences, like letters and characters, it is zero, otherwise, it is number pointing into the equivalence table

4.2.1 token.get_next

```
token t = token.get_next()
```

This fetches the next input token from the current input source, without expansion.

4.2.2 token.is_expandable

```
boolean b = token.is_expandable(token t)
```

This tests if the token `t` could be expanded.

4.2.3 token.expand

```
token.expand()
```

If a token is expandable, this will expand one level of it, so that the first token of the expansion will now be the next token to be read by `tex.get_next()`.

4.2.4 token.is_activechar

```
boolean b = token.is_activechar(token t)
```

This is a special test that is sometimes handy. Discovering whether some token is the result of an active character turned out to be very hard otherwise.

4.2.5 token.create

```
token t = token.create(string csname)
token t = token.create(number charcode)
token t = token.create(number charcode, number catcode)
```

This is the token factory. If you feed it a string, then it is a control sequence csname, and it will be looked up in the equivalence table.

If you feed it number, then this is assumed to be an input character, and an optional second number gives its category code. This means it is possible to overrule a character's category code, with a few exceptions: the category codes 0 (escape), 9 (ignored), 13 (active), 14 (comment), and 15 (invalid) cannot occur inside a token. The values 0, 9, 14 and 15 are therefore illegal as input to `token.create()`, and active characters will be resolved immediately.

Note: unknown string sequences and never defined active characters will result in a token representing an undefined control sequence with a near-random name. It is *not* possible to define brand new control sequences using `token.create`!

4.2.6 token.command_name

```
string commandname = token.command_name(token t)
```

This returns the name associated with the command value of the token in LuaTeX. There is no direct connection between these names and primitives. For instance, all `\ifxxx` tests are grouped under `if_fest`, and the command modifier defines which test is to be run.

4.2.7 token.command_id

```
number i = token.command_name(string commandname)
```

This returns a number that is the inverse operation of the previous command, to be used as the first item in a token table.

4.2.8 token.csname_name

```
string csname = token.csname_name(token t)
```

This returns the name associated with the equivalence table value of the token in LuaTeX. It returns the string value of the command used to create the current token, or an empty string if there is no associated control sequence.

4.2.9 token.csname_id

```
number i = token.csname_id(string csname)
```

This returns a number that is the inverse operation of the previous command, to be used as the third item in a token table.

4.3 The node library

The `node` library contains functions that facilitate dealing with (lists of) nodes and their values.

LuaTeX nodes are represented in Lua as userdata with the metadata type `luatex.node`. The various fields within a node can be accessed using fields that are indexed by strings.

Each node has at least the three fields `next`, `id`, and `subtype`:

- The `next` field returns the userdata object for the next node in a linked list of nodes, or nil, if there is no next node.
- The `id` indicates TEX's node type. The field `id` has a numeric value for efficiency reasons, but some of the library functions also accept a string value instead of `id`.
- The `subtype` is another number. It often gives further information about a node of a particular `id`, but it is most important when dealing with whatsits, because they are differentiated solely based on their `subtype`.

The other available fields depend on the `id` (and for whatsits, the `subtype`) of the node. Further details on the various fields and their meanings are given in the chapter 6.

TEX's math nodes are not yet supported, Sso, there is not yet an interface to the internals of the math lists, and it is not possible to create them from Lua. Support for `unset` (alignment) nodes is partial: they can be queried and modified from Lua code, but not created.

4.3.1 Node handling functions

4.3.1.1 `node.types`

```
table t = node.types()
```

This function returns an array that maps node id numbers to node type strings, providing an overview of the possible top-level `id` types.

4.3.1.2 `node.whatsits`

```
table t = node.whatsits()
```

TEX's whatsits all have the same `id`. The various subtypes are defined by their `subtype`. The function is much like `node.id`, except that it provides an array of `subtype` mappings.

4.3.1.3 `node.id`

```
number id = node.id(string type)
```

This converts a single type name to it's internal numeric representation.

4.3.1.4 `node.subtype`

```
number subtype = node.subtype(string type)
```

This converts a single whatsit name to it's internal numeric representation (subtype).

4.3.1.5 node.type

```
string type = node.type(number id)
```

This converts a internal numeric representation to an external string representation.

4.3.1.6 node.fields

```
table t = node.fields(string type)
table t = node.fields(string type, string subtype)
```

This function returns an array of valid field names for a particular type of node. If you want to the the valid fields for a whatsit, then you have to supply the second argument also. Otherwise, the second argument will be silently ignored.

This function accepts numeric id and subtype values as well.

4.3.1.7 node.new

```
node n = node.new(string type)
node n = node.new(string type, string subtype)
```

Creates a new node. All fields are initialized to either zero or null (depending on wether the field is a number or a pointer), except for id and subtype (if supplied). If you need a new whatsit, then the second argument is required.

This function accepts numeric id and subtype values as well.

4.3.1.8 node.free

```
node.free(node n)
```

Removes the node m from TEX's memory. Be careful: no checks are done on whether this node is still pointed to from a register or some next field: it is up to you to make sure that the internal data structures remain correct.

4.3.1.9 node.flush_list

```
node.flush_list(node n)
```


Removes the node list `m` and the complete node list following `m` from TEX's memory. Be careful: no checks are done on whether any of these nodes is still pointed to from a register or some `next` field: it is up to you to make sure that the internal data structures remain correct.

4.3.1.10 node.copy

```
node m = node.copy(node n)
```

Creates a deep copy of node `n`. Only the `next` field is not copied.

4.3.1.11 node.copy_list

```
node m = node.copy_list(node n)
```

Creates a deep copy of the node list that starts at `n`.

4.3.1.12 node.slide

```
node m = node.slide(node n)
```

Returns the last node of the node list that starts at `n`.

4.3.1.13 node.length

```
number i = node.length(node n)
number i = node.length(node n, node m)
```

Returns the number of nodes contained in the node list that starts at `n`. If `m` is also supplied it stops at `m` instead of at the end of the list. The node `m` is not counted.

4.3.1.14 node.count

```
number i = node.count(string type, node n)
number i = node.count(string type, node n, node m)
```

Returns the number of nodes contained in the node list that starts at `n` that have an `id` matching `type`. If `m` is also supplied, counting stops at `m` instead of at the end of the list. The node `m` is not counted.

4.3.1.15 node.traverse

```
node t = node.traverse(node n)
node t = node.traverse(node n, node m)
```

This is an iterator that loops over the node list that starts at `n`. If `m` is also supplied, the iterator stops at `m` instead of at the end of the list. The node `m` is not processed.

4.3.1.16 node.traverse_id

```
node t = node.traverse_id(string type, node n, node m)
node t = node.traverse_id(string type, node n)
```

This is an iterator that loops over all the nodes in the list that starts at `n` that have an `id` matching `type`. If `m` is also supplied, the iterator stops at `m` instead of at the end of the list. The node `m` is not processed.

4.3.2 Attribute handling

Attributes appear as linked list of userdata objects in the `attr` field of individual nodes. They can be handled individually, but it much safer and more efficient to use the dedicated functions associated with them.

4.3.2.1 node.has_attribute

```
number v = node.has_attribute(node n, number id)
number v = node.has_attribute(node n, number id, number val)
```

Tests if a node has the attribute with number `id` set. If `val` is also supplied, also tests if the value matches `val`. It returns the value, or, if no match is found, nil.

4.3.2.2 node.set_attribute

```
node.set_attribute(node n, number id, number val)
```

Sets the attribute with number `id` to the value `val`. Duplicate assignments are ignored.

4.3.2.3 node.unset_attribute

```
node.unset_attribute(node n, number id, number val)
node.unset_attribute(node n, number id)
```

Unsets the attribute with number `id`. If `val` is also supplied, it will only perform this operation if the value matches `val`. Missing attributes or attribute-value pairs are ignored.

4.4 The texio library

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

4.4.1 Printing functions

4.4.1.1 texio.write

```
texio.write(string target, string 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.

4.4.1.2 tex.write_nl

```
texio.write_nl(string target, string 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.

4.5 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, that can be used from within a \latelua argument. This function is not to be used inside \directlua unless you know *exactly* what you are doing.

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`.

4.6 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`.

4.6.1 File discovery callbacks

4.6.1.1 `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

This number is zero for the log or \input files. For TEX's \read or \write the number is incremented by one, so \read0 becomes 1.

asked_name
This is the user-supplied filename, as found by \input, \openin or \openout.

Return value:

actual_name
This is 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.
You have to return nil if the file cannot be found.

4.6.1.2 find_font_file

Your 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.

4.6.1.3 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.

4.6.1.4 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).

4.6.1.5 find_vf_file

Like find_font_file, but for virtual fonts. This applies to both ALEPH's OVF files and traditional Knuthian VF files.

4.6.1.6 find_ocp_file

Like find_font_file, but for ocp files.

4.6.1.7 find_map_file

Like find_font_file, but for map files.

4.6.1.8 find_enc_file

Like find_font_file, but for enc files.

4.6.1.9 find_sfd_file

Like find_font_file, but for subfont definition files.

4.6.1.10 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.

4.6.1.11 find_data_file

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

4.6.1.12 find_opentype_file

Like find_font_file, but for OpenType font files.

4.6.1.13 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.

4.6.1.14 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.

4.6.2 File reading callbacks

4.6.2.1 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.

4.6.2.1.1 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.

4.6.2.1.2 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.

4.6.2.2 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.

4.6.2.3 read_vf_file

Like read_font_file, but for virtual fonts.

4.6.2.4 read_ocp_file

Like read_font_file, but for ocp files.

4.6.2.5 read_map_file

Like read_font_file, but for map files.

4.6.2.6 read_enc_file

Like read_font_file, but for enc files.

4.6.2.7 read_sfd_file

Like read_font_file, but for subfont definition files.

4.6.2.8 read_pk_file

Like read_font_file, but for pk bitmap files.

4.6.2.9 read_data_file

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

4.6.2.10 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.

4.6.2.11 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.

4.6.2.12 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.

4.6.3 Data processing callbacks

4.6.3.1 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.

4.6.3.2 token_filter

This callback allows you to change the fetch and preprocess any lexical token that enters LuaTeX, before LuaTeX executes or expands the associated command.

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

The calling convention for this callback is bit more complicated then for most other callbacks. The function should either return a Lua table representing a valid to-be-processed token or tokenlist, 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, until it eventually does return a useful token or tokenlist (or until you reset the callback value to nil). See the description of `token` for some handy functions to be used in conjunction with this callback.

If your function returns a single usable token, then that token will be processed by LuaTeX immediately. If the function returns a token list (a table consisting of a list of consecutive token tables), then that list will be pushed to the input stack as completely new token list level, with it's token type set to inserted. In either case, the returned token(s) will not be fed back into the callback function.

4.6.4 Node list processing callbacks

The description of nodes and node lists is in the chapter 6.

4.6.4.1 buildpage_filter

This callback is called whenever LuaTeX is ready to move stuff to the main vertical list. You can use this callback to do specialized manipulation of the page building stage like imposition or column balancing.

```
function(node <head>)
    return true | false | node <newhead>
end
```

As for all the callbacks that deal with nodes, the return value can be one of three things:

- `boolean true` signals succesful processing
- `node` signals that the head node should be replaced by this node
- `boolean false` signals that the head node list should be ignored and flushed from memory

4.6.4.2 pre_linebreak_filter

This callback is called just before LuaTeX starts converting a list of nodes into a stack of `\hboxes`. The removal of any final skip and the subsequent insertion of `\parfillskip` has not happened yet at that moment.

```
function(node <head>, string <groupcode>, int <glyph_count>)
    return true | false | node <newhead>
end
```

The string called groupcode identifies the nodelist's context within TEX's processing. The range of possibilities is given in the table below, but not all of those can actually appear in pre_linebreak_filter, some are for the Xpack_filter callbacks that will be explained in the next two paragraphs.

value	explanation
hbox	\hbox in horizontal mode
adjusted_hbox	\hbox in vertical mode
vbox	\vbox
vtop	\vtop
align	\halign or \valign
disc	discretionaries
insert	packaging an insert
vcenter	\vcenter
local_box	\localleftbox or \localrightbox
split_off	top of a \vsplit
split_keep	remainder of a \vsplit
preamble	alignment preamble
align_set	alignment cell
fin_row	alignment row

4.6.4.3 hpack_filter

This callback is called when TEX is ready to start boxing some horizontal mode material. Math items are ignored at the moment.

```
function(node <head>, number <size>, string <packtype>,
         string <groupcode>, number <glyphcount>)
    return true | false | node <newhead>
end
```

The packtype is either additional or exactly. If additional, then the size is a \hbox spread ... argument. If exactly, then the size is a \hbox to .... In both cases, the number is in scaled points.

4.6.4.4 vpack_filter

This callback is called when TEX is ready to start boxing some vertical mode material. Math displays are ignored at the moment.

This function is very similar to the hpack_filter. Besides the fact that it is called at different moments, there is an extra variable that matches TEX's \maxdepth setting, but there is no glyph count.

```
function(node <head>, number <size>, string <packtype>,
        number <maxdepth>, string <groupcode>)
    return true | false | node <newhead>
end
```

4.6.4.5 pre_output_filter

This callback is called when TEX is ready to start boxing the box 255 for \output.

```
function(node <head>, number <size>, string <packtype>,
        number <maxdepth>, string <groupcode>)
    return true | false | node <newhead>
end
```

4.6.5 Information reporting callbacks

4.6.5.1 start_run

```
function()
```

Replaces the code that prints LUATEX's banner

4.6.5.2 stop_run

```
function()
```

Replaces the code that prints LUATEX's statistics and output written to messages.

4.6.5.3 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.

4.6.5.4 stop_page_number

```
function()
```

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

4.6.5.5 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 status 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.

4.6.6 Font-related callbacks

4.6.6.1 define_font

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

The string <name> is the filename 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.

4.7 The `lua` library

This library contains two read-only items:

4.7.1 Variables

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

This sets the id number of the instance.

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

This sets a LuaTeX version identifier string (currently `"0.1"`).

4.7.2 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)
```

4.8 The `kpse` library

4.8.1 `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)
string f = kpse.find_file(string filename, string ftype, number dpi)
```

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

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.

dpi
: This is used for the size argument of the formats `pk`, `gf`, and `bitmap font`.

4.8.2 kpse.set_program_name

Sets the KPATHSEA executable (and optionally program) name

```
kpse.set_program_name(string name)
kpse.set_program_name(string name, string progname)
```

The second argument controls the use of the dotted values in the `texmf.cnf` configuration file, and defaults to the first argument.

4.8.3 kpse.init_prog

Extra initialization for programs that need to generate bitmap fonts.

```
kpse.init_prog(string prefix, number base_dpi, string mfmode)
kpse.init_prog(string prefix, number base_dpi, string mfmode, string fallback)
```

4.8.4 kpse.readable_file

Test is a (absolute) file name is a readable file

```
string f = kpse.readable_file(string name)
```

The return value is the actual absolute filename you should use, because the disk name is not always the same as the requested name, due to aliases and system-specific handling under e.g. MSDOS.

Returns `nil` if the file does not exist or is not readable.

4.8.5 kpse.expand_path

Like kpsewhich's `-expand-path`:

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


4.8.6 kpse.expand_var

Like kpsewhich's -expand-var:

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

4.8.7 kpse.expand_braces

Like kpsewhich's -expand-braces:

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

4.8.8 kpse.var_value

Like kpsewhich's -var-value:

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

4.9 The status 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> = status.list()
```

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

Almost all of the values in status are fetched through a metatable at run-time whenever they are accessed, so you cannot use pairs on status, 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 status.

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
var_mem_max	number of allocated words for nodes
fix_mem_max	number of allocated words for tokens
fix_mem_end	maximum number of used tokens
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 Lua bytecode registers
luabytecode_bytes	number of bytes in Lua bytecode registers
luastates	number of active Lua interpreters

luastate_bytes	number of bytes in use by Lua interpreters
output_active	true if the \output routine is active

4.10 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
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 commandline, this key will be tested first instead of simply quitting
jobname	string		If no input file name was given on the commandline, this key will be tested first instead of simply giving up

4.11 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.

4.11.1 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.

4.11.2 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.

4.11.3 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**
table_version	number	indicates the `read_otf()` version
fontname	string	
fullname	string	
familyname	string	
weight	string	
copyright	string	
filename	string	
defbasefilename	string	
version	string	
italicangle	float	
upos	float	
uwidth	float	
units_per_em	number	
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	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	
gentags	table	
possub	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	

4.11.3.1 Glyph items

The glyphs is an array containing the per-character information (quite a few of these are only present if nonzero).

key	type	explanation
name	string	
unicodeenc	number	
boundingbox	array	array of four numbers
width	number	(only for horizontal fonts)
vwidth	number	(only for vertical fonts)
lsidebearing	number	(only if nonzero)
glyph_class	number	(only if nonzero)
kerns	array	(only for horizontal fonts, if set)
vkerns	array	(only for vertical fonts, if set)
dependents	array	linear array of glyph name strings (only if nonempty)
possub	table	(only if nonempty)
ligofme	table	(only if nonempty)
comment	string	(only if set)
color	number	(only if set)
tex_height	number	(only if set)
tex_depth	number	(only if set)
tex_sub_pos	number	(only if set)
tex_super_pos	number	(only if set)

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
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	

4.11.3.2 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	
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	(only if nonzero)
is_unicodefull	number	(only if nonzero)
is_custom	number	(only if nonzero)
is_original	number	(only if nonzero)
is_compact	number	(only if nonzero)
is_japanese	number	(only if nonzero)
is_korean	number	(only if nonzero)
is_tradchinese	number	(only if nonzero)
is_simplechinese	number	
low_page	number	
high_page	number	
iconv_name	string	
iso_2022_escape	string	

4.11.3.3 private table

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

4.11.3.4 cidinfo table

registry	string
ordering	string
supplement	number
version	number

4.11.3.5 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, Scri
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.

4.11.3.6 names table

Each item has two top-level keys:

	key	type	explanation
lang	`string`	language for this entry	
names	`table`		

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

key	type	explanation
`copyright`		
`family`		
`subfamily`		
`uniqueid`		
`fullname`		
`version`		
`postscriptname`		
`trademark`		
`manufacturer`		
`designer`		
`descriptor`		
`venderurl`		
`designerurl`		
`license`		
`licenseurl`		
`idontknow`		
`preffamilyname`		
`prefmodifiers`		
`compatfull`		
`sampletext`		
`cidfindfontname`		

4.11.3.7 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	

4.11.3.8 orders table

key	type	explanation
table_tag	string	
ordered_features	array	list of tag strings

4.11.3.9 ttf_tables table

key	type	explanation
tag	string	
len	number	
maxlen	number	
data	number	

4.11.3.10 script_lang table

key	type	explanation
script	string	
langs	array	list of language tags

4.11.3.11 kerns table

Substructure is identical to the per-glyph subtable.

4.11.3.12 vkerns table

Substructure is identical to the per-glyph subtable.

4.11.3.13 texdata table

key	type	explanation
type	string	unset, text, math, mathext
params	array	22 font numeric parameters

4.11.3.14 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	

4.11.3.15 possub table

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

key	type	explanation
type	number	
format	enum	glyphs, class, coverage, reversecoverage
script_lang_index	number	
flags	number	(only if nonzero)
tag	string	
nccnt	number	(only if nonzero)
bccnt	number	(only if nonzero)
fccnt	number	(only if nonzero)
nclass	array	
bclass	array	
fclass	array	
rules	array	an array of rule items

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
reversecoverage	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

reversecoverage:

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

4.11.4 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 five keys: `fontname`, `fullname`, `familyname`, `weight` and `table_version`.

4.11.5 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.

4.11.6 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).

4.11.7 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.

4.11.8 Currently active font

```
number i = font.currentid;
```

This is the currently used font number.

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)
encodingbytes	no	no	yes	number	default: depends on `format`
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	number	meaning
0	LT	8	TT
1	LL	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 TrueType) 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	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 applied, 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	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.

textual (Knuth)	number	string	result
l + r =: n	0	`=:`	|n
l + r =:| n	1	`=:|`	|nr
l + r |=: n	2	`|=:`	|ln
l + r |=:| n	3	`|=:|`	|lnr
l + r =:|> n	5	`=:|>`	n|r
l + r |=:> n	6	`|=:>`	l|n
l + r |=:|> n	7	`|=:|>`	l|nr
l + r |=:|>> n	11	`|=:|>>`	ln|r

The default value is 0, and can be left out. That signifies a normal ligature where the ligature replaces both original glyphs. In this table the | indicates the final insertion point.

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 rule is: if the font table has `encodingbytes` set to 2, then this is a wide font, in all other cases it isn't. The value 2 is the default for OpenType and TrueType fonts loaded via Lua.

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 (Type1 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, `subset` only works for PostScript-based non-cid OpenType fonts, every other font format essentially is treated as `full`.

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. Dealing with this is on the agenda.

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
node	1	node	output this node (list), 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 Nodes

6.1 LUA node representation

TEX's nodes are represented in LUA as userdata object with a variable set of fields.

The current return value of `node.types()` is: `vlist` (1), `rule` (2), `ins` (3), `mark` (4), `adjust` (5), `disc` (7), `whatsit` (8), `math` (9), `glue` (10), `kern` (11), `penalty` (12), `unset` (13), `style` (14), `choice` (15), `ord` (16), `op` (17), `bin` (18), `rel` (19), `open` (20), `close` (21), `punct` (22), `inner` (23), `radical` (24), `fraction` (25), `under` (26), `over` (27), `accent` (28), `vcenter` (29), `left` (30), `right` (31), `action` (39), `margin_kern` (40), `glyph` (41), `attribute` (42), `glue_spec` (43), `attribute_list` (44), `hlist` (0), but as already mentioned, the math and alignment nodes in this list are not supported at the moment. The useful list is described in the next sections.

6.1.1 Auxiliary items

A few node-typed userdata objects do not occur in the normal list of nodes, but can be pointed to from within that list. They are not quite really nodes, but it is easier for the library routines to treat them as if they were.

6.1.1.1 glue_spec items

Skips are about the only type of data objects in traditional TEX that are not a simple value. The structure that represents the glue components of a skip is called a `glue_spec`, and it has the following accessible fields:

key	type	explanation
`width`	number	
`stretch`	number	
`stretch_order`	number	
`shrink`	number	
`shrink_order`	number	

These objects are reference counted, so there is actually an extra field named `ref_count` as well, but that will become completely hidden in the near future.

6.1.1.2 attribute_list items

The newly introduced attribute registers are also non-trivial, because the value that is attached to a node is essentially a sparse array of key-value pairs.

It is generally easiest to deal with attributes by using the dedicated functions in the `node` library, but for completeness, here is the low-level interface:

field	type	explanation
next	node	pointer to the first attribute

There are no extra fields, this kind of item is only used as a head pointer for attribute items, making them easier to handle.

A normal node's attribute field will point to a field of this type, and the next field in that field will then point to the first defined attribute item, whose next will point to the second attribute item, etc.

6.1.1.3 attribute item

Valid fields:

field	type	explanation
next	node	pointer to the next attribute
number	number	the LHS
value	number	the RHS

6.1.2 Main text nodes

These are the nodes that comprise actual typesetting commands.

A few fields are present in all nodes regardless of their type, these are:

field	type	explanation
next	node	The next node in a list, or nil
id	number	The node's type (id) number
subtype	number	The node subtype identifier

The subtype is sometimes just a stub entry. Almost all nodes also have an attr field. In the following tables next and id are not explicitly mentioned.

6.1.2.1 hlist nodes

Valid fields: attr, width, depth, height, dir, shift, glue_order, glue_sign, glue_set, list

field	type	explanation
subtype	number	unused
attr	node	The head of the associated attribute list
width	number	
height	number	
depth	number	
shift	number	a displacement perpendicular to the character progression direction
glue_order	number	a number in the range 0–4, indicating the glue order
glue_set	number	the calculated glue ratio

glue_sign	number	
list	node	the body of this list
dir	number	the direction of this box

6.1.2.2 vlist nodes

Valid fields: As for hlist, except that shift is a displacement perpendicular to the line progression direction.

6.1.2.3 rule nodes

Valid fields: attr, width, depth, height, dir

field	type	explanation
subtype	number	unused
attr	node	
width	number	Rule size. The special value −1073741824 is used for running glue dimensions
height	number	' '
depth	number	' '
dir	number	the direction of this rule

6.1.2.4 ins nodes

Valid fields: attr, cost, depth, height, top_skip, list

field	type	explanation
subtype	number	the insertion class
attr	node	
cost	number	The penalty associated with this insert
height	number	
depth	number	
list	node	the body of this insert
top_skip	node	a pointer to the \splittopskip glue spec

6.1.2.5 mark nodes

Valid fields: attr, class, mark

field	type	explanation
subtype	number	unused
attr	node	
class	number	the mark class
mark	table	a table representing a token list

6.1.2.6 adjust nodes

Valid fields: attr, list

field	type	explanation
subtype	number	1 = pre
attr	node	
list	node	adjusted material

6.1.2.7 disc nodes

Valid fields: attr, pre, post, replace

field	type	explanation
subtype	number	unused
attr	node	
pre	node	pointer to the pre-break text
post	node	pointer to the post-break text
replace	number	the number of nodes to skip if this discretionary is chosen as a breakpoint

6.1.2.8 math nodes

Valid fields: attr, surround

field	type	explanation
subtype	number	0 = on, 1 = off
attr	node	
surround	number	width of the \mathsurround kern

6.1.2.9 glue nodes

Valid fields: attr, spec, leader

field	type	explanation
subtype	number	0 = \skip, 1–18 = internal glue parameters, 100 = \leaders, 101 = \cleaders, 102 = \xleaders
attr	node	
spec	node	pointer to a glue_spec item
leader	node	pointer to a box or rule for leaders

6.1.2.10 kern nodes

Valid fields: attr, width

field	type	explanation
subtype	number	0 = from font, 1 = from \kern or \/, 2 = from \accent
attr	node	
width	number	

6.1.2.11 penalty nodes

Valid fields: attr, penalty

field	type	explanation
subtype	number	not used
attr	node	
penalty	number	

6.1.2.12 glyph nodes

Valid fields: attr, char, font, components

field	type	explanation
subtype	number	0 = normal, 1 = ligature, 2 = leftboundary ligature, 3 = rightboundary ligature
attr	node	
char	number	
font	number	
components	node	pointer to ligature components

6.1.2.13 margin_kern nodes

Valid fields: attr, width, glyph

field	type	explanation
subtype	number	0 = left side, 1 = right side
attr	node	
width	number	
glyph	node	

6.1.3 whatsit nodes

Whatsit nodes come in many subtypes, that you can ask for my running node.whatsits(): write (1), close (2), special (3), language (4), local_par (6), dir (7), pdf_literal (8), pdf_refobj (10), pdf_refxform (12), pdf_refximage (14), pdf_annot (15), pdf_start_link (16), pdf_end_link (17), pdf_dest (19), pdf_thread (20), pdf_start_thread (21), pdf_end_thread (22), pdf_save_pos (23), pdf_restore (45), pdf_snap_ref_point (34), open (0), late_lua

(38), pdf_colorstack (42), pdf_save (44), user_defined (46), pdf_snapy (35), close_lua (39), pdf_setmatrix (43), pdf_snapy_comp (36),

6.1.3.1 open nodes

Valid fields: attr, stream, name, area, ext

field	type	explanation
attr	node	
stream	number	TEX's stream id number
name	string	file name
ext	string	file extension
area	string	file area

6.1.3.2 write nodes

Valid fields: attr, stream, data

field	type	explanation
attr	node	
stream	number	TEX's stream id number
data	table	a table representing the token list to be written

6.1.3.3 close nodes

Valid fields: attr, stream

field	type	explanation
attr	node	
stream	number	TEX's stream id number

6.1.3.4 special nodes

Valid fields: attr, data

field	type	explanation
attr	node	
data	string	The \special information

6.1.3.5 language nodes

Valid fields: attr, lang, left, right

field	type	explanation
attr	node	
lang	number	language id number
left	number	value of \lefthyphenmin
right	number	value of \righthyphenmin

6.1.3.6 local_par nodes

Valid fields: attr, pen_inter, pen_broken, dir, box_left, box_left_width, box_right, box_right_width

field	type	explanation
attr	node	
pen_inter	number	interline penalty
pen_broken	number	broken penalty
dir	number	the direction of this par
box_left	node	the \localleftbox
box_left_width	number	width of the \localleftbox
box_right	node	the \localrightbox
box_right_width	number	width of the \localrightbox

6.1.3.7 dir nodes

Valid fields: attr, dir, level, dvi_ptr, dvi_h

field	type	explanation
attr	node	
dir	number	the direction
level	number	nesting level of this direction whatsit
dvi_ptr	number	a saved dvi buffer byte offset
dir_h	number	a saved dvi position

6.1.3.8 pdf_literal nodes

Valid fields: attr, mode, data

field	type	explanation
attr	node	
mode	number	The mode setting of this literal
data	string	The \pdfliteral information

6.1.3.9 pdf_refobj nodes

Valid fields: attr, objnum

field	type	explanation
attr	node	
objnum	number	the referenced PDF object number

6.1.3.10 pdf_refxform nodes

Valid fields: attr, width, height, depth, objnum

field	type	explanation
attr	node	
width	number	
height	number	
depth	number	
objnum	number	the referenced PDF object number

6.1.3.11 pdf_refximage nodes

Valid fields: attr, width, height, depth, objnum

field	type	explanation
attr	node	
width	number	
height	number	
depth	number	
objnum	number	the referenced PDF object number

6.1.3.12 pdf_annot nodes

Valid fields: attr, width, height, depth, objnum, data

field	type	explanation
attr	node	
width	number	
height	number	
depth	number	
objnum	number	the referenced PDF object number
data	string	the annotation data

6.1.3.13 pdf_start_link nodes

Valid fields: attr, width, height, depth, objnum, link_attr, action

field	type	explanation
attr	node	
width	number	
height	number	
depth	number	
objnum	number	the referenced PDF object number
link_attr	table	the link attribute token list
action	node	the action to perform

6.1.3.14 action items

Valid fields: action_type, named_id, action_id, file, new_window, data, ref_count

These are a special kind of item that only appears inside pdf start link objects.

6.1.3.15 pdf_end_link nodes

Valid fields: attr

field	type	explanation
attr	node	

6.1.3.16 pdf_dest nodes

Valid fields: attr, width, height, depth, named_id, dest_id, dest_type, xyz_zoom, objnum

field	type	explanation
attr	node	
width	number	
height	number	
depth	number	
named_id	number	is the dest_id a string value?
dest_id	number or string	the destination id
dest_type	number	type of destination
xyz_zoom	number	
objnum	number	the PDF object number

6.1.3.17 pdf_thread nodes

Valid fields: attr, width, height, depth, named_id, thread_id, thread_attr

field	type	explanation
attr	node	
width	number	
height	number	
depth	number	
named_id	number	is the tread_id a string value?
tread_id	number or string	the thread id
thread_attr	number	extra thread information

6.1.3.18 pdf_start_thread nodes

Valid fields: attr, width, height, depth, named_id, thread_id, thread_attr

field	type	explanation
attr	node	
width	number	
height	number	
depth	number	
named_id	number	is the tread_id a string value?
tread_id	number or string	the thread id
thread_attr	number	extra thread information

6.1.3.19 pdf_end_thread nodes

Valid fields: attr

field	type	explanation
attr	node	

6.1.3.20 pdf_save_pos nodes

Valid fields: attr

field	type	explanation
attr	node	

6.1.3.21 pdf_snap_ref_point nodes

Valid fields: attr

field	type	explanation
attr	node	

6.1.3.22 pdf_snapy nodes

Valid fields: attr, final_skip, spec

field	type	explanation
attr	node	
final_skip	number	
spec	node	pointer to a glue spec

6.1.3.23 pdf_snapy_comp nodes

Valid fields: attr, comp_ratio

field	type	explanation
attr	node	
comp_ratio	number	

6.1.3.24 late_lua nodes

Valid fields: attr, reg, data

field	type	explanation
attr	node	
reg	number	LUA state id number
data	string	data to execute

6.1.3.25 close_lua nodes

Valid fields: attr, reg

field	type	explanation
attr	node	
reg	number	LUA state id number

6.1.3.26 pdf_colorstack nodes

Valid fields: attr, stack, cmd, data

field	type	explanation
attr	node	
stack	number	colorstack id number
cmd	number	command to execute
data	string	data

6.1.3.27 pdf_setmatrix nodes

Valid fields: attr, data

field	type	explanation
attr	node	
data	string	data

6.1.3.28 pdf_save nodes

Valid fields: attr

field	type	explanation
attr	node	

6.1.3.29 pdf_restore nodes

Valid fields: attr

field	type	explanation
attr	node	

6.1.3.30 user_defined nodes

Valid fields: attr, user_id, type, value

field	type	explanation
attr	node	
user_id	number	id number
type	number	type of the value
value	number	
	string	
	node	
	table	

6.2 User-defined whatsits

7 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.

7.1 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.
- The `\endlinechar` can be either added (values 0 or more), or not (negative values). If it is added, the character is always decimal 13 a/k/a `^^M` a/k/a carriage return (This change may be temporary).

7.2 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

7.3 Changes from PDFTEX 1.40

- A number of utility functions is removed:

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

- A few other experimental primitives are also provided without the extra `pdf` prefix, so they can also be called like this:

\primitive \ifabsnum
\ifprimitive \ifabsdim

- The definitions for new didot and new cicero are patched.
- The \pdfprimitive is bugfixed.
- The \pdftexversion is set to 200.

7.4 Changes from ALEPH RC4

- The input translations from ALEPH are not implemented, the related primitives are not available
  \DefaultInputMode \noDefaultInputTranslation
  \noDefaultInputMode \noInputTranslation
  \noInputMode \InputTranslation
  \InputMode \DefaultOutputTranslation
  \DefaultOutputMode \noDefaultOutputTranslation
  \noDefaultOutputMode \noOutputTranslation
  \noOutputMode \OutputTranslation
  \OutputMode
  \DefaultInputTranslation
- 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.
- The ^^ notation can come in five and six item repetitions also, to insert characters that do not fit in the BMP.

7.5 Changes from standard WEB2C

- There is no mltex
- There is no enctex
- The following commandline 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`).

8 Implementation notes

8.1 Primitives overlap

The primitives

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

are all aliases of each other.

8.2 Memory allocation

The single internal memory heap that traditional TEX used for tokens and nodes is split into two wholly separate arrays. Each of those can grow dynamically as needed.

The `texmf.cnf` settings related to main memory are no longer used (these are: `main_memory`, `mem_bot`, `extra_mem_top` and `extra_mem_bot`). Out of main memory errors can still occur, but the limiting factor is now solely the amount of RAM in your system.

Also, the memory (de)allocation routines for nodes are completely rewritten. The relevant code now lives in the C file `luanode.c`, and now normally uses a dozen or so avail lists instead of a doubly-linked model. At this moment, speed is still a little suboptimal because separate helper structures are maintained for debugging checks.

Because of the split in two arrays and the resulting differences in the data structures, some of the pascal web macros have been split. For instance, there are now `vlink` and `vinfo` as well as `link` and `info`. All access to the variable memory array is now hidden behind a macro called `vmem`.

The implementation of the growth of two arrays (via reallocation) introduces a potential pitfall: the memory arrays should never be used as the left hand side of a statement that can modify the array in question.

8.3 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 $2^{21} - 1$.

8.4 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.

8.5 Compressed format

The format is passed through zlib, allowing it to shrink to roughly half 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.

8.6 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).

9 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. And also, the `\textdir` command is broken.
- There is interference between rules and `\pdfliteral`. This is also likely related to the bidirectional algorithm.
- The TFM loader sometimes stores a full path instead of only a name, this prevents VF loading from working properly.
- The Lua setting of `false_boundarychar=65536` gives an error.
- 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).
- Handling of attributes in math mode is not complete. The data structures in math mode are quite different from those in text mode, so this will take some extra effort to implement correctly.
- When used inside `\directlua`, `pdf.print()` should create a literal node instead of flushin immedately.

10 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`?
- Create callback for the automatic creation of missing characters in fonts
- 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.
- Switch to FONTFORGE 2.0
- Complete the attributes in math and switch the nodes to a double-linked list.
- Move the raw bytes outside of the legal UNICODE range, to prevent collisions.
- Implement csname lookups for `tex.box` access.
- Integrate the various PDFTEX extended font codes for hz en protruding into the font table