LuaTEX Reference

beta 0.80.1

LuaTEX Reference Manual

copyright : LuaTEX development team
more info : www.luatex.org
version : October 6, 2015

Contents

Introduction

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

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

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

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

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

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

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

LuaTeX	:	Version 80.1
ConTeXt	:	2015.10.06 12:49
timestamp	:	October 6, 2015

1 Basic TEX enhancements

1.1 Introduction

From day one, LUATEX has offered extra features compared to the superset of PDFTEX and ALEPH. That has not been limited to the possibility to execute LUA code via `\directlua`, but LUATEX also adds functionality via new TEX-side primitives.

When LUATEX starts up in 'iniluatex' mode (`luatex -ini`), it defines only the primitive commands known by TEX82 and the one extra command `\directlua`. As is fitting, a LUA function has to be called to add the extra primitives to the user environment. The simplest method to get access to all of the new primitive commands is by adding this line to the format generation file:

```
\directlua { tex.enableprimitives('',tex.extraprimitives()) }
```

But be aware that the curly braces may not have the proper `\catcode` assigned to them at this early time (giving a 'Missing number' error), so it may be needed to put these assignments before the above line:

```
\catcode `\{=1
\catcode `\}=2
```

More fine-grained primitives control is possible, you can look up the details in section 7.14.12. For simplicity's sake, this manual assumes that you have executed the `\directlua` command as given above.

The startup behavior documented above is considered stable in the sense that there will not be backward-incompatible changes any more. However, we can decide to promite some primitives to the LUATEX namespace. For instance, after version 0.80.1 we promoted some rather generic PDFTEX primitives to core LUATEX ones, and the ones inherited frome ALEPH (OMEGA) are also promoted. Effectively this means that we now have the `tex`, `etex`, `luatex` and `pdftex` (sub)sets left.

1.2 Version information

There are three new primitives to test the version of LUATEX:

primitive	explanation	value
`\luatexbanner`	the banner reported on the command line	This is LuaTeX, Version beta-0.80.1 (TeX Live 2016/dev)
`\luatexversion`	a combination of major and minor number	80
`\luatexrevision`	the revision number, the current value is	1

The official LUATEX version is defined as follows:

- The major version is the integer result of `\luatexversion` divided by 100. The primitive is an 'internal variable', so you may need to prefix its use with `\the` depending on the context.
- The minor version is the two-digit result of `\luatexversion` modulo 100.
- The revision is the given by `\luatexrevision`. This primitive expands to a positive integer.
- The full version number consists of the major version, minor version and revision, separated by dots.

1.3 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} - 1 = $ 0x10FFFF). Later chapters will talk of characters and glyphs. Although these are not 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 no clear separation between the two concepts. Because the subtype of a glyph node can be changed in LUA it is lso up to the user.

A few primitives are affected by this, all in a similar fashion: each of them has to accommodate for a larger range of acceptable numbers. For instance, `\char` now accepts values between 0 and 1,114,111. 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 affected commands with an altered initial (left of the equals sign) or secondary (right of the equals sign) value are: `\char`, `\lccode`, `\uccode`, `\catcode`, `\sfcode`, `\efcode`, `\lpcode`, `\rpcode`, `\chardef`.

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 time comes to print a character $c \geq$ 1,114,112, LUATEX will actually print the single byte corresponding to c minus 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 7.1.2).

1.4 Extended tables

All traditional TEX and ε-TEX registers can be 16-bit numbers. The affected commands are:

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

The glyph properties `\efcode`, `\lpcode` and `\rpcode`, introduced in PDFTEX that deal with font expansion (hz) and character protruding, are also 16-bit. Because font memory management has been rewritten, these character properties are no longer shared among fonts instances that originate from the same metric file.

1.5 Attributes

1.5.1 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⟩ ⟨32-bit number⟩
`\attributedef` ⟨csname⟩ ⟨optional equals⟩ ⟨16-bit number⟩

Conceptually, an attribute is either 'set' or 'unset'. Unset attributes have a special negative value to indicate that they are unset, that value is the lowest legal value: `-"7FFFFFFF` in hexadecimal, a.k.a. −2147483647 in decimal. It follows that the value `-"7FFFFFFF` cannot be used as a legal attribute value, but you *can* assign `-"7FFFFFFF` to 'unset' an attribute. All attributes start out in this '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. Further information about how to use attributes for node list processing from LUA is given in chapter 6.

1.5.2 Box attributes

Nodes typically receive the list of attributes that is in effect when they are created. This moment can be quite asynchronous. For example: in paragraph building, the individual line boxes are created after the `\par` command has been processed, so they will receive the list of attributes that is in effect then, not the attributes that were in effect in, say, the first or third line of the paragraph.

Similar situations happen in LUATEX regularly. A few of the more obvious problematic cases are dealt with: the attributes for nodes that are created during hyphenation, kerning and ligaturing borrow their attributes from their surrounding glyphs, and it is possible to influence box attributes directly.

When you assemble a box in a register, the attributes of the nodes contained in the box are unchanged when such a box is placed, unboxed, or copied. In this respect attributes act the same as characters that have been converted to references to glyphs in fonts. For instance, when you use attributes to implement color support, each node carries information about its eventual color. In that case, unless you implement mechanisms that deal with it, applying a color to already boxed material will have no effect. Keep in mind that this incompatibility is mostly due to the fact that separate specials and literals are a more unnatural approach to colors than attributes.

It is possible to fine-tune the list of attributes that are applied to a `hbox`, `vbox` or `vtop` by the use of the keyword `attr`. An example:

```
\attribute2=5
\setbox0=\hbox {Hello}
\setbox2=\hbox attr1=12 attr2=-"7FFFFFFF{Hello}
```

This will set the attribute list of box 2 to $1 = 12$, and the attributes of box 0 will be $2 = 5$. As you can see, assigning the maximum negative value causes an attribute to be ignored.

The `attr` keyword(s) should come before a `to` or `spread`, if that is also specified.

1.6 LUA related primitives

1.6.1 \directlua

In order to merge LUA code with TEX input, a few new primitives are needed. The primitive `\directlua` is used to execute LUA code immediately. The syntax is

`\directlua` ⟨general text⟩
`\directlua name` ⟨general text⟩ ⟨general text⟩
`\directlua` ⟨16-bit number⟩ ⟨general text⟩

The last ⟨general text⟩ is expanded fully, and then fed into the LUA interpreter. After reading and expansion has been applied to the ⟨general text⟩, the resulting token list is converted to a string as if it was displayed using `\the\toks`. On the LUA side, each `\directlua` block is treated as a separate chunk. In such a chunk you can use the `local` directive to keep your variables from interfering with those used by the macro package.

The conversion to and from a token list means that you normally can not use LUA line comments (starting with `--`) within the argument. As there typically will be only one 'line' the first line comment will run on until the end of the input. You will either need to use TEX-style line comments (starting with %), or change the TEX category codes locally. Another possibility is to say:

```
\begingroup
\endlinechar=10
\directlua ...
\endgroup
```

Then LUA line comments can be used, since TEX does not replace line endings with spaces.

The `name` ⟨general text⟩ specifies the name of the LUA chunk, mainly shown in the stack backtrace of error messages created by LUA code. The ⟨general text⟩ is expanded fully, thus macros can be used to generate the chunk name, i.e.

```
\directlua name{\jobname:\the\inputlineno} ...
```

to include the name of the input file as well as the input line into the chunk name.

Likewise, the ⟨16-bit number⟩ designates a name of a LUA chunk, but in this case the name will be taken from the `lua.name` array (see the documentation of the `lua` table further in this manual).

The chunk name should not start with a `@`, or it will be displayed as a file name (this is a quirk in the current LUA implementation).

The \directlua command is expandable. Since it passes LUA code to the LUA interpreter its expansion from the TEX viewpoint is usually empty. However, there are some LUA functions that produce material to be read by TEX, the so called print functions. The most simple use of these is `tex.print(<string> s)`. The characters of the string `s` will be placed on the TEX input buffer, that is, 'before TEX's eyes' to be read by TEX immediately. For example:

```
\count10=20
a\directlua{tex.print(tex.count[10]+5)}b
```

expands to

a25b

Here is another example:

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

will result in

$\pi = 3.14159265358 98$

Note that the expansion of \directlua is a sequence of characters, not of tokens, contrary to all TEX commands. So formally speaking its expansion is null, but it places material on a pseudo-file to be immediately read by TEX, as ε-TEX's \scantokens. For a description of print functions look at section 7.14.10.

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. Often, 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.

The behavior documented in the above subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

1.6.2 \latelua

\latelua stores LUA code in a whatsit that will be processed at the time of shipping out. Its intended use is a cross between \pdfliteral and \write. Within the LUA code you can print PDF statements directly to the PDF file via `pdf.print`, or you can write to other output streams via `texio.write` or simply using LUA I/O routines.

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

Expansion of macros etcetera in the final `<general text>` is delayed until just before the whatsit is executed (like in \write). With regard to PDF output stream \latelua behaves as \pdfliteral page. The name ⟨general text⟩ and ⟨16-bit number⟩ behave in the same way as they do for \directlua

1.6.3 \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 and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to n and r respectively. The token sequence is fully expanded.

\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:

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

1.6.4 \luafunction

The \directlua commands involves tokenization of its argument (after picking up an optional name or number specification). The tokenlist is then converted into a string and given to LUA to turn into a function that is called. The overhead is rather small but when you use this primitive hundreds or thousands of times, it can become noticeable. For this reason there is a variant call available: \luafunction. This command is used as follows:

```
\directlua {
    local t = lua.get_functions_table()
    t[1] = function() tex.print("!") end
    t[2] = function() tex.print("?") end
}

\luafunction1
\luafunction2
```

Of course the functions can also be defined in a separate file. There is no limit on the number of functions apart from normal LUA limitations. Of course there is the limitation of no arguments but that would involve parsing and thereby give no gain. The function, when called in fact gets one argument, being the index, so in the following example the number 8 gets typeset.

```
\directlua {
    local t = lua.get_functions_table()
    t[8] = function(slot) tex.print(slot) end
}
```

1.7 \clearmarks

This primitive complements the ε-TEX mark primitives and clears a mark class completely, resetting all three connected mark texts to empty. It is an immediate command.

`\clearmarks` ⟨16-bit number⟩

1.8 `\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⟩

These primitives can now be implemented by overloading the ligature building and kerning functions, i.e. by assigning dummy functions to their associated callbacks.

1.9 `\formatname`

The `\formatname` 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.

1.10 `\scantextokens`

The syntax of `\scantextokens` is identical to `\scantokens`. This primitive 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.

1.11 Alignments

1.11.1 `\alignmark`

This primitive duplicates the functionality of `#` inside alignment preambles.

1.11.2 `\aligntab`

This primitive duplicates the functionality of `&` inside alignments and preambles.

1.12 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. This 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 from any other catcode tables, and their contents is stored and retrieved from the format file.

1.12.1 \catcodetable

`\catcodetable` ⟨15-bit number⟩

The primitive `\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.

1.12.2 \initcatcodetable

`\initcatcodetable` ⟨15-bit number⟩

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

0	`\letterbackslash`		`escape`	
5	`\letterhat \letterhat M`	return	`car_ret`	(this name may change)
9	`\letterhat \letterhat @`	null	`ignore`	
10	`<space>`	space	`spacer`	
11	`a - z`		`letter`	
11	`A - Z`		`letter`	
12	everything else		`other`	
14	`\letterpercent`		`comment`	
15	`\letterhat \letterhat ?`	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.

1.12.3 \savecatcodetable

`\savecatcodetable` ⟨15-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.

1.13 Suppressing errors

1.13.1 \suppressfontnotfounderror

```
\suppressfontnotfounderror = 1
```

If this new integer parameter is non-zero, then LuaTeX will not complain about font metrics that are not found. Instead it will silently skip the font assignment, making the requested csname for the font `\ifx` equal to `\nullfont`, so that it can be tested against that without bothering the user.

1.13.2 \suppresslongerror

```
\suppresslongerror = 1
```

If this new integer parameter is non-zero, then LuaTeX will not complain about `\par` commands encountered in contexts where that is normally prohibited (most prominently in the arguments of non-long macros).

1.13.3 \suppressifcsnameerror

```
\suppressifcsnameerror = 1
```

If this new integer parameter is non-zero, then LuaTeX will not complain about non-expandable commands appearing in the middle of a `\ifcsname` expansion. Instead, it will keep getting expanded tokens from the input until it encounters an `\endcsname` command. Use with care! This command is experimental: if the input expansion is unbalanced wrt. `\csname` ...`\endcsname` pairs, the LuaTeX process may hang indefinitely.

1.13.4 \suppressoutererror

```
\suppressoutererror = 1
```

If this new integer parameter is non-zero, then LuaTeX will not complain about `\outer` commands encountered in contexts where that is normally prohibited.

1.13.5 \suppressmathparerror

The following setting will permit

tokens in a math formula:

```
\suppressmathparerror = 1
```

So, the next code is valid then:

```
$ x + 1 =

a $
```

1.14 \matheqnogapstep

By default TeX will add one quad between the equation and the number. This is hardcoded. A new primitive can control this:

```
\matheqnogapstep = 1000
```

Because a math quad from the math text font is used instead of a dimension, we use a step to control the size. A value of zero will suppress the gap. The step is divided by 1000 which is the usual way to mimmick floating point factors in TEX.

1.15 \outputbox

```
\outputbox = 65535
```

This new integer parameter allows you to alter the number of the box that will be used to store the page sent to the output routine. Its default value is 255, and the acceptable range is from 0 to 65535.

1.16 \fontid

```
\fontid\font
```

This primitive expands into a number. It is not a register so there is no need to prefix with `\number` (and using `\the` gives an error). The currently used font id is 1. Here are some more:

`\bf` **13**
`\it` *17*
`\bi` ***20***

These numbers depend on the macro package used because each one has its own way of dealing with fonts. They can also differ per run, as they can depend on the order of loading fonts. For instance, when in ConTEXt virtual math Unicode fonts are used, we can easily get over a hundred ids in use. Not all ids have to be bound to a real font, after all it's just a number.

1.17 \gleaders

This type of leaders is anchored to the origin of the box to be shipped out. So they are like normal `\leaders` in that they align nicely, except that the alignment is based on the *largest* enclosing box instead of the *smallest*. The `g` stresses this global nature.

1.18 \Uchar

The expandable command `\Uchar` reads a number between 0 and 1,114,111 and expands to the associated Unicode character.

1.19 \hyphenationmin

This primitive can be used to set the minimal word length, so setting it to a value of 5 means that only words of 6 characters and more will be hyphenated, of course within the constraints of the `\lefthyphenmin` and `\righthyphenmin` values (as stored in the glyph node). This primitive accepts a number and stores the value with the language.

1.20 Debugging

If \tracingonline is larger than 2, the node list display will also print the node number of the nodes.

1.21 Images and Forms

LuaTeX accepts optional dimension parameters for \pdfrefximage and \pdfrefxform in the same format as for \pdfximage. With images, these dimensions are then used instead of the ones given to \pdfximage but the original dimensions are not overwritten, so that a \pdfrefximage without dimensions still provides the image with dimensions defined by \pdfximage. These optional parameters are not implemented for \pdfxform.

```
\pdfrefximage width 20mm height 10mm depth 5mm \pdflastximage
\pdfrefxform  width 20mm height 10mm depth 5mm \pdflastxform
```

1.22 File syntax

LuaTeX will accept a braced argument as a file name:

```
\input {plain}
\openin 0 {plain}
```

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

1.23 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.

2 LUA general

2.1 Initialization

2.1.1 LUATEX as a LUA interpreter

There are some situations that make LUATEX behave like a standalone LUA interpreter:

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

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

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

2.1.2 LUATEX as a LUA byte compiler

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

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

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

2.1.3 Other commandline processing

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

`--fmt=FORMAT`	load the format file `FORMAT`
`--lua=FILE`	load and execute a LUA initialization script
`--safer`	disable easily exploitable LUA commands
`--nosocket`	disable the LUA socket library
`--help`	display help and exit
`--ini`	be iniluatex, for dumping formats
`--interaction=STRING`	set interaction mode: `batchmode`, `nonstopmode` `scrollmode` or `errorstopmode`
`--halt-on-error`	stop processing at the first error
`--kpathsea-debug=NUMBER`	set path searching debugging flags according to the bits of `NUMBER`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Commandline processing happens very early on. So early, in fact, that none of TEX's initializations have taken place yet. For that reason, the tables that deal with typesetting, like `tex`, `token`, `node` and `pdf`, are off-limits during the execution of the startup file (they are nilled). Special care is taken that `texio.write` and `texio.write_nl` function properly, so that you can at least report your actions to the log file when (and if) it eventually becomes opened (note that TEX does not even know its `\jobname` yet at this point). See chapter 7 for more information about the LUATEX-specific LUA extension tables.

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

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

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

Unless the `texconfig` table tells LUATEX not to initialize KPATHSEA at all (set `texconfig.kpse_init` to `false` for that), LUATEX acts on some more commandline options after the initialization script is finished: in order to initialize the built-in KPATHSEA library properly, LUATEX needs to know the correct program name to use, and for that it needs to check `--progname`, or `--ini` and `--fmt`, if `--progname` is missing.

2.2 LUA behaviour

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

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

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

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

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

This functionality required an extension to kpathsea:

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

In keeping with the other TEX-like programs in TEXLIVE, the two LUA functions `os.execute` and `io.popen`, as well as the two new functions `os.exec` and `os.spawn` that are explained below, take the value of `shell_escape` and/or `shell_escape_commands` in account. Whenever LUATEX is run with the assumed intention to typeset a document (and by that we mean that it is called as `luatex`, as opposed to `texlua`, and that the commandline option `--luaonly` was not given), it will only run the four functions above if the matching `texmf.cnf` variable(s) or their `texconfig` (see section 7.15) counterparts allow execution of the requested system command. In 'script interpreter' runs of LuaTeX, these settings have no effect, and all four functions function as normal.

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

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

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

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

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

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

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

The `string` library functions `len`, `lower`, `sub` etc. are not Unicode-aware. For strings in the UTF8 encoding, i.e., strings containing characters above code point 127, the corresponding functions from the `slnunicode` library can be used, e.g., `unicode.utf8.len`, `unicode.utf8.lower` etc. The exceptions are `unicode.utf8.find`, that always returns byte positions in a string, and `unicode.utf8.match` and `unicode.utf8.gmatch`. While the latter two functions in general *are* Unicode-aware, they fall-back to non-Unicode-aware behavior when using the empty capture `()` but other captures work as expected. For the interpretation of character classes in `unicode.utf8` functions refer to the library sources at http://luaforge.net/projects/sln. Version 5.3 of Lua will provide some native UTF8 support.

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

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

simply `windows` and `msdos`
The list for the type `unix` is more precise: `linux`, `freebsd`, `kfreebsd`, `cygwin`, `openbsd`, `solaris`, `sunos` (pre-solaris), `hpux`, `irix`, `macosx`, `gnu` (hurd), `bsd` (unknown, but BSD-like), `sysv` (unknown, but SYSV-like), `generic` (unknown).

- `os.version` is planned as a future extension.
- `os.uname()` returns a table with specific operating system information acquired at runtime. The keys in the returned table are all string valued, and their names are: `sysname`, `machine`, `release`, `version`, and `nodename`.

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

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

2.3 LUA modules

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

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

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

3 Languages and characters, fonts and glyphs

LuaTeX's internal handling of the characters and glyphs that eventually become typeset is quite different from the way TeX82 handles those same objects. The easiest way to explain the difference is to focus on unrestricted horizontal mode (i.e. paragraphs) and hyphenation first. Later on, it will be easy to deal with the differences that occur in horizontal and math modes.

In TeX82, the characters you type are converted into `char_node` records when they are encountered by the main control loop. TeX attaches and processes the font information while creating those records, so that the resulting 'horizontal list' contains the final forms of ligatures and implicit kerning. This packaging is needed because we may want to get the effective width of for instance a horizontal box.

When it becomes necessary to hyphenate words in a paragraph, TeX converts (one word at time) the `char_node` records into a string array by replacing ligatures with their components and ignoring the kerning. Then it runs the hyphenation algorithm on this string, and converts the hyphenated result back into a 'horizontal list' that is consecutively spliced back into the paragraph stream. Keep in mind that the paragraph may contain unboxed horizontal material, which then already contains ligatures and kerns and the words therein are part of the hyphenation process.

The `char_node` records are somewhat misnamed, as they are glyph positions in specific fonts, and therefore not really 'characters' in the linguistic sense. There is no language information inside the `char_node` records. Instead, language information is passed along using `language whatsit` records inside the horizontal list.

In LuaTeX, the situation is quite different. The characters you type are always converted into `glyph_node` records with a special subtype to identify them as being intended as linguistic characters. LuaTeX stores the needed language information in those records, but does not do any font-related processing at the time of node creation. It only stores the index of the current font.

When it becomes necessary to typeset a paragraph, LuaTeX first inserts all hyphenation points right into the whole node list. Next, it processes all the font information in the whole list (creating ligatures and adjusting kerning), and finally it adjusts all the subtype identifiers so that the records are 'glyph nodes' from now on.

That was the broad overview. The rest of this chapter will deal with the minutiae of the new process.

3.1 Characters and glyphs

TeX82 (including pdfTeX) differentiates between `char_node`s and `lig_node`s. The former are simple items that contained nothing but a 'character' and a 'font' field, and they lived in the same memory as tokens did. The latter also contained a list of components, and a subtype indicating whether this ligature was the result of a word boundary, and it was stored in the same place as other nodes like boxes and kerns and glues.

In LuaTeX, these two types are merged into one, somewhat larger structure called a `glyph_node`. Besides having the old character, font, and component fields, and the new special fields like 'attr' (see section 6.1.2.12), these nodes also contain:

- A subtype, split into four main types:
  - `character`, for characters to be hyphenated: the lowest bit (bit 0) is set to 1.
  - `glyph`, for specific font glyphs: the lowest bit (bit 0) is not set.
  - `ligature`, for ligatures (bit 1 is set)
  - `ghost`, for 'ghost objects' (bit 2 is set)

  The latter two make further use of two extra fields (bits 3 and 4):
  - `left`, for ligatures created from a left word boundary and for ghosts created from `\leftghost`
  - `right`, for ligatures created from a right word boundary and for ghosts created from `\rightghost`

  For ligatures, both bits can be set at the same time (in case of a single-glyph word).
- `glyph_node`s of type 'character' also contain language data, split into four items that were current when the node was created: the `\setlanguage` (15 bits), `\lefthyphenmin` (8 bits), `\righthyphenmin` (8 bits), and `\uchyph` (1 bit).

Incidentally, LuaTeX allows 16383 separate languages, and words can be 256 characters long.

The new primitive `\hyphenationmin` can be used to signal the minimal length of a word. This value stored with the (current) language.

Because the `\uchyph` value is saved in the actual nodes, its handling is subtly different from TeX82: changes to `\uchyph` become effective immediately, not at the end of the current partial paragraph.

Typeset boxes now always have their language information embedded in the nodes themselves, so there is no longer a possible dependency on the surrounding language settings. In TeX82, a mid-paragraph statement like `\unhbox0` would process the box using the current paragraph language unless there was a `\setlanguage` issued inside the box. In LuaTeX, all language variables are already frozen.

3.2 The main control loop

In LuaTeX's main loop, almost all input characters that are to be typeset are converted into `glyph` node records with subtype 'character', but there are a few exceptions.

First, the `\accent` primitives creates nodes with subtype 'glyph' instead of 'character': one for the actual accent and one for the accentee. The primary reason for this is that `\accent` in TeX82 is explicitly dependent on the current font encoding, so it would not make much sense to attach a new meaning to the primitive's name, as that would invalidate many old documents and macro packages. A secondary reason is that in TeX82, `\accent` prohibits hyphenation of the current word. Since in LuaTeX hyphenation only takes place on 'character' nodes, it is possible to achieve the same effect.

This change of meaning did happen with `\char`, that now generates 'glyph' nodes with a character subtype. In traditional TeX there was a strong relationship betwene the 8-bit input encoding, hyphenation and glyph staken from a font. In LuaTeX we have UTF input, and in most cases this maps directly to a character in a font, apart from glyph replacement in the font engine. If you want to access arbitrary glyphs in a font directly you can alwasy use Lua to do so, because fonts are available as Lua table.

Second, all the results of processing in math mode eventually become nodes with 'glyph' subtypes.

Third, the ALEPH-derived commands `\leftghost` and `\rightghost` create nodes of a third subtype: ‘ghost’. These nodes are ignored completely by all further processing until the stage where inter-glyph kerning is added.

Fourth, automatic discretionaries are handled differently. TEX82 inserts an empty discretionary after sensing an input character that matches the `\hyphenchar` in the current font. This test is wrong, in our opinion: whether or not hyphenation takes place should not depend on the current font, it is a language property.

In LUATEX, it works like this: if LUATEX senses a string of input characters that matches the value of the new integer parameter `\exhyphenchar`, it will insert an explicit discretionary after that series of nodes. Initex sets the ``\exhyphenchar=`\-``. Incidentally, this is a global parameter instead of a language-specific one because it may be useful to change the value depending on the document structure instead of the text language.

The insertion of discretionaries after a sequence of explicit hyphens happens at the same time as the other hyphenation processing, *not* inside the main control loop.

The only use LUATEX has for `\hyphenchar` is at the check whether a word should be considered for hyphenation at all. If the `\hyphenchar` of the font attached to the first character node in a word is negative, then hyphenation of that word is abandoned immediately. **This behavior is added for backward compatibility only, and the use of `\hyphenchar=-1` as a means of preventing hyphenation should not be used in new LUATEX documents.**

Fifth, `\setlanguage` no longer creates whatsits. The meaning of `\setlanguage` is changed so that it is now an integer parameter like all others. That integer parameter is used in `\glyph_node` creation to add language information to the glyph nodes. In conjunction, the `\language` primitive is extended so that it always also updates the value of `\setlanguage`.

Sixth, the `\noboundary` command (this command prohibits word boundary processing where that would normally take place) now does create whatsits. These whatsits are needed because the exact place of the `\noboundary` command in the input stream has to be retained until after the ligature and font processing stages.

Finally, there is no longer a `main_loop` label in the code. Remember that TEX82 did quite a lot of processing while adding `char_nodes` to the horizontal list? For speed reasons, it handled that processing code outside of the ‘main control’ loop, and only the first character of any ‘word’ was handled by that ‘main control’ loop. In LUATEX, there is no longer a need for that (all hard work is done later), and the (now very small) bits of character-handling code have been moved back inline. When `\tracingcommands` is on, this is visible because the full word is reported, instead of just the initial character.

3.3 Loading patterns and exceptions

The hyphenation algorithm in LUATEX is quite different from the one in TEX82, although it uses essentially the same user input.

After expansion, the argument for `\patterns` has to be proper UTF8 with individual patterns separated by spaces, no `\char` or `\chardef`d commands are allowed. The current implementation is even more strict, and will reject all non-UNICODE characters, but that will be changed in the future. For now, the generated errors are a valuable tool in discovering font-encoding specific pattern files.

Likewise, the expanded argument for `\hyphenation` also has to be proper UTF8, but here a tiny little bit of extra syntax is provided:

1. Three sets of arguments in curly braces (`{}{}{}`) indicates a desired complex discretionary, with arguments as in `\discretionary`'s command in normal document input.
2. A `-` indicates a desired simple discretionary, cf. `\-` and `\discretionary{-}{}{}` in normal document input.
3. Internal command names are ignored. This rule is provided especially for `\discretionary`, but it also helps to deal with `\relax` commands that may sneak in.
4. An `=` indicates a (non-discretionary) hyphen in the document input.

The expanded argument is first converted back to a space-separated string while dropping the internal command names. This string is then converted into a dictionary by a routine that creates key-value pairs by converting the other listed items. It is important to note that the keys in an exception dictionary can always be generated from the values. Here are a few examples:

value	implied key (input)	effect
`ta-ble`	table	`ta\-ble` (= `ta\discretionary{-}{}{}ble`)
`ba{k-}{}{c}ken`	backen	`ba\discretionary{k-}{}{c}ken`

The resultant patterns and exception dictionary will be stored under the language code that is the present value of `\language`.

In the last line of the table, you see there is no `\discretionary` command in the value: the command is optional in the TEX-based input syntax. The underlying reason for that is that it is conceivable that a whole dictionary of words is stored as a plain text file and loaded into LUATEX using one of the functions in the LUA `lang` library. This loading method is quite a bit faster than going through the TEX language primitives, but some (most?) of that speed gain would be lost if it had to interpret command sequences while doing so.

It is possible to specify extra hyphenation points in compound words by using `{-}{}{-}` for the explicit hyphen character (replace `-` by the actual explicit hyphen character if needed). For example, this matches the word 'multi-word-boundaries' and allows an extra break inbetweem 'boun' and 'daries':

```
\hyphenation{multi{-}{}{-}word{-}{}{-}boun-daries}
```

The motivation behind the ε-TEX extension `\savinghyphcodes` was that hyphenation heavily depended on font encodings. This is no longer true in LUATEX, and the corresponding primitive is ignored pending complete removal. The future semantics of `\uppercase` and `\lowercase` are still under consideration, no changes have taken place yet.

3.4 Applying hyphenation

The internal structures LUATEX uses for the insertion of discretionaries in words is very different from the ones in TEX82, and that means there are some noticeable differences in handling as well.

First and foremost, there is no 'compressed trie' involved in hyphenation. The algorithm still reads PATGEN-generated pattern files, but LUATEX uses a finite state hash to match the patterns against the word to be hyphenated. This algorithm is based on the 'libhnj' library used by OpenOffice, which in turn is inspired by TeX. The memory allocation for this new implementation is completely dynamic, so the web2c setting for `trie_size` is ignored.

Differences between LuaTeX and TeX82 that are a direct result of that:

- LuaTeX happily hyphenates the full Unicode character range.
- Pattern and exception dictionary size is limited by the available memory only, all allocations are done dynamically. The trie-related settings in `texmf.cnf` are ignored.
- Because there is no 'trie preparation' stage, language patterns never become frozen. This means that the primitive `\patterns` (and its Lua counterpart `lang.patterns`) can be used at any time, not only in iniTeX.
- Only the string representation of `\patterns` and `\hyphenation` is stored in the format file. At format load time, they are simply re-evaluated. It follows that there is no real reason to preload languages in the format file. In fact, it is usually not a good idea to do so. It is much smarter to load patterns no sooner than the first time they are actually needed.
- LuaTeX uses the language-specific variables `\prehyphenchar` and `\posthyphenchar` in the creation of implicit discretionaries, instead of TeX82's `\hyphenchar`, and the values of the language-specific variables `\preexhyphenchar` and `\postexhyphenchar` for explicit discretionaries (instead of TeX82's empty discretionary).
- The value of the two counters related to hyphenation, `hyphenpenalty` and `exhyphenpenalty`, are now stored in the discretionary nodes. This permits a local overload for explicit `\discretionary` commands. The value current when the hyphenation pass is applied is used. When no callbacks are used this is compatible with traditional TeX. When you apply the Lua `lang.hyphenate` function the current values are used.

Inserted characters and ligatures inherit their attributes from the nearest glyph node item (usually the preceding one, but the following one for the items inserted at the left-hand side of a word).

Word boundaries are no longer implied by font switches, but by language switches. One word can have two separate fonts and still be hyphenated correctly (but it can not have two different languages, the `\setlanguage` command forces a word boundary).

All languages start out with `\prehyphenchar=`\-`, `\posthyphenchar=0`, `\preexhyphenchar=0` and `\postexhyphenchar=0`. When you assign the values of one of these four parameters, you are actually changing the settings for the current `\language`, this behavior is compatible with `\patterns` and `\hyphenation`.

LuaTeX also hyphenates the first word in a paragraph. Words can be up to 256 characters long (up from 64 in TeX82). Longer words generate an error right now, but eventually either the limitation will be removed or perhaps it will become possible to silently ignore the excess characters (this is what happens in TeX82, but there the behavior cannot be controlled).

If you are using the Lua function `lang.hyphenate`, you should be aware that this function expects to receive a list of 'character' nodes. It will not operate properly in the presence of 'glyph', 'ligature', or 'ghost' nodes, nor does it know how to deal with kerning. In the near future, it will be able to skip over 'ghost' nodes, and we may add a less fuzzy function you can call as well.

The hyphenation exception dictionary is maintained as key-value hash, and that is also dynamic, so the `hyph_size` setting is not used either.

3.5 Applying ligatures and kerning

After all possible hyphenation points have been inserted in the list, LuaTeX will process the list to convert the 'character' nodes into 'glyph' and 'ligature' nodes. This is actually done in two stages: first all ligatures are processed, then all kerning information is applied to the result list. But those two stages are somewhat dependent on each other: If the used font makes it possible to do so, the ligaturing stage adds virtual 'character' nodes to the word boundaries in the list. While doing so, it removes and interprets `noboundary` nodes. The kerning stage deletes those word boundary items after it is done with them, and it does the same for 'ghost' nodes. Finally, at the end of the kerning stage, all remaining 'character' nodes are converted to 'glyph' nodes.

This work separation is worth mentioning because, if you overrule from Lua only one of the two callbacks related to font handling, then you have to make sure you perform the tasks normally done by LuaTeX itself in order to make sure that the other, non-overruled, routine continues to function properly.

Work in this area is not yet complete, but most of the possible cases are handled by our rewritten ligaturing engine. We are working hard to make sure all of the possible inputs will become supported soon.

For example, take the word `office`, hyphenated `of-fice`, using a 'normal' font with all the `f-f` and `f-i` type ligatures:

Initial:	{o}{f}{f}{i}{c}{e}
After hyphenation:	{o}{f}{{-},{},{}}{f}{i}{c}{e}
First ligature stage:	{o}{{f-},{f},{<ff>}}{i}{c}{e}
Final result:	{o}{{f-},{<fi>},{<ffi>}}{c}{e}

That's bad enough, but let us assume that there is also a hyphenation point between the `f` and the `i`, to create `of-f-ice`. Then the final result should be:

```
{o}{{f-},
    {{f-},
     {i},
     {<fi>}},
    {{<ff>-},
     {i},
     {<ffi>}}}{c}{e}
```

with discretionaries in the post-break text as well as in the replacement text of the top-level discretionary that resulted from the first hyphenation point.

Here is that nested solution again, in a different representation:

	pre	post	replace
topdisc	f-[1]	sub1	sub2
sub1	f-[2]	i[3]	<fi>[4]
sub2	<ff>-[5]	i[6]	<ffi>[7]

When line breaking is choosing its breakpoints, the following fields will eventually be selected:

`of-f-ice`	`f-`[1]
	`f-`[2]
	`i`[3]
`of-fice`	`f-`[1]
	`<fi>`[4]
`off-ice`	`<ff>-`[5]
	`i`[6]
`office`	`<ffi>`[7]

The current solution in LuaTeX is not able to handle nested discretionaries, but it is in fact smart enough to handle this fictional `of-f-ice` example. It does so by combining two sequential discretionary nodes as if they were a single object (where the second discretionary node is treated as an extension of the first node).

One can observe that the `of-f-ice` and `off-ice` cases both end with the same actual post replacement list (`i`), and that this would be the case even if that `i` was the first item of a potential following ligature like `ic`. This allows LuaTeX to do away with one of the fields, and thus make the whole stuff fit into just two discretionary nodes.

The mapping of the seven list fields to the six fields in this discretionary node pair is as follows:

field	**description**
`disc1.pre`	`f-`[1]
`disc1.post`	`<fi>`[4]
`disc1.replace`	`<ffi>`[7]
`disc2.pre`	`f-`[2]
`disc2.post`	`i`[3,6]
`disc2.replace`	`<ff>-`[5]

What is actually generated after ligaturing has been applied is therefore:

```
{o}{{f-},
    {<fi>},
    {<ffi>}}
   {{f-},
    {i},
    {<ff>-}}{c}{e}
```

The two discretionaries have different subtypes from a discretionary appearing on its own: the first has subtype 4, and the second has subtype 5. The need for these special subtypes stems from the fact that not all of the fields appear in their ‘normal’ location. The second discretionary especially looks odd, with things like the `<ff>-` appearing in `disc2.replace`. The fact that some of the fields have different meanings (and different processing code internally) is what makes it necessary to have different subtypes: this enables LuaTeX to distinguish this sequence of two joined discretionary nodes from the case of two standalone discretionaries appearing in a row.

Of course there is still that relationship with fonts: ligatures can be implemented by mapping a sequence of glyphs onto one glyph, but also by selective replacement and kerning. This means that the above examples are just representing the traditional approach.

3.6 Breaking paragraphs into lines

This code is still almost unchanged, but because of the above-mentioned changes with respect to discretionaries and ligatures, line breaking will potentially be different from traditional TEX. The actual line breaking code is still based on the TEX82 algorithms, and it does not expect there to be discretionaries inside of discretionaries.

But that situation is now fairly common in LuaTEX, due to the changes to the ligaturing mechanism. And also, the LuaTEX discretionary nodes are implemented slightly different from the TEX82 nodes: the `no_break` text is now embedded inside the disc node, where previously these nodes kept their place in the horizontal list (the discretionary node contained a counter indicating how many nodes to skip).

The combined effect of these two differences is that LuaTEX does not always use all of the potential breakpoints in a paragraph, especially when fonts with many ligatures are used.

4 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 (TLT)
encodingbytes	no	no	yes	number	default: depends on format
encodingname	no	no	yes	string	encoding name
fonts	yes	no	yes	table	locally used fonts
psname	no	no	yes	string	actual (POSTSCRIPT) name (this is the PS fontname in the incoming font source, also used as fontname identifier in the PDF output, new in 0.43)
fullname	no	no	yes	string	output font name, used as a fallback in the PDF output if the psname is not set
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
tounicode	no	yes	yes	number	if 1, LUATEX assumes per-glyph tounicode entries are present in the font
stretch	no	no	yes	number	the 'stretch' value from \expandglyphsinfont

shrink	no	no	yes	number	the 'shrink' value from \expandglyphsinfont
step	no	no	yes	number	the 'step' value from \expandglyphsinfont
auto_expand	no	no	yes	boolean	the 'autoexpand' keyword from \expandglyphsinfont
expansion_factor	no	no	no	number	the actual expansion factor of an expanded font
attributes	no	no	yes	string	the \pdffontattr
cache	no	no	yes	string	this key controls caching of the lua table on the tex end. yes: use a reference to the table that is passed to LuaTeX (this is the default). no: don't store the table reference, don't cache any lua data for this font. renew: don't store the table reference, but save a reference to the table that is created at the first access to one of its fields in font.fonts. (new in 0.40.0, before that caching was always yes). Note: the saved reference is thread-local, so be careful when you are using coroutines: an error will be thrown if the table has been cached in one thread, but you reference it from another thread (≈ coroutine)
nomath	no	no	yes	boolean	this key allows a minor speedup for text fonts. if it is present and true, then LuaTeX will not check the character enties for math-specific keys.
slant	no	no	yes	number	This has the same semantics as the SlantFont operator in font map files.
extent	no	no	yes	number	This has the same semantics as the ExtendFont operator in font map files.

The key name is always required. The keys stretch, shrink, step and optionally auto_expand only have meaning when used together: they can be used to replace a post-loading \expandglyphsinfont command. The expansion_factor is value that can be present inside a font in font.fonts. It is the actual expansion factor (a value between -shrink and stretch, with step step) of a font that was automatically generated by the font expansion algorithm. The key attributes can be used to replace \pdffontattr. 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 are:

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 an 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)
top_accent	no	no	maybe	number	character's top accent alignment place, in sp (default zero)
bot_accent	no	no	maybe	number	character's bottom accent alignment place, in sp (default zero)
left_protruding	no	no	maybe	number	character's \lpcode
right_protruding	no	no	maybe	number	character's \rpcode
expansion_factor	no	no	maybe	number	character's \efcode
tounicode	no	no	maybe	string	character's Unicode equivalent(s), in UTF-16BE hexadecimal format
next	no	yes	yes	number	the 'next larger' character index
extensible	no	yes	yes	table	the constituent parts of an extensible recipe
vert_variants	no	no	yes	table	constituent parts of a vertical variant set
horiz_variants	no	no	yes	table	constituent parts of a horizontal variant set

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)?
mathkern	no	no	yes	table	math cut-in specifications

The values of top_accent, bot_accent and mathkern are used only for math accent and superscript placement, see the math chapter 55 in this manual for details.

The values of left_protruding and right_protruding are used only when \protrudechars is non-zero.

Whether or not expansion_factor is used depends on the font's global expansion settings, as well as on the value of \adjustspacing.

The usage of tounicode is this: if this font specifies a tounicode=1 at the top level, then LuaTeX will construct a /ToUnicode entry for the PDF font (or font subset) based on the character-level tounicode strings, where they are available. If a character does not have a sensible Unicode equivalent, do not provide a string either (no empty strings).

If the font-level tounicode is not set, then LuaTeX will build up /ToUnicode based on the TeX code points you used, and any character-level tounicodes will be ignored. *At the moment, the string format is exactly the format that is expected by Adobe CMap files (utf-16BE in hexadecimal encoding), minus the enclosing angle brackets. This may change in the future.* Small example: the tounicode for a fi ligature would be 00660069.

The presence of extensible will overrule next, if that is also present. It in in turn can be overruled by vert_variants.

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 horiz_variants and vert_variants are arrays of components. Each of those components is itself a hash of up to five keys:

key	type	explanation
glyph	number	The character index (note that this is an encoding number, not a name).
extender	number	One (1) if this part is repeatable, zero (0) otherwise.
start	number	Maximum overlap at the starting side (in scaled points).
end	number	Maximum overlap at the ending side (in scaled points).
advance	number	Total advance width of this item (can be zero or missing, then the natural size of the glyph for character component is used).

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.

4.1 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. For TYPE1 fonts, you have to set `encodingbytes` to 2 explicitly. For PK bitmap fonts, wide font encoding is not supported at all.

If no special care is needed, LUATEX currently falls back to the mapfile-based solution used by PDFTEX and DVIPS. This behavior 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

`type3` fonts are provided for backward compatibility only, and do not support the new wide encoding options.

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 its entirety

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. The way to get the correct `index` numbers for TYPE1 fonts is by loading the font via `fontloader.open`; use the table indices as `index` fields.

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.

4.2 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 existence 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. In case your font is referring to itself, you can use the `font.nextid()` function which returns the index of the next to be defined font which is probably the currently defined one.

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

command name	**arguments**	**arg type**	**description**
`font`	1	number	select a new font from the local `fonts` table

`char`	1	number	typeset this character number from the current font, and move right by the character's width
`node`	1	node	output this node (list), and move right by the width of this list
`slot`	2	number	a shortcut for the combination of a font and char command
`push`	0		save current position
`nop`	0		do nothing
`pop`	0		pop position
`rule`	2	2 numbers	output a rule $ht * wd$, 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
`lua`	1	string	execute a LUA script (at `\latelua` time)
`image`	1	image	output an image (the argument can be either an `<image>` variable or an `image_spec` table)
`comment`	any	any	the arguments of this command are ignored

Here is a rather elaborate glyph commands example:

```
...
commands = {
    { 'push' },                     -- remember where we are
    { 'right', 5000 },              -- move right about 0.08pt
    { 'font', 3 },                  -- select the fonts[3] entry
    { 'char', 97 },                 -- place character 97 (ASCII 'a')
    { 'pop' },                      -- go all the way back
    { 'down', -200000 },            -- move upwards by 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 at the start of the `commands` array. Therefore, if the virtual font is essentially only a re-encoding, then you do usually not have create an explicit 'font' command in the array.

Rules inside of `commands` arrays are built up using only two dimensions: they do not have depth. For correct vertical placement, an extra `down` command may be needed.

Regardless of the amount of movement you create within the `commands`, the output pointer will always move by exactly the width that was given in the `width` key of the character hash. Any movements that take place inside the `commands` array are ignored on the upper level.

4.2.1 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 practice, this means that you can create virtual duplicates of existing characters which is useful if you want to create composite characters.

Note: this feature does *not* work the other way around. There can not be 'real' characters in a virtual font! You cannot use this technique for font re-encoding either; you need a truly virtual font for that (because characters that are already present cannot be altered).

4.2.2 Example virtual font

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

```
\directlua {
  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
```

5 Math

The handling of mathematics in LuaTEX differs quite a bit from how TEX82 (and therefore PDFTEX) handles math. First, LuaTEX adds primitives and extends some others so that Unicode input can be used easily. Second, all of TEX82's internal special values (for example for operator spacing) have been made accessible and changeable via control sequences. Third, there are extensions that make it easier to use OpenType math fonts. And finally, there are some extensions that have been proposed in the past that are now added to the engine.

5.1 The current math style

It is possible to discover the math style that will be used for a formula in an expandable fashion (while the math list is still being read). To make this possible, LuaTEX adds the new primitive: `\mathstyle`. This is a 'convert command' like e.g. `\romannumeral`: its value can only be read, not set.

5.1.1 `\mathstyle`

The returned value is between 0 and 7 (in math mode), or -1 (all other modes). For easy testing, the eight math style commands have been altered so that the can be used as numeric values, so you can write code like this:

```
\ifnum\mathstyle=\textstyle
    \message{normal text style}
\else \ifnum\mathstyle=\crampedtextstyle
    \message{cramped text style}
\fi \fi
```

5.1.2 `\Ustack`

There are a few math commands in TEX where the style that will be used is not known straight from the start. These commands (`\over`, `\atop`, `\overwithdelims`, `\atopwithdelims`) would therefore normally return wrong values for `\mathstyle`. To fix this, LuaTEX introduces a special prefix command: `\Ustack`:

```
$\Ustack {a \over b}$
```

The `\Ustack` command will scan the next brace and start a new math group with the correct (numerator) math style.

5.2 Unicode math characters

Character handling is now extended up to the full Unicode range (the `\U` prefix), which is compatible with XeTEX.

The math primitives from TEX are kept as they are, except for the ones that convert from input to math commands: `mathcode`, and `delcode`. These two now allow for a 21-bit character argument on the left hand side of the equals sign.

Some of the new LUATEX primitives read more than one separate value. This is shown in the tables below by a plus sign in the second column.

The input for such primitives would look like this:

```
\def\overbrace{\Umathaccent 0 1 "23DE }
```

Altered TEX82 primitives:

primitive	value range (in hex)
`\mathcode`	0–10FFFF = 0–8000
`\delcode`	0–10FFFF = 0–FFFFFF

Unaltered:

primitive	value range (in hex)
`\mathchardef`	0–8000
`\mathchar`	0–7FFF
`\mathaccent`	0–7FFF
`\delimiter`	0–7FFFFFF
`\radical`	0–7FFFFFF

New primitives that are compatible with XETEX:

primitive	value range (in hex)
`\Umathchardef`	0+0+0–7+FF+10FFFF[1]
`\Umathcharnumdef`[5]	-80000000–7FFFFFFF[3]
`\Umathcode`	0–10FFFF = 0+0+0–7+FF+10FFFF[1]
`\Udelcode`	0–10FFFF = 0+0–FF+10FFFF[2]
`\Umathchar`	0+0+0–7+FF+10FFFF
`\Umathaccent`	0+0+0–7+FF+10FFFF[2,4]
`\Udelimiter`	0+0+0–7+FF+10FFFF[2]
`\Uradical`	0+0–FF+10FFFF[2]
`\Umathcharnum`	-80000000–7FFFFFFF[3]
`\Umathcodenum`	0–10FFFF = -80000000–7FFFFFFF[3]
`\Udelcodenum`	0–10FFFF = -80000000–7FFFFFFF[3]

Note 1: `\Umathchardef<csname>="8"0"0` and `\Umathchardef<number>="8"0"0` are also accepted.

Note 2: The new primitives that deal with delimiter-style objects do not set up a 'large family'. Selecting a suitable size for display purposes is expected to be dealt with by the font via the `\Umathoperatorsize` parameter (more information can be found in a following section).

Note 3: For these three primitives, all information is packed into a single signed integer. For the first two (`\Umathcharnum` and `\Umathcodenum`), the lowest 21 bits are the character code, the 3 bits above that represent the math class, and the family data is kept in the topmost bits (This means that the values for math families 128–255 are actually negative). For \Udelcodenum there is no math class; the math family information is stored in the bits directly on top of the character code. Using these three commands is not as natural as using the two- and three-value commands, so unless you know exactly what you are doing and absolutely require the speedup resulting from the faster input scanning, it is better to use the verbose commands instead.

Note 4: The \Umathaccent command accepts optional keywords to control various details regarding math accents. See section 5.7 below for details.

New primitives that exist in LuaTeX only (all of these will be explained in following sections):

primitive	value range (in hex)
\Uroot	0+0–FF+10FFFF[2]
\Uoverdelimiter	0+0–FF+10FFFF[2]
\Uunderdelimiter	0+0–FF+10FFFF[2]
\Udelimiterover	0+0–FF+10FFFF[2]
\Udelimiterunder	0+0–FF+10FFFF[2]

5.3 Cramped math styles

LuaTeX has four new primitives to set the cramped math styles directly:

```
\crampeddisplaystyle
\crampedtextstyle
\crampedscriptstyle
\crampedscriptscriptstyle
```

These additional commands are not all that valuable on their own, but they come in handy as arguments to the math parameter settings that will be added shortly.

5.4 Math parameter settings

In LuaTeX, the font dimension parameters that TeX used in math typesetting are now accessible via primitive commands. In fact, refactoring of the math engine has resulted in many more parameters than were accessible before.

primitive name	description
\Umathquad	the width of 18mu's
\Umathaxis	height of the vertical center axis of the math formula above the baseline
\Umathoperatorsize	minimum size of large operators in display mode
\Umathoverbarkern	vertical clearance above the rule
\Umathoverbarrule	the width of the rule
\Umathoverbarvgap	vertical clearance below the rule
\Umathunderbarkern	vertical clearance below the rule
\Umathunderbarrule	the width of the rule
\Umathunderbarvgap	vertical clearance above the rule

\Umathradicalkern	vertical clearance above the rule
\Umathradicalrule	the width of the rule
\Umathradicalvgap	vertical clearance below the rule
\Umathradicaldegreebefore	the forward kern that takes place before placement of the radical degree
\Umathradicaldegreeafter	the backward kern that takes place after placement of the radical degree
\Umathradicaldegreeraise	this is the percentage of the total height and depth of the radical sign that the degree is raised by. It is expressed in percents, so 60% is expressed as the integer 60.
\Umathstackvgap	vertical clearance between the two elements in a \atop stack
\Umathstacknumup	numerator shift upward in \atop stack
\Umathstackdenomdown	denominator shift downward in \atop stack
\Umathfractionrule	the width of the rule in a \over
\Umathfractionnumvgap	vertical clearance between the numerator and the rule
\Umathfractionnumup	numerator shift upward in \over
\Umathfractiondenomvgap	vertical clearance between the denominator and the rule
\Umathfractiondenomdown	denominator shift downward in \over
\Umathfractiondelsize	minimum delimiter size for \...withdelims
\Umathlimitabovevgap	vertical clearance for limits above operators
\Umathlimitabovebgap	vertical baseline clearance for limits above operators
\Umathlimitabovekern	space reserved at the top of the limit
\Umathlimitbelowvgap	vertical clearance for limits below operators
\Umathlimitbelowbgap	vertical baseline clearance for limits below operators
\Umathlimitbelowkern	space reserved at the bottom of the limit
\Umathoverdelimitervgap	vertical clearance for limits above delimiters
\Umathoverdelimiterbgap	vertical baseline clearance for limits above delimiters
\Umathunderdelimitervgap	vertical clearance for limits below delimiters
\Umathunderdelimiterbgap	vertical baseline clearance for limits below delimiters
\Umathsubshiftdrop	subscript drop for boxes and subformulas
\Umathsubshiftdown	subscript drop for characters
\Umathsupshiftdrop	superscript drop (raise, actually) for boxes and subformulas
\Umathsupshiftup	superscript raise for characters
\Umathsubsupshiftdown	subscript drop in the presence of a superscript
\Umathsubtopmax	the top of standalone subscripts cannot be higher than this above the baseline
\Umathsupbottommin	the bottom of standalone superscripts cannot be less than this above the baseline
\Umathsupsubbottommax	the bottom of the superscript of a combined super- and subscript be at least as high as this above the baseline
\Umathsubsupvgap	vertical clearance between super- and subscript
\Umathspaceafterscript	additional space added after a super- or subscript
\Umathconnectoroverlapmin	minimum overlap between parts in an extensible recipe

Each of the parameters in this section can be set by a command like this:

```
\Umathquad\displaystyle=1em
```

they obey grouping, and you can use `\the\Umathquad\displaystyle` if needed.

5.5 Font-based Math Parameters

While it is nice to have these math parameters available for tweaking, it would be tedious to have to set each of them by hand. For this reason, LuaTeX initializes a bunch of these parameters whenever you assign a font identifier to a math family based on either the traditional math font dimensions in the font (for assignments to math family 2 and 3 using TFM-based fonts like `cmsy` and `cmex`), or based on the named values in a potential `MathConstants` table when the font is loaded via Lua. If there is a `MathConstants` table, this takes precedence over font dimensions, and in that case no attention is paid to which family is being assigned to: the `MathConstants` tables in the last assigned family sets all parameters.

In the table below, the one-letter style abbreviations and symbolic tfm font dimension names match those using in the TeXbook. Assignments to `\textfont` set the values for the cramped and uncramped display and text styles. Use `\scriptfont` for the script styles, and `\scriptscriptfont` for the scriptscript styles (totalling eight parameters for three font sizes). In the TFM case, assignments only happen in family 2 and family 3 (and of course only for the parameters for which there are font dimensions).

Besides the parameters below, LuaTeX also looks at the 'space' font dimension parameter. For math fonts, this should be set to zero.

variable	style	default value opentype	default value tfm
`\Umathaxis`	–	AxisHeight	axis_height
`\Umathoperatorsize`	D, D'	DisplayOperatorMinHeight	[6]
`\Umathfractiondelsize`	D, D'	FractionDelimiterDisplayStyleSize[9]	delim1
"	T, T', S, S', SS, SS'	FractionDelimiterSize[9]	delim2
`\Umathfractiondenomdown`	D, D'	FractionDenominatorDisplayStyleShiftDown	denom1
"	T, T', S, S', SS, SS'	FractionDenominatorShiftDown	denom2
`\Umathfractiondenomvgap`	D, D'	FractionDenominatorDisplayStyleGapMin	3*default_rule_thickness
"	T, T', S, S', SS, SS'	FractionDenominatorGapMin	default_rule_thickness
`\Umathfractionnumup`	D, D'	FractionNumeratorDisplayStyleShiftUp	num1
"	T, T', S, S', SS, SS'	FractionNumeratorShiftUp	num2
`\Umathfractionnumvgap`	D, D'	FractionNumeratorDisplayStyleGapMin	3*default_rule_thickness
"	T, T', S, S', SS, SS'	FractionNumeratorGapMin	default_rule_thickness
`\Umathfractionrule`	–	FractionRuleThickness	default_rule_thickness
`\Umathlimitabovebgap`	–	UpperLimitBaselineRiseMin	big_op_spacing3
`\Umathlimitabovekern`	–	0[1]	big_op_spacing5
`\Umathlimitabovevgap`	–	UpperLimitGapMin	big_op_spacing1
`\Umathlimitbelowbgap`	–	LowerLimitBaselineDropMin	big_op_spacing4
`\Umathlimitbelowkern`	–	0[1]	big_op_spacing5
`\Umathlimitbelowvgap`	–	LowerLimitGapMin	big_op_spacing2
`\Umathoverdelimitervgap`	–	StretchStackGapBelowMin	big_op_spacing1
`\Umathoverdelimiterbgap`	–	StretchStackTopShiftUp	big_op_spacing3
`\Umathunderdelimitervgap`	–	StretchStackGapAboveMin	big_op_spacing2
`\Umathunderdelimiterbgap`	–	StretchStackBottomShiftDown	big_op_spacing4
`\Umathoverbarkern`	–	OverbarExtraAscender	default_rule_thickness
`\Umathoverbarrule`	–	OverbarRuleThickness	default_rule_thickness
`\Umathoverbarvgap`	–	OverbarVerticalGap	3*default_rule_thickness

\Umathquad	–	<font_size(f)>[1]	math_quad
\Umathradicalkern	–	RadicalExtraAscender	default_rule_thickness
\Umathradicalrule	–	RadicalRuleThickness	<not set>[2]
\Umathradicalvgap	D, D'	RadicalDisplayStyleVerticalGap	(default_rule_thickness+ (abs(math_x_height)/4))[3]
"	T, T', S, S', SS, SS'	RadicalVerticalGap	(default_rule_thickness+ (abs(default_rule_thickness)/4))[3]
\Umathradicaldegreebefore	–	RadicalKernBeforeDegree	<not set>[2]
\Umathradicaldegreeafter	–	RadicalKernAfterDegree	<not set>[2]
\Umathradicaldegreeraise	–	RadicalDegreeBottomRaisePercent	<not set>[2,7]
\Umathspaceafterscript	–	SpaceAfterScript	script_space[4]
\Umathstackdenomdown	D, D'	StackBottomDisplayStyleShiftDown	denom1
"	T, T', S, S', SS, SS'	StackBottomShiftDown	denom2
\Umathstacknumup	D, D'	StackTopDisplayStyleShiftUp	num1
"	T, T', S, S', SS, SS'	StackTopShiftUp	num3
\Umathstackvgap	D, D'	StackDisplayStyleGapMin	7*default_rule_thickness
"	T, T', S, S', SS, SS'	StackGapMin	3*default_rule_thickness
\Umathsubshiftdown	–	SubscriptShiftDown	sub1
\Umathsubshiftdrop	–	SubscriptBaselineDropMin	sub_drop
\Umathsubsupshiftdown	–	SubscriptShiftDownWithSuperscript[8] or SubscriptShiftDown	sub2
\Umathsubtopmax	–	SubscriptTopMax	(abs(math_x_height * 4) / 5)
\Umathsubsupvgap	–	SubSuperscriptGapMin	4*default_rule_thickness
\Umathsupbottommin	–	SuperscriptBottomMin	(abs(math_x_height) / 4)
\Umathsupshiftdrop	–	SuperscriptBaselineDropMax	sup_drop
\Umathsupshiftup	D	SuperscriptShiftUp	sup1
"	T, S, SS,	SuperscriptShiftUp	sup2
"	D', T', S', SS'	SuperscriptShiftUpCramped	sup3
\Umathsupsubbottommax	–	SuperscriptBottomMaxWithSubscript	(abs(math_x_height * 4) / 5)
\Umathunderbarkern	–	UnderbarExtraDescender	default_rule_thickness
\Umathunderbarrule	–	UnderbarRuleThickness	default_rule_thickness
\Umathunderbarvgap	–	UnderbarVerticalGap	3*default_rule_thickness
\Umathconnectoroverlapmin	–	MinConnectorOverlap	0[5]

Note 1: OPENTYPE fonts set \Umathlimitabovekern and \Umathlimitbelowkern to zero and set \Umathquad to the font size of the used font, because these are not supported in the MATH table,

Note 2: TFM fonts do not set \Umathradicalrule because TEX82 uses the height of the radical instead. When this parameter is indeed not set when LUATEX has to typeset a radical, a backward compatibility mode will kick in that assumes that an oldstyle TEX font is used. Also, they do not set \Umathradicaldegreebefore, \Umathradicaldegreeafter, and \Umathradicaldegreeraise. These are then automatically initialized to 5/18quad, −10/18quad, and 60.

Note 3: If tfm fonts are used, then the \Umathradicalvgap is not set until the first time LUATEX has to typeset a formula because this needs parameters from both family2 and family3. This provides a partial backward compatibility with TEX82, but that compatibility is only partial: once the \Umathradicalvgap is set, it will not be recalculated any more.

Note 4: (also if tfm fonts are used) A similar situation arises wrt. \Umathspaceafterscript: it is not set until the first time LUATEX has to typeset a formula. This provides some backward compatibility with TEX82. But once the \Umathspaceafterscript is set, \scriptspace will never be looked at again.

Note 5: Tfm fonts set \Umathconnectoroverlapmin to zero because TEX82 always stacks extensibles without any overlap.

Note 6: The \Umathoperatorsize is only used in \displaystyle, and is only set in OpenType fonts. In TFM font mode, it is artificially set to one scaled point more than the initial attempt's size, so that always the 'first next' will be tried, just like in TEX82.

Note 7: The \Umathradicaldegreeraise is a special case because it is the only parameter that is expressed in a percentage instead of as a number of scaled points.

Note 8: SubscriptShiftDownWithSuperscript does not actually exist in the 'standard' Opentype Math font Cambria, but it is useful enough to be added.

Note 9: FractionDelimiterDisplayStyleSize and FractionDelimiterSize do not actually exist in the 'standard' Opentype Math font Cambria, but were useful enough to be added.

5.6 Math spacing setting

Besides the parameters mentioned in the previous sections, there are also 64 new primitives to control the math spacing table (as explained in Chapter 18 of the TEXbook). The primitive names are a simple matter of combining two math atom types, but for completeness' sake, here is the whole list:

\Umathordordspacing
\Umathordopspacing
\Umathordbinspacing
\Umathordrelspacing
\Umathordopenspacing
\Umathordclosespacing
\Umathordpunctspacing
\Umathordinnerspacing
\Umathopordspacing
\Umathopopspacing
\Umathopbinspacing
\Umathoprelspacing
\Umathopopenspacing
\Umathopclosespacing
\Umathoppunctspacing
\Umathopinnerspacing
\Umathbinordspacing
\Umathbinopspacing
\Umathbinbinspacing
\Umathbinrelspacing
\Umathbinopenspacing
\Umathbinclosespacing
\Umathbinpunctspacing
\Umathbininnerspacing
\Umathrelordspacing
\Umathrelopspacing
\Umathrelbinspacing
\Umathrelrelspacing
\Umathrelopenspacing
\Umathrelclosespacing
\Umathrelpunctspacing
\Umathrelinnerspacing
\Umathopenordspacing
\Umathopenopspacing
\Umathopenbinspacing
\Umathopenrelspacing
\Umathopenopenspacing
\Umathopenclosespacing
\Umathopenpunctspacing
\Umathopeninnerspacing
\Umathcloseordspacing
\Umathcloseopspacing
\Umathclosebinspacing
\Umathcloserelspacing
\Umathcloseopenspacing
\Umathcloseclosespacing
\Umathclosepunctspacing
\Umathcloseinnerspacing
\Umathpunctordspacing
\Umathpunctopspacing

`\Umathpunctbinspacing`	`\Umathinneropspacing`
`\Umathpunctrelspacing`	`\Umathinnerbinspacing`
`\Umathpunctopenspacing`	`\Umathinnerrelspacing`
`\Umathpunctclosespacing`	`\Umathinneropenspacing`
`\Umathpunctpunctspacing`	`\Umathinnerclosespacing`
`\Umathpunctinnerspacing`	`\Umathinnerpunctspacing`
`\Umathinnerordspacing`	`\Umathinnerinnerspacing`

These parameters are of type `\muskip`, so setting a parameter can be done like this:

```
\Umathopordspacing\displaystyle=4mu plus 2mu
```

They are all initialized by initex to the values mentioned in the table in Chapter 18 of the TEXbook.

Note 1: for ease of use as well as for backward compatibility, `\thinmuskip`, `\medmuskip` and `\thickmuskip` are treated especially. In their case a pointer to the corresponding internal parameter is saved, not the actual `\muskip` value. This means that any later changes to one of these three parameters will be taken into account.

Note 2: Careful readers will realise that there are also primitives for the items marked `*` in the TEXbook. These will not actually be used as those combinations of atoms cannot actually happen, but it seemed better not to break orthogonality. They are initialized to zero.

5.7 Math accent handling

LuaTEX supports both top accents and bottom accents in math mode, and math accents stretch automatically (if this is supported by the font the accent comes from, of course). Bottom and combined accents as well as fixed-width math accents are controlled by optional keywords following `\Umathaccent`.

The keyword `bottom` after `\Umathaccent` signals that a bottom accent is needed, and the keyword `both` signals that both a top and a bottom accent are needed (in this case two accents need to be specified, of course).

Then the set of three integers defining the accent is read. This set of integers can be prefixed by the `fixed` keyword to indicate that a non-stretching variant is requested (in case of both accents, this step is repeated).

A simple example:

```
\Umathaccent both fixed 0 0 "20D7 fixed 0 0 "20D7 {example}
```

If a math top accent has to be placed and the accentee is a character and has a non-zero `top_accent` value, then this value will be used to place the accent instead of the `\skewchar` kern used by TEX82.

The `top_accent` value represents a vertical line somewhere in the accentee. The accent will be shifted horizontally such that its own `top_accent` line coincides with the one from the accentee. If the `top_accent` value of the accent is zero, then half the width of the accent followed by its italic correction is used instead.

The vertical placement of a top accent depends on the `x_height` of the font of the accentee (as explained in the TEXbook), but if value that turns out to be zero and the font had a MathConstants table, then `AccentBaseHeight` is used instead.

If a math bottom accent has to be placed, the `bot_accent` value is checked instead of `top_accent`. Because bottom accents do not exist in TEX82, the `\skewchar` kern is ignored.

The vertical placement of a bottom accent is straight below the accentee, no correction takes place.

5.8 Math root extension

The new primitive `\Uroot` allows the construction of a radical noad including a degree field. Its syntax is an extension of `\Uradical`:

```
\Uradical <fam integer> <char integer> <radicand>
\Uroot    <fam integer> <char integer> <degree> <radicand>
```

The placement of the degree is controlled by the math parameters `\Umathradicaldegreebefore`, `\Umathradicaldegreeafter`, and `\Umathradicaldegreeraise`. The degree will be typeset in `\scriptscriptstyle`.

5.9 Math kerning in super- and subscripts

The character fields in a lua-loaded OpenType math font can have a 'mathkern' table. The format of this table is the same as the 'mathkern' table that is returned by the `fontloader` library, except that all height and kern values have to be specified in actual scaled points.

When a super- or subscript has to be placed next to a math item, LUATEX checks whether the super- or subscript and the nucleus are both simple character items. If they are, and if the fonts of both character imtes are OpenType fonts (as opposed to legacy TEX fonts), then LUATEX will use the OpenType MATH algorithm for deciding on the horizontal placement of the super- or subscript.

This works as follows:

- The vertical position of the script is calculated.
- The default horizontal position is flat next to the base character.
- For superscripts, the italic correction of the base character is added.
- For a superscript, two vertical values are calculated: the bottom of the script (after shifting up), and the top of the base. For a subscript, the two values are the top of the (shifted down) script, and the bottom of the base.
- For each of these two locations:
  - find the mathkern value at this height for the base (for a subscript placement, this is the bottom_right corner, for a superscript placement the top_right corner)
  - find the mathkern value at this height for the script (for a subscript placement, this is the top_left corner, for a superscript placement the bottom_left corner)
  - add the found values together to get a preliminary result.
- The horizontal kern to be applied is the smallest of the two results from previous step.

The mathkern value at a specific height is the kern value that is specified by the next higher height and kern pair, or the highest one in the character (if there is no value high enough in the character), or simply zero (if the character has no mathkern pairs at all).

5.10 Scripts on horizontally extensible items like arrows

The primitives `\Uunderdelimiter` and `\Uoverdelimiter` allow the placement of a subscript or superscript on an automatically extensible item and `\Udelimiterunder` and `\Udelimiterover` allow the placement of an automatically extensible item as a subscript or superscript on a nucleus. The input:

```
$\Uoverdelimiter  0 "2194 {\hbox{\strut  overdelimiter}}$
$\Uunderdelimiter 0 "2194 {\hbox{\strut underdelimiter}}$
$\Udelimiterover  0 "2194 {\hbox{\strut  delimiterover}}$
$\Udelimiterunder 0 "2194 {\hbox{\strut delimiterunder}}$
```

will render this:

$$\overset{\text{overdelimiter}}{\longleftrightarrow}\ \underset{\text{underdelimiter}}{\longleftrightarrow}\ \overset{\longleftrightarrow}{\text{delimiterover}}\ \underset{\longleftrightarrow}{\text{delimiterunder}}$$

The vertical placements are controlled by `\Umathunderdelimiterbgap`, `\Umathunderdelimitervgap`, `\Umathoverdelimiterbgap`, and `\Umathoverdelimitervgap` in a similar way as limit placements on large operators. The superscript in `\Uoverdelimiter` is typeset in a suitable scripted style, the subscript in `\Uunderdelimiter` is cramped as well.

5.11 Extensible delimiters

LuaTeX internally uses a structure that supports OpenType 'MathVariants' as well as TFM 'extensible recipes'.

5.12 Other Math changes

5.12.1 Verbose versions of single-character math commands

LuaTeX defines six new primitives that have the same function as `^`, `_`, `$`, and `$$`.

primitive	**explanation**
`\Usuperscript`	Duplicates the functionality of `^`
`\Usubscript`	Duplicates the functionality of `_`
`\Ustartmath`	Duplicates the functionality of `$`, when used in non-math mode.
`\Ustopmath`	Duplicates the functionality of `$`, when used in inline math mode.
`\Ustartdisplaymath`	Duplicates the functionality of `$$`, when used in non-math mode.
`\Ustopdisplaymath`	Duplicates the functionality of `$$`, when used in display math mode.

The \Ustopmath and \Ustopdisplaymath primitives check if the current math mode is the correct one (inline vs. displayed), but you can freely intermix the four mathon/mathoff commands with explicit dollar sign(s).

5.12.2 Allowed math commands in non-math modes

The commands \mathchar, and \Umathchar and control sequences that are the result of \mathchardef or \Umathchardef are also acceptable in the horizontal and vertical modes. In those cases, the \textfont from the requested math family is used.

5.13 Math todo

The following items are still todo.

- Pre-scripts.
- Multi-story stacks.
- Flattened accents for high characters (maybe).
- Better control over the spacing around displays and handling of equation numbers.
- Support for multi-line displays using MATHML style alignment points.

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: `hlist` (0), `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), `noad` (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), `fence` (30), `math_char` (31), `sub_box` (32), `sub_mlist` (33), `math_text_char` (34), `delim` (35), `margin_kern` (36), `glyph` (37), `align_record` (38), `pseudo_file` (39), `pseudo_line` (40), `page_insert` (41), `split_insert` (42), `expr_stack` (43), `nested_list` (44), `span` (45), `attribute` (46), `glue_spec` (47), `attribute_list` (48), `action` (49), `temp` (50), `align_stack` (51), `movement_stack` (52), `if_stack` (53), `unhyphenated` (54), `hyphenated` (55), `delta` (56), `passive` (57), `shape` (58), `fake` (100).

The `\lastnodetype` primitive is ε-TEX compliant. The valid range is still $[-1, 15]$ and glyph nodes (formerly known as char nodes) have number 0 while ligature nodes are mapped to 7. That way macro packages can use the same symbolic names as in traditional ε-TEX. Keep in mind that the internal node numbers are different and that there are more node types than 15.

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	
`writable`	boolean	If this is true, you can't assign to this `glue_spec` because it is one of the preallocated special cases.

These objects are reference counted, so there is actually an extra read-only field named `ref_count` as well. This item type will likely disappear in the future, and the glue fields themselves will become part of the nodes referencing glue items.

The effective width of some glue subtypes depends on the stretch or shrink needed to make the encapsulating box fit its dimensions. For instance, in a paragraph lines normally have glue representing spaces and these stretch of shrink to make the content fit in the available space. The effective_glue function that takes a glue node and a parent (hlist or vlist) returns the effective width of that glue item.

6.1.1.2 attribute_list and attribute 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 attribute lists and attributes by using the dedicated functions in the node library, but for completeness, here is the low-level interface.

An attribute_list item is used as a head pointer for a list of attribute items. It has only one user-visible field:

field	type	explanation
next	<node>	pointer to the first attribute

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.

Valid fields in attribute items:

field	type	explanation
next	<node>	pointer to the next attribute
number	number	the attribute type id
value	number	the attribute value

As mentioned it's better to use the official helpers rather than edit these fields directly. For instance the prev field is used for other purposes and there is no double linked list.

6.1.1.3 action item

Valid fields: action_type, named_id, action_id, file, new_window, data, ref_count
Id: 49

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	read-only

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 a 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`, `head`
Id: `0`

field	type	explanation
`subtype`	number	`0` =unknown origin, `1` =created by linebreaking, `2` =explicit box command, `3` =paragraph indentation box, `4` =alignment column or row, `5` =alignment cell
`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	`0` =normal, `1` =stretching, `2` =shrinking
`head`	<node>	the first node of the body of this list
`dir`	string	the direction of this box, see 6.1.4.7

A warning: never assign a node list to the `head` field unless you are sure its internal link structure is correct, otherwise an error may result.

Note: the new field name `head` was introduced in 0.65 to replace the old name `list`. Use of the name `list` is now deprecated, but it will stay available until at least version 0.80.

6.1.2.2 vlist nodes

Valid fields: As for hlist, except that 'shift' is a displacement perpendicular to the line progression direction, and 'subtype' only has subtypes 0, 4, and 5.

6.1.2.3 rule nodes

Valid fields: attr, width, depth, height, dir
Id: 2

field	type	explanation
subtype	number	unused
attr	<node>	
width	number	the width of the rule; the special value −1073741824 is used for 'running' glue dimensions
height	number	the height of the rule (can be negative)
depth	number	the depth of the rule (can be negative)
dir	string	the direction of this rule, see 6.1.4.7

6.1.2.4 ins nodes

Valid fields: attr, cost, depth, height, spec, head
Id: 3

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

A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error may be result. You can use list instead (often in functions you want to use local variable swith similar names and both names are equally sensible).

6.1.2.5 mark nodes

Valid fields: attr, class, mark
Id: 4

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, head
Id: 5

field	type	explanation
subtype	number	0 =normal, 1 =‘pre’
attr	<node>	
head/list	<node>	adjusted material

A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error may be result.

6.1.2.7 disc nodes

Valid fields: attr, pre, post, replace, penalty
Id: 7

field	type	explanation
subtype	number	indicates the source of a discretionary: 0 =the \discretionary command, 1 =the \- command, 2 =added automatically following a -, 3 =added by the hyphenation algorithm (simple), 4 =added by the hyphenation algorithm (hard, first item), 5 =added by the hyphenation algorithm (hard, second item)
attr	<node>	
pre	<node>	pointer to the pre-break text
post	<node>	pointer to the post-break text
replace	<node>	pointer to the no-break text
penalty	number	the penalty associated with the break, normally \hyphenpenalty or \exhyphenpenalty

The subtype numbers 4 and 5 belong to the ‘of-f-ice’ explanation given elsewhere.

Warning: never assign a node list to the pre, post or replace field unless you are sure its internal link structure is correct, otherwise an error may be result. This limnitation will disappear in the future,

6.1.2.8 math nodes

Valid fields: attr, surround
Id: 9

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
Id: 10

field	type	explanation
subtype	number	0 =\skip, 1-18 =internal glue parameters, 100-103 ='leader' subtypes
attr	<node>	
spec	<node>	pointer to a glue_spec item
leader	<node>	pointer to a box or rule for leaders

The exact meanings of the subtypes are as follows:

1	\lineskip
2	\baselineskip
3	\parskip
4	\abovedisplayskip
5	\belowdisplayskip
6	\abovedisplayshortskip
7	\belowdisplayshortskip
8	\leftskip
9	\rightskip
10	\topskip
11	\splittopskip
12	\tabskip
13	\spaceskip
14	\xspaceskip
15	\parfillskip
16	\thinmuskip
17	\medmuskip
18	\thickmuskip
100	\leaders
101	\cleaders
102	\xleaders
103	\gleaders

6.1.2.10 kern nodes

Valid fields: attr, kern, expansion_factor
Id: 11

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
Id: 12

field	type	explanation
subtype	number	not used
attr	<node>	
penalty	number	

6.1.2.12 glyph nodes

Valid fields: attr, char, font, lang, left, right, uchyph, components, xoffset, yoffset, width, height, depth, expansion_factor
Id: 37

field	type	explanation
subtype	number	bitfield
attr	<node>	
char	number	
font	number	
lang	number	
left	number	
right	number	
uchyph	boolean	
components	<node>	pointer to ligature components
xoffset	number	
yoffset	number	
width	number	
height	number	
depth	number	
expansion_factor	number	

A warning: never assign a node list to the components field unless you are sure its internal link structure is correct, otherwise an error may be result. Valid bits for the subtype field are:

bit	meaning
0	character
1	ligature
2	ghost
3	left
4	right

See section 3.1 for a detailed description of the subtype field.

The expansion_factor has been introduced as part of the separation between font- and backend. It is the result of extensive experiments with a more efficient implementation of expansion. Early versions of LuaTeX already replaced multiple instances of fonts in the backend by scaling but contrary to pdfTeX in LuaTeX we now also got rid of font copies in the frontend and replaced them by expansion factors that travel with glyph nodes. Apart from a cleaner approach this is also a step towards a better separation between front- and backend.

The `is_char` function checks if a node is a glyphnode with a subtype still less than 256. This function can be used to determine if applying font logic to a glyph node makes sense.

6.1.2.13 margin_kern nodes

Valid fields: `attr`, `width`, `glyph`
Id: 36

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

6.1.3 Math nodes

These are the so–called 'noad's and the nodes that are specifically associated with math processing. Most of these nodes contain subnodes so that the list of possible fields is actually quite small. First, the subnodes:

6.1.3.1 Math kernel subnodes

Many object fields in math mode are either simple characters in a specific family or math lists or node lists. There are four associated subnodes that represent these cases (in the following node descriptions these are indicated by the word `<kernel>`).

The `next` and `prev` fields for these subnodes are unused.

6.1.3.1.1 math_char and math_text_char subnodes

Valid fields: `attr`, `fam`, `char`
Id: 31

field	type	explanation
attr	<node>	
char	number	
fam	number	

The `math_char` is the simplest subnode field, it contains the character and family for a single glyph object. The `math_text_char` is a special case that you will not normally encounter, it arises temporarily during math list conversion (its sole function is to suppress a following italic correction).

6.1.3.1.2 sub_box and sub_mlist subnodes

Valid fields: `attr`, `head`
Id: 32

field	type	explanation
attr	<node>	
head	<node>	

These two subnode types are used for subsidiary list items. For sub_box, the head points to a 'normal' vbox or hbox. For sub_mlist, the head points to a math list that is yet to be converted.

A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error may be result.

6.1.3.2 Math delimiter subnode

There is a fifth subnode type that is used exclusively for delimiter fields. As before, the next and prev fields are unused.

6.1.3.2.1 delim subnodes

Valid fields: attr, small_fam, small_char, large_fam, large_char
Id: 35

field	type	explanation
attr	<node>	
small_char	number	
small_fam	number	
large_char	number	
large_fam	number	

The fields large_char and large_fam can be zero, in that case the font that is sed for the small_fam is expected to provide the large version as an extension to the small_char.

6.1.3.3 Math core nodes

First, there are the objects (the TeXbook calls then 'atoms') that are associated with the simple math objects: Ord, Op, Bin, Rel, Open, Close, Punct, Inner, Over, Under, Vcent. These all have the same fields, and they are combined into a single node type with separate subtypes for differentiation.

6.1.3.3.1 simple nodes

Valid fields: attr, nucleus, sub, sup
Id: 16

field	type	explanation
subtype	number	see below
attr	<node>	
nucleus	<kernel>	
sub	<kernel>	
sup	<kernel>	

Operators are a bit special because they occupy three subtypes. subtype.

number	node subtype
0	Ord
1	Op: \displaylimits
2	Op: \limits
3	Op: \nolimits
4	Bin
5	Rel
6	Open
7	Close
8	Punct
9	Inner
10	Under
11	Over
12	Vcent

6.1.3.3.2 accent nodes

Valid fields: attr, nucleus, sub, sup, accent, bot_accent
Id: 28

field	type	explanation
subtype	number	the first bit is used for a fixed top accent flag (if the accent field is present), the second bit for a fixed bottom accent flag (if the bot_accent field is present); example: the actual value 3 means: do not stretch either accent
attr	<node>	
nucleus	<kernel>	
sub	<kernel>	
sup	<kernel>	
accent	<kernel>	
bot_accent	<kernel>	

6.1.3.3.3 style nodes

Valid fields: attr, style
Id: 14

field	type	explanation
style	string	contains the style

There are eight possibilities for the string value: one of 'display', 'text', 'script', or 'scriptscript'. Each of these can have a trailing ' to signify 'cramped' styles.

6.1.3.3.4 choice nodes

Valid fields: attr, display, text, script, scriptscript
Id: 15

field	type	explanation
attr	<node>	
display	<node>	
text	<node>	
script	<node>	
scriptscript	<node>	

A warning: never assign a node list to the display, text, script, or scriptscript field unless you are sure its internal link structure is correct, otherwise an error may be result.

6.1.3.3.5 radical nodes

Valid fields: attr, nucleus, sub, sup, left, degree
Id: 24

field	type	explanation
attr	<node>	
nucleus	<kernel>	
sub	<kernel>	
sup	<kernel>	
left	<delim>	
degree	<kernel>	Only set by \Uroot

A warning: never assign a node list to the nucleus, sub, sup, left, or degree field unless you are sure its internal link structure is correct, otherwise an error may be result.

The radical noad is also used for under- and overdelimiters, which is indicated by the subtypes:

0	\radical
1	\Uradical
2	\Uroot
3	\Uunderdelimiter
4	\Uoverdelimiter
5	\Udelimiterunder
6	\Udelimiterover

6.1.3.3.6 fraction nodes

Valid fields: attr, width, num, denom, left, right
Id: 25

field	type	explanation
attr	<node>	

width	number
num	<kernel>
denom	<kernel>
left	<delim>
right	<delim>

A warning: never assign a node list to the num, or denom field unless you are sure its internal link structure is correct, otherwise an error may be result.

6.1.3.3.7 fence nodes

Valid fields: attr, delim
Id: 30

field	type	explanation
subtype	number	1 =\left, 2 =\middle, 3 =\right
attr	<node>	
delim	<delim>	

6.1.4 whatsit nodes

Whatsit nodes come in many subtypes that you can ask for by running node.whatsits(): open (0), write (1), close (2), special (3), 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), save_pos (23), pdf_thread_data (24), pdf_link_data (25), late_lua (35), close_lua (36), pdf_colorstack (39), pdf_setmatrix (40), pdf_save (41), pdf_restore (42), cancel_boundary (43), user_defined (44), fake (100).

6.1.4.1 open nodes

Valid fields: attr, stream, name, area, ext
Id: 8, 0

field	type	explanation
attr	<node>	
stream	number	TeX's stream id number
name	string	file name
ext	string	file extension
area	string	file area (this may become obsolete)

6.1.4.2 write nodes

Valid fields: attr, stream, data
Id: 8, 1

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

6.1.4.3 close nodes

Valid fields: attr, stream
Id: 8, 2

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

6.1.4.4 special nodes

Valid fields: attr, data
Id: 8, 3

field	type	explanation
attr	<node>	
data	string	the \special information

6.1.4.5 language nodes

LUATEX does not have language whatsits any more. All language information is already present inside the glyph nodes themselves. This whatsit subtype will be removed in the next release.

6.1.4.6 local_par nodes

Valid fields: attr, pen_inter, pen_broken, dir, box_left, box_left_width, box_right, box_right_width
Id: 8, 6

field	type	explanation
attr	<node>	
pen_inter	number	local interline penalty (from \localinterlinepenalty)
pen_broken	number	local broken penalty (from \localbrokenpenalty)
dir	string	the direction of this par. see 6.1.4.7
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

A warning: never assign a node list to the box_left or box_right field unless you are sure its internal link structure is correct, otherwise an error may be result.

6.1.4.7 dir nodes

Valid fields: attr, dir, level, dvi_ptr, dvi_h
Id: 8, 7

field	type	explanation
attr	<node>	
dir	string	the direction (but see below)
level	number	nesting level of this direction whatsit
dvi_ptr	number	a saved dvi buffer byte offset
dir_h	number	a saved dvi position

A note on dir strings. Direction specifiers are three-letter combinations of T, B, R, and L.

These are built up out of three separate items:

- the first is the direction of the 'top' of paragraphs.
- the second is the direction of the 'start' of lines.
- the third is the direction of the 'top' of glyphs.

However, only four combinations are accepted: TLT, TRT, RTT, and LTL.

Inside actual dir whatsit nodes, the representation of dir is not a three-letter but a four-letter combination. The first character in this case is always either + or -, indicating whether the value is pushed or popped from the direction stack.

6.1.4.8 pdf_literal nodes

Valid fields: attr, mode, data
Id: 8, 8

field	type	explanation
attr	<node>	
mode	number	the 'mode' setting of this literal
data	string	the \pdfliteral information

Mode values:

value	corresponding \pdftex keyword
0	setorigin
1	page
2	direct

6.1.4.9 pdf_refobj nodes

Valid fields: attr, objnum
Id: 8, 10

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

6.1.4.10 pdf_refxform nodes

Valid fields: attr, width, depth, height, objnum
Id: 8, 12.

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.

6.1.4.11 pdf_refximage nodes

Valid fields: attr, width, depth, height, transform, index
Id: 8, 14

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.

6.1.4.12 pdf_annot nodes

Valid fields: attr, width, depth, height, objnum, data
Id: 8, 15

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

6.1.4.13 pdf_start_link nodes

Valid fields: attr, width, depth, height, objnum, link_attr, action
Id: 8, 16

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

6.1.4.14 pdf_end_link nodes

Valid fields: attr
Id: 8, 17

field	type	explanation
attr	<node>	

6.1.4.15 pdf_dest nodes

Valid fields: attr, width, depth, height, named_id, dest_id, dest_type, xyz_zoom, objnum
Id: 8, 19

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

6.1.4.16 pdf_thread nodes

Valid fields: attr, width, depth, height, named_id, thread_id, thread_attr
Id: 8, 20

field	type	explanation
attr	<node>	

width	number	
height	number	
depth	number	
named_id	number	is the tread_id a string value?
tread_id	number	the thread id
	string	the thread name
thread_attr	number	extra thread information

6.1.4.17 pdf_start_thread nodes

Valid fields: attr, width, depth, height, named_id, thread_id, thread_attr
Id: 8, 21

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

6.1.4.18 pdf_end_thread nodes

Valid fields: attr
Id: 8, 22

field	type	explanation
attr	<node>	

6.1.4.19 save_pos nodes

Valid fields: attr
Id: 8, 23

field	type	explanation
attr	<node>	

6.1.4.20 late_lua nodes

Valid fields: attr, reg, data, name, string
Id: 8, 35

field	type	explanation
attr	<node>	

data	string	data to execute
string	string	data to execute
name	string	the name to use for lua error reporting

The difference between `data` and `string` is that on assignment, the `data` field is converted to a token list, cf. use as `\latelua`. The `string` version is treated as a literal string.

6.1.4.21 pdf_colorstack nodes

Valid fields: `attr`, `stack`, `cmd`, `data`
Id: 8, 39

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

6.1.4.22 pdf_setmatrix nodes

Valid fields: `attr`, `data`
Id: 8, 40

field	type	explanation
attr	<node>	
data	string	data

6.1.4.23 pdf_save nodes

Valid fields: `attr`
Id: 8, 41

field	type	explanation
attr	<node>	

6.1.4.24 pdf_restore nodes

Valid fields: `attr`
Id: 8, 42

field	type	explanation
attr	<node>	

6.1.4.25 user_defined nodes

User-defined whatsit nodes can only be created and handled from LUA code. In effect, they are an extension to the extension mechanism. The LUATEX engine will simply step over such whatsits without ever looking at the contents.

Valid fields: attr, user_id, type, value
Id: 8, 44

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

The type can have one of five distinct values:

value	explanation
97	the value is an attribute node list
100	the value is a number
110	the value is a node list
115	the value is a string
116	the value is a token list in LUA table form

6.2 Two access models

After doing lots of tests with LUATEX and LUAJITTEX with and without just in time compilation enabled, and with and without using ffi, we came to the conclusion that userdata prevents a speedup. We also found that the checking of metatables as well as assignment comes with overhead that can't be neglected. This is normally not really a problem but when processing fonts for more complex scripts it could have quite some overhead.

Because the userdata approach has some benefits, this remains the recommended way to access nodes. We did several experiments with faster access using this model, but eventually settled for the 'direct' approach. For code that is proven to be okay, one can use this access model that operates on nodes more directly.

Deep down in TEX a node has a number which is an entry in a memory table. In fact, this model, where TEX manages memory is real fast and one of the reasons why plugging in callbacks that operate on nodes is quite fast. No matter what future memory model LUATEX has, an internal reference will always be a simple data type (like a number or light userdata in LUA speak). So, if you use the direct model, even if you know that you currently deal with numbers, you should not depend on that property but treat it an abstraction just like traditional nodes. In fact, the fact that we use a simple basic datatype has the penalty that less checking can be done, but less checking is also the reason why it's somewhat faster. An important aspect is that one cannot mix both methods, but you can cast both models.

So our advice is: use the indexed approach when possible and investigate the direct one when speed might be an issue. For that reason we also provide the get* and set* functions in the top level node namespace. There is a limited set of getters. When implementing this direct approach the regular index by key variant was also optimized, so direct access only makes sense when we're accessing nodes millions of times (which happens in some font processing for instance).

We're talking mostly of getters because setters are less important. Documents have not that many content related nodes and setting many thousands of properties is hardly a burden contrary to millions of consultations.

Normally you will access nodes like this:

```
local next = current.next
if next then
    -- do something
end
```

Here `next` is not a real field, but a virtual one. Accessing it results in a metatable method being called. In practice it boils down to looking up the node type and based on the node type checking for the field name. In a worst case you have a node type that sits at the end of the lookup list and a field that is last in the lookup chain. However, in successive versions of LuaTeX these lookups have been optimized and the most frequently accessed nodes and fields have a higher priority.

Because in practice the `next` accessor results in a function call, there is some overhead involved. The next code does the same and performs a tiny bit faster (but not that much because it is still a function call but one that knows what to look up).

```
local next = node.next(current)
if next then
    -- do something
end
```

There are several such function based accessors now:

`getnext`	parsing nodelist always involves this one
`getprev`	used less but is logical companion to getnext
`getboth`	returns the next and prev pointer of a node
`getid`	consulted a lot
`getsubtype`	consulted less but also a topper
`getfont`	used a lot in otf handling (glyph nodes are consulted a lot)
`getchar`	idem and also in other places
`getdisc`	returns the `pre`, `post` an `dreplace` fields
`getlist`	we often parse nested lists so this is a convenient one too (only works for hlist and vlist!)
`getleader`	comparable to list, seldom used in TeX (but needs frequent consulting like lists; leaders could have been made a dedicated node type)
`getfield`	generic getter, sufficient for the rest (other field names are often shared so a specific getter makes no sense then)

Some have setter counterparts:

There are several such function based accessors now:

setnext	assigns a value to the next field
setprev	assigns a value to the prev field
setboth	assigns a value to the prev and next field
setlink	links two noded
setchar	sets the character field
setdisc	sets the pre, post and replace fields and optionally the subtype and penalty fields

getfont	used a lot in otf handling (glyph nodes are consulted a lot)
getchar	idem and also in other places
getdisc	returns the pre, post an dreplace fields
getlist	we often parse nested lists so this is a convenient one too (only works for hlist and vlist!)
getleader	comparable to list, seldom used in TEX (but needs frequent consulting like lists; leaders could have been made a dedicated node type)
getfield	generic getter, sufficient for the rest (other field names are often shared so a specific getter makes no sense then)

It doesn't make sense to add more. Profiling demonstrated that these fields can get accesses way more times than other fields. Even in complex documents, many node and fields types never get seen, or seen only a few times. Most functions in the node namespace have a companion in node.direct, but of course not the ones that don't deal with nodes themselves. The following table summarized this:

function	node	direct
copy_list	+	+
copy	+	+
count	+	+
current_attr	+	+
dimensions	+	+
do_ligature_n	+	+
effective_glue	+	+
end_of_math	+	+
family_font	+	−
fields	+	−
first_character	+	−
first_glyph	+	+
flush_list	+	+
flush_node	+	+
free	+	+
getboth	+	+
getbox	−	+
getchar	+	+
getdisc	+	+
getfield	+	+
getfont	+	+
getid	+	+

getleader	+	+
getlist	+	+
getnext	+	+
getprev	+	+
getsubtype	+	+
has_attribute	+	+
has_field	+	+
has_glyph	+	+
hpack	+	+
id	+	−
insert_after	+	+
insert_before	+	+
is_char	+	+
is_direct	−	+
is_node	+	+
kerning	+	−
last_node	+	+
length	+	+
ligaturing	+	−
mlist_to_hlist	+	−
new	+	+
next	+	−
prev	+	−
protect_glyphs	+	+
protrusion_skippable	+	+
remove	+	+
set_attribute	+	+
setboth	+	+
setbox	+	+
setchar	+	+
setdisc	+	+
setfield	+	+
setlink	+	+
setnext	+	+
setprev	+	+
slide	+	+
subtype	+	−
tail	+	+
todirect	+	+
tonode	+	+
tostring	+	+
traverse_id	+	+
traverse	+	+
types	+	−
type	+	−

`unprotect_glyphs`	+	+
`unset_attribute`	+	+
`usedlist`	+	+
`vpack`	+	+
`whatsits`	+	−
`write`	+	+

The `node.next` and `node.prev` functions will stay but for consistency there are variants called `getnext` and `getprev`. We had to use `get` because `node.id` and `node.subtype` are already taken for providing meta information about nodes. Note: The getters do only basic checking for valid keys. You should just stick to the keys mentioned in the sections that describe node properties.

7 LuaTeX Lua Libraries

The implied use of the built-in Lua modules `epdf`, `fontloader`, `mplib`, and `pdfscanner` is deprecated. If you want to use these, please start your source file with a proper `require` line. In the future, LuaTeX will switch to loading these modules on demand.

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 TeX functions, and set up LuaTeX's hooks to execute Lua code.

The following sections are in alphabetical order.

7.1 The `callback` library

This library has functions that register, find and list callbacks. Callbacks are Lua functions that are called in well defined places. There are two kind of callbacks: those that mix with existing functionality, and those that (when enabled) replace functionality. In mosty cases the second category is expected to behave similar to the built in functiontionality because in a next step specific data is expected. For instance, you can replace the hyphenation routine. The function gets a list that can be hyphenated (or not). The final list should be valid and is (normally) used for constructing a paragraph. Another function can replace the ligature builder and/or kerner. Doing something else is possible but in the end might not give the user the expected outcome.

The first thing you need to do is registering a callback:

```
id, error = callback.register (<string> callback_name, <function> func)
id, error = callback.register (<string> callback_name, nil)
id, error = callback.register (<string> callback_name, false)
```

Here the `callback_name` is a predefined callback name, see below. The function returns the internal `id` of the callback or `nil`, if the callback could not be registered. In the latter case, `error` contains an error message, otherwise it is `nil`.

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.

For some minor speed gain, you can assign the boolean `false` to the non-file related callbacks, doing so will prevent LuaTeX from executing whatever it would execute by default (when no callback function is registered at all). Be warned: this may cause all sorts of grief unless you know *exactly* what you are doing!

Currently, callbacks are not dumped into the format file.

```
<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`.

7.1.1 File discovery callbacks

The behavior documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

7.1.1.1 `find_read_file` and `find_write_file`

Your 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.

7.1.1.2 `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.

7.1.1.3 `find_output_file`

Your 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.

7.1.1.4 find_format_file

Your 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).

7.1.1.5 find_vf_file

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

7.1.1.6 find_map_file

Like `find_font_file`, but for map files.

7.1.1.7 find_enc_file

Like `find_font_file`, but for enc files.

7.1.1.8 find_sfd_file

Like `find_font_file`, but for subfont definition files.

7.1.1.9 find_pk_file

Like `find_font_file`, but for pk bitmap files. The argument `asked_name` is a bit special in this case. Its 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.

7.1.1.10 find_data_file

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

7.1.1.11 find_opentype_file

Like `find_font_file`, but for OPENTYPE font files.

7.1.1.12 find_truetype_file and find_type1_file

Your 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.

7.1.1.13 find_image_file

Your 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.

7.1.2 File reading callbacks

The behavior documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

7.1.2.1 open_read_file

Your 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 function 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.

7.1.2.1.1 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.

7.1.2.1.2 close

LUATEX will run this optional function when it decides to close the file.

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

Your function should not return any value.

7.1.2.2 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 occurred (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 as follows:

`read_font_file`	ofm or tfm files
`read_vf_file`	virtual fonts
`read_map_file`	map files
`read_enc_file`	encoding files
`read_sfd_file`	subfont definition files
`read_pk_file`	pk bitmap files
`read_data_file`	embedded files (`\pdfobj file ...`)
`read_truetype_file`	TRUETYPE font files
`read_type1_file`	TYPE1 font files
`read_opentype_file`	OPENTYPE font files

7.1.3 Data processing callbacks

7.1.3.1 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.

This callback does not replace any internal code.

7.1.3.2 process_output_buffer

This callback allows you to change the contents of the line output buffer just before LuaTeX actually starts writing it to a file as the result of a \write command. It is only called for output to an actual file (that is, excluding the log, the terminal, and \write18 calls).

```
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.

This callback does not replace any internal code.

7.1.3.3 process_jobname

This callback allows you to change the jobname given by \jobname in TeX and tex.jobname in Lua. It does not affect the internal job name or the name of the output or log files.

```
function(<string> jobname)
    return <string> adjusted_jobname
end
```

The only argument is the actual job name; you should not use tex.jobname inside this function or infinite recursion may occur. If you return nil, LuaTeX will pretend your callback never happened.

This callback does not replace any internal code.

7.1.3.4 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 a 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 at a completely new token list level, with its token type set to 'inserted'. In either case, the returned token(s) will not be fed back into the callback function.

Setting this callback to false has no effect (because otherwise nothing would happen, forever).

7.1.4 Node list processing callbacks

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

7.1.4.1 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(<string> extrainfo)
end
```

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
after_output	an output routine has just finished
box	a typeset box is being added
new_graf	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
end	LUATEX is terminating (it's all over)

This callback does not replace any internal code.

7.1.4.2 `pre_linebreak_filter`

This callback is called just before LuaTeX starts converting a list of nodes into a stack of `\hbox`es, after the addition of `\parfillskip`.

```
function(<node> head, <string> groupcode)
    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**
`<empty>`	main vertical list
`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`
`align_set`	alignment cell
`fin_row`	alignment row

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 the returned node
- boolean `false` signals that the 'head' node list should be ignored and flushed from memory

This callback does not replace any internal code.

7.1.4.3 `linebreak_filter`

This callback replaces LuaTeX's line breaking algorithm.

```
function(<node> head, <boolean> is_display)
    return <node> newhead
end
```

The returned node is the head of the list that will be added to the main vertical list, the boolean argument is true if this paragraph is interrupted by a following math display.

If you return something that is not a `<node>`, LuaTeX will apply the internal linebreak algorithm on the list that starts at `<head>`. Otherwise, the `<node>` you return is supposed to be the head of a list of nodes that are all allowed in vertical mode, and at least one of those has to represent a hbox. Failure to do so will result in a fatal error.

Setting this callback to `false` is possible, but dangerous, because it is possible you will end up in an unfixable 'deadcycles loop'.

7.1.4.4 post_linebreak_filter

This callback is called just after LuaTeX has converted a list of nodes into a stack of `\hbox`es.

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

This callback does not replace any internal code.

7.1.4.5 hpack_filter

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

```
function(<node> head, <string> groupcode, <number> size,
         <string> packtype [, <string> direction])
    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.

The `direction` is either one of the three-letter direction specifier strings, or `nil`.

This callback does not replace any internal code.

7.1.4.6 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.

```
function(<node> head, <string> groupcode, <number> size, <string>
         packtype,  <number> maxdepth [, <string> direction])
```

```
    return true | false | <node> newhead
end
```

This callback does not replace any internal code.

7.1.4.7 pre_output_filter

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

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

This callback does not replace any internal code.

7.1.4.8 hyphenate

```
function(<node> head, <node> tail)
end
```

No return values. This callback has to insert discretionary nodes in the node list it receives.

Setting this callback to false will prevent the internal discretionary insertion pass.

7.1.4.9 ligaturing

```
function(<node> head, <node> tail)
end
```

No return values. This callback has to apply ligaturing to the node list it receives.

You don't have to worry about return values because the head node that is passed on to the callback is guaranteed not to be a glyph_node (if need be, a temporary node will be prepended), and therefore it cannot be affected by the mutations that take place. After the callback, the internal value of the 'tail of the list' will be recalculated.

The next of head is guaranteed to be non-nil.

The next of tail is guaranteed to be nil, and therefore the second callback argument can often be ignored. It is provided for orthogonality, and because it can sometimes be handy when special processing has to take place.

Setting this callback to false will prevent the internal ligature creation pass.

7.1.4.10 kerning

```
function(<node> head, <node> tail)
end
```

No return values. This callback has to apply kerning between the nodes in the node list it receives. See `ligaturing` for calling conventions.

Setting this callback to `false` will prevent the internal kern insertion pass.

7.1.4.11 `mlist_to_hlist`

This callback replaces LuaTeX's math list to node list conversion algorithm.

```
function(<node> head, <string> display_type, <boolean> need_penalties)
    return <node> newhead
end
```

The returned node is the head of the list that will be added to the vertical or horizontal list, the string argument is either 'text' or 'display' depending on the current math mode, the boolean argument is `true` if penalties have to be inserted in this list, `false` otherwise.

Setting this callback to `false` is bad, it will almost certainly result in an endless loop.

7.1.5 Information reporting callbacks

7.1.5.1 `pre_dump`

```
function()
end
```

This function is called just before dumping to a format file starts. It does not replace any code and there are neither arguments nor return values.

7.1.5.2 `start_run`

```
function()
end
```

This callback replaces the code that prints LuaTeX's banner. Note that for successful use, this callback has to be set in the lua initialization script, otherwise it will be seen only after the run has already started.

7.1.5.3 `stop_run`

```
function()
end
```

This callback replaces the code that prints LuaTeX's statistics and 'output written to' messages.

7.1.5.4 `start_page_number`

```
function()
```

```
end
```

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.

7.1.5.5 stop_page_number

```
function()
end
```

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

7.1.5.6 show_error_hook

```
function()
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.

This callback does not replace any internal code.

7.1.5.7 show_error_message

```
function()
end
```

This callback replaces the code that prints the error message. The usual interaction after the message is not affected.

7.1.5.8 show_lua_error_hook

```
function()
end
```

This callback replaces the code that prints the extra lua error message.

7.1.5.9 start_file

```
function(category,filename)
end
```

This callback replaces the code that prints LUATEX's when a file is opened like (filename for regular files. The category is a number:

1 a normal data file, like a TEX source
2 a font map coupling font names to resources
3 an image file (png, pdf, etc)
4 an embedded font subset
5 a fully embedded font

7.1.5.10 stop_file

```
function(category)
end
```

This callback replaces the code that prints LUATEX's when a file is closed like the) for regular files.

7.1.6 PDF-related callbacks

7.1.6.1 finish_pdffile

```
function()
end
```

This callback is called when all document pages are already written to the PDF file and LUATEX is about to finalize the output document structure. Its intended use is final update of PDF dictionaries such as /Catalog or /Info. The callback does not replace any code. There are neither arguments nor return values.

7.1.6.2 finish_pdfpage

```
function(shippingout)
end
```

This callback is called after the pdf page stream has been assembled and before the page object gets finalized.

7.1.7 Font-related callbacks

7.1.7.1 define_font

```
function(<string> name, <number> size, <number> id)
    return <table> font | <number> id
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 `id` is the internal number assigned to the font.

The internal structure of the `font` table that is to be returned is explained in chapter 4. That table is saved internally, so you can put extra fields in the table for your later LUA code to use. In alternative, retval can be a previously defined fontid. This is useful if a previous definition can be reused instead of creating a whole new font structure.

Setting this callback to `false` is pointless as it will prevent font loading completely but will nevertheless generate errors.

7.2 The epdf library

The `epdf` library provides Lua bindings to many PDF access functions that are defined by the poppler pdf viewer library (written in C++ by Kristian Høgsberg, based on xpdf by Derek Noonburg). Within LUATEX (and PDFTEX), xpdf functionality is being used since long time to embed PDF files. The `epdf` library shall allow to scrutinize an external PDF file. It gives access to its document structure, e.g., catalog, cross-reference table, individual pages, objects, annotations, info, and metadata. The LUATEX team is evaluating the possibility of reducing the binding to a basic low level PDF primitives and delegate the complete set of functions to an external shared object module.

The `epdf` library is still in alpha state: PDF access is currently read-only. Iit's not yet possible to alter a PDF file or to assemble it from scratch, and many function bindings are still missing, and it is unlikely that we to support that at all. At some point we might also decide to limit the interface to a reasonable subset.

For a start, a PDF file is opened by `epdf.open()` with file name, e.g.:

```
doc = epdf.open("foo.pdf")
```

This normally returns a `PDFDoc` userdata variable; but if the file could not be opened successfully, instead of a fatal error just the value `nil` is returned.

All Lua functions in the `epdf` library are named after the poppler functions listed in the poppler header files for the various classes, e.g., files `PDFDoc.h`, `Dict.h`, and `Array.h`. These files can be found in the poppler subdirectory within the LUATEX sources. Which functions are already implemented in the `epdf` library can be found in the LUATEX source file `lepdflib.cc`. For using the `epdf` library, knowledge of the PDF file architecture is indispensable.

There are many different userdata types defined by the `epdf` library, currently these are `AnnotBorderStyle`, `AnnotBorder`, `Annots`, `Annot`, `Array`, `Attribute`, `Catalog`, `Dict`, `EmbFile`, `GString`, `LinkDest`, `Links`, `Link`, `ObjectStream`, `Object`, `PDFDoc`, `PDFRectangle`, `Page`, `Ref`, `Stream`, `StructElement`, `StructTreeRoot` `TextSpan`, `XRefEntry` and `XRef`.

All these userdata names and the Lua access functions closely resemble the classes naming from the poppler header files, including the choice of mixed upper and lower case letters. The Lua function calls use object-oriented syntax, e.g., the following calls return the `Page` object for page 1:

```
pageref = doc:getCatalog():getPageRef(1)
pageobj = doc:getXRef():fetch(pageref.num, pageref.gen)
```

But writing such chained calls is risky, as an intermediate function may return nil on error; therefore between function calls there should be Lua type checks (e.g., against nil) done. If a non-object item is requested (e.g., a Dict item by calling page:getPieceInfo(), cf. Page.h) but not available, the Lua functions return nil (without error). If a function should return an Object, but it's not existing, a Null object is returned instead (also without error; this is in-line with poppler behavior).

All library objects have a __gc metamethod for garbage collection. The __tostring metamethod gives the type name for each object.

All object constructors:

```
<PDFDoc>       = epdf.open(<string> PDF filename)
<Annot>        = epdf.Annot(<XRef>, <Dict>, <Catalog>, <Ref>)
<Annots>       = epdf.Annots(<XRef>, <Catalog>, <Object>)
<Array>        = epdf.Array(<XRef>)
<Attribute>    = epdf.Attribute(<Type>,<Object>)| epdf.Attribute(<string>, <int>,
<Object>)
<Dict>         = epdf.Dict(<XRef>)
<Object>       = epdf.Object()
<PDFRectangle> = epdf.PDFRectangle()
```

The functions StructElement_Type, Attribute_Type and AttributeOwner_Type return a hash table {<string>,<integer>}.

Annot methods:

```
<boolean>     = <Annot>:isOK()
<Object>      = <Annot>:getAppearance()
<AnnotBorder> = <Annot>:getBorder()
<boolean>     = <Annot>:match(<Ref>)
```

AnnotBorderStyle methods:

```
<number> = <AnnotBorderStyle>:getWidth()
```

Annots methods:

```
<integer> = <Annots>:getNumAnnots()
<Annot>   = <Annots>:getAnnot(<integer>)
```

Array methods:

```
            <Array>:incRef()
            <Array>:decRef()
<integer> = <Array>:getLength()
            <Array>:add(<Object>)
<Object>  = <Array>:get(<integer>)
```

```
<Object>  = <Array>:getNF(<integer>)
<string>  = <Array>:getString(<integer>)
```

`Attribute` methods:

```
<boolean>  = <Attribute>:isOk()
<integer>  = <Attribute>:getType()
<integer>  = <Attribute>:getOwner()
<string>   = <Attribute>:getTypeName()
<string>   = <Attribute>:getOwnerName()
<Object>   = <Attribute>:getValue()
<Object>   = <Attribute>:getDefaultValue
<string>   = <Attribute>:getName()
<integer>  = <Attribute>:getRevision()
             <Attribute>:setRevision(<unsigned integer>)
<boolean>  = <Attribute>:istHidden()
             <Attribute>:setHidden(<boolean>)
<string>   = <Attribute>:getFormattedValue()
<string>   = <Attribute>:setFormattedValue(<string>)
```

`Catalog` methods:

```
<boolean>  = <Catalog>:isOK()
<integer>  = <Catalog>:getNumPages()
<Page>     = <Catalog>:getPage(<integer>)
<Ref>      = <Catalog>:getPageRef(<integer>)
<string>   = <Catalog>:getBaseURI()
<string>   = <Catalog>:readMetadata()
<Object>   = <Catalog>:getStructTreeRoot()
<integer>  = <Catalog>:findPage(<integer> object number, <integer> object gener-
ation)
<LinkDest> = <Catalog>:findDest(<string> name)
<Object>   = <Catalog>:getDests()
<integer>  = <Catalog>:numEmbeddedFiles()
<EmbFile>  = <Catalog>:embeddedFile(<integer>)
<integer>  = <Catalog>:numJS()
<string>   = <Catalog>:getJS(<integer>)
<Object>   = <Catalog>:getOutline()
<Object>   = <Catalog>:getAcroForm()
```

`EmbFile` methods:

```
<string>   = <EmbFile>:name()
<string>   = <EmbFile>:description()
<integer>  = <EmbFile>:size()
<string>   = <EmbFile>:modDate()
<string>   = <EmbFile>:createDate()
```

```
<string>   = <EmbFile>:checksum()
<string>   = <EmbFile>:mimeType()
<Object>   = <EmbFile>:streamObject()
<boolean>  = <EmbFile>:isOk()
```

Dict methods:

```
            <Dict>:incRef()
            <Dict>:decRef()
<integer> = <Dict>:getLength()
            <Dict>:add(<string>, <Object>)
            <Dict>:set(<string>, <Object>)
            <Dict>:remove(<string>)
<boolean> = <Dict>:is(<string>)
<Object>  = <Dict>:lookup(<string>)
<Object>  = <Dict>:lookupNF(<string>)
<integer> = <Dict>:lookupInt(<string>, <string>)
<string>  = <Dict>:getKey(<integer>)
<Object>  = <Dict>:getVal(<integer>)
<Object>  = <Dict>:getValNF(<integer>)
<boolean> = <Dict>:hasKey(<string>)
```

Link methods:

```
<boolean>  = <Link>:isOK()
<boolean>  = <Link>:inRect(<number>, <number>)
```

LinkDest methods:

```
<boolean>  = <LinkDest>:isOK()
<integer>  = <LinkDest>:getKind()
<string>   = <LinkDest>:getKindName()
<boolean>  = <LinkDest>:isPageRef()
<integer>  = <LinkDest>:getPageNum()
<Ref>      = <LinkDest>:getPageRef()
<number>   = <LinkDest>:getLeft()
<number>   = <LinkDest>:getBottom()
<number>   = <LinkDest>:getRight()
<number>   = <LinkDest>:getTop()
<number>   = <LinkDest>:getZoom()
<boolean>  = <LinkDest>:getChangeLeft()
<boolean>  = <LinkDest>:getChangeTop()
<boolean>  = <LinkDest>:getChangeZoom()
```

Links methods:

```
<integer>  = <Links>:getNumLinks()
<Link>     = <Links>:getLink(<integer>)
```

`Object` methods:

```
            <Object>:initBool(<boolean>)
            <Object>:initInt(<integer>)
            <Object>:initReal(<number>)
            <Object>:initString(<string>)
            <Object>:initName(<string>)
            <Object>:initNull()
            <Object>:initArray(<XRef>)
            <Object>:initDict(<XRef>)
            <Object>:initStream(<Stream>)
            <Object>:initRef(<integer> object number, <integer> object generation)
            <Object>:initCmd(<string>)
            <Object>:initError()
            <Object>:initEOF()
<Object>  = <Object>:fetch(<XRef>)
<integer> = <Object>:getType()
<string>  = <Object>:getTypeName()
<boolean> = <Object>:isBool()
<boolean> = <Object>:isInt()
<boolean> = <Object>:isReal()
<boolean> = <Object>:isNum()
<boolean> = <Object>:isString()
<boolean> = <Object>:isName()
<boolean> = <Object>:isNull()
<boolean> = <Object>:isArray()
<boolean> = <Object>:isDict()
<boolean> = <Object>:isStream()
<boolean> = <Object>:isRef()
<boolean> = <Object>:isCmd()
<boolean> = <Object>:isError()
<boolean> = <Object>:isEOF()
<boolean> = <Object>:isNone()
<boolean> = <Object>:getBool()
<integer> = <Object>:getInt()
<number>  = <Object>:getReal()
<number>  = <Object>:getNum()
<string>  = <Object>:getString()
<string>  = <Object>:getName()
<Array>   = <Object>:getArray()
<Dict>    = <Object>:getDict()
<Stream>  = <Object>:getStream()
<Ref>     = <Object>:getRef()
<integer> = <Object>:getRefNum()
<integer> = <Object>:getRefGen()
```

```
<string>  = <Object>:getCmd()
<integer> = <Object>:arrayGetLength()
          = <Object>:arrayAdd(<Object>)
<Object>  = <Object>:arrayGet(<integer>)
<Object>  = <Object>:arrayGetNF(<integer>)
<integer> = <Object>:dictGetLength(<integer>)
          = <Object>:dictAdd(<string>, <Object>)
          = <Object>:dictSet(<string>, <Object>)
<Object>  = <Object>:dictLookup(<string>)
<Object>  = <Object>:dictLookupNF(<string>)
<string>  = <Object>:dictgetKey(<integer>)
<Object>  = <Object>:dictgetVal(<integer>)
<Object>  = <Object>:dictgetValNF(<integer>)
<boolean> = <Object>:streamIs(<string>)
          = <Object>:streamReset()
<integer> = <Object>:streamGetChar()
<integer> = <Object>:streamLookChar()
<integer> = <Object>:streamGetPos()
          = <Object>:streamSetPos(<integer>)
<Dict>    = <Object>:streamGetDict()
```

Page methods:

```
<boolean>      = <Page>:isOk()
<integer>      = <Page>:getNum()
<PDFRectangle> = <Page>:getMediaBox()
<PDFRectangle> = <Page>:getCropBox()
<boolean>      = <Page>:isCropped()
<number>       = <Page>:getMediaWidth()
<number>       = <Page>:getMediaHeight()
<number>       = <Page>:getCropWidth()
<number>       = <Page>:getCropHeight()
<PDFRectangle> = <Page>:getBleedBox()
<PDFRectangle> = <Page>:getTrimBox()
<PDFRectangle> = <Page>:getArtBox()
<integer>      = <Page>:getRotate()
<string>       = <Page>:getLastModified()
<Dict>         = <Page>:getBoxColorInfo()
<Dict>         = <Page>:getGroup()
<Stream>       = <Page>:getMetadata()
<Dict>         = <Page>:getPieceInfo()
<Dict>         = <Page>:getSeparationInfo()
<Dict>         = <Page>:getResourceDict()
<Object>       = <Page>:getAnnots()
<Links>        = <Page>:getLinks(<Catalog>)
<Object>       = <Page>:getContents()
```

`PDFDoc` methods:

```
<boolean>   = <PDFDoc>:isOk()
<integer>   = <PDFDoc>:getErrorCode()
<string>    = <PDFDoc>:getErrorCodeName()
<string>    = <PDFDoc>:getFileName()
<XRef>      = <PDFDoc>:getXRef()
<Catalog>   = <PDFDoc>:getCatalog()
<number>    = <PDFDoc>:getPageMediaWidth()
<number>    = <PDFDoc>:getPageMediaHeight()
<number>    = <PDFDoc>:getPageCropWidth()
<number>    = <PDFDoc>:getPageCropHeight()
<integer>   = <PDFDoc>:getNumPages()
<string>    = <PDFDoc>:readMetadata()
<Object>    = <PDFDoc>:getStructTreeRoot()
<integer>   = <PDFDoc>:findPage(<integer> object number, <integer> object genera-
tion)
<Links>     = <PDFDoc>:getLinks(<integer>)
<LinkDest>  = <PDFDoc>:findDest(<string>)
<boolean>   = <PDFDoc>:isEncrypted()
<boolean>   = <PDFDoc>:okToPrint()
<boolean>   = <PDFDoc>:okToChange()
<boolean>   = <PDFDoc>:okToCopy()
<boolean>   = <PDFDoc>:okToAddNotes()
<boolean>   = <PDFDoc>:isLinearized()
<Object>    = <PDFDoc>:getDocInfo()
<Object>    = <PDFDoc>:getDocInfoNF()
<integer>   = <PDFDoc>:getPDFMajorVersion()
<integer>   = <PDFDoc>:getPDFMinorVersion()
```

`PDFRectangle` methods:

```
<boolean>   = <PDFRectangle>:isValid()
```

`Stream` methods:

```
<integer>   = <Stream>:getKind()
<string>    = <Stream>:getKindName()
            = <Stream>:reset()
            = <Stream>:close()
<integer>   = <Stream>:getChar()
<integer>   = <Stream>:lookChar()
<integer>   = <Stream>:getRawChar()
<integer>   = <Stream>:getUnfilteredChar()
            = <Stream>:unfilteredReset()
<integer>   = <Stream>:getPos()
<boolean>   = <Stream>:isBinary()
```

```
<Stream>    = <Stream>:getUndecodedStream()
<Dict>      = <Stream>:getDict()
```

`StructElement` methods:

```
<string>          = <StructElement>:getTypeName()
<integer>         = <StructElement>:getType()
<boolean>         = <StructElement>:isOk()
<boolean>         = <StructElement>:isBlock()
<boolean>         = <StructElement>:isInline()
<boolean>         = <StructElement>:isGrouping()
<boolean>         = <StructElement>:isContent()
<boolean>         = <StructElement>:isObjectRef()
<integer>         = <StructElement>:getMCID()
<Ref>             = <StructElement>:getObjectRef()
<Ref>             = <StructElement>:getParentRef()
<boolean>         = <StructElement>:hasPageRef()
<Ref>             = <StructElement>:getPageRef()
<StructTreeRoot> = <StructElement>:getStructTreeRoot()
<string>          = <StructElement>:getID()
<string>          = <StructElement>:getLanguage()
<integer>         = <StructElement>:getRevision()
                    <StructElement>:setRevision(<unsigned integer>)
<string>          = <StructElement>:getTitle()
<string>          = <StructElement>:getExpandedAbbr()
<integer>         = <StructElement>:getNumChildren()
<StructElement>  = <StructElement>:getChild()
                  = <StructElement>:appendChild<StructElement>)
<integer>         = <StructElement>:getNumAttributes()
<Attribute>       = <StructElement>:geAttribute(<integer>)
<string>          = <StructElement>:appendAttribute(<Attribute>)
<Attribute>       = <StructElement>:findAttribute(<Attribute::Type>,boolean,At-
tribute::Owner)
<string>          = <StructElement>:getAltText()
<string>          = <StructElement>:getActualText()
<string>          = <StructElement>:getText(<boolean>)
<table>           = <StructElement>:getTextSpans()
```

`StructTreeRoot` methods:

```
<StructElement> = <StructTreeRoot>:findParentElement
<PDFDoc>         = <StructTreeRoot>:getDoc
<Dict>           = <StructTreeRoot>:getRoleMap
<Dict>           = <StructTreeRoot>:getClassMap
<integer>        = <StructTreeRoot>:getNumChildren
<StructElement> = <StructTreeRoot>:getChild
                   <StructTreeRoot>:appendChild
```

```
<StructElement> = <StructTreeRoot>:findParentElement
```

`TextSpan` han only one method:

```
<string> = <TestSpan>:getText()
```

`XRef` methods:

```
<boolean>   = <XRef>:isOk()
<integer>   = <XRef>:getErrorCode()
<boolean>   = <XRef>:isEncrypted()
<boolean>   = <XRef>:okToPrint()
<boolean>   = <XRef>:okToPrintHighRes()
<boolean>   = <XRef>:okToChange()
<boolean>   = <XRef>:okToCopy()
<boolean>   = <XRef>:okToAddNotes()
<boolean>   = <XRef>:okToFillForm()
<boolean>   = <XRef>:okToAccessibility()
<boolean>   = <XRef>:okToAssemble()
<Object>    = <XRef>:getCatalog()
<Object>    = <XRef>:fetch(<integer> object number, <integer> object generation)
<Object>    = <XRef>:getDocInfo()
<Object>    = <XRef>:getDocInfoNF()
<integer>   = <XRef>:getNumObjects()
<integer>   = <XRef>:getRootNum()
<integer>   = <XRef>:getRootGen()
<integer>   = <XRef>:getSize()
<Object>    = <XRef>:getTrailerDict()
```

There is an experimental function `epdf.openMemStream` that takes three arguments:

`stream`	this is a (in low level LUA speak) light userdata object, i.e. a pointer to a sequence of bytes
`length`	this is the length of the stream in bytes
`name`	this is a unique identifier that us used for hashing the stream, so that mulltiple doesn't use more memory

Instead of a light userdata stream you can also pass a LUA string, in which case the given length is (at most) the string length.

The returned object can be used in the `img` library instead of a filename. Both the memory stream and it's use in the image library is experimental and can change. In case you wonder where this can be used: when you use the swiglib library for graphic magick, it can return such a userdata object. This permits conversion in memory and passing the result directly to the backend. This might save some runtime in one-pass workflows. This feature is currently not meant for production.

7.3 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 `fontloader` library that will be discussed in the next section.

7.3.1 Loading a TFM file

The behavior documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

```
<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 4.

7.3.2 Loading a VF file

The behavior documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

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

The meaning of the number `s` and the format of the returned table are similar to the ones in the `read_tfm()` function.

7.3.3 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 4 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)
```

Note that at the moment, each access to the `font.fonts` or call to `font.getfont` creates a lua table for the whole font. This process can be quite slow. In a later version of LUATEX, this interface will change (it will start using userdata objects instead of actual tables).

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

7.3.4 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).

7.3.5 Defining a font directly

You can define your own font into `font.fonts` by calling this function:

```
<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 4.

7.3.6 Projected next font id

```
<number> i = font.nextid()
```

This returns the font id number that would be returned by a `font.define` call if it was executed at this spot in the code flow. This is useful for virtual fonts that need to reference themselves.

7.3.7 Font id

```
<number> i = font.id(<string> csname)
```

This returns the font id associated with `csname` string, or -1 if `csname` is not defined.

7.3.8 Currently active font

```
<number> i = font.current()
font.current(<number> i)
```

This gets or sets the currently used font number.

7.3.9 Maximum font id

```
<number> i = font.max()
```

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

7.3.10 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.

7.4 The `fontloader` library

7.4.1 Getting quick information on a font

```
<table> info = fontloader.info(<string> 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 some 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
units_per_em	number	1000 for POSTSCRIPT-based fonts, usually 2048 for TRUETYPE
pfminfo	table	(see section 7.4.5.1.6)

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.

7.4.2 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 `fontloader` library, the simplest way to get that information is thus:

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

```
end

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

The main function call is

```
<userdata> f, <table> w = fontloader.open(<string> filename)
<userdata> f, <table> w = fontloader.open(<string> filename, <string> fontname)
```

The first return value is a userdata representation of the font. The second return value is a table containing any warnings and errors reported by fontloader while opening the font. In normal typesetting, you would probably ignore the second argument, but it can be useful for debugging purposes.

For TRUETYPE collections (when filename ends in 'ttc') and DFONT collections, you have to use a second string argument to specify which font you want from the collection. Use the `fontname` strings that are returned by `fontloader.info` for that.

To turn the font into a table, `fontloader.to_table` is used on the font returned by `fontloader.open`.

```
<table> f = fontloader.to_table(<userdata> font)
```

This table cannot be used directly by LUATEX and should be turned into another one as described in chapter 4. Do not forget to store the `fontname` value in the `psname` field of the metrics table to be returned to LUATEX, otherwise the font inclusion backend will not be able to find the correct font in the collection.

See section 7.4.5 for details on the userdata object returned by `fontloader.open()` and the layout of the `metrics` table returned by `fontloader.to_table()`.

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.

A loaded font is discarded with:

```
fontloader.close(<userdata> font)
```

7.4.3 Applying a 'feature file'

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

```
<table> errors = fontloader.apply_featurefile(<userdata> font, <string> file-
name)
```

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.

If the function fails, the return value is a table containing any errors reported by fontloader while applying the feature file. On success, `nil` is returned.

7.4.4 Applying an ‘AFM file’

You can apply an ‘AFM file’ to a loaded font:

```
<table> errors = fontloader.apply_afmfile(<userdata> font, <string> filename)
```

An AFM file is a textual representation of (some of) the meta information in a Type1 font. See

ftp://ftp.math.utah.edu/u/ma/hohn/linux/postscript/5004.AFM_Spec.pdf

for more information about AFM files.

Note: If you `fontloader.open()` a Type1 file named `font.pfb`, the library will automatically search for and apply `font.afm` if it exists in the same directory as the file `font.pfb`. In that case, there is no need for an explicit call to `apply_afmfile()`.

If the function fails, the return value is a table containing any errors reported by fontloader while applying the AFM file. On success, `nil` is returned.

7.4.5 Fontloader font tables

As mentioned earlier, the return value of `fontloader.open()` is a userdata object. One way to have access to the actual metrics is to call `fontloader.to_table()` on this object, returning the table structure that is explained in the following subsections.

However, it turns out that the result from `fontloader.to_table()` sometimes needs very large amounts of memory (depending on the font’s complexity and size) so it is possible to access the userdata object directly.

- All top-level keys that would be returned by `to_table()` can also be accessed directly.
-
- The top-level key ‘glyphs’ returns a *virtual* array that allows indices from `f.glyphmin` to (`f.glyphmax`).
- The items in that virtual array (the actual glyphs) are themselves also userdata objects, and each has accessors for all of the keys explained in the section ‘Glyph items’ below.

The top-level key 'subfonts' returns an *actual* array of userdata objects, one for each of the subfonts (or nil, if there are no subfonts).

A short example may be helpful. This code generates a printout of all the glyph names in the font `PunkNova.kern.otf`:

```
local f = fontloader.open('PunkNova.kern.otf')
print (f.fontname)
local i = 0
if f.glyphcnt > 0 then
    for i=f.glyphmin,f.glyphmax do
       local g = f.glyphs[i]
       if g then
          print(g.name)
       end
       i = i + 1
    end
end
fontloader.close(f)
```

In this case, the LuaTeX memory requirement stays below 100MB on the test computer, while the internal stucture generated by `to_table()` needs more than 2GB of memory (the font itself is 6.9MB in disk size).

Only the top-level font, the subfont table entries, and the glyphs are virtual objects, everything else still produces normal lua values and tables.

If you want to know the valid fields in a font or glyph structure, call the `fields` function on an object of a particular type (either glyph or font):

```
<table> fields = fontloader.fields(<userdata> font)
<table> fields = fontloader.fields(<userdata> font_glyph)
```

For instance:

```
local fields = fontloader.fields(f)
local fields = fontloader.fields(f.glyphs[0])
```

7.4.5.1 Table types

7.4.5.1.1 Top-level

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

key	type	explanation
table_version	number	indicates the metrics version (currently 0.3)
fontname	string	PostScript font name

fullname	string	official (human-oriented) 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	
uniqueid	number	
glyphs	array	
glyphcnt	number	number of included glyphs
glyphmax	number	maximum used index the glyphs array
glyphmin	number	minimum used index the glyphs array
hasvmetrics	number	
onlybitmaps	number	
serifcheck	number	
isserif	number	
issans	number	
encodingchanged	number	
strokedfont	number	
use_typo_metrics	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	
commments	string	
fontlog	string	
cvt_names	string	
anchor_classes	table	
ttf_tables	table	
ttf_tab_saved	table	
kerns	table	

vkerns	table
texdata	table
lookups	table
gpos	table
gsub	table
mm	table
chosenname	string
macstyle	number
fondname	string
fontstyle_id	number
fontstyle_name	table
strokewidth	float
mark_classes	table
creationtime	number
modificationtime	number
os2_version	number
sfd_version	number
math	table
validation_state	table
horiz_base	table
vert_base	table
extrema_bound	number

7.4.5.1.2 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
unicode	number	unicode code point, or -1
boundingbox	array	array of four numbers, see note below
width	number	only for horizontal fonts
vwidth	number	only for vertical fonts
tsidebearing	number	only for vertical ttf/otf fonts, and only if nonzero
lsidebearing	number	only if nonzero and not equal to boundingbox[1]
class	string	one of "none", "base", "ligature", "mark", "component" (if not present, the glyph class is 'automatic')
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
comment	string	only if set
tex_height	number	only if set

tex_depth	number	only if set
italic_correction	number	only if set
top_accent	number	only if set
is_extended_shape	number	only if this character is part of a math extension list
altuni	table	alternate UNICODE items
vert_variants	table	
horiz_variants	table	
mathkern	table	

On boundingbox: The boundingbox information for TRUETYPE fonts and TRUETYPE-based OTF fonts is read directly from the font file. POSTSCRIPT-based fonts do not have this information, so the boundingbox of traditional POSTSCRIPT fonts is generated by interpreting the actual bezier curves to find the exact boundingbox. This can be a slow process, so the boundingboxes of POSTSCRIPT-based OTF fonts (and raw CFF fonts) are calculated using an approximation of the glyph shape based on the actual glyph points only, instead of taking the whole curve into account. This means that glyphs that have missing points at extrema will have a too-tight boundingbox, but the processing is so much faster that in our opinion the tradeoff is worth it.

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 item in the lookups table explained above
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 a 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 } }
        }
    }
```

Note: The baselig table can be sparse!

7.4.5.1.3 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 [name?]
is_simplechinese	number	only if nonzero
low_page	number	
high_page	number	
iconv_name	string	
iso_2022_escape	string	

7.4.5.1.4 private table

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

7.4.5.1.5 cidinfo table

key	type	explanation
registry	string	
ordering	string	
supplement	number	
version	number	

7.4.5.1.6 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	
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	
codepages	table	A two-number array of encoded code pages
unicoderages	table	A four-number array of encoded unicode ranges
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.

7.4.5.1.7 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
wwsfamily
wwssubfamily

7.4.5.1.8 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

7.4.5.1.9 gpos table

The gpos table has one array entry for each lookup. (The gpos_ prefix is somewhat redundant.)

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
`mark_class`	string

The features subtable items of gpos have:

key	type	explanation
`tag`	string	
`scripts`	table	

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 or array	associated lookup(s)
`offsets`	array of numbers	

7.4.5.1.10 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`

7.4.5.1.11 ttf_tables and ttf_tab_saved tables

key	type	explanation
`tag`	string	
`len`	number	

maxlen	number
data	number

7.4.5.1.12 mm table

key	type	explanation
axes	table	array of axis names
instance_count	number	
positions	table	array of instance positions (#axes * instances)
defweights	table	array of default weights for instances
cdv	string	
ndv	string	
axismaps	table	

The axismaps:

key	type	explanation
blends	table	an array of blend points
designs	table	an array of design values
min	number	
def	number	
max	number	

7.4.5.1.13 mark_classes table

The keys in this table are mark class names, and the values are a space-separated string of glyph names in this class.

7.4.5.1.14 math table

```
ScriptPercentScaleDown
ScriptScriptPercentScaleDown
DelimitedSubFormulaMinHeight
DisplayOperatorMinHeight
MathLeading
AxisHeight
AccentBaseHeight
FlattenedAccentBaseHeight
SubscriptShiftDown
SubscriptTopMax
SubscriptBaselineDropMin
SuperscriptShiftUp
SuperscriptShiftUpCramped
SuperscriptBottomMin
SuperscriptBaselineDropMax
```

```
SubSuperscriptGapMin
SuperscriptBottomMaxWithSubscript
SpaceAfterScript
UpperLimitGapMin
UpperLimitBaselineRiseMin
LowerLimitGapMin
LowerLimitBaselineDropMin
StackTopShiftUp
StackTopDisplayStyleShiftUp
StackBottomShiftDown
StackBottomDisplayStyleShiftDown
StackGapMin
StackDisplayStyleGapMin
StretchStackTopShiftUp
StretchStackBottomShiftDown
StretchStackGapAboveMin
StretchStackGapBelowMin
FractionNumeratorShiftUp
FractionNumeratorDisplayStyleShiftUp
FractionDenominatorShiftDown
FractionDenominatorDisplayStyleShiftDown
FractionNumeratorGapMin
FractionNumeratorDisplayStyleGapMin
FractionRuleThickness
FractionDenominatorGapMin
FractionDenominatorDisplayStyleGapMin
SkewedFractionHorizontalGap
SkewedFractionVerticalGap
OverbarVerticalGap
OverbarRuleThickness
OverbarExtraAscender
UnderbarVerticalGap
UnderbarRuleThickness
UnderbarExtraDescender
RadicalVerticalGap
RadicalDisplayStyleVerticalGap
RadicalRuleThickness
RadicalExtraAscender
RadicalKernBeforeDegree
RadicalKernAfterDegree
RadicalDegreeBottomRaisePercent
MinConnectorOverlap
FractionDelimiterSize
FractionDelimiterDisplayStyleSize
```

7.4.5.1.15 validation_state table

key	explanation
bad_ps_fontname	
bad_glyph_table	
bad_cff_table	
bad_metrics_table	
bad_cmap_table	
bad_bitmaps_table	
bad_gx_table	
bad_ot_table	
bad_os2_version	
bad_sfnt_header	

7.4.5.1.16 horiz_base and vert_base table

key	type	explanation
tags	table	an array of script list tags
scripts	table	

The `scripts` subtable:

key	type	explanation
baseline	table	
default_baseline	number	
lang	table	

The `lang` subtable:

key	type	explanation
tag	string	a script tag
ascent	number	
descent	number	
features	table	

The `features` points to an array of tables with the same layout except that in those nested tables, the tag represents a language.

7.4.5.1.17 altuni table

An array of alternate UNICODE values. Inside that array are hashes with:

key	type	explanation
unicode	number	this glyph is also used for this unicode
variant	number	the alternative is driven by this unicode selector

7.4.5.1.18 vert_variants and horiz_variants table

key	type	explanation
variants	string	
italic_correction	number	
parts	table	

The parts table is an array of smaller tables:

key	type	explanation
component	string	
extender	number	
start	number	
end	number	
advance	number	

7.4.5.1.19 mathkern table

key	type	explanation
top_right	table	
bottom_right	table	
top_left	table	
bottom_left	table	

Each of the subtables is an array of small hashes with two keys:

key	type	explanation
height	number	
kern	number	

7.4.5.1.20 kerns table

Substructure is identical to the per-glyph subtable.

7.4.5.1.21 vkerns table

Substructure is identical to the per-glyph subtable.

7.4.5.1.22 texdata table

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

7.4.5.1.23 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	string	
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
glyphs	array	only if the parent's format is glyphs
class	array	only if the parent's format is class
coverage	array	only if the parent's format is coverage
reversecoverage	array	only if the parent's format is reversecoverage

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	

7.5 The `img` library

The `img` library can be used as an alternative to `\pdfximage` and `\pdfrefximage`, and the associated 'satellite' commands like `\pdfximagebbox`. Image objects can also be used within virtual fonts via the `image` command listed in section 4.2.

7.5.1 `img.new`

```
<image> var = img.new()
<image> var = img.new(<table> image_spec)
```

This function creates a userdata object of type 'image'. The `image_spec` argument is optional. If it is given, it must be a table, and that table must contain a `filename` key. A number of other keys can also be useful, these are explained below.

You can either say

```
a = img.new()
```

followed by

```
a.filename = "foo.png"
```

or you can put the file name (and some or all of the other keys) into a table directly, like so:

```
a = img.new({filename='foo.pdf', page=1})
```

The generated `<image>` userdata object allows access to a set of user-specified values as well as a set of values that are normally filled in and updated automatically by LuaTeX itself. Some of those are derived from the actual image file, others are updated to reflect the PDF output status of the object.

There is one required user-specified field: the file name (`filename`). It can optionally be augmented by the requested image dimensions (`width`, `depth`, `height`), user-specified image attributes (`attr`), the requested PDF page identifier (`page`), the requested boundingbox (`pagebox`) for PDF inclusion, the requested color space object (`colorspace`).

The function `img.new` does not access the actual image file, it just creates the `<image>` userdata object and initializes some memory structures. The `<image>` object and its internal structures are automatically garbage collected.

Once the image is scanned, all the values in the `<image>` except `width`, `height` and `depth`, become frozen, and you cannot change them any more.

7.5.2 `img.keys`

```
<table> keys = img.keys()
```

This function returns a list of all the possible `image_spec` keys, both user-supplied and automatic ones.

field name	type	description
attr	string	the image attributes for LuaTeX
bbox	table	table with 4 boundingbox dimensions `llx`, `lly`, `urx`, and `ury` overruling the `pagebox` entry
colordepth	number	the number of bits used by the color space
colorspace	number	the color space object number
depth	number	the image depth for LuaTeX (in scaled points)
filename	string	the image file name
filepath	string	the full (expanded) file name of the image
height	number	the image height for LuaTeX (in scaled points)
imagetype	string	one of `pdf`, `png`, `jpg`, `jp2`, `jbig2`, or `nil`
index	number	the PDF image name suffix
objnum	number	the PDF image object number
page	??	the identifier for the requested image page (type is number or string, default is the number 1)
pagebox	string	the requested bounding box, one of `none`, `media`, `crop`, `bleed`, `trim`, `art`
pages	number	the total number of available pages
rotation	number	the image rotation from included PDF file, in multiples of 90 deg.
stream	string	the raw stream data for an `/Xobject /Form` object
transform	number	the image transform, integer number 0..7
width	number	the image width for LuaTeX (in scaled points)
xres	number	the horizontal natural image resolution (in DPI)
xsize	number	the natural image width
yres	number	the vertical natural image resolution (in DPI)
ysize	number	the natural image height

A running (undefined) dimension in `width`, `height`, or `depth` is represented as `nil` in Lua, so if you want to load an image at its 'natural' size, you do not have to specify any of those three fields.

The `stream` parameter allows to fabricate an `/XObject /Form` object from a string giving the stream contents, e.g., for a filled rectangle:

```
a.stream = "0 0 20 10 re f"
```

When writing the image, an `/Xobject /Form` object is created, like with embedded PDF file writing. The object is written out only once. The `stream` key requires that also the `bbox` table is given. The `stream` key conflicts with the `filename` key. The `transform` key works as usual also with `stream`.

The `bbox` key needs a table with four boundingbox values, e.g.:

```
a.bbox = {"30bp", 0, "225bp", "200bp"}
```

This replaces and overrules any given `pagebox` value; with given `bbox` the box dimensions coming with an embedded PDF file are ignored. The `xsize` and `ysize` dimensions are set accordingly, when the image is scaled. The `bbox` parameter is ignored for non-PDF images.

The `transform` allows to mirror and rotate the image in steps of 90 deg. The default value 0 gives an unmirrored, unrotated image. Values 1 − 3 give counterclockwise rotation by 90, 180, or 270 degrees, whereas with values 4 − 7 the image is first mirrored and then rotated counterclockwise by 90, 180, or 270 degrees. The transform operation gives the same visual result as if you would externally preprocess the image by a graphics tool and then use it by LuaTeX. If a PDF file to be embedded already contains a /Rotate specification, the rotation result is the combination of the /Rotate rotation followed by the transform operation.

7.5.3 img.scan

```
<image> var = img.scan(<image> var)
<image> var = img.scan(<table> image_spec)
```

When you say img.scan(a) for a new image, the file is scanned, and variables such as xsize, ysize, image type, number of pages, and the resolution are extracted. Each of the width, height, depth fields are set up according to the image dimensions, if they were not given an explicit value already. An image file will never be scanned more than once for a given image variable. With all subsequent img.scan(a) calls only the dimensions are again set up (if they have been changed by the user in the meantime).

For ease of use, you can do right-away a

```
<image> a = img.scan ({ filename = "foo.png" })
```

without a prior img.new.

Nothing is written yet at this point, so you can do a=img.scan, retrieve the available info like image width and height, and then throw away a again by saying a=nil. In that case no image object will be reserved in the PDF, and the used memory will be cleaned up automatically.

7.5.4 img.copy

```
<image> var = img.copy(<image> var)
<image> var = img.copy(<table> image_spec)
```

If you say a = b, then both variables point to the same <image> object. if you want to write out an image with different sizes, you can do a b=img.copy(a).

Afterwards, a and b still reference the same actual image dictionary, but the dimensions for b can now be changed from their initial values that were just copies from a.

7.5.5 img.write

```
<image> var = img.write(<image> var)
<image> var = img.write(<table> image_spec)
```

By img.write(a) a PDF object number is allocated, and a whatsit node of subtype pdf_refximage is generated and put into the output list. By this the image a is placed into the page stream, and the image file is written out into an image stream object after the shipping of the current page is finished.

Again you can do a terse call like

```
img.write ({ filename = "foo.png" })
```

The <image> variable is returned in case you want it for later processing.

7.5.6 img.immediatewrite

```
<image> var = img.immediatewrite(<image> var)
<image> var = img.immediatewrite(<table> image_spec)
```

By img.immediatewrite(a) a PDF object number is allocated, and the image file for image a is written out immediately into the PDF file as an image stream object (like with \immediate\pdfximage). The object number of the image stream dictionary is then available by the objnum key. No pdf_refximage whatsit node is generated. You will need an img.write(a) or img.node(a) call to let the image appear on the page, or reference it by another trick; else you will have a dangling image object in the PDF file.

Also here you can do a terse call like

```
a = img.immediatewrite ({ filename = "foo.png" })
```

The <image> variable is returned and you will most likely need it.

7.5.7 img.node

```
<node> n = img.node(<image> var)
<node> n = img.node(<table> image_spec)
```

This function allocates a PDF object number and returns a whatsit node of subtype pdf_refximage, filled with the image parameters width, height, depth, and objnum. Also here you can do a terse call like:

```
n = img.node ({ filename = "foo.png" })
```

This example outputs an image:

```
node.write(img.node{filename="foo.png"})
```

7.5.8 img.types

```
<table> types = img.types()
```

This function returns a list with the supported image file type names, currently these are pdf, png, jpg, jp2 (JPEG 2000), and jbig2.

7.5.9 img.boxes

```
<table> boxes = img.boxes()
```

This function returns a list with the supported PDF page box names, currently these are `media`, `crop`, `bleed`, `trim`, and `art` (all in lowercase letters).

7.6 The kpse library

This library provides two separate, but nearly identical interfaces to the KPATHSEA file search functionality: there is a 'normal' procedural interface that shares its kpathsea instance with LuaTeX itself, and an object oriented interface that is completely on its own.

7.6.1 `kpse.set_program_name` and `kpse.new`

Before the search library can be used at all, its database has to be initialized. There are three possibilities, two of which belong to the procedural interface.

First, when LuaTeX is used to typeset documents, this initialization happens automatically and the KPATHSEA executable and program names are set to `luatex` (that is, unless explicitly prohibited by the user's startup script. See section 2.1 for more details).

Second, in TeXLua mode, the initialization has to be done explicitly via the `kpse.set_program_name` function, which 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.

Third, if you prefer the object oriented interface, you have to call a different function. It has the same arguments, but it returns a userdata variable.

```
local kpathsea = kpse.new(<string> name)
local kpathsea = kpse.new(<string> name, <string> progname)
```

Apart from these two functions, the calling conventions of the interfaces are identical. Depending on the chosen interface, you either call `kpse.find_file()` or `kpathsea:find_file()`, with identical arguments and return vales.

7.6.2 `find_file`

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

The default type is `tex`. Note: this is different from KPSEWHICH, which tries to deduce the file type itself from looking at the supplied extension.

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`.

7.6.3 lookup

A more powerful (but slower) generic method for finding files is also available. It returns a string for each found file.

```
<string> f, ... = kpse.lookup(<string> filename, <table> options)
```

The options match commandline arguments from `kpsewhich`:

key	type	description
debug	number	set debugging flags for this lookup
format	string	use specific file type (see list above)
dpi	number	use this resolution for this lookup; default 600
path	string	search in the given path
all	boolean	output all matches, not just the first
mustexist	boolean	search the disk as well as ls-R if necessary
mktexpk	boolean	disable/enable mktexpk generation for this lookup
mktextex	boolean	disable/enable mktextex generation for this lookup
mktexmf	boolean	disable/enable mktexmf generation for this lookup
mktextfm	boolean	disable/enable mktextfm generation for this lookup
subdir	string or table	only output matches whose directory part ends with the given string(s)

7.6.4 init_prog

Extra initialization for programs that need to generate bitmap fonts.

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

7.6.5 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.

7.6.6 expand_path

Like kpsewhich's `-expand-path`:

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

7.6.7 expand_var

Like kpsewhich's -expand-var:

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

7.6.8 expand_braces

Like kpsewhich's -expand-braces:

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

7.6.9 show_path

Like kpsewhich's -show-path:

```
<string> r = kpse.show_path(<string> ftype)
```

7.6.10 var_value

Like kpsewhich's -var-value:

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

7.6.11 version

Returns the kpathsea version string.

```
<string> r = kpse.version()
```

7.7 The lang library

This library provides the interface to LuaTeX's structure representing a language, and the associated functions.

```
<language> l = lang.new()
<language> l = lang.new(<number> id)
```

This function creates a new userdata object. An object of type `<language>` is the first argument to most of the other functions in the `lang` library. These functions can also be used as if they were object methods, using the colon syntax.

Without an argument, the next available internal id number will be assigned to this object. With argument, an object will be created that links to the internal language with that id number.

```
<number> n = lang.id(<language> l)
```

returns the internal `\language` id number this object refers to.

```
<string> n = lang.hyphenation(<language> l)
lang.hyphenation(<language> l, <string> n)
```

Either returns the current hyphenation exceptions for this language, or adds new ones. The syntax of the string is explained in section 3.3.

```
lang.clear_hyphenation(<language> l)
```

Clears the exception dictionary for this language.

```
<string> n = lang.clean(<string> o)
```

Creates a hyphenation key from the supplied hyphenation value. The syntax of the argument string is explained in section 3.3. This function is useful if you want to do something else based on the words in a dictionary file, like spell-checking.

```
<string> n = lang.patterns(<language> l)
lang.patterns(<language> l, <string> n)
```

Adds additional patterns for this language object, or returns the current set. The syntax of this string is explained in section 3.3.

```
lang.clear_patterns(<language> l)
```

Clears the pattern dictionary for this language.

```
<number> n = lang.prehyphenchar(<language> l)
lang.prehyphenchar(<language> l, <number> n)
```

Gets or sets the 'pre-break' hyphen character for implicit hyphenation in this language (initially the hyphen, decimal 45).

```
<number> n = lang.posthyphenchar(<language> l)
lang.posthyphenchar(<language> l, <number> n)
```

Gets or sets the 'post-break' hyphen character for implicit hyphenation in this language (initially null, decimal 0, indicating emptiness).

```
<number> n = lang.preexhyphenchar(<language> l)
lang.preexhyphenchar(<language> l, <number> n)
```

Gets or sets the 'pre-break' hyphen character for explicit hyphenation in this language (initially null, decimal 0, indicating emptiness).

```
<number> n = lang.postexhyphenchar(<language> l)
lang.postexhyphenchar(<language> l, <number> n)
```

Gets or sets the 'post-break' hyphen character for explicit hyphenation in this language (initially null, decimal 0, indicating emptiness).

```
<boolean> success = lang.hyphenate(<node> head)
<boolean> success = lang.hyphenate(<node> head, <node> tail)
```

Inserts hyphenation points (discretionary nodes) in a node list. If `tail` is given as argument, processing stops on that node. Currently, `success` is always true if `head` (and `tail`, if specified) are proper nodes, regardless of possible other errors.

Hyphenation works only on 'characters', a special subtype of all the glyph nodes with the node subtype having the value `1`. Glyph modes with different subtypes are not processed. See section 3.1 for more details.

7.8 The `lua` library

This library contains one read-only item:

```
<string> s = lua.version
```

This returns the LUA version identifier string. The value is currently Lua 5.2.

7.8.1 LUA bytecode registers

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

```
lua.bytecode[<number> n] = <function> f
lua.bytecode[<number> 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.

Note: The function must not contain any upvalues. Currently, functions containing upvalues can be stored (and their upvalues are set to `nil`), but this is an artifact of the current LUA implementation and thus subject to change.

The associated function calls are

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

Note: Since a LUA file loaded using `loadfile(filename)` is essentially an anonymous function, a complete file can be stored in a bytecode register like this:

```
lua.bytecode[n] = loadfile(filename)
```

Now all definitions (functions, variables) contained in the file can be created by executing this bytecode register:

```
lua.bytecode[n]()
```

Note that the path of the file is stored in the LUA bytecode to be used in stack backtraces and therefore dumped into the format file if the above code is used in INITEX. If it contains private information, i.e.

the user name, this information is then contained in the format file as well. This should be kept in mind when preloading files into a bytecode register in INITEX.

7.8.2 LUA chunk name registers

There is an array of 65536 (0–65535) potential chunk names for use with the `\directlua` and `\latelua` primitives.

```
lua.name[<number> n] = <string> s
<string> s = lua.name[<number> n]
```

If you want to unset a lua name, you can assign `nil` to it.

7.9 The `mplib` library

The METAPOST library interface registers itself in the table `mplib`. It is based on MPLIB version 1.999.

7.9.1 `mplib.new`

To create a new METAPOST instance, call

```
<mpinstance> mp = mplib.new({...})
```

This creates the `mp` instance object. The argument hash can have a number of different fields, as follows:

name	**type**	**description**	**default**
`error_line`	number	error line width	79
`print_line`	number	line length in ps output	100
`random_seed`	number	the initial random seed	variable
`interaction`	string	the interaction mode, one of `batch`, `nonstop`, `scroll`, `errorstop`	`errorstop`
`job_name`	string	`--jobname`	`mpout`
`find_file`	function	a function to find files	only local files

The `find_file` function should be of this form:

```
<string> found = finder (<string> name, <string> mode, <string> type)
```

with:

name	**the requested file**
`mode`	the file mode: `r` or `w`
`type`	the kind of file, one of: `mp`, `tfm`, `map`, `pfb`, `enc`

Return either the full pathname of the found file, or `nil` if the file cannot be found.

Note that the new version of MPLIB no longer uses binary mem files, so the way to preload a set of macros is simply to start off with an `input` command in the first `mp:execute()` call.

7.9.2 `mp:statistics`

You can request statistics with:

```
<table> stats = mp:statistics()
```

This function returns the vital statistics for an MPLIB instance. There are four fields, giving the maximum number of used items in each of four allocated object classes:

`main_memory`	number	memory size
`hash_size`	number	hash size
`param_size`	number	simultaneous macro parameters
`max_in_open`	number	input file nesting levels

Note that in the new version of MPLIB, this is informational only. The objects are all allocated dynamically, so there is no chance of running out of space unless the available system memory is exhausted.

7.9.3 `mp:execute`

You can ask the METAPOST interpreter to run a chunk of code by calling

```
<table> rettable = mp:execute('metapost language chunk')
```

for various bits of METAPOST language input. Be sure to check the `rettable.status` (see below) because when a fatal METAPOST error occurs the MPLIB instance will become unusable thereafter.

Generally speaking, it is best to keep your chunks small, but beware that all chunks have to obey proper syntax, like each of them is a small file. For instance, you cannot split a single statement over multiple chunks.

In contrast with the normal standalone `mpost` command, there is *no* implied 'input' at the start of the first chunk.

7.9.4 `mp:finish`

```
<table> rettable = mp:finish()
```

If for some reason you want to stop using an MPLIB instance while processing is not yet actually done, you can call `mp:finish`. Eventually, used memory will be freed and open files will be closed by the LUA garbage collector, but an explicit `mp:finish` is the only way to capture the final part of the output streams.

7.9.5 Result table

The return value of `mp:execute` and `mp:finish` is a table with a few possible keys (only `status` is always guaranteed to be present).

log	string	output to the 'log' stream
term	string	output to the 'term' stream
error	string	output to the 'error' stream (only used for 'out of memory')
status	number	the return value: `0` =good, `1` =warning, `2` =errors, `3` =fatal error
fig	table	an array of generated figures (if any)

When `status` equals 3, you should stop using this MPLIB instance immediately, it is no longer capable of processing input.

If it is present, each of the entries in the `fig` array is a userdata representing a figure object, and each of those has a number of object methods you can call:

boundingbox	function	returns the bounding box, as an array of 4 values
postscript	function	returns a string that is the ps output of the `fig`. this function accepts two optional integer arguments for specifying the values of `prologues` (first argument) and `procset` (second argument)
svg	function	returns a string that is the svg output of the `fig`. This function accepts an optional integer argument for specifying the value of `prologues`
objects	function	returns the actual array of graphic objects in this `fig`
copy_objects	function	returns a deep copy of the array of graphic objects in this `fig`
filename	function	the filename this `fig`'s POSTSCRIPT output would have written to in stand-alone mode
width	function	the `fontcharwd` value
height	function	the `fontcharht` value
depth	function	the `fontchardp` value
italcorr	function	the `fontcharit` value
charcode	function	the (rounded) `charcode` value

Note: you can call `fig:objects()` only once for any one `fig` object!

When the boundingbox represents a 'negated rectangle', i.e. when the first set of coordinates is larger than the second set, the picture is empty.

Graphical objects come in various types that each has a different list of accessible values. The types are: `fill`, `outline`, `text`, `start_clip`, `stop_clip`, `start_bounds`, `stop_bounds`, `special`.

There is helper function (`mplib.fields(obj)`) to get the list of accessible values for a particular object, but you can just as easily use the tables given below.

All graphical objects have a field `type` that gives the object type as a string value; it is not explicit mentioned in the following tables. In the following, `number`s are POSTSCRIPT points represented as a floating point number, unless stated otherwise. Field values that are of type `table` are explained in the next section.

7.9.5.1 fill

path	table	the list of knots
htap	table	the list of knots for the reversed trajectory
pen	table	knots of the pen
color	table	the object's color
linejoin	number	line join style (bare number)
miterlimit	number	miterlimit
prescript	string	the prescript text
postscript	string	the postscript text

The entries `htap` and `pen` are optional.

There is helper function (`mplib.pen_info(obj)`) that returns a table containing a bunch of vital characteristics of the used pen (all values are floats):

width	number	width of the pen
sx	number	x scale
rx	number	xy multiplier
ry	number	yx multiplier
sy	number	y scale
tx	number	x offset
ty	number	y offset

7.9.5.2 outline

path	table	the list of knots
pen	table	knots of the pen
color	table	the object's color
linejoin	number	line join style (bare number)
miterlimit	number	miterlimit
linecap	number	line cap style (bare number)
dash	table	representation of a dash list
prescript	string	the prescript text
postscript	string	the postscript text

The entry `dash` is optional.

7.9.5.3 text

text	string	the text
font	string	font tfm name
dsize	number	font size
color	table	the object's color
width	number	
height	number	
depth	number	

transform	table	a text transformation
prescript	string	the prescript text
postscript	string	the postscript text

7.9.5.4 special

prescript	string	special text

7.9.5.5 start_bounds, start_clip

path	table	the list of knots

7.9.5.6 stop_bounds, stop_clip

Here are no fields available.

7.9.6 Subsidiary table formats

7.9.6.1 Paths and pens

Paths and pens (that are really just a special type of paths as far as MPLIB is concerned) are represented by an array where each entry is a table that represents a knot.

`left_type`	string	when present: endpoint, but usually absent
`right_type`	string	like `left_type`
`x_coord`	number	X coordinate of this knot
`y_coord`	number	Y coordinate of this knot
`left_x`	number	X coordinate of the precontrol point of this knot
`left_y`	number	Y coordinate of the precontrol point of this knot
`right_x`	number	X coordinate of the postcontrol point of this knot
`right_y`	number	Y coordinate of the postcontrol point of this knot

There is one special case: pens that are (possibly transformed) ellipses have an extra string-valued key `type` with value `elliptical` besides the array part containing the knot list.

7.9.6.2 Colors

A color is an integer array with 0, 1, 3 or 4 values:

0	marking only	no values
1	greyscale	one value in the range $(0, 1)$, 'black' is 0
3	RGB	three values in the range $(0, 1)$, 'black' is $0, 0, 0$
4	CMYK	four values in the range $(0, 1)$, 'black' is $0, 0, 0, 1$

If the color model of the internal object was `uninitialized`, then it was initialized to the values representing 'black' in the colorspace `defaultcolormodel` that was in effect at the time of the `shipout`.

7.9.6.3 Transforms

Each transform is a six-item array.

1	number	represents x
2	number	represents y
3	number	represents xx
4	number	represents yx
5	number	represents xy
6	number	represents yy

Note that the translation (index 1 and 2) comes first. This differs from the ordering in PostScript, where the translation comes last.

7.9.6.4 Dashes

Each `dash` is two-item hash, using the same model as PostScript for the representation of the dashlist. `dashes` is an array of 'on' and 'off', values, and `offset` is the phase of the pattern.

dashes	hash	an array of on-off numbers
offset	number	the starting offset value

7.9.7 Character size information

These functions find the size of a glyph in a defined font. The `fontname` is the same name as the argument to `infont`; the `char` is a glyph id in the range 0 to 255; the returned `w` is in AFM units.

7.9.7.1 `mp:char_width`

```
<number> w = mp:char_width(<string> fontname, <number> char)
```

7.9.7.2 `mp:char_height`

```
<number> w = mp:char_height(<string> fontname, <number> char)
```

7.9.7.3 `mp:char_depth`

```
<number> w = mp:char_depth(<string> fontname, <number> char)
```

7.10 The node library

The `node` library contains functions that facilitate dealing with (lists of) nodes and their values. They allow you to create, alter, 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.

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 freeing 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.

There are statistics available with regards to the allocated node memory, which can be handy for tracing.

7.10.1 Node handling functions

7.10.1.1 node.is_node

```
<boolean> t = node.is_node(<any> item)
```

This function returns true if the argument is a userdata object of type <node>.

7.10.1.2 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.

7.10.1.3 node.whatsits

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

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

7.10.1.4 node.id

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

This converts a single type name to its internal numeric representation.

7.10.1.5 node.subtype

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

This converts a single whatsit name to its internal numeric representation (`subtype`).

7.10.1.6 node.type

```
<string> type = node.type(<any> n)
```

In the argument is a number, then this function converts an internal numeric representation to an external string representation. Otherwise, it will return the string `node` if the object represents a node, and `nil` otherwise.

7.10.1.7 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.

7.10.1.8 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.

7.10.1.9 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 a node on the TeX level.

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

7.10.1.10 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.

7.10.1.11 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.

7.10.1.12 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.

7.10.1.13 node.copy_list

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

Creates a deep copy of the node list that starts at `n`. If `m` is also given, the copy stops just before node `m`.

Note that you cannot copy attribute lists this way, specialized functions for dealing with attribute lists will be provided later but are not there yet. However, there is normally no need to copy attribute lists as when you do assignments to the `attr` field or make changes to specific attributes, the needed copying and freeing takes place automatically.

7.10.1.14 node.next

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

Returns the node following this node, or `nil` if there is no such node.

7.10.1.15 node.prev

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

Returns the node preceding this node, or `nil` if there is no such node.

7.10.1.16 node.current_attr

```
<node> m = node.current_attr()
```

Returns the currently active list of attributes, if there is one.

The intended usage of `current_attr` is as follows:

```
local x1 = node.new("glyph")
x1.attr = node.current_attr()
local x2 = node.new("glyph")
x2.attr = node.current_attr()
```

or:

```
local x1 = node.new("glyph")
local x2 = node.new("glyph")
local ca = node.current_attr()
x1.attr = ca
x2.attr = ca
```

The attribute lists are ref counted and the assignment takes care of incrementing the refcount. You cannot expect the value `ca` to be valid any more when you assign attributes (using `tex.setattribute`) or when control has been passed back to TEX.

Note: this function is somewhat experimental, and it returns the *actual* attribute list, not a copy thereof. Therefore, changing any of the attributes in the list will change these values for all nodes that have the current attribute list assigned to them.

7.10.1.17 node.hpack

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

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 its 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. The second return value is the badness of the generated box.

Caveat: at this moment, there can be unexpected side-effects to this function, like updating some of the \marks and \inserts. Also note that the content of h is the original node list n: if you call node.free(h) you will also free the node list itself, unless you explicitly set the list field to nil beforehand. And in a similar way, calling node.free(n) will invalidate h as well!

7.10.1.18 node.vpack

```
<node> h, <number> b = node.vpack(<node> n)
<node> h, <number> b = node.vpack(<node> n, <number> w, <string> info)
<node> h, <number> b = node.vpack(<node> n, <number> w, <string> info, <string>
dir)
```

This function creates a new vlist by packaging the list that begins at node n into a vertical box. With only a single argument, this box is created using the natural height of its components. In the three argument form, info must be either additional or exactly, and w is the additional (\vbox spread) or exact (\vbox to) height to be used.

The second return value is the badness of the generated box.

See the description of node.hpack() for a few memory allocation caveats.

7.10.1.19 node.dimensions

```
<number> w, <number> h, <number> d  = node.dimensions(<node> n)
<number> w, <number> h, <number> d  = node.dimensions(<node> n, <string> dir)
<number> w, <number> h, <number> d  = node.dimensions(<node> n, <node> t)
<number> w, <number> h, <number> d  = node.dimensions(<node> n, <node> t, <string>
dir)
```

This function calculates the natural in-line dimensions of the node list starting at node n and terminating just before node t (or the end of the list, if there is no second argument). The return values are scaled points. An alternative format that starts with glue parameters as the first three arguments is also possible:

```
<number> w, <number> h, <number> d  =
  node.dimensions(<number> glue_set, <number> glue_sign,
                  <number> glue_order, <node> n)
<number> w, <number> h, <number> d  =
  node.dimensions(<number> glue_set, <number> glue_sign,
                  <number> glue_order, <node> n, <string> dir)
<number> w, <number> h, <number> d  =
  node.dimensions(<number> glue_set, <number> glue_sign,
                  <number> glue_order, <node> n, <node> t)
<number> w, <number> h, <number> d  =
  node.dimensions(<number> glue_set, <number> glue_sign,
```

```
                    <number> glue_order, <node> n, <node> t, <string> dir)
```

This calling method takes glue settings into account and is especially useful for finding the actual width of a sublist of nodes that are already boxed, for example in code like this, which prints the width of the space inbetween the a and b as it would be if \box0 was used as-is:

```
\setbox0 = \hbox to 20pt {a b}

\directlua{print (node.dimensions(
    tex.box[0].glue_set,
    tex.box[0].glue_sign,
    tex.box[0].glue_order,
    tex.box[0].head.next,
    node.tail(tex.box[0].head)
)) }
```

7.10.1.20 node.mlist_to_hlist

```
<node> h = node.mlist_to_hlist(<node> n,
             <string> display_type, <boolean> penalties)
```

This runs the internal mlist to hlist conversion, converting the math list in n into the horizontal list h. The interface is exactly the same as for the callback mlist_to_hlist.

7.10.1.21 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.

7.10.1.22 node.tail

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

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

7.10.1.23 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.

7.10.1.24 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 a 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.

7.10.1.25 node.traverse

```
<node> t = node.traverse(<node> n)
```

This is a lua iterator that loops over the node list that starts at `n`. Typically code looks like this:

```
for n in node.traverse(head) do
   ...
end
```

is functionally equivalent to:

```
do
  local n
  local function f (head,var)
    local t
    if var == nil then
       t = head
    else
       t = var.next
    end
    return t
  end
  while true do
    n = f (head, n)
    if n == nil then break end
    ...
  end
end
```

It should be clear from the definition of the function `f` that even though it is possible to add or remove nodes from the node list while traversing, you have to take great care to make sure all the `next` (and `prev`) pointers remain valid.

If the above is unclear to you, see the section 'For Statement' in the LUA Reference Manual.

7.10.1.26 node.traverse_id

```
<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.

See the previous section for details. The change is in the local function `f`, which now does an extra while loop checking against the upvalue `id`:

```
local function f(head,var)
  local t
  if var == nil then
     t = head
  else
     t = var.next
  end
  while not t.id == id do
     t = t.next
  end
  return t
end
```

7.10.1.27 node.end_of_math

`<node> t = node.end_of_math(<node> start)`

Looks for and returns the next `math_node` following the `start`. If the given node is a math endnode this helper return that node, else it follows the list and return the next math endnote. If no such node is found nil is returned.

7.10.1.28 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 following the `current` in the calling argument, and is only passed back as a convenience (or `nil`, if there is no such node). The returned `head` is more important, because if the function is called with `current` equal to `head`, it will be changed.

7.10.1.29 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 node `new`, set up to be part of the list (with correct `next` field). If `head` is initially `nil`, it will become `new`.

7.10.1.30 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 node `new`, set up to be part of the list (with correct `next` field). If `head` is initially `nil`, it will become `new`.

7.10.1.31 node.first_glyph

```
<node> n = node.first_glyph(<node> n)
<node> n = node.first_glyph(<node> n, <node> m)
```

Returns the first node in the list starting at `n` that is a glyph node with a subtype indicating it is a glyph, or `nil`. If `m` is given, processing stops at (but including) that node, otherwise processing stops at the end of the list.

7.10.1.32 node.ligaturing

```
<node> h, <node> t, <boolean> success = node.ligaturing(<node> n)
<node> h, <node> t, <boolean> success = node.ligaturing(<node> n, <node> m)
```

Apply TeX-style ligaturing to the specified nodelist. The tail node `m` is optional. The two returned nodes `h` and `t` are the new head and tail (both `n` and `m` can change into a new ligature).

7.10.1.33 node.kerning

```
<node> h, <node> t, <boolean> success = node.kerning(<node> n)
<node> h, <node> t, <boolean> success = node.kerning(<node> n, <node> m)
```

Apply TeX-style kerning to the specified nodelist. The tail node `m` is optional. The two returned nodes `h` and `t` are the head and tail (either one of these can be an inserted kern node, because special kernings with word boundaries are possible).

7.10.1.34 node.unprotect_glyphs

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

Subtracts 256 from all glyph node subtypes. This and the next function are helpers to convert from `characters` to `glyphs` during node processing.

7.10.1.35 node.protect_glyphs

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

Adds 256 to all glyph node subtypes in the node list starting at n, except that if the value is 1, it adds only 255. The special handling of 1 means that characters will become glyphs after subtraction of 256.

7.10.1.36 node.last_node

```
<node> n = node.last_node()
```

This function pops the last node from TEX's 'current list'. It returns that node, or nil if the current list is empty.

7.10.1.37 node.write

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

This is an experimental function that will append a node list to TEX's 'current list' The node list is not deep-copied! There is no error checking either!

7.10.1.38 node.protrusion_skippable

```
<boolean> skippable = node.protrusion_skippable(<node> n)
```

Returns true if, for the purpose of line boundary discovery when character protrusion is active, this node can be skipped.

7.10.2 Attribute handling

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

7.10.2.1 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.

7.10.2.2 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. *[needs explanation]*

7.10.2.3 `node.unset_attribute`

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

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`.

7.11 The pdf library

This contains variables and functions that are related to the PDF backend.

7.11.1 `pdf.mapfile`, `pdf.mapline`

```
pdf.mapfile(<string> map file)
pdf.mapline(<string> map line)
```

These two functions can be used to replace primitives `\pdfmapfile` and `\pdfmapline` from PDFTEX. They expect a string as only parameter and have no return value.

The also functions replace the former variables `pdf.pdfmapfile` and `pdf.pdfmapline`.

7.11.2 `pdf.catalog`, `pdf.info`,`pdf.names`, `pdf.trailer`

These variables offer a read-write interface to the corresponding PDFTEX token lists. The value types are strings and they are written out to the PDF file directly after the PDFTEX token registers.

The preferred interface is now `pdf.setcatalog`, `pdf.setinfo` `pdf.setnames` and `pdf.settrailer` for setting these properties and `pdf.getcatalog`, `pdf.getinfo` `pdf.getnames` and `pdf.gettrailer` for querying them,

The corresponding 'pdf' parameter names `pdf.pdfcatalog`, `pdf.pdfinfo`, `pdf.pdfnames`, and `pdf.pdftrailer` are not available.

7.11.3 pdf.<set/get>pageattributes, pdf.<set/get>pageresources, pdf.<set/get>p sattributes

These variables offer a read-write interface to related token lists. The value types are strings. The variables have no interaction with the corresponding PDFTEX token registers `\pdfpageattr`, `\pdfpageresources`, and `\pdfpagesattr`. They are written out to the PDF file directly after the PDFTEX token registers.

The preferred interface is now `pdf.setpageattributes`, `pdf.setpagesattributes` and `pdf.setpageresources` for setting these properties and `pdf.getpageattributes`, `pdf.getpageattributes` and `pdf.getpageresources` for querying them.

7.11.4 `pdf.h`, `pdf.v`

These are the `h` and `v` values that define the current location on the output page, measured from its lower left corner. The values can be queried using scaled points as units.

```
local h = pdf.h
local v = pdf.v
```

7.11.5 `pdf.getpos`, `pdf.gethpos`, `pdf.getvpos`

These are the function variants of `pdf.h` and `pdf.v`. Sometimes using a function is preferred over a key so this saves wrapping. Also, these functions are faster then the key based access, as `h` and `v` keys are not real variables but looked up using a metatable call. The `getpos` function returns two values, the other return one.

```
local h, v = pdf.getpos()
```

7.11.6 `pdf.hasmatrix`, `pdf.getmatrix`

The current matrix transformation is available via the `getmatrix` command, which returns 6 values: `sx`, `rx`, `ry`, `sy`, `tx`, and `ty`. The `hasmatrix` function returns `true` when a matrix is applied.

```
if pdf.hasmatrix() then
    local sx, rx, ry, sy, tx, ty = pdf.getmatrix()
    -- do something useful or not
end
```

7.11.7 `pdf.print`

A print function to write stuff to the PDF document that can be used from within a `\latelua` argument. This function is not to be used inside `\directlua` unless you know *exactly* what you are doing.

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

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

7.11.8 `pdf.immediateobj`

This function creates a PDF object and immediately writes it to the PDF file. It is modelled after PDFTEX's `\immediate` `\pdfobj` primitives. All function variants return the object number of the newly generated object.

```
<number> n = pdf.immediateobj(<string> objtext)
```

```
<number> n = pdf.immediateobj("file", <string> filename)
<number> n = pdf.immediateobj("stream", <string> streamtext, <string> attrtext)
<number> n = pdf.immediateobj("streamfile", <string> filename, <string> attr-
text)
```

The first version puts the `objtext` raw into an object. Only the object wrapper is automatically generated, but any internal structure (like `<< >>` dictionary markers) needs to provided by the user. The second version with keyword `"file"` as 1st argument puts the contents of the file with name `filename` raw into the object. The third version with keyword `"stream"` creates a stream object and puts the `streamtext` raw into the stream. The stream length is automatically calculated. The optional `attrtext` goes into the dictionary of that object. The fourth version with keyword `"streamfile"` does the same as the 3rd one, it just reads the stream data raw from a file.

An optional first argument can be given to make the function use a previously reserved PDF object.

```
<number> n = pdf.immediateobj(<integer> n, <string> objtext)
<number> n = pdf.immediateobj(<integer> n, "file", <string> filename)
<number> n = pdf.immediateobj(<integer> n, "stream", <string> streamtext, <string>
attrtext)
<number> n = pdf.immediateobj(<integer> n, "streamfile", <string> filename,
<string> attrtext)
```

7.11.9 `pdf.obj`

This function creates a PDF object, which is written to the PDF file only when referenced, e.g., by `pdf.refobj()`.

All function variants return the object number of the newly generated object, and there are two separate calling modes.

The first mode is modelled after PDFTEX's `\pdfobj` primitive.

```
<number> n = pdf.obj(<string> objtext)
<number> n = pdf.obj("file", <string> filename)
<number> n = pdf.obj("stream", <string> streamtext, <string> attrtext)
<number> n = pdf.obj("streamfile", <string> filename, <string> attrtext)
```

An optional first argument can be given to make the function use a previously reserved PDF object.

```
<number> n = pdf.obj(<integer> n, <string> objtext)
<number> n = pdf.obj(<integer> n, "file", <string> filename)
<number> n = pdf.obj(<integer> n, "stream", <string> streamtext, <string> attr-
text)
<number> n = pdf.obj(<integer> n, "streamfile", <string> filename, <string> at-
trtext)
```

The second mode accepts a single argument table with key–value pairs.

```
<number> n = pdf.obj {
```

```
    type           = <string>,
    immmediate     = <boolean>,
    objnum         = <number>,
    attr           = <string>,
    compresslevel  = <number>,
    objcompression = <boolean>,
    file           = <string>,
    string         = <string>
}
```

The `type` field can have the values `raw` and `stream`, this field is required, the others are optional (within constraints).

Note: this mode makes `pdf.obj` look more flexible than it actually is: the constraints from the separate parameter version still apply, so for example you can't have both `string` and `file` at the same time.

7.11.10 pdf.refobj

This function, the LUA version of the `\pdfrefobj` primitive, references an object by its object number, so that the object will be written out.

```
pdf.refobj(<integer> n)
```

This function works in both the `\directlua` and `\latelua` environment. Inside `\directlua` a new whatsit node 'pdf_refobj' is created, which will be marked for flushing during page output and the object is then written directly after the page, when also the resources objects are written out. Inside `\latelua` the object will be marked for flushing.

This function has no return values.

7.11.11 pdf.reserveobj

This function creates an empty PDF object and returns its number.

```
<number> n = pdf.reserveobj()
<number> n = pdf.reserveobj("annot")
```

7.11.12 pdf.registerannot

This function adds an object number to the `/Annots` array for the current page without doing anything else. This function can only be used from within `\latelua`.

```
pdf.registerannot (<number> objnum)
```

7.12 The pdfscanner library

The `pdfscanner` library allows interpretation of PDF content streams and `/ToUnicode` (cmap) streams. You can get those streams from the `epdf` library, as explained in an earlier section. There is only a single top-level function in this library:

`pdfscanner.scan (<Object> stream, <table> operatortable, <table> info)`

The first argument, `stream`, should be either a PDF stream object, or a PDF array of PDF stream objects (those options comprise the possible return values of `<Page>:getContents()` and `<Object>:getStream()` in the `epdf` library).

The second argument, `operatortable`, should be a Lua table where the keys are PDF operator name strings and the values are Lua functions (defined by you) that are used to process those operators. The functions are called whenever the scanner finds one of these PDF operators in the content stream(s). The functions are called with two arguments: the `scanner` object itself, and the `info` table that was passed are the third argument to `pdfscanner.scan`.

Internally, `pdfscanner.scan` loops over the PDF operators in the stream(s), collecting operands on an internal stack until it finds a PDF operator. If that PDF operator's name exists in `operatortable`, then the associated function is executed. After the function has run (or when there is no function to execute) the internal operand stack is cleared in preparation for the next operator, and processing continues.

The `scanner` argument to the processing functions is needed because it offers various methods to get the actual operands from the internal operand stack.

A simple example of processing a PDF's document stream could look like this:

```
function Do (scanner, info)
   local val       = scanner:pop()
   local name      = val[2] -- val[1] == 'name'
   local resources = info.resources
   local xobject   = resources:lookup("XObject"):getDict():lookup(name)
   print (info.space ..'Use XObject '.. name)
   if xobject and xobject:isStream() then
      local dict = xobject:getStream():getDict()
      if dict then
        local name = dict:lookup("Subtype")
        if name:getName() == "Form" then
          local newinfo =  {
            space = info.space .. "  " ,
            resources = dict:lookup("Resources"):getDict()
          }
          pdfscanner.scan(xobject, operatortable, newinfo)
        end
      end
   end
end

operatortable = { Do = Do }

doc       = epdf.open(arg[1])
pagenum   = 1
```

```
while pagenum <= doc:getNumPages() do
   local page = doc:getCatalog():getPage(pagenum)
   local info = {
     space     = "  " ,
     resources = page:getResourceDict()
   }
   print('Page ' .. pagenum)
   pdfscanner.scan(page:getContents(), operatortable, info)
   pagenum = pagenum + 1
end
```

This example iterates over all the actual content in the PDF, and prints out the found XObject names. While the code demonstrates quite some of the `epdf` functions, let's focus on the type `pdfscanner` specific code instead.

From the bottom up, the line

```
pdfscanner.scan(page:getContents(), operatortable, info)
```

runs the scanner with the PDF page's top-level content.

The third argument, `info`, contains two entries: `space` is used to indent the printed output, and `resources` is needed so that embedded `XForms` can find their own content.

The second argument, `operatortable` defines a processing function for a single PDF operator, `Do`.

The function `Do` prints the name of the current XObject, and then starts a new scanner for that object's content stream, under the condition that the XObject is in fact a `/Form`. That nested scanner is called with new `info` argument with an updated `space` value so that the indentation of the output nicely nests, and with an new `resources` field to help the next iteration down to properly process any other, embedded XObjects.

Of course, this is not a very useful example in practise, but for the purpose of demonstrating `pdfscanner`, it is just long enough. It makes use of only one `scanner` method: `scanner:pop()`. That function pops the top operand of the internal stack, and returns a lua table where the object at index one is a string representing the type of the operand, and object two is its value.

The list of possible operand types and associated lua value types is:

`integer`	<number>
`real`	<number>
`boolean`	<boolean>
`name`	<string>
`operator`	<string>
`string`	<string>
`array`	<table>
`dict`	<table>

In case of `integer` or `real`, the value is always a LUA (floating point) number.

In case of `name`, the leading slash is always stripped.

In case of `string`, please bear in mind that PDF actually supports different types of strings (with different encodings) in different parts of the PDF document, so may need to reencode some of the results; `pdfscanner` always outputs the byte stream without reencoding anything. `pdfscanner` does not differentiate between literal strings and hexidecimal strings (the hexadecimal values are decoded), and it treats the stream data for inline images as a string that is the single operand for `EI`.

In case of `array`, the table content is a list of `pop` return values.

In case of `dict`, the table keys are PDF name strings and the values are `pop` return values.

There are few more methods defined that you can ask `scanner`:

`pop`	as explained above
`popNumber`	return only the value of a `real` or `integer`
`popName`	return only the value of a `name`
`popString`	return only the value of a `string`
`popArray`	return only the value of a `array`
`popDict`	return only the value of a `dict`
`popBool`	return only the value of a `boolean`
`done`	abort further processing of this `scan()` call

The `popXXX` are convenience functions, and come in handy when you know the type of the operands beforehand (which you usually do, in PDF). For example, the `Do` function could have used `local name = scanner:popName()` instead, because the single operand to the `Do` operator is always a PDF name object.

The `done` function allows you to abort processing of a stream once you have learned everything you want to learn. This comes in handy while parsing `/ToUnicode`, because there usually is trailing garbage that you are not interested in. Without `done`, processing only end at the end of the stream, possibly wasting CPU cycles.

7.13 The `status` library

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

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

The keys in the table are the known items, the value is the current value. Almost all of the values in `status` are fetched through a metatable at run-time whenever they are accessed, so you cannot use `pairs` on `status`, but you *can* use `pairs` on `info`, of course. If you do not need the full list, you can also ask for a single item by using its 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
node_mem_usage	a string giving insight into currently used nodes
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
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
`luastate_bytes`	number of bytes in use by LUA interpreters
`output_active`	`true` if the `\output` routine is active
`callbacks`	total number of executed callbacks so far
`indirect_callbacks`	number of those that were themselves a result of other callbacks (e.g. file readers)
`luatex_svn`	the luatex repository id
`luatex_version`	the luatex version number
`luatex_revision`	the luatex revision string
`ini_version`	`true` if this is an INITEX run

7.14 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 front-ends 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.
- Some special others, see the list below

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`).

7.14.1 Internal parameter values

For all the parameters in this section, it is possible to access them directly using their names as index in the `tex` table, or by using one of the functions `tex.get()` and `tex.set()`.

The exact parameters and return values differ depending on the actual parameter, and so does whether `tex.set` has any effect. For the parameters that *can* be set, it is possible to use `global` as the first argument to `tex.set`; this makes the assignment global instead of local.

```
tex.set (<string> n, ...)
tex.set ('global', <string> n, ...)
... = tex.get (<string> n)
```

7.14.1.1 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.pdfadjustspacing
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.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.parshape	tex.spacefactor
tex.insertpenalties	tex.prevgraf	

7.14.1.2 Dimension parameters

The dimension parameters accept LUA numbers (signifying scaled points) or strings (with included dimension). The result is always a number in scaled points.

Read-write:

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

Read-only:

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

7.14.1.3 Direction parameters

The direction parameters are read-only and return a LUA string.

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

7.14.1.4 Glue parameters

The glue parameters accept and return a userdata object that represents a glue_spec node.

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

7.14.1.5 Muglue parameters

All muglue parameters are to be used read-only and return a LUA string.

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

7.14.1.6 Tokenlist parameters

The tokenlist parameters accept and return LUA strings. LUA strings are converted to and from token lists using `\the \toks` style expansion: all category codes are either space (10) or other (12). It follows that assigning to some of these, like 'tex.output', is actually useless, but it feels bad to make exceptions in view of a coming extension that will accept full-blown token strings.

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

7.14.2 Convert commands

All 'convert' commands are read-only and return a LUA string. The supported commands at this moment are:

`tex.eTeXVersion`	`tex.pdffontobjnum(number)`
`tex.eTeXrevision`	`tex.pdffontsize(number)`
`tex.formatname`	`tex.uniformdeviate(number)`
`tex.jobname`	`tex.number(number)`
`tex.luatexbanner`	`tex.romannumeral(number)`
`tex.luatexrevision`	`tex.pdfpageref(number)`
`tex.pdfnormaldeviate`	`tex.pdfxformname(number)`
`tex.fontname(number)`	`tex.fontidentifier(number)`
`tex.pdffontname(number)`	

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, as well as the ones that require only a simple numeric value.

The special (lua-only) case of `tex.fontidentifier` returns the `csname` string that matches a font id number (if there is one).

if these are really needed in a macro package.

7.14.3 Last item commands

All 'last item' commands are read-only and return a number.

The supported commands at this moment are:

tex.lastpenalty	tex.pdflastximagepages	tex.eTeXversion
tex.lastkern	tex.pdflastannot	tex.currentgrouplevel
tex.lastskip	tex.pdflastxpos	tex.currentgrouptype
tex.lastnodetype	tex.pdflastypos	tex.currentiflevel
tex.inputlineno	tex.pdfrandomseed	tex.currentiftype
tex.pdflastobj	tex.pdflastlink	tex.currentifbranch
tex.pdflastxform	tex.luatexversion	tex.pdflastximagecol-ordepth
tex.pdflastximage	tex.eTeXminorversion	

7.14.4 Attribute, count, dimension, skip and token registers

TEX's attributes (`\attribute`), counters (`\count`), dimensions (`\dimen`), skips (`\skip`) and token (`\toks`) registers can be accessed and written to using two times five virtual sub-tables of the `tex` table:

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

It is possible to use the names of relevant `\attributedef`, `\countdef`, `\dimendef`, `\skipdef`, 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 `\skipdef`, 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 and from token lists using `\the \toks` style expansion: all category codes are either space (10) or other (12).

The skip registers accept and return `glue_spec` userdata node objects (see the description of the node interface elsewhere in this manual).

As an alternative to array addressing, there are also accessor functions defined for all cases, for example, here is the set of possibilities for `\skip` registers:

```
tex.setskip (<number> n, <node> s)
tex.setskip (<string> s, <node> s)
tex.setskip ('global',<number> n, <node> s)
tex.setskip ('global',<string> s, <node> s)
<node> s = tex.getskip (<number> n)
```

```
<node> s = tex.getskip (<string> s)
```

In the function-based interface, it is possible to define values globally by using the string `global` as the first function argument.

7.14.5 Character code registers

TEX's character code tables (`\lccode`, `\uccode`, `\sfcode`, `\catcode`, `\mathcode`, `\delcode`) can be accessed and written to using six virtual subtables of the `tex` table

`tex.lccode`	`tex.sfcode`	`tex.mathcode`
`tex.uccode`	`tex.catcode`	`tex.delcode`

The function call interfaces are roughly as above, but there are a few twists. `sfcode`s are the simple ones:

```
tex.setsfcode (<number> n, <number> s)
tex.setsfcode ('global', <number> n, <number> s)
<number> s = tex.getsfcode (<number> n)
```

The function call interface for `lccode` and `uccode` additionally allows you to set the associated sibling at the same time:

```
tex.setlccode (['global'], <number> n, <number> lc)
tex.setlccode (['global'], <number> n, <number> lc, <number> uc)
<number> lc = tex.getlccode (<number> n)
tex.setuccode (['global'], <number> n, <number> uc)
tex.setuccode (['global'], <number> n, <number> uc, <number> lc)
<number> uc = tex.getuccode (<number> n)
```

The function call interface for `catcode` also allows you to specify a category table to use on assignment or on query (default in both cases is the current one):

```
tex.setcatcode (['global'], <number> n, <number> c)
tex.setcatcode (['global'], <number> cattable, <number> n, <number> c)
<number> lc = tex.getcatcode (<number> n)
<number> lc = tex.getcatcode (<number> cattable, <number> n)
```

The interfaces for `delcode` and `mathcode` use small array tables to set and retrieve values:

```
tex.setmathcode (['global'], <number> n, <table> mval )
<table> mval = tex.getmathcode (<number> n)
tex.setdelcode (['global'], <number> n, <table> dval )
<table> dval = tex.getdelcode (<number> n)
```

Where the table for `mathcode` is an array of 3 numbers, like this:

```
{<number> mathclass, <number> family, <number> character}
```

And the table for delcode is an array with 4 numbers, like this:

```
{<number> small_fam, <number> small_char, <number> large_fam, <number> large_char}
```

Normally, the third and fourth values in a delimiter code assignment will be zero according to \Udelcode usage, but the returned table can have values there (if the delimiter code was set using \delcode, for example). Unset delcode's can be recognized because dval[1] is −1.

7.14.6 Box registers

It is 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)
tex.setbox(<string> cs, <node> s)
tex.setbox('global', <number> n, <node> s)
tex.setbox('global', <string> cs, <node> s)
<node> n = tex.getbox(<number> n)
<node> n = tex.getbox(<string> cs)
```

for function-based access. In the function-based interface, it is possible to define values globally by using the string global as the first function argument.

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])
```

7.14.7 Math parameters

It is possible to set and query the internal math parameters using:

```
tex.setmath(<string> n, <string> t, <number> n)
tex.setmath('global', <string> n, <string> t, <number> n)
<number> n = tex.getmath(<string> n, <string> t)
```

As before an optional first parameter global indicates a global assignment.

The first string is the parameter name minus the leading 'Umath', and the second string is the style name minus the trailing 'style'.

Just to be complete, the values for the math parameter name are:

quad	axis	operatorsize	
overbarkern	overbarrule	overbarvgap	
underbarkern	underbarrule	underbarvgap	
radicalkern	radicalrule	radicalvgap	
radicaldegreebefore	radicaldegreeafter	radicaldegreeraise	
stackvgap	stacknumup	stackdenomdown	
fractionrule	fractionnumvgap	fractionnumup	
fractiondenomvgap	fractiondenomdown	fractiondelsize	
limitabovevgap	limitabovebgap	limitabovekern	
limitbelowvgap	limitbelowbgap	limitbelowkern	
underdelimitervgap	underdelimiterbgap		
overdelimitervgap	overdelimiterbgap		
subshiftdrop	supshiftdrop	subshiftdown	
subsupshiftdown	subtopmax	supshiftup	
supbottommin	supsubbottommax	subsupvgap	
spaceafterscript	connectoroverlapmin		
ordordspacing	ordopspacing	ordbinspacing	ordrelspacing
ordopenspacing	ordclosespacing	ordpunctspacing	ordinnerspacing
opordspacing	opopspacing	opbinspacing	oprelspacing
opopenspacing	opclosespacing	oppunctspacing	opinnerspacing
binordspacing	binopspacing	binbinspacing	binrelspacing
binopenspacing	binclosespacing	binpunctspacing	bininnerspacing
relordspacing	relopspacing	relbinspacing	relrelspacing
relopenspacing	relclosespacing	relpunctspacing	relinnerspacing
openordspacing	openopspacing	openbinspacing	openrelspacing
openopenspacing	openclosespacing	openpunctspacing	openinnerspacing
closeordspacing	closeopspacing	closebinspacing	closerelspacing
closeopenspacing	closeclosespacing	closepunctspacing	closeinnerspacing
punctordspacing	punctopspacing	punctbinspacing	punctrelspacing
punctopenspacing	punctclosespacing	punctpunctspacing	punctinnerspacing
innerordspacing	inneropspacing	innerbinspacing	innerrelspacing
inneropenspacing	innerclosespacing	innerpunctspacing	innerinnerspacing

The values for the style parameter name are:

display	crampeddisplay
text	crampedtext
script	crampedscript
scriptscript	crampedscriptscript

7.14.8 Special list heads

The virtual table `tex.lists` contains the set of internal registers that keep track of building page lists.

field	description
page_ins_head	circular list of pending insertions

contrib_head	the recent contributions
page_head	the current page content
hold_head	used for held-over items for next page
adjust_head	head of the current \vadjust list
pre_adjust_head	head of the current \vadjust pre list

7.14.9 Semantic nest levels

The virtual table tex.nest contains the currently active semantic nesting state. It has two main parts: a zero-based array of userdata for the semantic nest itself, and the numerical value tex.nest.ptr, which gives the highest available index. Neither the array items in tex.nest[] nor tex.nest.ptr can be assigned to (as this would confuse the typesetting engine beyond repair), but you can assign to the individual values inside the array items, e.g. tex.nest[tex.nest.ptr].prevdepth.

tex.nest[tex.nest.ptr] is the current nest state, tex.nest[0] the outermost (main vertical list) level.

The known fields are:

key	**type**	**modes**	**explanation**
mode	number	all	The current mode. This is a number representing the main mode at this level: 0 =no mode (this happens during \write) 1 =vertical, 127 =horizontal, 253 =display math. -1 =internal vertical, -127 =restricted horizontal, -253 =inline math.
modeline	number	all	source input line where this mode was entered in, negative inside the output routine
head	node	all	the head of the current list
tail	node	all	the tail of the current list
prevgraf	number	vmode	number of lines in the previous paragraph
prevdepth	number	vmode	depth of the previous paragraph (equal to \pdfignoreddimen when it is to be ignored)
spacefactor	number	hmode	the current space factor
dirs	node	hmode	used for temporary storage by the line break algorithm
noad	node	mmode	used for temporary storage of a pending fraction numerator, for \over etc.
delimptr	node	mmode	used for temporary storage of the previous math delimiter, for \middle
mathdir	boolean	mmode	true when during math processing the \mathdir is not the same as the surrounding \textdir
mathstyle	number	mmode	the current \mathstyle

7.14.10 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.

The result of using these functions from inside callbacks is undefined at the moment.

7.14.10.1 `tex.print`

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

Each string argument is treated by TeX as a separate input line. If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process).

The optional parameter can be used to print the strings using the catcode regime defined by `\catcodetable` `n`. If `n` is -1, the currently active catcode regime is used. If `n` is -2, the resulting catcodes are the result of `\the` `\toks`: all category codes are 12 (other) except for the space character, that has category code 10 (space). Otherwise, 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.

7.14.10.2 `tex.sprint`

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

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.
  Note that this does not prevent TeX itself from eating spaces as result of interpreting the line. For example, in

  ```
  before\directlua{tex.sprint("\\relax")tex.sprint(" inbetween")}after
  ```

the space before `inbetween` will be gobbled as a result of the 'normal' scanning of `\relax`.

If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process).

The optional argument sets the catcode regime, as with `tex.print()`.

7.14.10.3 `tex.tprint`

```
tex.tprint({<number> n, <string> s, ...}, {...})
```

This function is basically a shortcut for repeated calls to `tex.sprint(<number> n, <string> s, ...)`, once for each of the supplied argument tables.

7.14.10.4 `tex.write`

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

Each string argument is treated by TEX as a special kind of input line that makes it 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.

If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process).

7.14.11 Helper functions

7.14.11.1 `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.

7.14.11.2 `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 multiplied number(s) are of range, it generates 'number to big' error(s) as well.

Note: the precision of the output of this function will depend on your computer's architecture and operating system, so use with care! An interface to LuaTeX's internal, 100% portable scale function will be added at a later date.

7.14.11.3 `tex.sp`

```
<number> n = tex.sp(<number> o)
<number> n = tex.sp(<string> s)
```

Converts the number `o` or a string `s` that represents an explicit dimension into an integer number of scaled points.

For parsing the string, the same scanning and conversion rules are used that LuaTeX would use if it was scanning a dimension specifier in its TeX-like input language (this includes generating errors for bad values), expect for the following:

1. only explicit values are allowed, control sequences are not handled
2. infinite dimension units (`fil...`) are forbidden
3. `mu` units do not generate an error (but may not be useful either)

7.14.11.4 `tex.definefont`

```
tex.definefont(<string> csname, <number> fontid)
tex.definefont(<boolean> global, <string> csname, <number> fontid)
```

Associates `csname` with the internal font number `fontid`. The definition is global if (and only if) `global` is specified and true (the setting of `globaldefs` is not taken into account).

7.14.11.5 `tex.error`

```
tex.error(<string> s)
tex.error(<string> s, <table> help)
```

This creates an error somewhat like the combination of `\errhelp` and `\errmessage` would. During this error, deletions are disabled.

The array part of the `help` table has to contain strings, one for each line of error help.

7.14.11.6 `tex.hashtokens`

```
for i,v in pairs (tex.hashtokens()) do ... end
```

Returns a name and token table pair (see section 7.17 about token tables) iterator for every non-zero entry in the hash table. This can be useful for debugging, but note that this also reports control sequences that may be unreachable at this moment due to local redefinitions: it is strictly a dump of the hash table.

7.14.12 Functions for dealing with primitives

7.14.12.1 tex.enableprimitives

```
tex.enableprimitives(<string> prefix, <table> primitive names)
```

This function accepts a prefix string and an array of primitive names.

For each combination of 'prefix' and 'name', the `tex.enableprimitives` first verifies that 'name' is an actual primitive (it must be returned by one of the `tex.extraprimitives()` calls explained below, or part of TEX82, or `\directlua`). If it is not, `tex.enableprimitives` does nothing and skips to the next pair.

But if it is, then it will construct a csname variable by concatenating the 'prefix' and 'name', unless the 'prefix' is already the actual prefix of 'name'. In the latter case, it will discard the 'prefix', and just use 'name'.

Then it will check for the existence of the constructed csname. If the csname is currently undefined (note: that is not the same as `\relax`), it will globally define the csname to have the meaning: run code belonging to the primitive 'name'. If for some reason the csname is already defined, it does nothing and tries the next pair.

An example:

```
tex.enableprimitives('LuaTeX', {'formatname'})
```

will define `\LuaTeXformatname` with the same intrinsic meaning as the documented primitive `\formatname`, provided that the control sequences `\LuaTeXformatname` is currently undefined.

Second example:

```
tex.enableprimitives('Omega',tex.extraprimitives ('omega'))
```

will define a whole series of csnames like `\Omegatextdir`, `\Omegapardir`, etc., but it will stick with `\OmegaVersion` instead of creating the doubly-prefixed `\OmegaOmegaVersion`.

When LUATEX is run with `--ini` only the TEX82 primitives and `\directlua` are available, so no extra primitives **at all**.

If you want to have all the new functionality available using their default names, as it is now, you will have to add

```
\ifx\directlua\undefined \else
   \directlua {tex.enableprimitives('',tex.extraprimitives ())}
\fi
```

near the beginning of your format generation file. Or you can choose different prefixes for different subsets, as you see fit.

Calling some form of `tex.enableprimitives()` is highly important though, because if you do not, you will end up with a TEX82-lookalike that can run LUA code but not do much else. The defined csnames are (of course) saved in the format and will be available at runtime.

7.14.12.2 `tex.extraprimitives`

```
<table> t = tex.extraprimitives(<string> s, ...)
```

This function returns a list of the primitives that originate from the engine(s) given by the requested string value(s). The possible values and their (current) return values are:

name	values
tex	vskip write vsize \normalcontrolspace unhcopy output - / unskip unvbox boxmaxdepth muskipdef string toksdef floatingpenalty righthyphenmin voffset escapechar topmark splitfirstmark vsplit everydisplay badness xleaders textfont showlists language mathchoice topskip abovedisplayshortskip underline tracinglostchars pagefillstretch unvcopy splitbotmark finalhyphendemerits atopwithdelims pretolerance fi dp setlanguage ht nulldelimiterspace or wd pagegoal advance chardef catcode mathchar scriptscriptfont mathcode leftskip pageshrink pagefilstretch delcode fontname lastkern belowdisplayshortskip tolerance mathopen exhyphenpenalty maxdepth futurelet abovewithdelims hangindent lastskip linepenalty everyjob xspaceskip globaldefs everypar scriptfont delimiter afterassignment firstmark lineskiplimit lineskip def fam day iffalse textstyle end mag box belowdisplayskip ifx let errmessage exhyphenchar hss expandafter hfilneg the displaywidth mathsurround pagedepth looseness leaders vss ifhmode botmark ifinner displaystyle accent immediate ifmmode parshape meaning abovedisplayskip medmuskip emergencystretch rightskip mathclose hangafter hoffset aftergroup cleaders romannumeral hbadness mathbin showboxbreadth jobname vbadness patterns nonstopmode errhelp predisplaypenalty endlinechar mathinner lastbox showboxdepth postdisplaypenalty mathrel holdinginserts radical mathord pagetotal everycr adjdemerits halign defaultskewchar errorcontextlines splitmaxdepth ifcase noindent tracingmacros moveright predisplaysize tracingrestores message ifhbox deadcycles interlinepenalty mathpunct lccode noboundary displayindent nonscript everyhbox global penalty tracingcommands everymath nolimits noalign inputlineno pagestretch parskip indent dimendef widowpenalty ifvbox above spaceskip middle displaylimits pausing everyvbox iftrue moveleft mathop endcsname dimen ifcat clubpenalty splittopskip doublehyphendemerits ifdim limits ifeof insert delimitershortfall ifodd insertpenalties tracingpages vadjust tracingonline count ifnum edef char begingroup tracingparagraphs hyphenation uccode hfuzz openout leqno hyphenpenalty vcenter hfil thickmuskip maxdeadcycles mkern hbox overfullrule else hsize raise thinmuskip spacefactor input hrule left eqno parfillskip font valign dump relax prevdepth read shipout batchmode right skipdef setbox baselineskip special mskip endgroup uchyph binoppenalty endinput omit pagefilllstretch overwithdelims newlinechar vfilneg time vfill span prevgraf over show vbox tracingstats year defaulthyphenchar nullfont muskip closeout toks outer multiply tracingoutput parindent displaywidowpenalty unhbox lefthyphenmin vtop mathaccent discretionary vfuzz overline unkern showthe showbox uppercase lowercase closein openin errorstopmode scrollmode skewchar hyphenchar sfcode countdef mathchardef xdef gdef

	long atop scriptscriptstyle scriptstyle unpenalty copy lower kern vfil hfill hskip crcr cr ifvoid ifvmode if number lastpenalty skip par vrule noexpand mark ignorespaces fontdimen divide csname scriptspace outputpenalty month delimiterfactor relpenalty brokenpenalty tabskip
core	directlua
etex	unless botmarks currentiftype pagediscards mutoglue displaywidowpenalties fontcharic fontchardp fontcharwd tracingifs iffontchar eTeXVersion protected topmarks showgroups glueexpr splitfirstmarks predisplaydirection gluetomu everyeof eTeXversion clubpenalties savingvdiscards splitbotmarks showtokens tracingassigns dimexpr parshapedimen readline eTeXminorversion glueshrinkorder ifdefined currentifbranch firstmarks lastnodetype marks currentgrouplevel interlinepenalties muexpr unexpanded ifcsname parshapeindent showifs parshapelength currentgrouptype widowpenalties splitdiscards glueshrink gluestretch gluestretchorder numexpr scantokens interactionmode detokenize fontcharht currentiflevel savinghyphcodes lastlinefit tracingnesting tracingscantokens tracinggroups eTeXrevision
pdftex	pdfximage pdfpxdimen pdftrailer pdfuniqueresname pdfoutput pdfgentounicode pdfoutline pdfoptionpdfminorversion pdfendthread pdfimagehicolor pdflastximagecolordepth pdfpkresolution pdfthreadmargin pdfimageapplygamma pdfobjcompresslevel pdfreplacefont pdfxformresources pdfcopyfont pdfvorigin pdflastximagepages pdfcatalog pdfpageattr pdfgamma pdffontname pdfannot pdflastlink pdfnames pdfstartthread pdffontsize pdfimagegamma pdfglyphtounicode pdffontobjnum pdfcolorstack pdfxform pdfcompresslevel pdfinsertht pdfstartlink pdfmapfile pdftracingfonts pdfpagebox pdfcreationdate pdfpageref pdfcolorstackinit pdfdest pdfinclusionerrorlevel pdfinfo pdfxformattr pdfxformname pdfpagesattr pdflastannot pdfsave pdfhorigin pdfrefxform pdfliteral pdflastximage pdfimageresolution pdfdestmargin pdfobj pdfminorversion pdflastxform pdfximagebbox pdfincludechars pdfpkmode pdfretval pdfdecimaldigits pdflinkmargin pdfdraftmode expandglyphsinfont pdfmapline pdffontattr pdfthread pdfendlink pdfrefximage pdfrefobj pdfrestore pdfsetmatrix pdflastobj pdfinclusioncopyfonts pdfpageresources
omega	
aleph	
luatex	Umathcloseopspacing textdir Umathordpunctspacing Udelimiterunder luastartup Umathopenpunctspacing pagebottomoffset Umathordinnerspacing Umathbinclosespacing rightghost Umathlimitbelowbgap Umathopeninnerspacing Uoverdelimiter Umathpunctpunctspacing Umathclosepunctspacing Umathrelordspacing Umathsupbottommin Umathlimitbelowkern Umathstackdenomdown localrightbox Umathfractionrule Umathpunctinnerspacing Umathcloseinnerspacing Umathopenrelspacing Umathsupsubbottommax Umathcloserelspacing ifincsname Umathcharnum Umathinnerordspacing synctex formatname letterspacefont Umathrelinnerspacing Umathsubtopmax randomseed suppressoutererror Umathsubsupshiftdown Umathopbinspacing Umathordbinspacing Umathrelopspacing Umathopenbinspacing Umathoverdelimiterbgap localleftbox alignmark Uunderdelimiter hyphenationmin Umathclosebinspacing Umathcodenum luafunction Umath-

punctopenspacing Umathconnectoroverlapmin crampedscriptscriptstyle Umath-radicaldegreeafter uniformdeviate luatexversion Umathfractionnumup right-marginkern Umathopclosespacing Umathordclosespacing Umathoverdelimiter-vgap lastxpos expanded suppressmathparerror Udelcode bodydir Umathopen-closespacing attribute Umathsubshiftdrop Umathsubshiftdown matheqnogap-step Umathpunctrelspacing Umathradicaldegreeraise adjustspacing Umathsup-shiftdrop Umathpunctclosespacing Umathcloseclosespacing luatexrevision localinterlinepenalty Umathchar Udelimiterover Ustack Umathcode Udelcode-num suppresslongerror ignoreligaturesinfont Umathaxis Umathfractionnumvgap Umathrelclosespacing Umathpunctbinspacing luatexdatestamp Ustopdisplay-math quitvmode crampedscriptstyle setrandomseed crampedtextstyle pagedir Umathbinrelspacing Umathopordspacing attributedef Umathordordspacing Umath-openordspacing mathdir outputbox pagewidth Ustopmath aligntab Umathpunc-topspacing Umathsubsupvgap luaescapestring Umathfractiondenomvgap Umathrad-icalrule Umathunderbarrule postexhyphenchar Umathradicaldegreebefore Umath-stacknumup normaldeviate Umathbinopspacing boxdir Ustartdisplaymath save-catcodetable Umathbinpunctspacing tagcode Uroot Umathoverbarkern Umathoper-atorsize Uradical mathstyle Umathopopenspacing Umathordopenspacing Umath-bininnerspacing Umathinnerrelspacing clearmarks Umathoverbarvgap fontid Umathopenopenspacing Umathunderdelimiterbgap Umathoverbarrule crampeddis-playstyle ifabsdim Umathlimitabovebgap Umathstackvgap Umathinneropspacing Umathrelbinspacing Umathcloseopenspacing pardir initcatcodetable nokerns pageleftoffset Umathrelopenspacing Umathlimitabovekern Udelimiter savepos localbrokenpenalty Umathfractiondelsize gleaders Umathunderdelimitervgap Umathinnerbinspacing noligs Ustartmath Usubscript Umathcharnumdef rpcode Umathaccent pagetopoffset pageheight catcodetable Umathspaceafterscript primitive Umathinneropenspacing Umathordopspacing Umathopenopspacing ifab-snum scantextokens suppressifcsnameerror suppressfontnotfounderror latelua pagerightoffset efcode lpcode preexhyphenchar posthyphenchar prehyphen-char Umathinnerinnerspacing Umathinnerpunctspacing Umathinnerclosespacing Umathpunctordspacing Umathcloseordspacing Umathrelpunctspacing Umath-relrelspacing Umathbinopenspacing Umathbinbinspacing Umathbinordspacing Umathopinnerspacing Umathoppunctspacing Umathoprelspacing Umathopopspacing Umathordrelspacing Umathsupshiftup Umathlimitbelowvgap Umathlimitabovevgap Umathfractiondenomdown Umathradicalvgap Umathradicalkern Umathunderbarv-gap Umathunderbarkern Umathquad Umathchardef Usuperscript ifprimitive Uchar leftmarginkern luatexbanner lastypos leftghost protrudechars

umath Umathcloseopspacing Umathordpunctspacing Udelimiterunder Umathopen-punctspacing Umathordinnerspacing Umathbinclosespacing Umathlimitbelow-bgap Umathopeninnerspacing Uoverdelimiter Umathpunctpunctspacing Umath-closepunctspacing Umathrelordspacing Umathsupbottommin Umathlimitbelowk-ern Umathstackdenomdown Umathfractionrule Umathpunctinnerspacing Umathclo-seinnerspacing Umathopenrelspacing Umathsupsubbottommax Umathcloserelspac-ing Umathcharnum Umathinnerordspacing Umathrelinnerspacing Umathsubtopmax Umathsubsupshiftdown Umathopbinspacing Umathordbinspacing Umathrelopspacing Umathopenbinspacing Umathoverdelimiterbgap Uunderdelimiter Umathclosebinspacing Umathcodenum Umathpunctopenspacing Umathconnectoroverlapmin Umathradicaldegreeafter Umathfractionnumup Umathopclosespacing Umathordclosespacing Umathoverdelimitervgap Udelcode Umathopenclosespacing Umathsubshiftdrop Umathsubshiftdown Umathpunctrelspacing Umathradicaldegreeraise Umathsupshiftdrop Umathpunctclosespacing Umathcloseclosespacing Umathchar Udelimiterover Ustack Umathcode Udelcodenum Umathaxis Umathfractionnumvgap Umathrelclosespacing Umathpunctbinspacing Ustopdisplaymath Umathbinrelspacing Umathopordspacing Umathordordspacing Umathopenordspacing Ustopmath Umathpunctopspacing Umathsubsupvgap Umathfractiondenomvgap Umathradicalrule Umathunderbarrule Umathradicaldegreebefore Umathstacknumup Umathbinopspacing Ustartdisplaymath Umathbinpunctspacing Uroot Umathoverbarkern Umathoperatorsize Uradical Umathopopenspacing Umathordopenspacing Umathbininnerspacing Umathinnerrelspacing Umathoverbarvgap Umathopenopenspacing Umathunderdelimiterbgap Umathoverbarrule Umathlimitabovebgap Umathstackvgap Umathinneropspacing Umathrelbinspacing Umathcloseopenspacing Umathrelopenspacing Umathlimitabovekern Udelimiter Umathfractiondelsize Umathunderdelimitervgap Umathinnerbinspacing Ustartmath Usubscript Umathcharnumdef Umathaccent Umathspaceafterscript Umathinneropenspacing Umathordopspacing Umathopenopspacing Umathinnerinnerspacing Umathinnerpunctspacing Umathinnerclosespacing Umathpunctordspacing Umathcloseordspacing Umathrelpunctspacing Umathrelrelspacing Umathbinopenspacing Umathbinbinspacing Umathbinordspacing Umathopinnerspacing Umathoppunctspacing Umathoprelspacing Umathopopspacing Umathordrelspacing Umathsupshiftup Umathlimitbelowvgap Umathlimitabovevgap Umathfractiondenomdown Umathradicalvgap Umathradicalkern Umathunderbarvgap Umathunderbarkern Umathquad Umathchardef Usuperscript

Note that `'luatex'` does not contain `directlua`, as that isconsidered to be a core primitive, along with all the TEX82 primitives, so it is part of the list that is returned from `'core'`.

`'umath'` is a subset of `'luatex'` that covers the Unicode math primitives as it might be desired to handle the prefixing of that subset differently.

Running `tex.extraprimitives()` will give you the complete list of primitives `-ini` startup. It is exactly equivalent to `tex.extraprimitives('etex', 'pdftex' and 'luatex')`.

7.14.12.3 tex.primitives

```
<table> t = tex.primitives()
```

This function returns a hash table listing all primitives that LUATEX knows about. The keys in the hash are primitives names, the values are tables representing tokens (see section 7.17). The third value is always zero.

7.14.13 Core functionality interfaces

7.14.13.1 tex.badness

```
<number> b = tex.badness(<number> t, <number> s)
```

This helper function is useful during linebreak calculations. t and s are scaled values; the function returns the badness for when total t is supposed to be made from amounts that sum to s. The returned number is a reasonable approximation of $100(t/s)^3$;

7.14.13.2 tex.linebreak

```
local <node> nodelist, <table> info =
      tex.linebreak(<node> listhead, <table> parameters)
```

The understood parameters are as follows:

name	type	description
pardir	string	
pretolerance	number	
tracingparagraphs	number	
tolerance	number	
looseness	number	
hyphenpenalty	number	
exhyphenpenalty	number	
pdfadjustspacing	number	
adjdemerits	number	
pdfprotrudechars	number	
linepenalty	number	
lastlinefit	number	
doublehyphendemerits	number	
finalhyphendemerits	number	
hangafter	number	
interlinepenalty	number or table	if a table, then it is an array like \interlinepenalties
clubpenalty	number or table	if a table, then it is an array like \clubpenalties
widowpenalty	number or table	if a table, then it is an array like \widowpenalties
brokenpenalty	number	
emergencystretch	number	in scaled points
hangindent	number	in scaled points
hsize	number	in scaled points
leftskip	glue_spec node	
rightskip	glue_spec node	
pdfignoreddimen	number	in scaled points
parshape	table	

Note that there is no interface for \displaywidowpenalties, you have to pass the right choice for widowpenalties yourself.

The meaning of the various keys should be fairly obvious from the table (the names match the TEX and PDFTEX primitives) except for the last 5 entries. The four `pdf...line...` keys are ignored if their value equals `pdfignoreddimen`.

It is your own job to make sure that `listhead` is a proper paragraph list: this function does not add any nodes to it. To be exact, if you want to replace the core line breaking, you may have to do the following (when you are not actually working in the `pre_linebreak_filter` or `linebreak_filter` callbacks, or when the original list starting at listhead was generated in horizontal mode):

- add an 'indent box' and perhaps a `local_par` node at the start (only if you need them)
- replace any found final glue by an infinite penalty (or add such a penalty, if the last node is not a glue)
- add a glue node for the `\parfillskip` after that penalty node
- make sure all the `prev` pointers are OK

The result is a node list, it still needs to be vpacked if you want to assign it to a `\vbox`.

The returned `info` table contains four values that are all numbers:

prevdepth	depth of the last line in the broken paragraph
prevgraf	number of lines in the broken paragraph
looseness	the actual looseness value in the broken paragraph
demerits	the total demerits of the chosen solution

Note there are a few things you cannot interface using this function: You cannot influence font expansion other than via `pdfadjustspacing`, because the settings for that take place elsewhere. The same is true for hbadness and hfuzz etc. All these are in the `hpack()` routine, and that fetches its own variables via globals.

7.14.13.3 `tex.shipout`

```
tex.shipout(<number> n)
```

Ships out box number `n` to the output file, and clears the box register.

7.15 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**
`kpse_init`	boolean	true	`false` totally disables KPATHSEA initialisation, and enables interpretation of the following numeric key–value pairs. (only ever unset this if you implement *all* file find callbacks!)
`shell_escape`	string	`'f'`	Use `'y'` or `'t'` or `'1'` to enable `\write 18` unconditionally, `'p'` to enable the commands that are listed in `shell_escape_commands`

shell_escape_commands	string		Comma-separated list of command names that may be executed by \write 18 even if shell_escape is set to 'p'. Do *not* use spaces around commas, separate any required command arguments by using a space, and use the ASCII double quote (") for any needed argument or path quoting
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
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
hash_extra	number	0	cf. web2c docs
pk_dpi	number	72	cf. web2c docs
trace_file_names	boolean	true	false disables TeX's normal file open-close feedback (the assumption is that callbacks will take care of that)
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

Note: the numeric values that match web2c parameters are only used if kpse_init is explicitly set to false. In all other cases, the normal values from texmf.cnf are used.

7.16 The texio library

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

7.16.1 Printing functions

7.16.1.1 texio.write

```
texio.write(<string> target, <string> s, ...)
```

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

Without the `target` argument, writes all given strings 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`.

Note: If several strings are given, and if the first of these strings is or might be one of the targets above, the `target` must be specified explicitly to prevent LUA from interpreting the first string as the target.

7.16.1.2 `texio.write_nl`

```
texio.write_nl(<string> target, <string> s, ...)
texio.write_nl(<string> s, ...)
```

This function behaves like `texio.write`, but make sure that the given strings will appear at the beginning of a new line. You can pass a single empty string if you only want to move to the next line.

7.17 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:

index	**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 the result of control sequences, like letters and characters, it is zero, otherwise, it is a number pointing into the 'equivalence table'

7.17.1 `token.get_next`

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

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

7.17.2 `token.is_expandable`

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

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

7.17.3 `token.expand`

```
token.expand(<token> t)
```

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 `token.get_next()`.

7.17.4 token.is_activechar

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

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

7.17.5 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 *not* possible to define brand new control sequences using `token.create`!

7.17.6 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_test`, and the 'command modifier' defines which test is to be run.

7.17.7 token.command_id

```
<number> i = token.command_id(<string> commandname)
```

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

7.17.8 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.

Keep in mind that there are potentially two control sequences that return the same csname string: single character control sequences and active characters have the same 'name'.

7.17.9 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.

7.17.10 The newtoken libray

The current `token` library will be replaced by a new one that is more flexible and powerful. The transition takes place in steps. In version 0.80 we have `newtoken` and in version 0.85 the old lib will be replaced completely. So if you use this new mechanism in production code you need to be aware of incompatible updates between 0.80 and 0.90. Because the related in- and output code will also be cleaned up and rewritten you should be aware of incompatible logging and error reporting too.

The old library presents tokens as triplets or numbers, the new library presents a userdata object. The old library used a callback to intercept tokens in the input but the new library provides a basic scanner infrastructure that can be used to write macros that accept a wide range of arguments. This interface is on purpose kept general and as performance is quite ok one can build additional parsers without too much overhead. It's up to macro package writers to see how they can benefit from this as the main principle behind LuaTeX is to provide a minimal set of tools and no solutions.

The current functions in the `newtoken` namespace are given in the next table:

function	argument	result
is_token	token	checks if the given argument is a token userdatum
get_next		returns the next token in the input
scan_keyword	string	returns true if the given keyword is gobbled
scan_int		returns a number
scan_dimen	infinity, mu-units	returns a number representing a dimension and or two numbers being the filler and order
scan_glue	mu-units	returns a glue spec node
scan_toks	definer, expand	returns a table of tokens token list (this can become a linked list in later releases)
scan_code	bitset	returns a character if its category is in the given bitset (representing catcodes)
scan_string		returns a string given between `{}`, as `\macro` or as sequence of characters with catcode 11 or 12
scan_word		returns a sequence of characters with catcode 11 or 12 as string

create	returns a userdata token object of the given control sequence name (or character); this interface can change

The scanners can be considered stable apart from the one scanning for a token. This is because futures releases can return a linked list instead of a table (as with nodes). The `scan_code` function takes an optional number, the `keyword` function a normal LUA string. The `infinity` boolean signals that we also permit `fill` as dimension and the `mu-units` flags the scanner that we expect math units. When scanning tokens we can indicate that we are defining a macro, in which case the result will also provide information about what arguments are expected and in the result this is separated from the meaning by a separator token. The `expand` flag determines if the list will be expanded.

The string scanner scans for something between curly braces and expands on the way, or when it sees a control sequence it will return its meaning. Otherwise it will scan characters with catcode `letter` or `other`. So, given the following definition:

```
\def\bar{bar}
\def\foo{foo-\bar}
```

we get:

`\directlua{newtoken.scan_string()}{foo}`	`foo`	full expansion
`\directlua{newtoken.scan_string()}foo`	`foo`	letters and others
`\directlua{newtoken.scan_string()}\foo`	`foo-bar`	meaning

The `\foo` case only gives the meaning, but one can pass an already expanded definition (`\edef`'d). In the case of the braced variant one can of course use the `\detokenize` and `\unexpanded` primitives as there we do expand.

The `scan_word` scanner can be used to implement for instance a number scanner:

```
function newtokens.scan_number(base)
    return tonumber(newtoken.scan_word(),base)
end
```

This scanner accepts any valid LUA number so it is a way to pick up floats in the input.

The creator function can be used as follows:

```
local t = newtoken("relax")
```

This gives back a token object that has the properties of the `\relax` primitive. The possible properties of tokens are:

command	a number representing the internal command number
cmdname	the type of the command (for instance the catcode in case of a character or the classifier that determines the internal treatment
csname	the associated control sequence (if applicable)
id	the unique id of the token
active	a boolean indicating the active state of the token
expandable	a boolean indicating if the token (macro) is expandable
protected	a boolean indicating if the token (macro) is protected

The numbers that represent a catcode are the same as in TeX itself, so using this information assumes that you know a bit about TeX's internals. The other numbers and names are used consistently but are not frozen. So, when you use them for comparing you can best query a known primitive or character first to see the values.

More interesting are the scanners. You can use the Lua interface as follows:

```
\directlua {
    function mymacro(n)
        ...
    end
}

\def\mymacro#1{%
    \directlua {
        mymacro(\number\dimexpr#1)
    }%
}

\mymacro{12pt}
\mymacro{\dimen0}
```

You can also do this:

```
\directlua {
    function mymacro()
        local d = newtoken.scan_dimen()
        ...
    end
}

\def\mymacro{%
    \directlua {
        mymacro()
    }%
}

\mymacro 12pt
\mymacro \dimen0
```

It is quite clear from looking at the code what the first method needs as argument(s). For the second method you need to look at the Lua code to see what gets picked up. Instead of passing from TeX to Lua we let Lua fetch from the input stream.

In the first case the input is tokenized and then turned into a string when it's passed to Lua where it gets interpreted. In the second case only a function call gets interpreted but then the input is picked up by explicitly calling the scanner functions. These return proper Lua variables so no further conversion has to be done. This is more efficient but in practice (given what TeX has to do) this effect should not be overestimated. For numbers and dimensions it saves a bit but for passing strings conversion to and from tokens has to be done anyway (although we can probably speed up the process in later versions if needed).

When the interface is stable and has replaced the old one completely we will add some more information here. By that time the internals have been cleaned up a bit more so we know then what will stay and go. A positive side effect of this transition is that we can simplify the input part because we no longer need to intercept using callbacks.

8 Modifications

8.1 The merged engines

8.1.1 The need for change

The first version of LuaTeX only had a few extra primitives and it was largely the same as pdfTeX. Then we merged substantial parts of Aleph into the code and got more primitives. When we got more stable teh decision was made to clean up the rather hybrid nature of the program. This means that some primnitives have been promoted to core primitives, often with a different name, and that others were removed. This made it possible to start cleaning up the code base. We will describe most in following paragraphs.

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 neccessary to clean up the internal interfaces. These will also be mentioned.

8.1.2 Changes from TeX 3.1415926

Of course it all starts with traditional TeX. Even if we started with pdfTeX, most still comes from the original. But we divert a bit.

- The current code base is written in C, not Pascal. We use cweb when possible.
- See chapter 3 for many small changes related to paragraph building, language handling and hyphenation. The most important change is that adding a brace group in the middle of a word (like in `of{}fice`) does not prevent ligature creation.
- There is no pool file, all strings are embedded during compilation.
- The specifier `plus 1 fillll` does not generate an error. The extra ‘l’ is simply typeset.
- The upper limit to `\endlinechar` and `\newlinechar` is 127.
- The hz optimization code has been partially redone so that we no longer need to create extra font instances. The front- and backend have been decoupled and more efficient (pdf) code is generated.

8.1.3 Changes from ε-TeX 2.2

Being the de factor standard extension of course we provide the ε-TeX functionality, but with a few small adaptions.

- The ε-TeX functionality is always present and enabled so the prepended asterisk or `-etex` switch for iniTeX is not needed.
- The TeXXeT extension is not present, so the primitives `\TeXXeTstate`, `\beginR`, `\beginL`, `\endR` and `\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.
- The `\savinghyphcodes` command is a no-op. Chapter 3 explains why.
- When kpathsea is used to find files, LuaTeX uses the `ofm` file format to search for font metrics. In turn, this means that LuaTeX looks at the `OFMFONTS` configuration variable (like Omega and Aleph) instead of `TFMFONTS` (like TeX and pdfTeX). Likewise for virtual fonts (LuaTeX uses the variable `OVFFONTS` instead of `VFFONTS`).

8.1.4 Changes from PDFTeX 1.40

Because we want to produce PDF the most natural starting point was the popular pdfTeX program. We inherit the stable features, dropped most of the experimental code and promoted some functionality to core LuaTeX functionality which in turn triggered renaming primitives.

- The (experimental) support for snap nodes has been removed, because it is 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`.
- The (experimental) support for specialized spacing around nodes has also been removed. The associated primitives that are now gone are: `\pdfadjustinterwordglue`, `\pdfprependkern`, and `\pdfappendkern`, as well as the five supporting primitives `\knbscode`, `\stbscode`, `\shbscode`, `\knbccode`, and `\knaccode`.
- A number of 'pdftex primitives' have been removed as they can be implemented using Lua: `\pdfelapsedtime`, `\pdfescapehex`, `\pdfescapename`, `\pdfescapestring`, `\pdffiledump`, `\pdffilemoddate`, `\pdffilesize`, `\pdfforcepagebox`, `\pdflastmatch`, `\pdfmatch`, `\pdfmdfivesum`, `\pdfmovechars`, `\pdfoptionalwaysusepdfpagebox`, `\pdfoptionpdfinclusionerrorlevel`, `\pdfresettimer`, `\pdfshellescape`, `\pdfstrcmp` and `\pdfunescapehex`
- The version related primitives `\pdftexbanner`, `\pdftexversion` and `\pdftexrevision` are no longer present as there is no longer a strict relationship with pdfTeX development.
- The experimental snapper mechanism has been removed and therefore also the primitives: `\pdfignoreddimen`, `\pdffirstlineheight`, `\pdfeachlineheight`, `\pdfeachlinedepth` and `\pdflastlinedepth`
- The experimental primitives `\primitive`, `\ifprimitive`, `\ifabsnum` and `\ifabsdim` are promoted to core primitives. The `\pdf*` prefixed originals are not available.
- The PNG transparency fix from 1.40.6 is not applied as high-level support is pending.
- Two extra token lists are provides, `\pdfxformresources` and `\pdfxformattr`, as an alternative to `\pdfxform` keywords.
- The current version of LuaTeX no longer replaces and/or merges fonts in embedded pdf files with fonts of the enveloping PDF document. This regression may be temporary, depending on how the rewritten font backend will look like.
- The primitives `\pdfpagewidth` and `\pdfpageheight` have been removed because `\pagewidth` and `\pageheight` have that purpose.
- The primitives `\pdfnormaldeviate`, `\pdfuniformdeviate`, `\pdfsetrandomseed` and `\pdfrandomseed` have been promoted to core primitives without `pdf` prefix so the original commands are no longer recognized.
- The primitives `\ifincsname`, `\expanded` and `\quitvmode` are now core primitives.

- As the hz and protrusion mechanism are part of the core the related primitives `\lpcode`, `\rpcode`, `\efcode`, `\leftmarginkern`, `\rightmarginkern` are promoted to core primitives. The two commands `\protrudechars` and `\adjustspacing` replace their prefixed with `\pdf` originals.
- The `\tagcode` primitive is promoted to core primitive.
- The `\letterspacefont` feature is now part of the core but will not be changed (improved). We just provide it for legacy use.
- The `\pdfnoligatures` primitive is now `\ignoreligaturesinfont`.
- The `\pdffontexpand` primitive is now `\expandglyphsinfont`.
- Because position tracking is also available in DVI mode the `\savepos`, `\lastxpos` and `\lastypos` commands now replace their `pdf` prefixed originals.
- Candidates for removal are `\pdfcolorstackinit` and `\pdfcolorstack`.
- Candidates for replacement are `\pdfoutput` (`\outputmode`) and `\pdfmatrix` (something with a normal syntax).

8.1.5 Changes from ALEPH RC4

Because we wanted proper directional typesetting the ALEPH mechanisms looked most attractive. These are rather close to the ones provided by OMEGA, so what we say next applies to both these programs.

- The extended 16-bit math primitives (`\omathcode` etc.) have been removed.
- The OCP processing is no longer supported at all. As a consequence, the following primitives have been removed:
  `\ocp`, `\externalocp`, `\ocplist`, `\pushocplist`, `\popocplist`, `\clearocplists`, `\addbeforeocplist`, `\addafterocplist`, `\removebeforeocplist`, `\removeafterocplist` and `\ocptracelevel`
- LuaTeX only understands 4 of the 16 direction specifiers of ALEPH: `TLT` (latin), `TRT` (arabic), `RTT` (cjk), `LTL` (mongolian). All other direction specifiers generate an error.
- The input translations from ALEPH are not implemented, the related primitives are not available: `\DefaultInputMode`, `\noDefaultInputMode`, `\noInputMode`, `\InputMode`, `\DefaultOutputMode`, `\noDefaultOutputMode`, `\noOutputMode`, `\OutputMode`, `\DefaultInputTranslation`, `\noDefaultInputTranslation`, `\noInputTranslation`, `\InputTranslation`, `\DefaultOutputTranslation`, `\noDefaultOutputTranslation`, `\noOutputTranslation` and `\OutputTranslation`
- Several bugs hav ebeen fixed. The `\hoffset` bug when `\pagedir TRT` is gone, removing the need for an explicit fix to `\hoffset`. Also bug causing `\fam` to fail for family numbers above 15 is fixed. A fair amount of other minor bugs are fixed as well, most of these related to `\tracingcommands` output.
- The scanner for direction specifications now allows an optional space after the direction is completely parsed.
- The `^^` notation can come in five and six item repetitions also, to insert characters that do not fit in the BMP.
- Glues *immediately after* direction change commands are not legal breakpoints.
- Several mechanisms that need to be right-to-left aware have been improved. For instance placement of formula numbers.

- The page dimension related primitives `\pagewidth` and `\pageheight` have been promoted to core primitives.
- The primitives `\charwd`, `\charht`, `\chardp` and `\charit` have been removes as we have the ε-TEX variants `\fontchar*`.
- The two dimension registers `\pagerightoffset` and `\pagebottomoffset` are now core primitives.
- The direction related primitives `\pagedir`, `\bodydir`, `\pardir`, `\textdir`, `\mathdir` and `\boxdir` are now core primitives.
- The promotion of primitives to core primitives as well as the removed of all others mean that the initialization namespace `aleph` is gone.

8.1.6 Changes from standard WEB2C

The compilation framework is WEB2C and we keep using that but without the PASCAL to C step. This framework also provides some common features that deal with reading bytes from files and locating files in TDS. This is what we do different:

- There is no mltex support.
- There is no enctex support.
- The following commandline switches are silently ignored, even in non-LUA mode: `-8bit`, `-translate-file`, `-mltex`, `-enc` and `-etex`.
- The `\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`).
- The 'E' option does not do anything useful.

8.2 Implementation notes

8.2.1 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 `texnode.c`, and basically uses a dozen or so 'avail' lists instead of a doubly-linked model. An extra function layer is added so that the code can ask for nodes by type instead of directly requisitioning a certain amount of memory words.

Because of the split into two arrays and the resulting differences in the data structures, some of the macros have been duplicated. For instance, there are now `vlink` and `vinfo` as well as `token_link` and `token_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.

8.2.2 Sparse arrays

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

The `\catcode`, `\sfcode`, `\lccode` and `\uccode` 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 $2^{21} - 1$.

8.2.3 Simple single-character csnames

Single-character commands are no longer treated specially 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.

Active characters are internally implemented as a special type of multi-letter control sequences that uses a prefix that is otherwise impossible to obtain.

8.2.4 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 IO, so it should still be faster.

8.2.5 Binary file reading

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