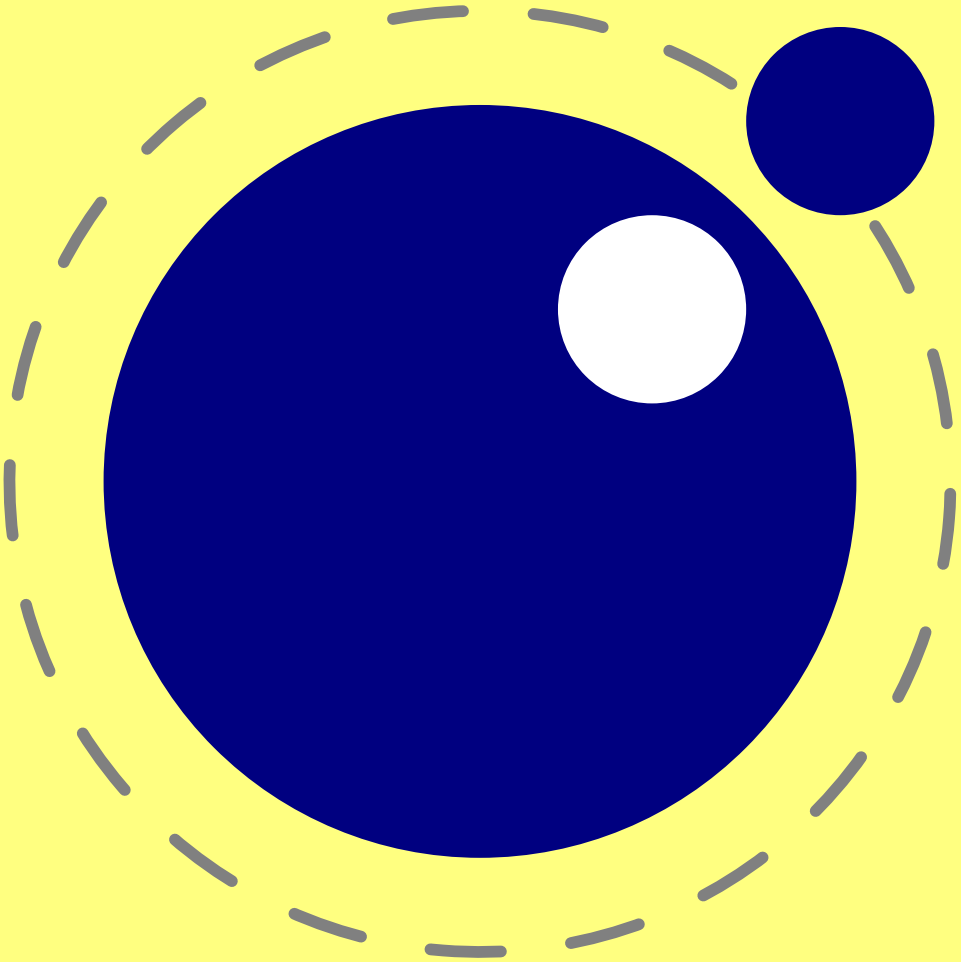
LuaTEX Reference

snapshot 2007-08-10



LuaTEX Reference Manual

more info: www.luatex.org
version: August 13, 2007(snapshot 2007-08-10)

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 or beta release 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 stability, nor on functionality staying the same.

Nothing 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 not willing to deal with this situation, you should wait for the stable version. Currently we expect the first release to be available sometime in the summer of 2008.

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

- *pdfTeX* version 1.40.3
- *Aleph* RC4 (from the *TeXLive* repository)
- *Lua* 5.1.2
- Dedicated *Lua* libraries
- Various *TeX* extensions
- Parts of *FontForge* 2007.06.07
- Newly written compiled source code to glue it all together

Neither *Aleph*'s I/O translation processes, nor tcx files, nor *encTeX* can be used, these encoding-related functions are superseded by a *Lua*-based solution (reader callbacks). Also, some experimental *pdfTeX* features are removed. These can be implemented in Lua instead.

2 Basic TEX enhancements

2.1 Version information

There are three new primitives to test the version of *LuaTEX*:

primitive	explanation
`\luatexversion`	A combination of major an minor number, as in pdfTeX. Current value: **10**
`\luatexrevision`	The revision, as in pdfTeX. Current value: **2**
`\luatexdatestamp`	A combination of the local date and hour when the current executable was compiled, the syntax is identical to `\luatexrevision`. Value for the executable that generated this document: **2007081018**.

Note that the `\luatexdatestamp` depends on both the compilation time and compilation place of the current executable, it is defined in terms of the local time. The purpose of this primitive is solely to be an aid in the development process, do not use it for anything besides debugging.

2.2 UNICODE text support

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

Later chapters will talk of characters and glyphs. Although these are not the interchangeable, they are closely related. During typesetting, a character is always converted to a suitable graphic representation of that character in a specific font. However, while processing a list of to-be-typeset nodes, its contents may still be seen as a character. Inside *LuaTEX* there is not yet a clear separation between the two concepts yet. Until this is implemented, please do not be too harsh on us if we make errors in the usage of the terms.

Note: for now, it only makes sense to use values above the base plane (`"0xFFFF`) 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.

A few primitives affected by this, all in a similar fashion: each of them has to accomodate for a larger range of acceptable numbers. 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

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 just outside of the valid unicode range, starting at the value 1.114.112 (0x110000). When the times comes to print a character $c >= 1.114.112$, *LuaTEX* will actually print the single byte corresponding to $c - 1.114.112$.

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 (this will be explained in section 4.6.2).

2.3 Wide math characters

Text handling 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. The extension from 8-bit to 16-bit was already present in *Aleph* by means of a set of extra primitives.

Therefore, the math primitives from *TEX* and *Aleph* are kept mostly as they are, except for the ones that convert from input to math commands like `matcode` and `omathcode`. The traditional *TEX* primitives are unchanged, their arguments are upscaled from 8 to 16 bits internally (as in *Aleph*).

primitive	max index/bits	hex			numeric
\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			$2^3 \times 2^4 \times 2^8 \times 2^4 \times 2^8$
\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	$+2^8 * 2^{16}$

2.4 Extended tables

All traditional *TEX* and *ε-TEX* 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	

The same is true for the font-related pdfTeX tables like \rpcode etc.

2.5 Attribute registers

Attributes are a completely new concept in 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⟩
\attributedef ⟨csname⟩ ⟨optional equals⟩ ⟨16-bit number⟩

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 an special negative value meaning 'unset' (currently that value is −1, but please test on negativity, not on a specific value). All attributes start out in the 'unset' state (in iniTeX).

Attributes can be used as extra counter values, but their usefulness comes mostly 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. Future versions of LuaTeX will propably be using specific negative attribute ids for internal use. Further information about how to use attributes for node list processing from lua is given in chapter 6.

2.6 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.

2.6.1 \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, it will be initialized automatically. The current category codes are applied to the ⟨general text⟩, and it is passed on as if it was displayed using \the\toks. On the Lua side, each of these blocks is treated as a chunk comprising a single line. This means that you can

not use *Lua* line comments (starting with `--`) within the argument, as that will last for the rest of the input. You need to use *TeX*-style comments (starting with %) instead.

This command is expandable. As an example, the following input:

```
$\pi = \directlua0{tex.print(math.pi)}$
```

will result in $\pi = 3.1415926535898$

Because the ⟨general text⟩ is a chunk, the normal *Lua* error handling is triggered if there is a problem in the included code. The *Lua* error messages should be clear enough, but the contextual information is still pretty bad. Typically, you will only see the line number of the right brace at the end of the code.

While on the subject of errors: some of the things you can do inside *Lua* code can break up *LuaTeX* pretty bad. If you are not careful while working with the node list interface, you may even end up with assertion errors from within the *TeX* portion of the executable.

2.6.2 \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 can print *PDF* statements directly to the *PDF* file.

`\latelua` ⟨16-bit number⟩ ⟨general text⟩

2.6.3 \luaescapestring

This primitive converts a *TeX* token sequence 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 category code 12.

`\luaescapestring` ⟨general text⟩

Most often, this command is not actually the best way to deal with the differences between the *TeX* and *Lua*. In very short bits of *Lua* code it is often not needed, and for longer stretches of *Lua* code it is easier to keep the code in a separate file and load it using *Lua*'s `dofile`:

```
\directlua0 { dofile('mysetups.lua')}
```

2.6.4 \closelua

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

`\closelua` ⟨16-bit number⟩

You cannot close the initial *Lua* state (0), attempts to do so will be silently ignored.

States are never closed automatically except when a fatal out of memory error occurs, at which point LuaTeX will exit anyway.

Also be aware that Lua states are not closed immediately, but only when the \output routine comes into play next (because there may be pending \latelua calls).

2.7 New ε-TeX primitives

2.7.1 \clearmarks

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

\clearmarks ⟨16-bit number⟩

2.7.2 \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. They are part of a temporary trick and will be removed in the near future. For now, you need to enable these primitives when you want to do node list processing of 'characters', where TeX's normal processing would get in the way.

\noligs ⟨integer⟩
\nokerns ⟨integer⟩

2.7.3 \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.

2.7.4 \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 ..' error tests are not executed, allowing the expansion to end on a different grouping level or while a conditional is still incomplete.

2.7.5 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.

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.

2.7.5.1 \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).

2.7.5.2 \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 identical to the currently active table, an error is raised.

2.7.5.3 \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.

2.7.6 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 inside the argument.

3 LUA general

3.1 Initialization

3.1.1 LUATEX as a LUA interpreter

There are some situations that make LUATEX behaves like it is a LUA interpreter only:

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

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

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

3.1.2 LUATEX as a LUA byte compiler

There are two situations that make LUATEX behaves like the LUA byte compiler:

- If a `--luaconly` option is given on the commandline
- If the executable is named `texluac`

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

3.1.3 Other commandline processing

When the LUATEX executable starts, it looks for the `--lua` commandline option. If there is no `--lua` option, the commandline is interpreted in a similar fashion as in traditional PDFTEX and ALEPH. But if the option is present, LUATEX will enter an alternative mode of commandline parsing in comparison to the standard web2c programs.

In this mode, a small series of actions is taken in order. At first, it will only interpret a small subset of the commandline directly:

–lua=s	load and execute a LUA initialization script
–safer	disable easily exploitable LUA commands
–help	display help and exit
–version	display version and exit

Now it searches for the requested *Lua* initialization script. If it 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 it checks the `--safer` switch. You can use that to disable some *Lua* 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.

Next the initialization script is loaded and executed. From within the script, the entire commandline in available in the *Lua* table `arg`, beginning with `arg[0]`, containing the name of the executable.

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). See chapter 4 for more information about the *LuaTEX*-specific *Lua* extension tables.

The *Lua* initialization script is loaded into *Lua* state 0, and everything you do will remain visible during the rest of the run, with the exception of the aforementioned `tex`, `token`, `node` and `pdf` tables: those will be initialized to their documented state after the execution of the script. You should not store anything in variables or within tables with these four global names, as they will be overwritten completely.

We recommend you use the startup file only for your own *TEX*-independant initializations (if you need any), to parse the commandline, set values in the `texconfig` table, and register the callbacks you need. *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 for more details on which ones exactly).

Unless the `texconfig` table tells it not to start *kpathsea* at all (set `texconfig.kpse_init` to `false` for that), *LuaTEX* acts on three more commandline options after the initialization script is finished:

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).

3.2 LUA changes

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 acceptable line endings.

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 it's attributes table (`permissions`).

The `string` library has an extra function: `string.explode(s[,m])`. This function returns an array containing the string argument `s` split into substrings based on the value of the string argument `m`. The second argument is a string that is either empty (this splits the string into characters), a single character (this splits on each occurrence of that character, possibly introducing empty strings), or a single character followed by the plus sign `+` (this special version does not create empty substrings). The default value for `m` is `' +'` (multiple spaces).

The `string` library also 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 containing one byte)
- `string.characterpairs(s)` (two strings each containing one byte) 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 `string.characterpairs()` and `string.bytepairs()` are useful especially in the conversion of UTF-16 encoded data into UTF-8.

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.env` This is a hash table containing a dump of the variables and values in the process environment at the start of the run. It is writeable, but the actual environment is updated automatically.

3.3 LUA Modules

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

- `slnunicode`, from the Selene libraries, http://luaforge.net/projects/sln. (version 1.1)
- `luazip`, from the kepler project, http://www.keplerproject.org/luazip/. (version 1.2.1, but patched for compilation with lua 5.1)
- `luafilesystem`, also from the kepler project, http://www.keplerproject.org/luafilesystem/. (version 1.2, but patched for compilation with lua 5.1)
- `lpeg`, by Roberto Ierusalimschy, http://www.inf.puc-rio.br/~roberto/lpeg.html. (version 0.6)
- `lzlib`, by Tiago Dionizio, http://mega.ist.utl.pt/~tngd/lua/. (version 0.2)
- `md5`, by Roberto Ierusalimschy http://www.inf.puc-rio.br/~roberto/md5/md5-5/md5.html.

4 LUATEX LUA Libraries

The interfacing between TEX and LUA is facilitated by a set of library modules. The LUA libraries in this chapter are all defined and initialized by the LUATEX executable. Together, they allow LUA scripts to query and change a number of TEX's internal variables, run various internal functions TEX, and set up LUATEX's hooks to execute LUA code.

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.pdfgamma
tex.pdfgentounicode
tex.pdfimageapplygamma
tex.pdfimagegamma
tex.pdfimagehicolor
tex.pdfimageresolution
tex.pdfinclusionerrorlevel
tex.pdfminorversion
tex.pdfobjcompresslevel
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

The 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.luatexrevision
tex.luatexdatestamp
tex.pdfnormaldeviate
tex.pdftexbanner
tex.pdftexrevision

All 'convert' commands are read-only and return a *Lua* string

If you are wondering why this list looks haphazard; 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 subtables :

```
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.

4.1.11 Helper functions

4.1.11.1 tex.round

```
number n =  tex.round(number o)
```

Rounds lua number `o`, and returns a number that is in the range of a valid TEX register value. If the number starts out of range, it generates a `Number to big' error as well.

4.1.11.2 tex.scale

```
number n =  tex.scale(number o, number delta)
table n  =  tex.scale(table o, number delta)
```

Multiplies the lua numbers `o` and `delta`, and returns a rounded number that is in the range of a valid TEX register value. In the table version, it creates a copy of the table with all numeric top–level values scaled in that manner. If the mutiplied number(s) are of range, it generates `Number to big' error(s) as well.

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 the name of a control sequence (without leading backslash), 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 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 not always a 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_idtring 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. They allow you to alter, create, copy, delete, and insert LuaTeX node objects, the core objects within the typesetter.

LuaTeX nodes are represented in Lua as userdata with the metadata type luatex.node. The various parts within a node can be accessed using named fields.

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 chapter 6.

TeX's math nodes are not yet supported: there is not yet an interface to the internals of the math list 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.

Nodes can be compared to each other, but: you are actually comparing indices into the node memory. This means that equality tests can only be trusted under very limited conditions. It will not work correctly in any situation where one of the two nodes has been freed and/or reallocated: in that case, there will be false positives.

At the moment, memory management of nodes should still be done explicitly by the user. Nodes are not 'seen' by the *Lua* garbage collector, so you have to call the node free-ing functions yourself when you are no longer in need of a node (list). Nodes form linked lists without reference counting, so you have to be careful that when control returns back to *LuaTEX* itself, you have not deleted nodes that are still referenced from a `next` pointer elsewhere, and that you did not create nodes that are referenced more than once.

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(number id)
table t = node.fields(number id, number subtype)
```

This function returns an array of valid field names for a particular type of node. If you want to get the valid fields for a 'whatsit', you have to supply the second argument also. In other cases, any given second argument will be silently ignored.

This function accepts string `id` and `subtype` values as well.

4.3.1.7 node.has_field

```
boolean t = node.has_field(<node> n, string field)
```

This function returns a boolean that is only true if `n` is actually a node, and it has the field.

4.3.1.8 node.new

```
<node> n = node.new(number id)
<node> n = node.new(number id, number subtype)
```

Creates a new node. All of the new node's fields are initialized to either zero or nil except for `id` and `subtype` (if supplied). If you want to create a new whatsit, then the second argument is required, otherwise it need not be present. As with all node functions, this function creates an node on the TEX level.

This function accepts string `id` and `subtype` values as well.

4.3.1.9 node.free

```
node.free(<node> n)
```

Removes the node `n` 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.

4.3.1.10 node.flush_list

```
node.flush_list(<node> n)
```

Removes the node list `n` and the complete node list following `n` 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.

4.3.1.11 node.copy

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

Creates a deep copy of node `n`, including all nested lists as in the case of a hlist or vlist node. Only the `next` field is not copied.

4.3.1.12 node.copy_list

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

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

4.3.1.13 node.hpack

```
<node> h = node.hpack(<node> n)
<node> h = node.hpack(<node> n, number w, string info)
```

This function creates a new hlist by packaging the list that begins at node `n` into a horizontal box. With only a single argument, this box is created using the natural width of it's components. In the three argument form, `info` must be either `additional` or `exactly`, and `w` is the additional (`\hbox spread`) or exact (`\hbox to`) width to be used.

Caveat: at this moment, there can be unexpected side-effects to this function, like updating some of the `\marks` and `\inserts`.

4.3.1.14 node.slide

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

Returns the last node of the node list that starts at `n`. As a side-effect, it also creates a reverse chain of `prev` pointers between nodes.

4.3.1.15 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.

4.3.1.16 node.count

```
number i = node.count(number id, <node> n)
number i = node.count(number id, <node> n, <node> m)
```

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

This function also accept string `id`'s.

4.3.1.17 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.

4.3.1.18 node.traverse_id

```
<node> t = node.traverse_id(number id, <node> n, <node> m)
<node> t = node.traverse_id(number id, <node> n)
```

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

This function also accept string `id`'s.

4.3.1.19 node.remove

```
<node> head, current = node.remove(<node> head, <node> current)
```

This function removes the node `current` from the list following `head`. It is your responsibility to make sure it is really part of that list. The return values are the new `head` and `current` nodes. The returned `current` is the node in the calling argument, and is only passed back as a convenience (it's `next` field will be cleared). The returned `head` is more important, because if the function is called with `current` equal to `head`, it will be changed.

4.3.1.20 node.insert_before

```
<node> head, new = node.insert_before(<node> head, <node> current, <node>
new)
```

This function inserts the node new before current into the list following head. It is your responsibility to make sure that current is really part of that list. The return values are the (potentially mutated) head and the new, set up to be part of the list (with correct next field). If head is initially nil, it will become new.

4.3.1.21 node.insert_after

```
<node> head, new = node.insert_after(<node> head, <node> current, <node>
new)
```

This function inserts the node new after current into the list following head. It is your responsibility to make sure that current is really part of that list. The return values are the head and the new, set up to be part of the list (with correct next field). If head is initially nil, it will become new.

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

```
number v = node.unset_attribute(<node> n, number id, number val)
number v = 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.

If the attribute was actually deleted, returns its old value. Otherwise, returns nil.

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 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 of the log file and output file after LuaTEX is finished, and an eventual format filename will become mangled. That is because these file names 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 run this optional function when it decides to close the file.

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

Your function should not return any value.

4.6.2.2 General file readers

There is a set of callbacks for the loading of binary data files. These all use the same interface:

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

The `name` will normally be a full path name as it is returned by either one of the file discovery callbacks or the internal version of `kpse.find_file()`.

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.

The list of functions is:

`read_font_file`	This function is called when TEX needs to read a `ofm` or `tfm` file.
`read_vf_file`	for virtual fonts.
`read_ocp_file`	for ocp files.
`read_map_file`	for map files.
`read_enc_file`	for encoding files.
`read_sfd_file`	for subfont definition files.
`read_pk_file`	for pk bitmap files.
`read_data_file`	for embedded files (`\pdfobj file '...'`).

`read_truetype_file` for *TrueType* font files.
`read_type1_file` for *Type1* font files.
`read_opentype_file` for *OpenType* font files.

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 replace the way *LuaTeX* fetches lexical tokens.

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

The calling convention for this callback is bit more complicated than 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 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, string extrainfo)
    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

The string `extrainfo` gives some additional information about what *TeX*'s state is with respect to the `current page'. The possible values are:

value	explanation
alignment	a (partial) alignment is being added
box	a typeset box is being added
begin_of_par	the beginning of a new paragraph
vmode_par	\par was found in vertical mode
hmode_par	\par was found in horizontal mode
insert	an insert is added
penalty	a penalty (in vertical mode)
before_display	immediately before a display starts
after_display	a display is finished

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 a possible 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 `hpack_filter` and `vpack_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 for 'testing the water' only. 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. This is why the exact definition of `indicator` is not given here.

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 returns the id number of the instance.

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

This returns a LuaTeX version identifier string. The value is currently lua.version, but it is soon to be replaced by something more elaborate.

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.

ftype
: maps to the `-format` argument of *kpsewhich*. The supported `ftype` values are the same as the ones supported by the standalone `kpsewhich` program:

```
'gf'                    'afm'
'pk'                    'base'
'bitmap font'           'bib'
'tfm'                   'bst'
```


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

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 fall-
back)
```

4.8.4 kpse.readable_file

Test if an (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 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
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	current size allocated for 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	current allocated size of the line buffer
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
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
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 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 provides the interface into the internals of the font system, and also it contains helper functions to load traditional *TEX* font metrics formats. Other font loading functionality is provided by the `fontforge` library that will be discussed in the next section.

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 metrics 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 meaning of the number `s`, and the format of the returned table is the silimar to the `read_tfm()` function.

4.11.3 The fonts array

The whole table of TEX fonts is accessible from lua using a virtual array.

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

See chapter 5 for the structure of the tables. Because this is a virtual array, you cannot call `pairs` on it, but see below for the `font.each` iterator.

The two metatable functions implementing the virtual array are:

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

Also note the following: assignments can only be made to fonts that have already been defined in TEX, but have not been accessed since that definition. This limits the usability of the write access to `font.fonts` quite a lot, a less stringent ruleset will likely be implemented later.

4.11.4 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).

4.11.5 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.

4.11.6 Currently active font

```
number i = font.current();
```

This is the currently used font number.

4.11.7 Maximum font id

```
number i = font.max();
```

This is the largest used index in `font.fonts`.

4.11.8 Iterating over all fonts

```
for i,v in font.each() do
  ...
end
```

This is an iterator over each of the defined *TEX* fonts. The first returned value is the index in `font.fonts`, the second the font itself, as a lua table. The indices are listed incrementally, but they do not always form an array of consecutive numbers: in some cases there can be holes in the sequence.

4.12 The `fontforge` library

4.12.1 Getting quick information on a font

```
local info = fontforge.info('filename')
```

This function returns either `nil`, or a `table`, or an array of small tables (in the case of a TrueType collection). The returned table(s) will contain six fairly interesting information items from the font(s) defined by the file:

key	type	explanation
`fontname`	string	the 'PostScript' name of the font
`fullname`	string	The formal name of the font
`familyname`	string	The family name this font belongs to

`weight`	string	A string indicating the color value of the font
`version`	string	The internal font version
`italicangle`	float	The slant angle

Getting information through this function is (sometimes much) more efficient than loading the font properly, and is therefore handy when you want to create a dictionary of available fonts based on a directory contents.

4.12.2 Loading an OPENTYPE or TRUETYPE file

If you want to use an *OpenType* font, you have to get the metric information from somewhere. Using the `fontforge` library, the basic way to get that information is thus:

```
function load_font (filename)
  local metrics = nil
  local font = fontforge.open(filename)
  if font then
     metrics = fontforge.to_table(font)
     fontforge.close(font)
  end
  return metrics
end

myfont = load_font('/opt/tex/texmf/fonts/data/arial.ttf')
```

The main function call is

```
local f = fontforge.open('filename')
```

For *TrueType* collections (when filename ends in 'ttc'), you have to use a second string argument to specify which font you want from the collection. Use one of the `fullname` strings that are returned by `fontforge.info` for that.

```
local f = fontforge.open('filename','fullname')
```

The font file is parsed and partially interpreted by the font loading routines from *FontForge*. The file format can be *OpenType*, *TrueType*, *TrueType* Collection, CFF, or *Type1*.

There are a few advantages to this approach compared to reading the actual font file ourselves:

- The font is automatically re-encoded, so that the `metrics` table for *TrueType* and *OpenType* fonts 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 character boundingbox has to be calculated in some way.
- In the future, it may be interesting to allow *Lua* scripts access to the font program itself, perhaps even creating or changing the font.

4.12.3 Applying a `feature file'

You can apply a `feature file' to a loaded font:

```
fontforge.apply_featurefile(f,'filename')
```

A `feature file' is a textual representation of the features in an OpenType font. See http://www.adobe.com/devnet/opentype/afdko/topic_feature_file_syntax.html and http://fontforge.sourceforge.net/featurefile.html for a more detailed description of feature files.

4.12.4 Applying an `afm file'

You can apply a `afm file' to a loaded font:

```
fontforge.apply_afmfile(f,'filename')
```

An `afm file' is a textual representation of (some of) the metainformation in a Type 1 font. See http://www.adobe.com/devnet/font/pdfs/5004.AFM_Spec.pdf for more information about afm files.

Note: if you `fontforge.open()` a PFB file named `font.pfb`, the library will automatically search for, and apply, `font.afm` if it exists in the same directory as `font.pfb`. In that case, there is no need for an explicit call to `apply_afmfile()`.

4.13 Fontforge font tables

The top-level keys in the returned table are (the explanations in this part of the documentation is not yet finished):

key	type	explanation
table_version	number	indicates the metrics version
fontname	string	*PostScript* font name
fullname	string	official font name
familyname	string	family name
weight	string	weight indicator
copyright	string	copyright information
filename	string	the file name
version	string	font version
italicangle	float	slant angle

units_per_em	number	1000 for *PostScript*-based fonts, usually 2048 for *TrueType*
ascent	number	height of ascender in units_per_em
descent	number	depth of descender in units_per_em
upos	float	
uwidth	float	
vertical_origin	number	
uniqueid	number	
glyphcnt	number	number of included glyphs
glyphs	array	
glyphmax	number	maximum used index the glyphs array
hasvmetrics	number	
order2	number	set to 1 for *TrueType* splines, 0 otherwise
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
origname	string	the file name, as supplied by the user
map	table	
private	table	
xuid	string	
pfminfo	table	
names	table	
cidinfo	table	
subfonts	array	
cidmaster	array	
commments	string	
anchor_classes	table	
ttf_tables	table	
kerns	table	
vkerns	table	
texdata	table	
lookups	table	
gpos	table	
gsub	table	
chosenname	string	
macstyle	number	
fondname	string	
design_size	number	
fontstyle_id	number	
fontstyle_name	table	
design_range_bottom	number	
design_range_top	number	

strokewidth	float
mark_classes	array
mark_class_names	array
creationtime	number
modificationtime	number
os2_version	number

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	the glyph name
unicodeenc	number	unicode code point, or -1
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)
lookups	table	(only if nonempty)
ligatures	table	(only if nonempty)
anchors	table	(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)
comment	string	(only if set)

The kerns and vkerns are linear arrays of small hashes:

key	type	explanation
char	string	
off	number	
lookup	string	

The lookups is a hash based on lookup subtable names, with the value of each key inside that 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
specification	table	extra data

For the first seven values of type, there can be additional sub-information, stored in the sub-table specification:

value	type	explanation
position	table	a table of the offset_specs type
pair	table	one string: paired, and an array of one or two offset_specs tables: offsets
substitution	table	one string: variant
alternate	table	one string: components
multiple	table	one string: components
ligature	table	two strings: components, char
lcaret	array	linear array of numbers

Tables for offset_specs contain up to four number-valued fields: x (a horizontal offset), y (a vertical offset), h (an advance width correction) and v (an advance height correction).

The ligatures 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	

The anchor table is indexed by a string signifying the anchor type, which is one of

key	type	explanation
mark	table	placement mark
basechar	table	mark for attaching combining items to a base char
baselig	table	mark for attaching combining items to a ligature
basemark	table	generic mark for attaching combining items to connect to
centry	table	cursive entry point
cexit	table	cursive exit point

The content of these is an short array of defined anchors, with the entry keys being the anchor names. For all except baselig, the value is a single table with this definition:

key	type	explanation
x	number	x location
y	number	y location
ttf_pt_index	number	truetype point index, only if given

For baselig, the value is a small array of such anchor sets sets, one for each constituent item of the ligature.

For clarification, an anchor table could for example look like this :

```
['anchor']={
  ['basemark']={
   ['Anchor-7']={ ['x']=170, ['y']=1080 }
  },
  ['mark']={
   ['Anchor-1'] ={ ['x']=160, ['y']=810 },
   ['Anchor-4']= { ['x']=160, ['y']=800 }
  },
  ['baselig']={
   [1] = { ['Anchor-2'] ={ ['x']=160, ['y']=650 } },
   [2] = { ['Anchor-2']= { ['x']=460, ['y']=640 } }
  }
 },
```

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	(only if nonzero)
low_page	number	
high_page	number	
iconv_name	string	
iso_2022_escape	string	

3 private table

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

4 cidinfo table

key	type	explanation
registry	string	
ordering	string	
supplement	number	
version	number	

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	explanation
familytype	string	Values as in the *OpenType* font specification: Any, No Fit, Text and Display, Script, Decorative, Pictorial
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.

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

7 anchor_classes table

The anchor_classes classes:

key	type	explanation
name	string	A descriptive id of this anchor class
lookup	string	
type	string	One of 'mark', 'mkmk', 'curs', 'mklg'

8 gpos table

Th gpos table has one array entry for each lookup.

key	type	explanation
type	string	One of 'gpos_single', 'gpos_pair', 'gpos_cursive', 'gpos_mark2base', 'gpos_mark2ligature', 'gpos_mark2mark', 'gpos_context', 'gpos_contextchain'
flags	table	
name	string	
features	array	
subtables	array	

The flags table has a true value for each of the lookup flags that is actually set:

key	type	explanation
r2l	boolean	
ignorebaseglyphs	boolean	
ignoreligatures	boolean	
ignorecombiningmarks	boolean	

The features table has:

key	type	explanation
tag	string	
scripts	table	
ismax	number	(only if true)

The scripts table within features has:

key	type	explanation
script	string	
langs	array of strings	

The subtables table has:

key	type	explanation
name	string	
suffix	string	(only if used)

anchor_classes	number	(only if used)
vertical_kerning	number	(only if used)
kernclass	table	(only if used)

The kernclass with subtables table has:

key	type	explanation
firsts	array of strings	
seconds	array of strings	
lookup	string	associated lookup
offsets	array of numbers	

9 gsub table

This has identical layout to the gpos table, except for the type:

key	type	explanation
type	string	One of 'gsub_single', 'gsub_multiple', 'gsub_alternate', 'gsub_ligature', 'gsub_context', 'gsub_contextchain', 'gsub_reversecontextchain'

10 ttf_tables table

key	type	explanation
tag	string	
len	number	
maxlen	number	
data	number	

11 kerns table

Substructure is identical to the per-glyph subtable.

12 vkerns table

Substructure is identical to the per-glyph subtable.

13 texdata table

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

14 lookups table

Top-level `lookups` is quite different from the ones at character level. The keys in this hash are strings, the values the actual lookups, represented as dictionary tables.

key	type	explanation
`type`	number	
`format`	enum	One of 'glyphs', 'class', 'coverage', 'reversecoverage'
`tag`	string	
`current_class`	array	
`before_class`	array	
`after_class`	array	
`rules`	array	an array of rule items

Rule items have one common item and one specialized item:

key	type	explanation
`lookups`	array	A linear array of lookup names
`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'

A glyph table is:

key	type	explanation
`names`	string	
`back`	string	
`fore`	string	

A class table is:

key	type	explanation
`current`	array	of numbers
`before`	array	of numbers
`after`	array	of numbers

coverage:

key	type	explanation
`current`	array	of strings
`before`	array	of strings
`after`	array	of strings

reversecoverage:

key	type	explanation
`current`	array	of strings

before	array	of strings
after	array	of strings
replacements	string	

5 Font structure

All TEX fonts are represented to LUA code as tables, and 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] = {
  ['width'] = 200250
  ['height'] = 455111,
  ['depth'] = 0,
  ['italic'] = 50973,
  ['kerns'] = {
     [63] = 50973,
     [93] = 50973,
     [39] = 50973,
```

```
      [33] = 50973,
      [41] = 50973
  },
  ['ligatures'] = {
    [102] = {
       ['char'] = 11,
       ['type'] = 0
    },
    [108] = {
       ['char'] = 13,
       ['type'] = 0
    },
    [105] = {
       ['char'] = 12,
       ['type'] = 0
    }
  },
}
```

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 parts 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` are:

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

Currently, only *TrueType* and *OpenType* fonts can be 'wide' fonts (*Type1 PostScript* fonts are not supported).

Values for `embedding` are:

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* and *TrueType* 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 *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 high 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 glyph commands 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 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`.

5.2.1 Artificial fonts

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 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 virtual font demonstration:

```
\directlua0 {
  callback.register('define_font',
    function (name,size)
```


```
      if name == 'cmr10-red' then
        f = font.read_tfm('cmr10',size)
        f.name = 'cmr10-red'
        f.type = 'virtual'
        f.fonts = {{ name = 'cmr10', size = 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 at 10pt \myfont  This is a line of text \par
\font\myfontx= cmr10 at 10pt \myfontx Here is another line of text \par
```

6 Nodes

6.1 LUA node representation

TEX's nodes are represented in LUA as userdata object with a variable set of fields. In the following syntax tables, such the type of such a userdata object is represented as ⟨node⟩.

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 the same as regular 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. This item type will likely disappear in the near future, and the glue fields themselves will become part of the nodes referencing glue items.

6.1.1.2 attribute_list items

The newly introduced attribute registers are 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 an item of type attribute_list, and the next field in that item will 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 attribute type id
value	number	the attribute value

6.1.1.4 action item

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.

field	type	explanation
action_type	number	
action_id	number or string	
named_id	number	
file	string	
new_window	number	
data	string	
ref_count	number	

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. Not all nodes actually use the subtype, but this way you can be sure that all nodes accept it as a valid field name, and that is often handy in node list traversal. In the following tables next and id are not explicitly mentioned.

Besides these three fields, almost all nodes also have an attr field, and there is a also field called prev. That last field is always present, but only initialized on explicit request: when the function node.slide() is called, it will set up the prev fields to be a backwards pointer in the argument node list.

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	0 = normal, 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, kern

field	type	explanation
subtype	number	0 = from font, 1 = from \kern or \/, 2 = from \accent
attr	<node>	
kern	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, xoffset, yoffset

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
xoffset	number	
yoffset	number	

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), late_lua (35), close_lua (36), user_defined (43), pdf_restore (42), pdf_colorstack (39), pdf_setmatrix (40), pdf_save (41), open (0),

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

Be aware that pdf_refxform nodes have dimensions that are used by *LuaTEX*.

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

Be aware that pdf_refximage nodes have dimensions that are used by *LuaTEX*.

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 pdf_end_link nodes

Valid fields: attr

field	type	explanation
attr	<node>	

6.1.3.15 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

6.1.3.16 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

6.1.3.17 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

6.1.3.18 pdf_end_thread nodes

Valid fields: attr

field	type	explanation
attr	<node>	

6.1.3.19 pdf_save_pos nodes

Valid fields: attr

field	type	explanation
attr	<node>	

6.1.3.20 late_lua nodes

Valid fields: attr, reg, data

field	type	explanation
attr	<node>	
reg	number	*Lua* state id number
data	string	data to execute

6.1.3.21 close_lua nodes

Valid fields: attr, reg

field	type	explanation
attr	<node>	
reg	number	*Lua* state id number

6.1.3.22 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

6.1.3.23 pdf_setmatrix nodes

Valid fields: attr, data

field	type	explanation
attr	<node>	
data	string	data

6.1.3.24 pdf_save nodes

Valid fields: attr

field	type	explanation
attr	<node>	

6.1.3.25 pdf_restore nodes

Valid fields: attr

field	type	explanation
attr	<node>	

6.1.3.26 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	

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
- Some of the tracing information that is output by ε-TEX's `\tracingassigns` and `\tracingrestores` is not there.
- Register management in LUATEX uses the ALEPH model, so the maximum value is 65535 and the implementation uses a flat array instead of the mixed flat&sparse model from ε-TEX.

7.3 Changes from PDFTEX 1.40

- The (experimental) support for snap nodes has been removed, because it much more natural to build this functionality on top of node processing and attributes. The associated primitives that are now gone are: `\pdfsnaprefpoint`, `\pdfsnapy`, and `\pdfsnapycomp`.
- 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 separate arrays. Each of these will grow dynamically when 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 the amount of RAM in your system, not a predefined limit.

Also, the memory (de)allocation routines for nodes are completely rewritten. The relevant code now lives in the C file `luanode.c`, and basically 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 into two arrays and the resulting differences in the data structures, some of the Pascal WEB macros have been duplicated. 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.

The input line buffer and pool size are now also reallocated when needed, and the `texmf.cnf` settings `buf_size` and `pool_size` are silently ignored.

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 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 and limitations

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, or it may not be worth the hassle and the limitations will eventually be documented.

- Seaching of directly-generated PDF documents that use wide fonts is impossible, due to explicit referencing (by *LuaTeX*) of font glyph indices without a ToUnicode map being included.
- 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 flushing immediately.
- At the moment, only characters in plane 0 and plane 1 can be assigned catcode 13 (i.e. turned into active characters). This is a temporary measure to reduce the memory requirements of *LuaTeX*. In general, *LuaTeX*'s memory footprint is a bit larger that we would like (with `plain.fmt` preloaded it needs about 55MB).
- Not all of *Aleph*'s direction commands are handled properly in *pdf* mode, and especially the vertical scripts support is missing almost completely (only TRT and TLT are routinely tested).
- 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.
- Node pointers are not always checked for validity, so if you make a mistake in the node list processing, *LuaTeX* may about with an assertion error.

10 TODO

On top of the 'normal' extensions that are planned, there are some more specific small feature requests. Whether these will all be included is not certain yet, (and new requests are welcome).

- Implement the *TEX* primitive `\dimension`, cf. `\number`
- Change the *Lua* table `tex.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.
- Complete the attributes in math and switch all the nodes to a double-linked list.
- Finish the interface from Lua to TEX's internals, specially the hash and equivalence table (a small subpart is implementing `\csname` lookups for `tex.box` access).
- Integrate the various *pdfTEX* extended font codes for hz en protruding into the font table
- Use of Type1C for embedded PostScript font subsets.
- Support font reencoding of 8-bit fonts via char index.
- Attempt to parse OFM level 0 fonts that are masquerading as level 1.
- Add line numbers and input context information to the lua errors