LuaTEX Reference

snapshot 2007-09-17

LuaTEX Reference Manual

more info: www.luatex.org
version: September 17, 2007(snapshot 2007-09-17)

Contents

1 Introduction

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

Features may come and go. The current version of LuaTEX is not meant for production and users cannot depend on stability, nor on functionality staying the same.

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

If you are not willing to deal with this situation, you should wait for the stable version. Currently we expect the first release to be available sometime in the summer of 2008.

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

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

Neither Aleph's I/O translation processes, nor tcx files, nor encTEX can be used, these encoding-related functions are superseded by a Lua-based solution (reader callbacks).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

2 Basic TEX enhancements

2.1 Version information

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

primitive	explanation
\luatexversion	A combination of major an minor number, as in pdfTeX. Current value: **11**
\luatexrevision	The revision, as in pdfTeX. Current value: **1**
\luatexdatestamp	A combination of the local date and hour when the current executable was compiled, the syntax is identical to \luatexrevision. Value for the executable that generated this document: **2007091716**.

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

2.2 UNICODE text support

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

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

Note: for now, it only makes sense to use values above the base plane ("0xFFFF) for \mathcode and \catcode assignments, since the hyphenation patterns are still limited to at the most 16-bit values, so the other commands will not know what to do with those high values.

A few primitives affected by this, all in a similar fashion: each of them has to accomodate for a larger range of acceptable numbers. For instance, \char now accepts values between 0 and 1114111. This should not be a problem for well-behaved input files, but it could create incompatibilities for input that would have generated an error when processed by older TEX-based engines. The maximum number of allocations is "10FFFF or $2^{20} + 2^{16}$ (21 bits). The maximum value that can be assigned are:

primitive	bits	hex	numeric
\char	21	10FFFF	$2^{20} + 2^{16}$
\chardef	21	10FFFF	$2^{20} + 2^{16}$
\lccode	21	10FFFF	$2^{20} + 2^{16}$

`\uccode`	21	10FFFF	$2^{20}+2^{16}$
`\sfcode`	15	7FFF	2^{15}
`\catcode`	4	F	2^4

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

Output in byte-sized chunks can be achieved by using characters just outside of the valid unicode range, starting at the value 1.114.112 (0x110000). When the times comes to print a character $c >= 1.114.112$, LuaTEX will actually print the single byte corresponding to $c - 1.114.112$.

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

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

2.3 Wide math characters

Text handling is now extended up to the full Unicode range, but math mode deals mostly with glyphs in fonts directly and fonts tend to be 16-bit at maximum. The extension from 8-bit to 16-bit was already present in Aleph by means of a set of extra primitives.

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

primitive	max index/bits	hex	numeric
`\mathchardef`	15	8000	$2^3 \times 2^8 \times 2^4$
`\mathcode`	15	8000	$2^3 \times 2^8 \times 2^4$
`\delcode`	27	7FFFFFF	$2^3 \times 2^4 \times 2^8 \times 2^4 \times 2^8$
`\mathchar`	15	7FFF	$2^3 * 2^8 * 2^4$
`\delimiter`	27	7FFFFFF	$2^3 * 2^4 * 2^8 * 2^4 * 2^8$
`\omathchar`	27	7FFFFFF	$2^3 * 2^{16} * 2^8$
`\odelimiter`	27+24	7FFFFFF + FFFFFF	$2^3 * 2^8 * 2^{16} + 2^8 * 2^{16}$
`\omathchardef`	21=27	10FFFF = 8000000	$2^{20}+2^{16} = 2^3 * 2^{16} * 2^8$
`\omathcode`	21=27	10FFFF = 8000000	$2^{20}+2^{16} = 2^3 * 2^{16} * 2^8$
`\odelcode`	21=27+24	10FFFF = 7FFFFFF + FFFFFF	$2^{20}+2^{16} = 2^3 * 2^8 * 2^{16}$ $+2^8 * 2^{16}$

2.4 Extended tables

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

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

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

2.5 Attribute registers

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

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

Conceptually, an attribute is either 'set' or 'unset'. Set attributes can only have values of 0 or more, otherwise they are considered unset and automatically remapped to an special negative value meaning 'unset' (currently that value is −1, but please test on negativity, not on a specific value). All attributes start out in the 'unset' state (in INITEX).

Attributes can be used as extra counter values, but their usefulness comes mostly from the fact that the numbers and values of all 'set' attributes are attached to all nodes created in their scope. These can then be queried from any LUA code that deals with node processing. Future versions of LUATEX will propably be using specific negative attribute ids for internal use. Further information about how to use attributes for node list processing from lua is given in chapter ??.

2.6 LUA related primitives

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

2.6.1 \directlua

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

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

The ⟨general text⟩ is fed into the LUA interpreter state indicated by the ⟨16-bit number⟩. If the state does not exist yet, it will be initialized automatically. The current category codes are applied to the ⟨general text⟩, and it is passed on as if it was displayed using \the\toks. On the LUA side, each of these blocks is treated as a chunk comprising a single line. This means that you can not use LUA line comments (starting with --) within the argument, as that will last for the rest of the input. You need to use TEX-style comments (starting with %) instead.

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

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

will result in $\pi = 3.1415926535898$

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

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

2.6.2 \latelua

\latelua stores LUA code in a whatsit that will be processed inside the output routine. It's intended use is very similar to \pdfliteral. Within the LUA code, you can print PDF statements directly to the PDF file.

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

2.6.3 \luaescapestring

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

\luaescapestring ⟨general text⟩

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

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

2.6.4 \closelua

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

\closelua ⟨16-bit number⟩

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

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

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

2.7 New ε-TEX primitives

2.7.1 \clearmarks

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

\clearmarks ⟨16-bit number⟩

2.7.2 \noligs and \nokerns

These primitives prohibit ligature and kerning insertion at the time when the initial node list is built by LUATEX's main control loop. They are part of a temporary trick and will be removed in the near future. For now, you need to enable these primitives when you want to do node list processing of 'characters', where TEX's normal processing would get in the way.

\noligs ⟨integer⟩
\nokerns ⟨integer⟩

2.7.3 \formatname

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

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

2.7.4 \scantextokens

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

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

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

2.7.5 Catcode tables

Catcode tables are a new feature that allows you to switch to a predefined catcode regime in a single statement. You can have a practically unlimited number of different tables.

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

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

2.7.5.1 \catcodetable

\catcodetable ⟨28-bit number⟩

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

2.7.5.2 \initcatcodetable

\initcatcodetable ⟨28-bit number⟩

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

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

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

2.7.5.3 \savecatcodetable

\savecatcodetable ⟨28-bit number⟩

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

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

2.7.6 \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.

2.7.7 Font syntax

LuaTeX will accept a braced argument as a font name:

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

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

3 LUA general

3.1 Initialization

3.1.1 LUATEX as a LUA interpreter

There are some situations that make LuaTEX behaves like it is a Lua interpreter only:

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

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

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

3.1.2 LUATEX as a LUA byte compiler

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

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

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

3.1.3 Other commandline processing

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

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

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

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

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

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

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

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

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

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

We recommend you use the startup file only for your own TEX-independant initializations (if you need any), to parse the commandline, set values in the `texconfig` table, and register the callbacks you need. LUATEX will fetch some of the other commandline options from the `texconfig` table at the end of script execution (see the description of the `texconfig` table later on in this document for more details on which ones exactly).

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

flag	**meaning**
`--fmt=s`	set the format name
`--progname=s`	set the progname (only for KPATHSEA)
`--ini`	enable INITEX mode

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

3.2 LUA changes

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