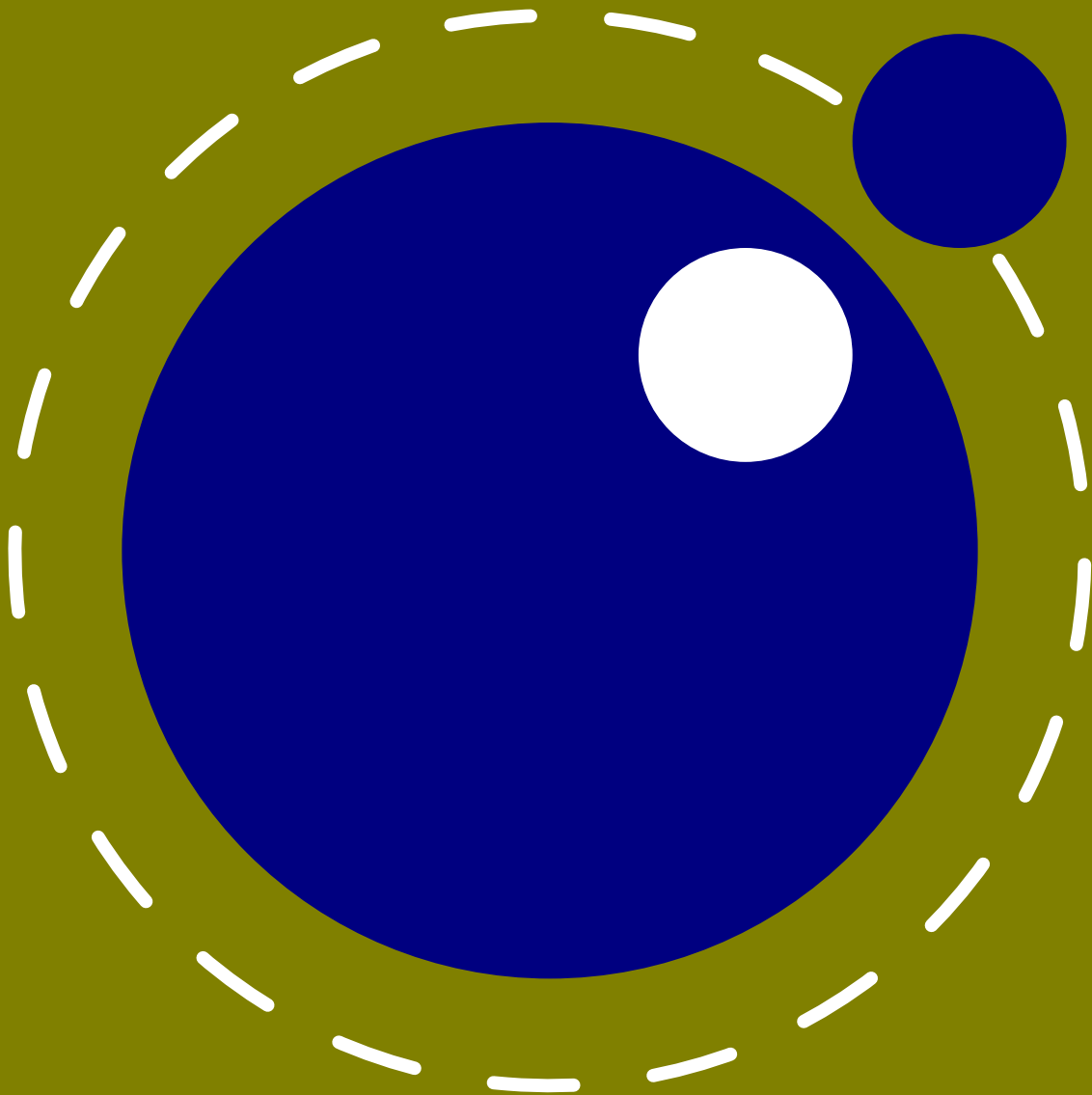


# **LuaMetaT<sub>E</sub>X**

## **Reference**

## **Manual**



**March 2022**  
**Version 2.09.42**



# **LuaMetaT<sub>E</sub>X**

## **Reference**

### **Manual**

**copyright** : LuaMetaT<sub>E</sub>X development team  
              : LuaT<sub>E</sub>X development team  
              : ConT<sub>E</sub>Xt development team  
**more info** : [www.luatex.org](http://www.luatex.org)  
              : [contextgarden.net](http://contextgarden.net)  
              : [build.contextgarden.net](http://build.contextgarden.net)  
**version**   : March 10, 2022



# Contents

<b>Introduction</b>	<b>11</b>
<b>1 The internals</b>	<b>13</b>
<b>2 Differences with LuaTeX</b>	<b>17</b>
<b>3 The original engines</b>	<b>29</b>
3.1 The merged engines	29
3.1.1 The rationale	29
3.1.2 Changes from T <sub>E</sub> X 3.1415926...	29
3.1.3 Changes from $\varepsilon$ -T <sub>E</sub> X 2.2	31
3.1.4 Changes from pdfT <sub>E</sub> X 1.40	31
3.1.5 Changes from Aleph RC4	32
3.1.6 Changes from standard web2c	33
3.2 Implementation notes	33
3.2.1 Memory allocation	33
3.2.2 Sparse arrays	33
3.2.3 Simple single-character csnames	33
3.2.4 Binary file reading	33
3.2.5 Tabs and spaces	34
3.2.6 Logging	34
3.2.7 Parsing	35
3.2.8 Changes in keyword scanning	35
<b>4 Using LuaMetaT<sub>E</sub>X</b>	<b>37</b>
4.1 Initialization	37
4.1.1 A bare bone engine	37
4.1.2 LuaMetaT <sub>E</sub> X as a Lua interpreter	37
4.1.3 Other commandline processing	37
4.2 Lua behaviour	39
4.2.1 The Lua version	39
4.2.2 Locales	39
4.3 Lua modules	39
4.4 Testing	40
<b>5 Basic T<sub>E</sub>X enhancements</b>	<b>41</b>
5.1 Introduction	41
5.1.1 Primitive behaviour	41
5.1.2 Rationale	41
5.1.3 Version information	42
5.2 Unicode text support	44
5.2.1 Extended ranges	44
5.2.2 \Uchar	44
5.2.3 Extended tables	44



5.3	Attributes	45
5.3.1	Nodes	45
5.3.2	Attribute registers	46
5.3.3	Box attributes	46
5.4	Lua related primitives	48
5.4.1	\directlua	48
5.4.2	\luaescapestring	49
5.4.3	\luafunction, \luafunctioncall and \luadef	49
5.4.4	\luabytecode and \luabytecodecall	50
5.5	Catcode tables	51
5.5.1	Catcodes	51
5.5.2	\catcodetable	51
5.5.3	\initcatcodetable	51
5.5.4	\savecatcodetable	52
5.6	Tokens, commands and strings	52
5.6.1	\scantextokens, \tokenized and \retokenized	52
5.6.2	\toksapp, \tokspre, \etoksapp, \etokspre, \gtoksapp, \gtokspre, \xtoksapp, \xtokspre	52
5.6.3	\csstring, \begincsname and \lastnamedcs	52
5.6.4	\clearmarks, \flushmarks, \currentmarks	53
5.6.5	\alignmark, \aligntab, \aligncontent, \tabsize and \everytab	53
5.6.6	\letcharcode	54
5.6.7	\lettonothing and \glettonothing	54
5.6.8	\glet	54
5.6.9	\defcsname, \edefcsname, \edefcsname and \xdefcsname	54
5.6.10	\expanded	55
5.6.11	\ignorepars	55
5.6.12	\futureexpand, \futureexpandis, \futureexpandisap	55
5.6.13	\aftergrouped	55
5.7	Conditions	56
5.7.1	\ifabsnum and \ifabsdim	56
5.7.2	\ifcmpnum, \ifcmpdim, \ifnumval, \ifdimval, \ifchknum and \ifchkdim	56
5.7.3	\ifmathstyle and \ifmathparameter	57
5.7.4	\ifempty	57
5.7.5	\ifrelax	58
5.7.6	\ifboolean	58
5.7.7	\iftok and \ifcstok	58
5.7.8	\ifarguments, \ifparameters and \ifparameter	58
5.7.9	\ifcondition	59
5.7.10	\orelse and \orunless	60
5.7.11	\ifflags	61
5.8	Boxes, rules and leaders	61
5.8.1	\outputbox	61
5.8.2	\hrule, \vrule, \nohrule and \novrule	61
5.8.3	\vsplit	62



5.8.4	Images and reused box objects	62
5.8.5	\hpack, \vpack and \tpack	64
5.8.6	\gleaders	64
5.9	Languages	64
5.9.1	\hyphenationmin	64
5.9.2	\boundary, \noboundary, \protrusionboundary and \wordboundary	64
5.10	Control and debugging	64
5.10.1	Tracing	64
5.10.2	\lastnodetype, \lastnodesubtype, \currentifttype	65
5.11	Files	65
5.11.1	File syntax	65
5.11.2	Writing to file	66
5.12	Math	66
5.13	Fonts	66
5.14	Directions	66
5.14.1	Two directions	66
5.14.2	How it works	67
5.14.3	Normalizing lines	68
5.14.4	Orientations	69
5.15	Keywords	69
5.16	Expressions and \numericsscale	70
5.17	Macro arguments	70
5.18	Overload protection	71
5.19	Constants with \integerdef and \dimensiondef	73
5.20	Serialization with \todimension, \toscaled and \tointeger	74
5.21	Expressions with \numexpression	74
5.22	Nodes	76
<b>6</b>	<b>Fonts</b>	<b>77</b>
6.1	Introduction	77
6.2	Defining fonts	77
6.3	Virtual fonts	81
6.4	Additional T <sub>E</sub> X commands	84
6.4.1	Font syntax	84
6.4.2	\fontid and \setfontid	84
6.4.3	\glyphoptions	85
6.4.4	\glyphscale, \glyphxscale, \glyphyscale and \scaledfontdimen	85
6.4.5	\fontspecdef, \fontspecid, \fontspecscale, , \fontspecxscale, \fontspecyscale	85
6.4.6	\glyphxoffset, \glyphyoffset	86
6.4.7	\glyph	86
6.4.8	\nospaces	86
6.4.9	\protrusionboundary	87



6.5	The Lua font library	87
6.5.1	Introduction	87
6.5.2	Defining a font with <code>define</code> , <code>addcharacters</code> and <code>setfont</code>	88
6.5.3	Font ids: <code>id</code> , <code>max</code> and <code>current</code>	88
6.5.4	Glyph data: <code>\glyphdatafield</code> , <code>\glyphscriptfield</code> , <code>\glyphstatefield</code>	88
<b>7</b>	<b>Languages, characters, fonts and glyphs</b>	<b>89</b>
7.1	Introduction	89
7.2	Characters, glyphs and discretionaries	89
7.3	The main control loop	94
7.4	Loading patterns and exceptions	96
7.5	Applying hyphenation	98
7.6	Applying ligatures and kerning	99
7.7	Breaking paragraphs into lines	100
7.8	The language library	100
7.8.1	<code>new</code> and <code>id</code>	100
7.8.2	hyphenation	100
7.8.3	<code>clearhyphenation</code> and <code>clean</code>	101
7.8.4	<code>patterns</code> and <code>clearpatterns</code>	101
7.8.5	<code>hyphenationmin</code>	101
7.8.6	<code>[pre post][ex ]hyphenchar</code>	101
7.8.7	<code>hyphenate</code>	102
7.8.8	<code>[set get]hjcode</code>	102
<b>8</b>	<b>Math</b>	<b>103</b>
8.1	Traditional alongside OpenType	103
8.2	Unicode math characters	103
8.3	Setting up the engine	105
8.4	Math styles	106
8.4.1	<code>\mathstyle</code>	106
8.4.2	<code>\Ustack</code>	107
8.4.3	The new <code>\cramped ...style</code> commands	108
8.5	Math parameter settings	109
8.5.1	Many new <code>\Umath*</code> primitives	109
8.5.2	Font-based math parameters	111
8.6	Math spacing	115
8.6.1	Setting inline surrounding space with <code>\mathsurround</code> and <code>\mathsurroundskip</code>	115
8.6.2	Pairwise spacing	116
8.6.3	Local <code>\frozen</code> settings with	117
8.6.4	Checking a state with <code>\ifmathparameter</code>	117
8.6.5	Skips around display math and <code>\mathdisplayskipmode</code>	118
8.6.6	Nolimit correction with <code>\mathnolimitsmode</code>	118
8.6.7	Influencing script kerning with <code>\mathscriptboxmode</code>	118
8.6.8	Forcing fixed scripts with <code>\mathscriptsmode</code>	119
8.6.9	Penalties: <code>\mathpenaltiesmode</code>	120
8.6.10	Equation spacing: <code>\matheqnogapstep</code>	120





8.7	Math constructs	120
8.7.1	Unscaled fences and <code>\mathdelimitersmode</code>	120
8.7.2	Accent handling with <code>\Umathaccent</code>	122
8.7.3	Building radicals with <code>\Uradical</code> and <code>\Uroot</code>	122
8.7.4	Super- and subscripts	123
8.7.5	Scripts on extensibles: <code>\Uunderdelimit</code> , <code>\Uoverdelimit</code> , <code>\Udelimitover</code> , <code>\Udelimitunder</code> and <code>\Uhexensible</code>	123
8.7.6	Fractions and the new <code>\Uskewed</code> and <code>\Uskewedwithdelims</code>	124
8.7.7	Math styles: <code>\Ustyle</code>	125
8.7.8	Delimiters: <code>\Uleft</code> , <code>\Umiddle</code> and <code>\Uright</code>	126
8.7.9	Accents: <code>\mathlimitsmode</code>	126
8.8	Extracting values	127
8.8.1	Codes and using <code>\Umathcode</code> , <code>\Umathcharclass</code> , <code>\Umathcharfam</code> and <code>\Umathcharslot</code>	127
8.8.2	Last lines and <code>\predisplaygapfactor</code>	127
8.9	Math mode	128
8.9.1	Verbose versions of single-character math commands like <code>\Usuperscript</code> and <code>\Usubscript</code>	128
8.9.2	Script commands <code>\Unosuperscript</code> and <code>\Unosubscript</code>	128
8.9.3	Injecting primes with <code>Uprimescript</code>	128
8.9.4	Prescripts with <code>\USuperprescript</code> and <code>Usubprescript</code>	128
8.9.5	Allowed math commands in non-math modes	129
8.10	Goodies	129
8.10.1	Less Tracing	129
8.11	Experiments	129
8.11.1	Forcing classes with <code>\Umathclass</code>	130
8.11.2	Scaling spacing with <code>\Umathxscale</code> and <code>\Umathyscale</code>	130
<b>9</b>	<b>Building paragraphs and pages</b>	<b>131</b>
9.1	Introduction	131
9.2	Paragraphs	131
9.3	Inserts	131
9.4	Marks	132
9.5	Adjusts	132
9.6	Migration	133
9.7	Pages	133
9.8	Local boxes	133
<b>10</b>	<b>Nodes</b>	<b>135</b>
10.1	Lua node representation	135
10.2	Main text nodes	136
10.2.1	<code>hlist</code> and <code>vlist</code> nodes	136
10.2.2	rule nodes	137
10.2.3	insert nodes	138
10.2.4	mark nodes	138
10.2.5	adjust nodes	138
10.2.6	disc nodes	139



10.2.7	math nodes	139
10.2.8	glue nodes	140
10.2.9	glue_spec nodes	140
10.2.10	kern nodes	141
10.2.11	penalty nodes	141
10.2.12	glyph nodes	142
10.2.13	boundary nodes	143
10.2.14	par nodes	143
10.2.15	dir nodes	143
10.2.16	Whatsits	144
10.2.17	Math noads	144
10.3	The node library	148
10.3.1	Introduction	148
10.3.2	Housekeeping	149
10.3.3	Manipulating lists	152
10.3.4	Glue handling	156
10.3.5	Attribute handling	157
10.3.6	Glyph handling	159
10.3.7	Packaging	160
10.3.8	Math	162
10.4	Two access models	162
10.5	Normalization	169
10.6	Properties	169
<b>11</b>	<b>Lua callbacks</b>	<b>173</b>
11.1	Registering callbacks	173
11.2	File related callbacks	174
11.2.1	find_format_file and find_log_file	174
11.2.2	open_data_file	174
11.3	Data processing callbacks	174
11.3.1	process_jobname	174
11.4	Node list processing callbacks	174
11.4.1	contribute_filter	174
11.4.2	buildpage_filter	175
11.4.3	build_page_insert	175
11.4.4	pre_linebreak_filter	176
11.4.5	linebreak_filter	177
11.4.6	append_to_vlist_filter	177
11.4.7	post_linebreak_filter	177
11.4.8	glyph_run	177
11.4.9	hpack_filter	178
11.4.10	vpack_filter	178
11.4.11	hpack_quality	178
11.4.12	vpack_quality	179
11.4.13	process_rule	179
11.4.14	pre_output_filter	179
11.4.15	hyphenate	179



11.4.16	ligaturing	179
11.4.17	kerning	180
11.4.18	insert_par	180
11.4.19	mlist_to_hlist	180
11.5	Information reporting callbacks	181
11.5.1	pre_dump	181
11.5.2	start_run	181
11.5.3	stop_run	181
11.5.4	intercept_tex_error, intercept_lua_error	181
11.5.5	show_error_message and show_warning_message	181
11.5.6	start_file	182
11.5.7	stop_file	182
11.5.8	wrapup_run	182
11.6	Font-related callbacks	182
11.6.1	define_font	182
11.6.2	show_whatsit	183
<b>12</b>	<b>The TeX related libraries</b>	<b>185</b>
12.1	The lua library	185
12.1.1	Version information	185
12.1.2	Table allocators	185
12.1.3	Bytecode registers	185
12.1.4	Introspection	186
12.2	The status library	186
12.3	The tex library	195
12.3.1	Introduction	195
12.3.2	Internal parameter values, set and get	195
12.3.3	Convert commands	198
12.3.4	Item commands	198
12.3.5	Accessing registers: set*, get* and is*	198
12.3.6	Character code registers: [get set]*code[s]	200
12.3.7	Box registers: [get set]box	201
12.3.8	triggerbuildpage	202
12.3.9	splitbox	202
12.3.10	Accessing math parameters: [get set]math	202
12.3.11	Special list heads: [get set]list	203
12.3.12	Semantic nest levels: getnest and ptr	204
12.3.13	Print functions	205
12.3.14	Helper functions	207
12.3.15	Functions for dealing with primitives	210
12.3.16	Core functionality interfaces	215
12.3.17	Functions related to synctex	217
12.4	The texconfig table	217



12.5	The texio library	218
12.5.1	write and writeselector	218
12.5.2	writenl and writeselectornl	219
12.5.3	setescape	219
12.5.4	closeinput	219
12.6	The token library	219
12.6.1	The scanner	219
12.6.2	Picking up one token	222
12.6.3	Creating tokens	222
12.6.4	Macros	223
12.6.5	Pushing back	224
12.6.6	Nota bene	225
<b>13</b>	<b>The MetaPost library mplib</b>	<b>227</b>
13.1	Introduction	227
13.2	Process management	227
13.2.1	new	227
13.2.2	getstatistics	229
13.2.3	execute	229
13.2.4	finish	230
13.2.5	settolerance and gettolerance	230
13.2.6	Errors	230
13.2.7	The scanner status	230
13.2.8	The hash	230
13.2.9	Callbacks	231
13.3	The end result	231
13.3.1	The figure	231
13.3.2	fill	232
13.3.3	outline	232
13.3.4	start_bounds, start_clip, start_group	232
13.3.5	stop_bounds, stop_clip, stop_group	233
13.4	Subsidiary table formats	233
13.4.1	Paths and pens	233
13.4.2	Colors	233
13.4.3	Transforms	233
13.4.4	Dashes	234
13.4.5	Pens and peninfo	234
13.4.6	Character size information	234
13.5	Scanners	235
13.6	Injectors	236
13.7	To be checked	237
<b>14</b>	<b>The pdf related libraries</b>	<b>239</b>
14.1	The pdfe library	239
14.1.1	Introduction	239
14.1.2	open, openfile, new, getstatus, close, unencrypt	239
14.1.3	getsize, getversion, getnofobjects, getnofpages	240



14.1.4	get[catalog trailer info]	240
14.1.5	getpage, getbox	240
14.1.6	get[string integer number boolean name]	240
14.1.7	get[dictionary array stream]	241
14.1.8	[open close readfrom whole ]stream	241
14.1.9	getfrom[dictionary array]	242
14.1.10	[dictionary array]totable	242
14.1.11	getfromreference	242
14.2	Memory streams	243
14.3	The pdfscanner library	243
<b>15</b>	<b>Extra libraries</b>	<b>245</b>
15.1	Introduction	245
15.2	File and string readers: fio and type sio	245
15.3	md5	245
15.4	sha2	246
15.5	xzip	246
15.6	xmath	246
15.7	xcomplex	248
15.8	xdecimal	249
15.9	lfs	249
15.10	pngdecode	250
15.11	basexx	250
15.12	Multibyte string functions	251
15.13	Extra os library functions	252
15.14	The lua library functions	252
	<b>Primitive codes</b>	<b>255</b>
	<b>Topics</b>	<b>275</b>
	<b>Primitives</b>	<b>279</b>
	<b>Callbacks</b>	<b>285</b>
	<b>Nodes</b>	<b>287</b>
	<b>Libraries</b>	<b>289</b>
	<b>Statistics</b>	<b>297</b>
	<b>Some remarks</b>	<b>299</b>





# Introduction

Around 2005 we started the LuaTeX project and it took about a decade to reach a state where we could consider the experiments to have reached a stable state. Pretty soon LuaTeX could be used in production, even if some of the interfaces evolved, but ConTeXt was kept in sync so that was not really a problem. In 2018 the functionality was more or less frozen. Of course we might add some features in due time but nothing fundamental will change as we consider version 1.10 to be reasonable feature complete. Among the reasons is that this engine is now used outside ConTeXt too which means that we cannot simply change much without affecting other macro packages.

In reaching that state some decisions were delayed because they didn't go well with a current stable version. This is why at the 2018 ConTeXt meeting those present agreed that we could move on with a follow up tagged MetaTeX, a name we already had in mind for a while, but as Lua is an important component, it got expanded to LuaMetaTeX. This follow up is a lightweight companion to LuaTeX that will be maintained alongside. More about the reasons for this follow up as well as the philosophy behind it can be found in the document(s) describing the development. During LuaTeX development I kept track of what happened in a series of documents, parts of which were published as articles in user group journals, but all are in the ConTeXt distribution. I did the same with the development of LuaMetaTeX.

The LuaMetaTeX engine is, as said, a follow up on LuaTeX. Just as we have ConTeXt MkII for pdfTeX and XeTeX, we have MkIV for LuaTeX so for LuaMetaTeX we have yet another version of ConTeXt: LMTX. By freezing MkII, and at some point freezing MkIV, we can move on as we like, but we try to remain downward compatible where possible, something that the user interface makes possible. Although LuaMetaTeX can be used for production we can also use it for possibly drastic experiments but without affecting LuaTeX. Because we can easily adapt ConTeXt to support both, no other macro package will be harmed when (for instance) the interface that the engine provides change as part of an experiment or cleanup of code. Of course, when we consider something to be useful, it can be ported back to LuaTeX, but only when there are good reasons for doing so and when no compatibility issues are involved.

By now the code of these two related engines differs a lot so in retrospect it makes less sense to waste time on porting back. When considering this follow up one consideration was that a lean and mean version with an extension mechanism is a bit closer to original TeX. Of course, because we also have new primitives, this is not entirely true. The basic algorithms remain the same but code got reshuffled and because we expose internal names of variables and such that is reflected in the code base (like more granularity in nodes and token commands). Delegating tasks to Lua already meant that some aspects, especially system dependent ones, no longer made sense and therefore had consequences for the interface at the system level. In LuaMetaTeX more got delegated, like all file related operations. The penalty of moving even more responsibility to Lua has been compensated by (hopefully) harmless optimization of code in the engine and some more core functionality. In the process system dependencies have been minimalized.

One side effect of opening up is that what normally is hidden gets exposed and this is also true for all kind of codes that are used internally to distinguish states and properties of commands, tokens, nodes and more. Especially during development these can change but the good news is that they can be queried so on can write in code independent ways (in LuaTeX node id's are



an example). That also means more interface related commands, so again lean and mean is not applicable here, especially because the detailed control over the text, math, font and language subsystems also results in additional commands to query their state. And, as the MetaPost got extended, that subsystem is on the one hand leaner and meaner because backend code was dropped but on the other hand got a larger code base due to opening up the scanner and adding a feedback mechanism.

This manual started as an adaptation of the Lua<sub>T</sub><sub>E</sub><sub>X</sub> manual and therefore looks similar. Some chapters are removed, others were added and the rest has been (and will be further) adapted. It also discusses the (main) differences. Some of the new primitives or functions that show up in LuaMeta<sub>T</sub><sub>E</sub><sub>X</sub> might show up in Lua<sub>T</sub><sub>E</sub><sub>X</sub> at some point, but most will be exclusive to LuaMeta<sub>T</sub><sub>E</sub><sub>X</sub>, so don't take this manual as reference for Lua<sub>T</sub><sub>E</sub><sub>X</sub>! As long as we're experimenting we can change things at will but as we keep Con<sub>T</sub><sub>E</sub><sub>X</sub>t LMTX synchronized users normally won't notice this. Often you can find examples of usage in Con<sub>T</sub><sub>E</sub><sub>X</sub>t related documents and the source code so that serves a reference too. More detailed explanations can be found in documents in the Con<sub>T</sub><sub>E</sub><sub>X</sub>t distribution, if only because there we can present features in the perspective of useability.

For Con<sub>T</sub><sub>E</sub><sub>X</sub>t users the LuaMeta<sub>T</sub><sub>E</sub><sub>X</sub> engine will become the default. As mentioned, the Con<sub>T</sub><sub>E</sub><sub>X</sub>t variant for this engine is tagged LMTX. The pair can be used in production, just as with Lua<sub>T</sub><sub>E</sub><sub>X</sub> and MkIV. In fact, most users will probably not really notice the difference. In some cases there will be a drop in performance, due to more work being delegated to Lua, but on the average performance is much better, due to some changes below the hood of the engine. Memory consumption is also less. The timeline of development is roughly: from 2018 upto 2020 engine development, 2019 upto 2021 the stepwise code split between MkIV and LMTX, while in 2021 and 2022 we will (mostly) freeze MkIV and LMTX will be the default.

As this follow up is closely related to Con<sub>T</sub><sub>E</sub><sub>X</sub>t development, and because we expect stock Lua<sub>T</sub><sub>E</sub><sub>X</sub> to be used outside the Con<sub>T</sub><sub>E</sub><sub>X</sub>t proper, there will be no special mailing list nor coverage (or pollution) on the Lua<sub>T</sub><sub>E</sub><sub>X</sub> related mailing lists. We have the Con<sub>T</sub><sub>E</sub><sub>X</sub>t mailing lists for that. In due time the source code will be part of the regular Con<sub>T</sub><sub>E</sub><sub>X</sub>t distribution so that is then also the reference implementation: if needed users can compile the binary themselves.

This manual sometimes refers to Lua<sub>T</sub><sub>E</sub><sub>X</sub>, especially when we talk of features common to both engine, as well as to LuaMeta<sub>T</sub><sub>E</sub><sub>X</sub>, when it is more specific to the follow up. A substantial amount of time went into the transition and more will go in, so if you want to complain about LuaMeta<sub>T</sub><sub>E</sub><sub>X</sub>, don't bother me. Of course, if you really need professional support with these engines (or <sub>T</sub><sub>E</sub><sub>X</sub> in general), you can always consider contacting the developers.

Hans Hagen

LuaMeta<sub>T</sub><sub>E</sub><sub>X</sub> Banner : luametate<sub>X</sub> 2.0942 / 20220308

LuaMeta<sub>T</sub><sub>E</sub><sub>X</sub> Version : March 10, 2022

Con<sub>T</sub><sub>E</sub><sub>X</sub>t Version : LMTX 2022.03.10 17:22

Lua<sub>T</sub><sub>E</sub><sub>X</sub> Team : Hans Hagen, Hartmut Henkel, Taco Hoekwater, Luigi Scarso

LuaMeta<sub>T</sub><sub>E</sub><sub>X</sub> Team : Hans Hagen, Alan Braslau, Mojca Miklavc and Wolfgang Schuster





# 1 The internals

This is a reference manual and not a tutorial. This means that we discuss changes relative to traditional  $\text{\TeX}$  and also present new (or extended) functionality. As a consequence we will refer to concepts that we assume to be known or that might be explained later. Because the  $\text{\LuaTeX}$  and  $\text{\LuaMetaTeX}$  engines open up  $\text{\TeX}$  there's suddenly quite some more to explain, especially about the way a (to be) typeset stream moves through the machinery. However, discussing all that in detail makes not much sense, because deep knowledge is only relevant for those who write code not possible with regular  $\text{\TeX}$  and who are already familiar with these internals (or willing to spend time on figuring it out).

So, the average user doesn't need to know much about what is in this manual. For instance fonts and languages are normally dealt with in the macro package that you use. Messing around with node lists is also often not really needed at the user level. If you do mess around, you'd better know what you're dealing with. Reading “The  $\text{\TeX}$  Book” by Donald Knuth is a good investment of time then also because it's good to know where it all started. A more summarizing overview is given by “ $\text{\TeX}$  by Topic” by Victor Eijkhout. You might want to peek in “The  $\varepsilon\text{\TeX}$  manual” too.

But ... if you're here because of Lua, then all you need to know is that you can call it from within a run. If you want to learn the language, just read the well written Lua book. The macro package that you use probably will provide a few wrapper mechanisms but the basic `\directlua` command that does the job is:

```
\directlua{tex.print("Hi there")}
```

You can put code between curly braces but if it's a lot you can also put it in a file and load that file with the usual Lua commands. If you don't know what this means, you definitely need to have a look at the Lua book first.

If you still decide to read on, then it's good to know what nodes are, so we do a quick introduction here. If you input this text:

```
Hi There ...
```

eventually we will get a linked lists of nodes, which in ascii art looks like:

```
H <=> i <=> [glue] <=> T <=> h <=> e <=> r <=> e ...
```

When we have a paragraph, we actually get something like this, where a `par` node stores some metadata and is followed by a `hlist` flagged as indent box:

```
[par] <=> [hlist] <=> H <=> i <=> [glue] <=> T <=> h <=> e <=> r <=> e ...
```

Each character becomes a so called glyph node, a record with properties like the current font, the character code and the current language. Spaces become glue nodes. There are many node types and nodes can have many properties but that will be discussed later. Each node points back to a previous node or next node, given that these exist. Sometimes multiple characters are represented by one glyph (shape), so one can also get:

```
[par] <=> [hlist] <=> H <=> i <=> [glue] <=> Th <=> e <=> r <=> e ...
```



And maybe some characters get positioned relative to each other, so we might see:

```
[par] <=> [hlist] <=> H <=> [kern] <=> i <=> [glue] <=> Th <=> e <=> r <=> e ...
```

Actually, the above representation is one view, because in LuaMetaT<sub>E</sub>X we can choose for this:

```
[par] <=> [glue] <=> H <=> [kern] <=> i <=> [glue] <=> Th <=> e <=> r <=> e ...
```

where glue (currently fixed) is used instead of an empty hlist (think of a `\hbox`). Options like this are available because want a certain view on these lists from the Lua end and the result being predicable is part of that.

It's also good to know beforehand that T<sub>E</sub>X is basically centered around creating paragraphs and pages. The par builder takes a list and breaks it into lines. At some point horizontal blobs are wrapped into vertical ones. Lines are so called boxes and can be separated by glue, penalties and more. The page builder accumulates lines and when feasible triggers an output routine that will take the list so far. Constructing the actual page is not part of T<sub>E</sub>X but done using primitives that permit manipulation of boxes. The result is handled back to T<sub>E</sub>X and flushed to a (often pdf) file.

```
\setbox\scratchbox\vbox\bgroup
  line 1\par line 2
\egroup
```

```
\showbox\scratchbox
```

The above code produces the next log lines that reveal how the engines sees a paragraph (wrapped in a `\vbox`):

```
1:4: > \box257=
1:4: \vbox[normal][16=1,17=1,47=1], width 483.69687, height 27.58083, depth 0.1416, direction l2r
1:4: .\list
1:4: ..\hbox[line][16=1,17=1,47=1], width 483.69687, height 7.59766, depth 0.1416, glue 455.40097fil, direction l2r
1:4: ...list
1:4: ....\glue[left hang][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[left][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[parfillleft][16=1,17=1,47=1] 0.0pt
1:4: ....\par[newgraf][16=1,17=1,47=1], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 3000, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000, finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, emergencystretch 12.0, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
1:4: ....\glue[indent][16=1,17=1,47=1] 0.0pt
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+00006C l
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+000069 i
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+00006E n
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+000065 e
1:4: ....\glue[space][16=1,17=1,47=1] 3.17871pt plus 1.58936pt minus 1.05957pt, font 30
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+000031 1
1:4: ....\penalty[line][16=1,17=1,47=1] 10000
1:4: ....\glue[parfill][16=1,17=1,47=1] 0.0pt plus 1.0fil
```



```

1:4: ....\glue[right][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[right hang][16=1,17=1,47=1] 0.0pt
1:4: ..\glue[par][16=1,17=1,47=1] 5.44995pt plus 1.81665pt minus 1.81665pt
1:4: ..\glue[baseline][16=1,17=1,47=1] 6.79396pt
1:4: ..\hbox[line][16=1,17=1,47=1], width 483.69687, height 7.59766, depth 0.1416, glue 455.40097fil, direction l2r
1:4: ...list
1:4: ....\glue[left hang][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[left][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[parfillleft][16=1,17=1,47=1] 0.0pt
1:4: ....\par[newgraf][16=1,17=1,47=1], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 3000, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000, finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, emergencystretch 12.0, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
1:4: ....\glue[indent][16=1,17=1,47=1] 0.0pt
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+00006C l
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+000069 i
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+00006E n
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+000065 e
1:4: ....\glue[space][16=1,17=1,47=1] 3.17871pt plus 1.58936pt minus 1.05957pt, font 30
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+000032 2
1:4: ....\penalty[line][16=1,17=1,47=1] 10000
1:4: ....\glue[parfill][16=1,17=1,47=1] 0.0pt plus 1.0fil
1:4: ....\glue[right][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[right hang][16=1,17=1,47=1] 0.0pt

```

The Lua $\TeX$  engine provides hooks for Lua code at nearly every reasonable point in the process: collecting content, hyphenating, applying font features, breaking into lines, etc. This means that you can overload  $\TeX$ 's natural behaviour, which still is the benchmark. When we refer to ‘callbacks’ we means these hooks. The  $\TeX$  engine itself is pretty well optimized but when you kick in much Lua code, you will notices that performance drops. Don't blame and bother the authors with performance issues. In Con $\TeX$ t over 50% of the time can be spent in Lua, but so far we didn't get many complaints about efficiency. Adding more callbacks makes no sense, also because at some point the performance hit gets too large. There are plenty of ways to achieve goals. For that reason: take remarks about Lua $\TeX$ , features, potential, performance etc. with a natural grain of salt.

Where plain  $\TeX$  is basically a basic framework for writing a specific style, macro packages like Con $\TeX$ t and L<sup>A</sup> $\TeX$  provide the user a whole lot of additional tools to make documents look good. They hide the dirty details of font management, language support, turning structure into typeset results, wrapping pages, including images, and so on. You should be aware of the fact that when you hook in your own code to manipulate lists, this can interfere with the macro package that you use. Each successive step expects a certain result and if you mess around too much, the engine eventually might bark and quit. It can even crash, because testing everywhere for what users can do wrong is no real option.

When you read about nodes in the following chapters it's good to keep in mind what commands relate to them. Here are a few:

COMMAND	NODE	EXPLANATION
<code>\hbox</code>	hlist	horizontal box



<code>\vbox</code>	<code>vlist</code>	vertical box with the baseline at the bottom
<code>\vtop</code>	<code>vlist</code>	vertical box with the baseline at the top
<code>\hskip</code>	<code>glue</code>	horizontal skip with optional stretch and shrink
<code>\vskip</code>	<code>glue</code>	vertical skip with optional stretch and shrink
<code>\kern</code>	<code>kern</code>	horizontal or vertical fixed skip
<code>\discretionary</code>	<code>disc</code>	hyphenation point (pre, post, replace)
<code>\char</code>	<code>glyph</code>	a character
<code>\hrule</code>	<code>rule</code>	a horizontal rule
<code>\vrule</code>	<code>rule</code>	a vertical rule
<code>\textdirection</code>	<code>dir</code>	a change in text direction

---

Whatever we feed into  $\text{T}_{\text{E}}\text{X}$  at some point becomes a token which is either interpreted directly or stored in a linked list. A token is just a number that encodes a specific command (operator) and some value (operand) that further specifies what that command is supposed to do. In addition to an interface to nodes, there is an interface to tokens, as later chapters will demonstrate.

Text (interspersed with macros) comes from an input medium. This can be a file, token list, macro body cq. arguments, some internal quantity (like a number), Lua, etc. Macros get expanded. In the process  $\text{T}_{\text{E}}\text{X}$  can enter a group. Inside the group, changes to registers get saved on a stack, and restored after leaving the group. When conditionals are encountered, another kind of nesting happens, and again there is a stack involved. Tokens, expansion, stacks, input levels are all terms used in the next chapters. Don't worry, they lose their magic once you use  $\text{T}_{\text{E}}\text{X}$  a lot. You have access to most of the internals and when not, at least it is possible to query some state we're in or level we're at.

When we talk about pack(ag)ing it can mean two things. When  $\text{T}_{\text{E}}\text{X}$  has consumed some tokens that represent text they are added to the current list. When the text is put into a so called `\hbox` (for instance a line in a paragraph) it (normally) first gets hyphenated, next ligatures are build, and finally kerns are added. Each of these stages can be overloaded using Lua code. When these three stages are finished, the dimension of the content is calculated and the box gets its width, height and depth. What happens with the box depends on what macros do with it.

The other thing that can happen is that the text starts a new paragraph. In that case some information is stored in a leading par node. Then indentation is appended and the paragraph ends with some glue. Again the three stages are applied but this time afterwards, the long line is broken into lines and the result is either added to the content of a box or to the main vertical list (the running text so to say). This is called par building. At some point  $\text{T}_{\text{E}}\text{X}$  decides that enough is enough and it will trigger the page builder. So, building is another concept we will encounter. Another example of a builder is the one that turns an intermediate math list into something typeset.

Wrapping something in a box is called packing. Adding something to a list is described in terms of contributing. The more complicated processes are wrapped into builders. For now this should be enough to enable you to understand the next chapters. The text is not as enlightening and entertaining as Don Knuths books, sorry.



## 2 Differences with LuaTeX

As LuaMetaTeX is a leaner and meaner LuaTeX. This means that substantial parts and dependencies are gone: quite some font code, all backend code with related frontend code and of course image and font inclusion. There is also new functionality which makes for less lean but in the end we still have less, also in terms of dependencies. This chapter will discuss what is gone. We start with the primitives that were dropped.

<b>fonts</b>	<code>\letterspacefont \copyfont \expandglyphsinfont \ignoreligaturesinfont \tagcode \leftghost \rightghost</code>
<b>backend</b>	<code>\dviextension \dvivariable \dvifedback \pdfextension \pdfvariable \pdffeedback \dviextension \draftmode \outputmode</code>
<b>dimensions</b>	<code>\pageleftoffset \pagerightoffset \pagetopoffset \pagebottomoffset \pageheight \pagewidth</code>
<b>resources</b>	<code>\saveboxresource \useboxresource \lastsavedboxresourceindex \saveimageresource \useimageresource \lastsavedimageresourceindex \lastsavedimageresourcepages</code>
<b>positioning</b>	<code>\savepos \lastxpos \lastypos</code>
<b>directions</b>	<code>\textdir \linedir \mathdir \pardir \pagedir \bodydir \pagedirection \bodydirection</code>
<b>randomizer</b>	<code>\randomseed \setrandomseed \normaldeviate \uniformdeviate</code>
<b>utilities</b>	<code>\synctex</code>
<b>extensions</b>	<code>\latelua \lateluafunction \openout \write \closeout \openin \read \readline \closein \ifeof</code>
<b>control</b>	<code>\suppressfontnotfounderror \suppresslongerror \suppressprimitiveerror \suppressmathparerror \suppressifcsnameerror \suppressoutererror \mathoption</code>
<b>system</b>	<code>\primitive \ifprimitive \formatname</code>
<b>ignored</b>	<code>\long \outer \mag</code>

The math machinery has been overhauled stepwise. In the process detailed control has been added but later some of that got removed or replaced. The engine now assumes that OpenType fonts are used but you do need to set up the engine properly, something that has to be done with respect to fonts anyway. By enabling and/disabling certain features you can emulate the traditional engine. Font parameters no longer are taken from the traditional parameters when they are not set. We just assume properly passed so called math constants and quite a few new ones have been added.

The resources and positioning primitives are actually useful but can be defined as macros that (via Lua) inject nodes in the input that suit the macro package and backend. The three-letter direction primitives are gone and the numeric variants are now leading. There is no need for page and body related directions and they don't work well in LuaTeX anyway. We only have two directions left. Because we can hook in Lua functions that get information about what is expected (consumer or provider) there are plenty possibilities for adding functionality using this scripting language.



The primitive related extensions were not that useful and reliable so they have been removed. There are some new variants that will be discussed later. The `\outer` and `\long` prefixes are gone as they don't make much sense nowadays and them becoming dummies opened the way to something new: control sequence properties that permit protection against as well as controlled overloading of definitions. I don't think that (ConT<sub>E</sub>Xt) users will notice these prefixes being gone. The definition and parsing related `\suppress...` features are now default and can't be changed so related primitives are gone.

The `\shipout` primitive does no ship out but just erases the content of the box unless of course that has happened already in another way. A macro package should implement its own backend and related shipout. Talking of backend, the extension primitives that relate to backends can be implemented as part of a backend design using generic whatsits. There is only one type of whatsit now. In fact we're now closer to original T<sub>E</sub>X with respect to the extensions.

The `img` library has been removed as it's rather bound to the backend. The `slunicode` library is also gone. There are some helpers in the string library that can be used instead and one can write additional Lua code if needed. There is no longer a pdf backend library but we have an up to date pdf parsing library on board.

In the `node`, `tex` and `status` library we no longer have helpers and variables that relate to the backend. The LuaMetaT<sub>E</sub>X engine is in principle dvi and pdf unaware. There are, as mentioned, only generic whatsit nodes that can be used for some management related tasks. For instance you can use them to implement user nodes. More extensive status information is provided in the overhauled status library. All libraries have additional functionality and names of functions have been normalized (for as far as possible).

The margin kern nodes are gone and we now use regular kern nodes for them. As a consequence there are two extra subtypes indicating the injected left or right kern. The glyph field served no real purpose so there was no reason for a special kind of node.

The `kpse` library is no longer built-in, but one can use an external `kpse` library, assuming that it is present on the system, because the engine has a so called optional library interface to it. Because there is no backend, quite some file related callbacks could go away. The following file related callbacks remained (till now):

```
find_write_file find_format_file open_data_file
```

The callbacks related to errors are changed:

```
intercept_tex_error intercept_lua_error  
show_error_message show_warning_message
```

There is a hook that gets called when one of the fundamental memory structures gets reallocated.

```
trace_memory
```

When you use the overload protect mechanisms, a callback can be plugged in to handle exceptions:

```
handle_overload
```

The (job) management hooks are kept:



```
process_jobname
start_run stop_run wrapup_run
pre_dump
start_file stop_file
```

Because we use a more generic whatsit model, there is a new callback:

```
show_whatsit
```

Because tracing boxes now reports a lot more information, we have a plug in for detail:

```
get_attribute
```

Being the core of extensibility, the typesetting callbacks of course stayed. This is what we ended up with:

```
alignment_filter, append_line_filter, append_to_vlist_filter, begin_paragraph,
build_page_insert, buildpage_filter, define_font, find_format_file,
find_log_file, get_attribute, glyph_run, handle_overload, hpack_filter,
hpack_quality, hyphenate, insert_par, intercept_lua_error, intercept_tex_error,
kerning, ligaturing, linebreak_filter, local_box_filter, make_extensible,
math_rule, missing_character, mlist_to_hlist, open_data_file, paragraph_context,
post_linebreak_filter, pre_dump, pre_linebreak_filter, pre_output_filter,
process_jobname, register_extensible, show_error_message, show_lua_call,
show_warning_message, show_whatsit, start_file, start_run, stop_file, stop_run,
trace_memory, vpack_filter, vpack_quality, wrapup_run
```

As in Lua $\TeX$  font loading happens with the following callback. This time it really needs to be set because there is no built-in font loader.

```
define_font
```

There are all kinds of subtle differences in the implementation, for instance we no longer intercept `*` and `&` as these were already replaced long ago in  $\TeX$  engines by command line options. Talking of options, only a few are left. All input goes via Lua, even the console. One can program a terminal if needed.

We took our time for reaching a stable state in Lua $\TeX$ . Among the reasons is the fact that most was experimented with in Con $\TeX$ t, which we can adapt to the engine as we go. It took many years to decide what to keep and how to do things. Of course there are places when things can be improved but that most likely only happens in LuaMeta $\TeX$ . Contrary to what is sometimes suggested, the Lua $\TeX$ -Con $\TeX$ t MkIV combination (assuming matched versions) has been quite stable. It made no sense otherwise. Most Con $\TeX$ t functionality didn't change much at the user level. Of course there have been issues, as is natural with everything new and beta, but we have a fast update cycle.

The same is true for LuaMeta $\TeX$  and Con $\TeX$ t LMTX: it can be used for production as usual and in practice Con $\TeX$ t users tend to use the beta releases, which proves this. Of course, if you use low level features that are experimental you're on your own. Also, as with Lua $\TeX$  it might take many years before a long term stable is defined. The good news is that, when the source code





has become part of the ConT<sub>E</sub>Xt distribution, there is always a properly working, more or less long term stable, snapshot.

The error reporting subsystem has been redone quite a bit but is still fundamentally the same. We don't really assume interactive usage but if someone uses it, it might be noticed that it is not possible to backtrack or inject something. Of course it is no big deal to implement all that in Lua if needed. It removes a system dependency and makes for a bit cleaner code. In ConT<sub>E</sub>Xt we quit on an error simply because one has to fix source anyway and runs are fast enough. Logging provides more detail and new primitives can be used to prevent clutter in tracing (the more complex a macro package becomes, the more extreme tracing becomes).

There are new primitives as well as some extensions to existing primitive functionality. These are described in following chapters but there might be hidden treasures in the binary. If you locate them, don't automatically assume them to stay, some might be part of experiments! There are for instance a few csname related definers, we have integer and dimension constants, the macro argument parser can be brought in tolerant mode, the repertoire of conditionals has been extended, some internals can be controlled (think of normalization of lines, hyphenation etc.), and macros can be protected against user overload. Not all is discussed in detail in this manual but there are introductions in the ConT<sub>E</sub>Xt distribution that explain them. But the T<sub>E</sub>X kernel is of course omnipresent.

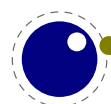
The following primitives are available in LuaT<sub>E</sub>X but not in LuaMetaT<sub>E</sub>X. Some of these are emulated in ConT<sub>E</sub>Xt.

Umathbinbinspacing	Umathopbinspacing
Umathbinclonespacing	Umathopclonespacing
Umathbininnerspacing	Umathopenbinspacing
Umathbinopenspacing	Umathopenclosespacing
Umathbinopspacing	Umathopeninnerspacing
Umathbinordspacing	Umathopenopenspacing
Umathbinpunctspacing	Umathopenopspacing
Umathbinrelspacing	Umathopenordspacing
Umathclosebinspacing	Umathopenpunctspacing
Umathcloseclonespacing	Umathopenrelspacing
Umathcloseinnerspacing	Umathoppinnerspacing
Umathcloseopenspacing	Umathopopenspacing
Umathcloseopspacing	Umathopopspacing
Umathcloseordspacing	Umathopordspacing
Umathclosepunctspacing	Umathoppunctspacing
Umathcloserelspacing	Umathoprelspacing
Umathinnerbinspacing	Umathordbinspacing
Umathinnerclonespacing	Umathordclonespacing
Umathinnerinnerspacing	Umathordinnerspacing
Umathinneropenspacing	Umathordopenspacing
Umathinneropspacing	Umathordopspacing
Umathinnerordspacing	Umathordordspacing
Umathinnerpunctspacing	Umathordpunctspacing
Umathinnerrelspacing	Umathordrelspacing





Umathpunctbinspacing	immediateassignment
Umathpunctclosespacing	insertht
Umathpunctinnerspacing	lastsavedboxresourceindex
Umathpunctopenspacing	lastsavedimageresourceindex
Umathpunctopspacing	lastsavedimageresourcepages
Umathpunctordspacing	lastxpos
Umathpunctpunctspacing	lastypos
Umathpunctrelspacing	latelua
Umathrelbinspacing	lateluafunction
Umathrelclosespacing	leftghost
Umathrelinnerspacing	letterspacefont
Umathreloppspacing	linedir
Umathreloppspacing	mag
Umathrelordspacing	mathdefaultsmode
Umathrelpunctspacing	mathdir
Umathrelrelspacing	mathflattenmode
automatichyphenmode	mathitalicsmode
bodydir	mathoption
bodydirection	mathrulethicknessmode
boxdir	nokerns
breakafterdirmode	noligs
closein	nolocaldirs
closeout	nolocalwhatsits
compoundhyphenmode	normaldeviate
copyfont	openin
discretionaryligaturemode	openout
draftmode	outputmode
dviextension	pagebottomoffset
dvifedback	pagedir
dvivariable	pagedirection
eTeXVersion	pageheight
eTeXglueshrinkorder	pageleftoffset
eTeXgluestretchorder	pagerightoffset
eTeXminorversion	pagetopoffset
eTeXrevision	pagewidth
eTeXversion	pardir
expandglyphsinfont	pdfextension
fixupboxesmode	pdffeedback
glyphdimensionsmode	pdfvariable
hoffset	primitive
hyphenationbounds	randomseed
hyphenpenaltymode	read
ifeof	readline
ifprimitive	rightghost
ignoreligaturesinfont	saveboxresource
immediateassigned	saveimageresource



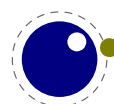
savepos	synctex
setrandomseed	tagcode
shapemode	texdir
special	tracingscantokens
suppressfontnotfounderror	uniformdeviate
suppressifcsnameerror	useboxresource
suppresslongerror	useimageresource
suppressmathparerror	voffset
suppressoutererror	write
suppressprimitiveerror	

The following primitives are available in LuaMetaTeX only. At some point in time some might be added to LuaTeX.

Uabove	Umathnolimits
Uabovewithdelims	Umathnumeratorvariant
Uatop	Umathopenupdepth
Uatopwithdelims	Umathopenupheight
Umathaccentbasedepth	Umathoverdelimitervariant
Umathaccentbaseheight	Umathoverlayaccentvariant
Umathaccentbottomshiftup	Umathoverlinevariant
Umathaccenttopshiftup	Umathphantom
Umathaccentvariant	Umathpresubshiftdistance
Umathadapttoleft	Umathpresupshiftdistance
Umathadapttoright	Umathprimeraise
Umathbotaccentvariant	Umathprimeshiftup
Umathclass	Umathprimespaceafter
Umathdegreevariant	Umathprimevariant
Umathdelimiterovervariant	Umathprimewidth
Umathdelimiterundervariant	Umathradicalvariant
Umathdenominatorvariant	Umathruleddepth
Umathextrasubpreshift	Umathruleheight
Umathextrasubprespace	Umathskeweddelimitertolerance
Umathextrasubshift	Umathspacebeforescript
Umathextrasubspace	Umathstackvariant
Umathextrasuppreshift	Umathsubscriptvariant
Umathextrasupprespace	Umathsubshiftdistance
Umathextrasupshift	Umathsuperscriptvariant
Umathextrasupspace	Umathsupshiftdistance
Umathflattenedaccentbasedepth	Umathtopaccentvariant
Umathflattenedaccentbaseheight	Umathunderdelimitervariant
Umathflattenedaccentbottomshiftup	Umathunderlinevariant
Umathflattenedaccenttopshiftup	Umathuseaxis
Umathfractionvariant	Umathvextensiblevariant
Umathhextensiblevariant	Umathvoid
Umathlimits	Umathxscale
Umathnoaxis	



Umathyscale	boxxoffset
Unosubprescript	boxymove
Unosuperprescript	boxyoffset
Uoperator	copymathatomrule
Uover	copymathparent
Uoverwithdelims	copymathspacing
Uprimescript	currentloopiterator
Ushiftedsubprescript	currentloopnesting
Ushiftedsubscript	currentmarks
Ushiftedsuperprescript	defcsname
Ushiftedsuperscript	dimensiondef
Ustyle	dimexpression
Usubprescript	edefcsname
Usuperprescript	endmathgroup
adjustspacingshrink	endsimplegroup
adjustspacingstep	enforced
adjustspacingstretch	etoks
afterassigned	everybeforepar
aftergrouped	everytab
aliased	expand
aligncontent	expandafterpars
allcrampedstyles	expandafterspaces
alldisplaystyles	expandcstoken
allmathstyles	expandedafter
allscriptscriptstyles	expandedloop
allscriptstyles	expandtoken
allsplitstyles	flushmarks
alltextstyles	fontmathcontrol
alluncrampedstyles	fontspecdef
atendofgroup	fontspecid
atendofgrouped	fontspecifiedname
automigrationmode	fontspecifiedsize
autoparagraphmode	fontspecscale
beginlocalcontrol	fontspecxscale
beginmathgroup	fontspecyscale
beginsimplegroup	fonttextcontrol
boxanchor	frozen
boxanchors	futurecsname
boxattribute	futuredef
boxgeometry	futureexpand
boxorientation	futureexpandis
boxshift	futureexpandisap
boxsource	gdefcsname
boxtarget	gletcsname
boxtotal	glettonothing
boxxmove	gluespecdef



glyph	insertdistance
glyphdatafield	insertheight
glyphoptions	insertheights
glyphscale	insertlimit
glyphscriptfield	insertmaxdepth
glyphscriptscale	insertmode
glyphscriptscriptscale	insertmultiplier
glyphstatefield	insertpenalty
glyphtextscale	insertprogress
glyphxoffset	insertstorage
glyphxscale	insertstoring
glyphyoffset	insertunbox
glyphyscale	insertuncopy
hccode	insertwidth
hyphenationmode	instance
ifarguments	integerdef
ifboolean	lastarguments
ifchkdim	lastchkdim
ifchknum	lastchknum
ifcmpdim	lastleftclass
ifcmpnum	lastloopiterator
ifcstok	lastnodesubtype
ifdimexpression	lastparcontext
ifdimval	lastrightclass
ifempty	letcsname
ifflags	letfrozen
ifhaschar	letmathatomrule
ifhastok	letmathparent
ifhastoks	letmathspacing
ifhasxtoks	letprotected
ifinsert	lettonothing
ifmathparameter	localcontrol
ifmathstyle	localcontrolled
ifnumexpression	localcontrolledloop
ifnumval	lcalleftboxbox
ifparameter	localmiddlebox
ifparameters	localmiddleboxbox
ifrelax	localrightboxbox
iftok	mathatom
ignorearguments	mathatomskip
ignorepars	mathbackwardpenalties
immutable	mathfenced
inherited	mathfencesmode
insertbox	mathfontcontrol
insertcopy	mathforwardpenalties
insertdepth	mathfrac



mathghost	scaledinterwordstretch
mathlimitsmode	scaledslantperpoint
mathmiddle	semiexpanded
mathrad	semiprotected
mathscale	setdefaultmathcodes
mathslackmode	setmathatomrule
mathspacingmode	setmathignore
mathstackstyle	setmathoptions
mathstylefontid	setmathpostpenalty
maththreshold	setmathprepenalty
meaningasis	setmathspacing
meaningfull	shapingpenaltiesmode
meaningless	shapingpenalty
mugluespecdef	shownodedetails
mutable	skewed
noaligned	skewedwithdelims
norelax	snapshotpar
normalizelinemode	srule
numericsscale	supmarkmode
numexpression	swapcsvalues
orelse	tabsize
orphanpenalties	thewithoutunit
orphanpenalty	tinymuskip
orunless	todimension
overloaded	tointeger
overloadmode	tokenized
overshoot	tolerant
pageboundary	tomathstyle
pageboundarypenalty	toscaled
pagevsize	tracingadjusts
parametercount	tracingalignments
parametermark	tracingexpressions
parattribute	tracingfullboxes
parfillleftskip	tracinghyphenation
permanent	tracinginserts
postinlinepenalty	tracinglevels
preinlinepenalty	tracingmarks
quitloop	tracingmath
resetmathspacing	tracingnodes
retokenized	uleaders
scaledemwidth	undent
scaledexheight	unexpandedloop
scaledextraspaces	unhpack
scaledfontdimen	unletfrozen
scaledinterwordshrink	unletprotected
scaledinterwordspace	untraced



unvpack  
wrapuppar

~~xdelkssname~~

As part of a bit more consistency some function names also changed. Names with an \_ got that removed (as that was the minority). It's easy to provide a back mapping if needed (just alias the functions).

*Todo: only mention the LuaTeX ones.*

LIBRARY	OLD NAME	NEW NAME	COMMENT
language	clear_patterns	clearpatterns	
	clear_hyphenation	clearhyphenation	
mplib	italcor	italic	
	pen_info	peninfo	
	solve_path	solvepath	
texio	write_nl	writenl	old name stays
node	protect_glyph	protectglyph	
	protect_glyphs	protectglyphs	
	unprotect_glyph	unprotectglyph	
	unprotect_glyphs	unprotectglyphs	
	end_of_math	endofmath	
	mlist_to_hlist	mlisttohlist	
	effective_glue	effectiveglue	
	has_glyph	hasglyph	
	first_glyph	firstglyph	
	has_field	hasfield	
	copy_list	copylist	
	flush_node	flushnode	
	flush_list	flushlist	
	insert_before	insertbefore	
	insert_after	insertafter	
	last_node	lastnode	
	is_zero_glue	iszeroglue	
	make_extensible	makeextensible	
	uses_font	usesfont	
	is_char	ischar	
	is_direct	isdirect	
	is_glyph	isglyph	
	is_node	isnode	
token	scan_keyword	scankeyword	
	scan_keywordcs	scankeywordcs	
	scan_int	scanint	
	scan_real	scanreal	
	scan_float	scanfloat	
	scan_dimen	scandimen	
	scan_glue	scanglue	
	scan_toks	scantoks	
	scan_code	scancode	



<code>scan_string</code>	<code>scanstring</code>
<code>scan_argument</code>	<code>scanargument</code>
<code>scan_word</code>	<code>scanword</code>
<code>scan_csname</code>	<code>scancsname</code>
<code>scan_list</code>	<code>scanlist</code>
<code>scan_box</code>	<code>scanbox</code>

---

It's all part of trying to make the code base consistent but it is sometimes a bit annoying. However, that's why we develop this engine independent of the LuaT<sub>E</sub>X code base. It's anyway a change that has been on my todo list for quite a while because those inconsistencies annoyed me. It might take some years to get all done.







## 3 The original engines

### 3.1 The merged engines

#### 3.1.1 The rationale

The first version of LuaTeX, made by Hartmut after we discussed the possibility of an extension language, only had a few extra primitives and it was largely the same as pdfTeX. It was presented to the public in 2005. As part of the Oriental TeX project, Taco merged some parts of Aleph into the code and some more primitives were added. Then we started more fundamental experiments. After many years, when the engine had become more stable, the decision was made to clean up the rather hybrid nature of the program. This means that some primitives were promoted to core primitives, often with a different name, and that others were removed. This also made it possible to start cleaning up the code base, which showed decades of stepwise additions to original TeX. In chapter 5 we discuss some new primitives, here we will cover most of the adapted ones.

During more than a decade stepwise new functionality was added and after 10 years the more or less stable version 1.0 was presented. But we continued and after some 15 years the LuaMetaTeX follow up entered its first testing stage. But before details about the engine are discussed in successive chapters, we first summarize where we started from. Keep in mind that in LuaMetaTeX we have a bit less than in LuaTeX, so this section differs from the one in the LuaTeX manual.

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

Again we stress that *this is not a TeX manual, nor a tutorial*. If you are unfamiliar with TeX first play a little with a macro package, take a look at the TeX book, make yourself familiar with the concepts and macro language. That will likely take days and not hours. Also, many of the new concepts introduced in LuaTeX and LuaMetaTeX are explained in documents that come with the ConTeXt distribution, articles and presentations. It doesn't pay off to repeat that here, especially not in a time when users often search instead of read from cover to cover.

Occasionally there are extensions to pdfTeX and LuaTeX but these are unlikely to end up in LuaMetaTeX. If needed one can add functionality using Lua. Another reason is that because the way we handle files and generate output being compatible would only harm the engine. We have some fundamental extensions that overcome limitations anyway. One area where there are significant changes is in logging: at some point it no longer made sense to be compatible (with LuaTeX) because we carry around more information.

#### 3.1.2 Changes from TeX 3.1415926...

Of course it all starts with traditional TeX. Even if we started with pdfTeX, most still comes from original Knuthian TeX. But we divert a bit.



- ▶ The current code base is written in C, not Pascal. The original web documentation is kept when possible and not wrapped in tagged comments. As a consequence instead of one large file plus change files, we now have multiple files organized in categories like `tex`, `lua`, `languages`, `fonts`, `libraries`, etc. There are some artifacts of the conversion to C, but these got (and get) removed stepwise. The documentation, which actually comes from the mix of engines (via so called change files), is a mix of what authors of the engines wove into the source, and most is of course from Don Knuths original. In `LuaMetaTeX` we try to stay as close as possible to the original so that the documentation of the fundamentals behind `TeX` by Don Knuth still applies. However, because we use C, some documentation is a bit off. Also, most global variables are now collected in structures, but the original names and level of abstraction were mostly kept. On the other hand, opening up had its impact on the code, so that makes some documentation a bit off too. Adapting that all will take time.
- ▶ See chapter 7 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. Also, the hyphenation, ligature building and kerning has been split so that we can hook in alternative or extra code wherever we like. There are various options to control discretionary injection and related penalties are now integrated in these nodes. Language information is now bound to glyphs. The number of languages in `LuaMetaTeX` is smaller than in `LuaTeX`. Control over discretionaries is more granular and now managed by less variables.
- ▶ There is no pool file, all strings are embedded during compilation. This also removed some memory constraints. We kept token and node memory management because it is convenient and efficient but parts were reimplemented in order to remove some constraints. Token memory management is largely the same. All the other large memory structures, like those related to nesting, the save stack, input levels, the hash table and table of equivalents, etc. now all start out small and are enlarged when needed, where maxima are controlled in the usual way. In principle the initial memory footprint is smaller while at the same time we can go real large. Because we have wide memory words some data (arrays) used for housekeeping could be reorganized a bit.
- ▶ The specifier `plus 1 filllll` does not generate an error. The extra 'l' is simply typeset.
- ▶ The upper limit to `\endlinechar` and `\newlinechar` is 127.
- ▶ Because the backend is not built-in, the magnification (`\mag`) primitive is gone. A `\shipout` command just discards the content of the given box. The write related primitives have to be implemented in the used macro package using Lua. None of the `pdfTeX` derived primitives is present.
- ▶ Because there is no font loader, a Lua variant is free to either support or not the Omega `ofm` file format. As there are hardly any such fonts it probably makes no sense. There is plenty of control over the way glyphs get treated and scaling of fonts and glyphs is also more dynamic.
- ▶ There is more control over some (formerly hard-coded) math properties. In fact, there is a whole extra bit of math related code because we need to deal with OpenType fonts. The math processing has been adapted to the new (dynamic) font and glyph scaling features.
- ▶ The `\outer` and `\long` prefixed are silently ignored. It is permitted to use `\par` in math.
- ▶ The lack of a backend means that some primitives related to it are not implemented. This is no big deal because it is possible to use the scanner library to implement them as needed, which depends on the macro package and backend.
- ▶ The math style related primitives can use numbers as well as symbolic names. There is some



more (control over) math anyway, which is a side effect of supporting OpenType math.

### 3.1.3 Changes from $\varepsilon$ -T<sub>E</sub>X 2.2

Being the de-facto standard extension of course we provide the  $\varepsilon$ -T<sub>E</sub>X features, but with a few small adaptations.

- ▶ The  $\varepsilon$ -T<sub>E</sub>X functionality is always present and enabled so the prepended asterisk or `-etex` switch for `iniTEX` is not needed.
- ▶ The T<sub>E</sub>X<sub>Xe</sub>T extension is not present, so the primitives `\TeXXeTstate`, `\beginR`, `\beginL`, `\endR` and `\endL` are missing. Instead we used the Omega/Aleph approach to directionality as starting point, albeit it has been changed quite a bit, so that we're probably not that far from T<sub>E</sub>X<sub>Xe</sub>T.
- ▶ Some of the tracing information that is output by  $\varepsilon$ -T<sub>E</sub>X's `\tracingassigns` and `\tracingrestores` is not there. Also keep in mind that tracing doesn't involve what Lua does.
- ▶ Register management in LuaMetaT<sub>E</sub>X uses the Omega/Aleph model, so the maximum value is 65535 and the implementation uses a flat array instead of the mixed flat & sparse model from  $\varepsilon$ -T<sub>E</sub>X.
- ▶ Because we have more nodes, conditionals, etc. the  $\varepsilon$ -T<sub>E</sub>X status related variables are adapted to LuaMetaT<sub>E</sub>X: we use different 'constants', but that should be no problem because any sane macro package uses abstraction.
- ▶ The `\scantokens` primitive is now using the same mechanism as Lua print-to-T<sub>E</sub>X uses, which simplifies the code. There is a little performance hit but it will not be noticed in ConT<sub>E</sub>Xt, because we never use this primitive.
- ▶ Because we don't use change files on top of original T<sub>E</sub>X, the integration of  $\varepsilon$ -T<sub>E</sub>X functionality is bit more natural, code wise.
- ▶ The `\readline` primitive has to be implemented in Lua. This is a side effect of delegating all file io.
- ▶ Most of the code is rewritten but the original primitives are still tagged as coming from  $\varepsilon$ -T<sub>E</sub>X.

### 3.1.4 Changes from pdfT<sub>E</sub>X 1.40

Because we want to produce pdf the most natural starting point was the popular pdfT<sub>E</sub>X program. We inherit the stable features, dropped most of the experimental code and promoted some functionality to core LuaT<sub>E</sub>X functionality which in turn triggered renaming primitives. However, as the backend was dropped, not that much from pdfT<sub>E</sub>X is present any more. Basically all we now inherit from pdfT<sub>E</sub>X is expansion and protrusion but even that has been adapted. So don't expect LuaMetaT<sub>E</sub>X to be compatible.

- ▶ The experimental primitives `\ifabsnum` and `\ifabsdim` have been promoted to core primitives.
- ▶ The primitives `\ifincsname`, `\expanded` and `\quitvmode` have become core primitives.
- ▶ As the hz (expansion) 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` control these processes. The protrusion and kern related primitives are now dimensions while expansion is still one of these 1000 based scales.



- ▶ In LuaMetaT<sub>E</sub>X three extra primitives can be used to overload the font specific settings: `\adjustspacingstep` (max: 100), `\adjustspacingstretch` (max: 1000) and `\adjustspacingshrink` (max: 500).
- ▶ 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 the glyph and kern nodes carry the used values. In LuaT<sub>E</sub>X that made a more efficient generation of pdf code possible. It also resulted in much cleaner code. The backend code is gone, but of course the information is still carried around.
- ▶ When `\adjustspacing` has value 2, hz optimization will be applied to glyphs and kerns. When the value is 3, only glyphs will be treated. A value smaller than 2 disables this feature.
- ▶ When `\protrudechars` has a value larger than zero characters at the edge of a line can be made to hang out. A value of 2 will take the protrusion into account when breaking a paragraph into lines. A value of 3 will try to deal with right-to-left rendering; this is a still experimental feature.
- ▶ The pixel multiplier dimension `\pxdimen` has been inherited as core primitive.
- ▶ The primitive `\tracingfonts` is now a core primitive but doesn't relate to the backend.

### 3.1.5 Changes from Aleph RC4

In LuaT<sub>E</sub>X we took the 32 bit aspects and much of the directional mechanisms and merged it into the pdfT<sub>E</sub>X code base as starting point for further development. Then we simplified directionality, fixed it and opened it up. In LuaMetaT<sub>E</sub>X not that much of the later is left. We only have two horizontal directions. Instead of vertical directions we introduce an orientation model bound to boxes.

The already reduced-to-four set of directions now only has two members: left-to-right and right-to-left. They don't do much as it is the backend that has to deal with them. When paragraphs are constructed a change in horizontal direction is irrelevant for calculating the dimensions. So, basically most that we do is registering state and passing that on till the backend can do something with it.

Here is a summary of inherited functionality:

- ▶ The `^^` notation has been extended: after `^^^^` four hexadecimal characters are expected and after `^^^^^^` six hexadecimal characters have to be given. The original T<sub>E</sub>X interpretation is still valid for the `^^` case but the four and six variants do no backtracking, i.e. when they are not followed by the right number of hexadecimal digits they issue an error message. Because `^^^` is a normal T<sub>E</sub>X case, we don't support the odd number of `^^^^^` either.
- ▶ Glues *immediately after* direction change commands are not legal breakpoints. There is a bit more sanity testing for the direction state. This can be configured.
- ▶ The placement of math formula numbers is direction aware and adapts accordingly. Boxes carry directional information but rules don't.
- ▶ There are no direction related primitives for page and body directions. The paragraph, text and math directions are specified using primitives that take a number. The three letter codes are dropped.



### 3.1.6 Changes from standard web2c

The LuaMetaT<sub>E</sub>X codebase is not dependent on the web2c framework. The interaction with the file system and tds is up to Lua. There still might be traces but eventually the code base should be lean and mean. The MetaPost library is coded in cweb and in order to be independent from related tools, conversion to C is done with a Lua script ran by, surprise, LuaMetaT<sub>E</sub>X.

## 3.2 Implementation notes

### 3.2.1 Memory allocation

The single internal memory heap that traditional T<sub>E</sub>X used for tokens and nodes is split into two separate arrays. Each of these will grow dynamically when needed. Internally a token or node is an index into these arrays. This permits for an efficient implementation and is also responsible for the performance of the core. All other data structures are mostly the same but managed dynamically too. Because we operate in a 64 bit world, the parallel table of equivalents needed for managing levels, is gone. Anyhow, the original documentation in T<sub>E</sub>X The Program mostly applies!

### 3.2.2 Sparse arrays

The `\mathcode`, `\delcode`, `\catcode`, `\sfcode`, `\lccode` and `\uccode` (and the new `\hjcode`) tables are now sparse arrays that are implemented in C. They are no longer part of the T<sub>E</sub>X ‘equivalence table’ and because each had 1.1 million entries with a few memory words each, this makes a major difference in memory usage. Performance is not really hurt by this.

The `\catcode`, `\sfcode`, `\lccode`, `\uccode` and `\hjcode` assignments don't show up when using the  $\varepsilon$ -T<sub>E</sub>X tracing routines `\tracingassigns` and `\tracingrestores` but we don't see that as a real limitation. It also saves a lot of clutter.

The glyph ids within a font are also managed by means of a sparse array as glyph ids can go up to index  $2^{21} - 1$  but these are never accessed directly so again users will not notice this.

### 3.2.3 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 control sequences. This is a side effect of going Unicode and utf. Where using 256 slots in an array add no burden supporting the whole Unicode range is a waste of space. Therefore, also active characters are internally implemented as a special type of multi-letter control sequences that uses a prefix that is otherwise impossible to obtain.

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.

### 3.2.4 Binary file reading

All input now goes via Lua: files loaded with `\input` as well as files that are opened with `\openin`. Actually the later has to be implemented in terms of macros and Lua calls. This also means that compared to LuaT<sub>E</sub>X the internal handling of input has been changed but users won't notice that.



Setting a callback is expected now. Although reading input natively using `getc` calls is more efficient, we now fetch lines from Lua, put them in a buffer and then pick successive bytes (keep in mind that we read utf) from that. The performance is quite ok, also because Lua is fast, today's operating systems cache, and storage media have become very fast. Also, TeX is spending more time messing around with what it has input than actually reading input.

### 3.2.5 Tabs and spaces

We conform to the way other TeX engines handle trailing tabs and spaces. For decades trailing tabs and spaces (before a newline) were removed from the input but this behaviour was changed in September 2017 to only handle spaces. We are aware that this can introduce compatibility issues in existing workflows but because we don't want too many differences with upstream TeXLive we just follow up on that patch (which is a functional one and not really a fix). It is up to macro packages maintainers to deal with possible compatibility issues and in LuaMetaTeX they can do so via the callbacks that deal with reading from files.

The previous behaviour was a known side effect and (as that kind of input normally comes from generated sources) it was normally dealt with by adding a comment token to the line in case the spaces and/or tabs were intentional and to be kept. We are aware of the fact that this contradicts some of our other choices but consistency with other engines. We still stick to our view that at the log level we can (and might be) more incompatible. We already expose some more details anyway.

### 3.2.6 Logging

When detailed logging is enabled more detail is output with respect to what nodes are involved. This is a side effect of the core nodes having more detailed subtype information. The benefit of more detail wins from any wish to be byte compatible in the logging. One can always write additional logging in Lua.

The information that goes into the log file can be different from LuaTeX, and might even differ a bit more in the future. The main reason is that inside the engine we have more granularity, which for instance means that we output subtype and attribute related information when nodes are printed. Of course we could have offered a compatibility mode but it serves no purpose. Over time there have been many subtle changes to control logs in the TeX ecosystems so another one is bearable.

In a similar fashion, there is a bit different behaviour when TeX expects input, which in turn is a side effect of removing the interception of `*` and `&` which made for cleaner code (quite a bit had accumulated as side effect of continuous adaptations in the TeX ecosystems). There was already code that was never executed, simply as side effect of the way LuaTeX initializes itself (one needs to enable classes of primitives for instance). Keep in mind that over time system dependencies have been handles with TeX change files, the web2c infrastructure, kpse features, compilation variables and flags, etc. In LuaMetaTeX we try to minimize all that.

When it became unavoidable that we output more detail, it also became clear that it made no sense to stay log and trace compatible. Some is controlled by parameters in order to stay close the original, but ConTeXt is configured such that we benefit from the new possibilities. Examples are that in addition to `\meaning` we have `\meaningfull` that also exposes macro properties, and



`\meaningless` that only exposes the body. The `\untraced` prefix will suppress some in the log, and we set `\tracinglevels` to 3 in order to get details about the input and grouping level. When there's less shown than expected keep in mind that LuaMetaTeX has a somewhat optimized saving and restoring of meanings so less can happen which is reflected in tracing. When node lists are serialized (as with `\showbox`) some nodes, like discretionaries report more detail. The compact serializer, used for instance to signal overfull boxes, also shows a bit more detail with respect to non-content nodes. I math more is shown if only because we have more control and additional mechanisms.

### 3.2.7 Parsing

Token parsers have been upgraded for the sake of Lua, `\csname` handling has been extended, macro definitions can be more flexible so there code was adapted, more conditionals also brought some changes. But we build upon the (reorganized) TeX foundation so the basics can definitely be recognized.

Because of interfacing in Lua the internal token and node organization has been normalized (read: we cannot cheat because all is kind of visible). On the one hand this can come with a performance penalty but that is more than compensated by extensions, optimized parsers and such. Still the fact that we are utf based (32 bit) makes the machinery slower than the 8 bit original. The reworked LuaMetaTeX engine is substantially faster than the LuaTeX predecessor.

The handling of conditionals has been adapted so that we can have flatter branches (`\orelse cum suis`). This again has some consequences for parsing. Because parsing alignments is rather interwoven in general parsing and expansion the handling of related primitives has been slightly adapted (also for the sake of Lua interfacing) and dealing with `\noalign` situations is a bit more convenient.

This are just a few of the adaptations and most of this happened stepwise with testing in the ConTeXt code base. It will be clear that LuaMetaTeX is a quite different extension to the original. You're warned.

### 3.2.8 Changes in keyword scanning

Some primitives accept (optional) keywords and in LuaMetaTeX there are more keywords than in LuaTeX. Scanning can trigger error messages and lookahead side effects and in LuaMetaTeX these can be different. This is no big deal because errors are still errors.







# 4 Using LuaMetaTeX

## 4.1 Initialization

### 4.1.1 A bare bone engine

Although the LuaMetaTeX engine will start up when you call the program it will not do much useful. You can compare it to computer hardware without (high level) operating system with a TeX kernel being the bios. It can interpret TeX code but for typesetting you need a reasonable setup. You also need to load fonts, and for output you need a backend, and both can be implemented in Lua. If you don't like that and want to get up and running immediately, you will be more happy with LuaTeX, pdfTeX or XeTeX, combined with your favorite macro package.

If you just want to play around you can install the ConTeXt distribution which (including manuals and some fonts) is tiny compared to a full TeXLive installation and can be run alongside it without problems. If there are issues you can go to the usual ConTeXt support platforms and seek help where you can find the people who made LuaTeX and LuaMetaTeX.

### 4.1.2 LuaMetaTeX as a Lua interpreter

Although LuaMetaTeX is primarily meant as a TeX engine, it can also serve as a stand alone Lua interpreter. There are two ways to make LuaMetaTeX behave like a standalone Lua interpreter. The first method uses the command line option `--luaonly` followed by a filename. The second is more automatic: if the only non-option argument (file) on the commandline has the extension `lmt` or `lua`. The `luc` extension has been dropped because bytecode compiled files are not portable and one can always load `indirect`. The `lmt` suffix is more ConTeXt specific and makes it possible to have files for LuaTeX and LuaMetaTeX alongside.

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 command line in the positive values, just like the Lua interpreter does.

LuaMetaTeX will exit immediately after executing the specified Lua script and is, in effect, a somewhat bulky stand alone Lua interpreter with a bunch of extra preloaded libraries. But we really want to keep the binary small, if possible below the 3MB which is okay for a script engine.

When no argument is given, LuaMetaTeX will look for a Lua file with the same name as the binary and run that one when present. This makes it possible to use the engine as a stub. For instance, in ConTeXt a symlink from `mtxrun` to type `luametatex` will run the `mtxrun.lmt` or `mtxrun.lua` script when present in the same path as the binary itself. As mentioned before first checking for (ConTeXt) `lmt` files permits different files for different engines in the same path.

### 4.1.3 Other commandline processing

When the LuaMetaTeX executable starts, it looks for the `--lua` command line option. If there is no `--lua` option, the command line is interpreted in a similar fashion as the other TeX engines. All options are accepted but only some are understood by LuaMetaTeX itself:



COMMANDLINE ARGUMENT	EXPLANATION
--credits	display credits and exit
--fmt=FORMAT	load the format file FORMAT
--help	display help and exit
--ini	be iniluatex, for dumping formats
--jobname=STRING	set the job name to STRING
--lua=FILE	load and execute a Lua initialization script
--version	display version and exit

There are less options than with LuaT<sub>E</sub>X, because one has to deal with them in Lua anyway. So for instance there are no options to enter a safer mode or control executing programs because this can easily be achieved with a startup Lua script.

Next the initialization script is loaded and executed. From within the script, the entire command line is available in the Lua table `arg`, beginning with `arg[0]`, containing the name of the executable. As consequence warnings about unrecognized options are suppressed.

Command line processing happens very early on. So early, in fact, that none of T<sub>E</sub>X's initializations have taken place yet. The Lua libraries that don't deal with T<sub>E</sub>X are initialized rather soon so you have these available.

LuaMetaT<sub>E</sub>X allows some of the command line 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).

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.
- ▶ There is 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.

So let's summarize this. The handling of what is called `jobname` is a bit complex. There can be explicit names set on the command line but when not set they can be taken from the `texconfig` table.

startup filename	--lua	a Lua file	
startup jobname	--jobname	a T <sub>E</sub> X tex	texconfig.jobname
startup dumpname	--fmt	a format file	texconfig.formatname

These names are initialized according to `--luaonly` or the first filename seen in the list of options. Special treatment of `&` and `*` as well as interactive startup is gone but we still enter T<sub>E</sub>X via an forced `\input` into the input buffer.<sup>1</sup>

<sup>1</sup> This might change at some point into an explicit loading triggered via Lua.



When we are in  $\text{T}_{\text{E}}\text{X}$  mode at some point the engine needs a filename, for instance for opening a log file. At that moment the set jobname becomes the internal one and when it has not been set which internalized to jobname but when not set becomes texput. When you see a texput.log file someplace on your system it normally indicates a bad run.

When running on MS Windows the command line, filenames, environment variable access etc. internally uses the current code page but to the user is exposed as utf8. Normally users won't notice this.

There is an extra options `--permitloadlib` that needs to be given when you load external libraries via Lua. Although you could manage this via Lua itself in a startup script, the reason for having this as option is the wish for security (at some point that became a demand for  $\text{LuaT}_{\text{E}}\text{X}$ ), so this might give an extra feeling of protection.

## 4.2 Lua behaviour

### 4.2.1 The Lua version

We currently use Lua 5.4 and will follow developments of the language but normally with some delay. Therefore the user needs to keep an eye on (subtle) differences in successive versions of the language. Here is an example of one aspect.

Luas `tostring` function (and `string.format`) may return values in scientific notation, thereby confusing the  $\text{T}_{\text{E}}\text{X}$  end of things when it is used as the right-hand side of an assignment to a `\dimen` or `\count`. The output of these serializers also depend on the Lua version, so in Lua 5.3 you can get different output than from 5.2. It is best not to depend the automatic cast from string to number and vice versa as this can change in future versions.

### 4.2.2 Locales

In stock Lua, many things depend on the current locale. In  $\text{LuaMetaT}_{\text{E}}\text{X}$ , we can't do that, because it makes documents unportable. While  $\text{LuaMetaT}_{\text{E}}\text{X}$  is running it forces the following locale settings:

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

There is no way to change that as it would interfere badly with the often language specific conversions needed at the  $\text{T}_{\text{E}}\text{X}$  end.

## 4.3 Lua modules

Of course the regular Lua modules are present. In addition we provide the `lpeg` library by Roberto Ierusalimsky. 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 utf8 characters that need more than one byte, 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('ä')` results in the message `bad argument #1 to 'R' (range must have two characters)`, since to `lpeg`, `ä` is two 'characters' (bytes), so `ä` totals three. In practice this is no real issue and with some care you can deal with Unicode just fine.

There are some more libraries present. These are discussed on a later chapter. For instance we embed `luasocket` but contrary to `LuaTeX` don't embed the related Lua code. The `luafilesystem` module has been replaced by a more efficient one that also deals with the MS Windows file and environment properties better (Unicode support in MS Windows dates from before `utf8` became dominant so we need to deal with wide Unicode16).

There are more extensive math libraries and there are libraries that deal with encryption and compression. There are also some optional libraries that we do interface but that are loaded on demand. The interfaces are as minimal as can be because we so much in Lua, which also means that one can tune behaviour to usage better.

## 4.4 Testing

For development reasons you can influence the used startup date and time. By setting the `start_time` variable in the `texconfig` table; as with other variables we use the internal name there. When Universal Time is needed, set the entry `use_utc_time` in the `texconfig` table.

In `ConTeXt` we provide the command line argument `--nodates` that does a bit more than disabling dates; it avoids time dependent information in the output file for instance.



# 5 Basic T<sub>E</sub>X enhancements

## 5.1 Introduction

### 5.1.1 Primitive behaviour

From day one, LuaT<sub>E</sub>X has offered extra features compared to the superset of pdfT<sub>E</sub>X, which includes  $\epsilon$ -T<sub>E</sub>X, and Aleph. This has not been limited to the possibility to execute Lua code via `\directlua`, but LuaT<sub>E</sub>X also adds functionality via new T<sub>E</sub>X-side primitives or extensions to existing ones. The same is true for LuaMetaT<sub>E</sub>X. Some primitives have `luatex` in their name and there will be no `luametatex` variants. This is because we consider LuaMetaT<sub>E</sub>X to be LuaT<sub>E</sub>X<sup>+</sup>.

Contrary to the LuaT<sub>E</sub>X engine LuaMetaT<sub>E</sub>X enables all its primitives. You can clone (a selection of) primitives with a different prefix, like this:

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

The `extraprimitives` function returns the whole list or a subset, specified by one or more keywords `tex`, `etex` or `luatex`. When you clone all primitives you can also do this:

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

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 and you can look up the details in section 12.3.15. There are only three kinds of primitives: `tex`, `etex` and `luatex` but a future version might drop this and no longer make that distinction as it no longer serves a purpose apart from the fact that it reveals some history.

### 5.1.2 Rationale

One can argue that T<sub>E</sub>X should stay as it is but over decades usage of this program has evolved and resulted in large macro packages that often need to rely on what the T<sub>E</sub>X books calls ‘dirty tricks’. When you look deep down in the code of ConT<sub>E</sub>Xt MkII, MkIV and MkXL (aka LMTX) you will see plenty of differences but quite a bit of the functionality in the most recent versions is also available in MkII. Of course more has been added over time, and some mechanisms could be made more efficient and reliable but plenty was possible.

So, when you see something done in ConT<sub>E</sub>Xt LMTX using new LuaMetaT<sub>E</sub>X primitives you can assume that somehow the same is done in ConT<sub>E</sub>Xt MkIV. We don't really need LuaMetaT<sub>E</sub>X instead of LuaT<sub>E</sub>X. Among the main reasons for still going for this new engine are:



- ▶ some new primitives make for less tracing and tracing has become rather verbose over years (just try `tracingall`); examples are the new macro argument handling and some new hooks
- ▶ some new primitives permits more efficient coding and have a positive impact on performance (this sort of compensates a performance hit due to delegating work to Lua)
- ▶ other primitives are there because they make the code look better; good examples are the extensions to conditionals; they remove the necessity for all kind of (somewhat unnatural) middle layers; take `local control` as example
- ▶ a few primitives make complex and demanding mechanism a bit easier to grasp and explain; think of alignments, inserts and marks
- ▶ more access from the Lua end to  $\text{\TeX}$  internals: a few more callbacks, more options, more robust interfaces, etc
- ▶ some mechanisms are very specific but can be made more generic (and powerful), like inserts, marks, adjusts and local boxes

I realize that new primitives also can make some  $\text{\TeX}$  code look less threatening to new users. It removes a bit of hackery and limits the level of guru that comes with showing off the mastery of expansion and lookahead. So be it. I wonder if those objecting to some of the extensions (with the argument that they are not needed, and `Con $\text{\TeX}$ t MkIV` is proof of that) can resist using them. I admit that it sometimes hurt to throw away good working but cumbersome code that took a while to evolve, but I also admit that I favor long distance traveling by bike or car over riding horseback.

It took a few years for `LuaMeta $\text{\TeX}$`  to evolve to what it is now and most extensions are not there “because they were easy” or “could be done”. If that were the case, there would be plenty more. In many aspects it has been a balancing act and much also relates to looking at the `Con $\text{\TeX}$ t` source code ( $\text{\TeX}$  as well as Lua) and wondering why it looks that way. It is also driven by the fact that I want to be able to explain to users why things are done in a certain way. In fact, I want users to be able to look at the code and understand it (apart from maybe a few real dirty low level helpers that are also dirty because of performance reasons). Just take this into account when reading on.

And yes, there are still a few possibilities I want to explore . . . some might show up temporarily so don't be surprised. I'm also aware that some new features can have bugs or side effects that didn't show up in `Con $\text{\TeX}$ t`, which after all is the benchmark and environment in which all this evolves.

Over time, the other  $\text{\TeX}$  engines might have an occasional feature (primitive) added and it is very unlikely that `LuaMeta $\text{\TeX}$`  will follow up on that. First of all we have different internals but most of all because plenty of time went into considering what got added and what not, apart from the fact that we have callbacks. Decades of  $\text{\TeX}$  development never really have lead to an extensive wish list so there is no real need why there should be a demand on anything other than we offer here. If  $\text{\TeX}$  worked well for ages, it can as well do for more, so there is no need to cripple the code base simply in order to be compatible with other engines; `LuaMeta $\text{\TeX}$`  is already quite different anyway.

### 5.1.3 Version information

There are three primitives to test the version of `Lua $\text{\TeX}$`  (and `LuaMeta $\text{\TeX}$` ):



PRIMITIVE	VALUE	EXPLANATION
<code>\luatexbanner</code>	This is LuaMetaTeX, Version 2.09.42	the banner reported on the console
<code>\luatexversion</code>	209	major and minor number combined
<code>\luatexrevision</code>	42	the revision number

A 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` or `\number` depending on the context.
- ▶ The minor version is a number running from 0 upto 99.
- ▶ The revision is reported by `\luatexrevision`. Contrary to other engines in LuaMetaTeX is also a number so one needs to prefix it with `\the` or `\number`.<sup>2</sup>
- ▶ The full version number consists of the major version (X), minor version (YY) and revision (ZZ), separated by dots, so X.YY.ZZ.

The LuaTeX binary has companions like LuajitTeX and a version that has a font rendering library on board. Both introduce dependencies that don't fit into the LuaMetaTeX agenda: compilation should be easy and future proof and not depend on code outside the source tree. It means that for instance the ConTeXt runners don't really need to check much more than the basic name. It also means that the context and mtxt-run stubs can be symbolic links to the main program that itself is about 3MB, so we can keep the binary footprint small. For normal ConTeXt LMTX processing no other binaries are needed because whatever support we need is done in Lua.

The LuaMetaTeX version number starts at 2 in order to prevent a clash with LuaTeX, and the version commands are the same. This is a way to indicate that these projects are related.

The status library also provides some information including what we get with the three mentioned primitives:

FIELD	VALUE
filename	E:/context/manuals/mkiv/external/luametateX/luametateX-enhancements.tex
banner	This is LuaMetaTeX, Version 2.09.42
luatex_engine	luametateX
luatex_version	209
luatex_revision	42
luatex_verbose	2.09.42
copyright	Taco Hoekwater, Hans Hagen & Wolfgang Schuster
development_id	20220308
format_id	642
used_compiler	gcc

<sup>2</sup> In the past it always was good to prefix the revision with `\number` anyway, just to play safe, although there have for instance been times that pdfTeX had funny revision indicators that at some point ended up as letters due to the internal conversions.



## 5.2 Unicode text support

### 5.2.1 Extended ranges

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 the engine there is no clear separation between the two concepts. Because the subtype of a glyph node can be changed in Lua it is up to the user. Subtypes larger than 255 indicate that font processing has happened.

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  $\text{\TeX}$ -based engines. The affected commands with an altered initial (left of the equal sign) or secondary (right of the equal sign) value are: `\char`, `\lccode`, `\uccode`, `\hjcode`, `\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 section ?? . Normalization of the Unicode input is on purpose not built-in and can be handled by a macro package during callback processing. We have made some practical choices and the user has to live with those.

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  $c \geq 1,114,112$ , Lua $\text{\TeX}$  will actually print the single byte corresponding to  $c$  minus 1,114,112.

Contrary to other  $\text{\TeX}$  engines, the output to the terminal is as-is so there is no escaping with `^^`. We operate in a utf universe. Because we operate in a C universum, zero characters are special but because we also live in a Unicode galaxy that is no real problem.

### 5.2.2 `\Uchar`

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

### 5.2.3 Extended tables

All traditional  $\text{\TeX}$  and  $\varepsilon\text{\TeX}$  registers can be 16-bit numbers. The affected commands are:

<code>\count</code>	<code>\muskip</code>	<code>\countdef</code>	<code>\muskipdef</code>
<code>\dimen</code>	<code>\marks</code>	<code>\dimendef</code>	<code>\toksdef</code>
<code>\skip</code>	<code>\toks</code>	<code>\skipdef</code>	<code>\insert</code>





<code>\box</code>	<code>\copy</code>	<code>\wd</code>	<code>\setbox</code>
<code>\unhbox</code>	<code>\unhcopy</code>	<code>\ht</code>	<code>\vsplit</code>
<code>\unvbox</code>	<code>\unvcopy</code>	<code>\dp</code>	

Fonts are loaded via Lua and a minimal amount of information is kept at the  $\text{\TeX}$  end. Sharing resources is up to the loaders. The engine doesn't really care about what a character (or glyph) number represents (a Unicode or index) as it only is interested in dimensions.

In  $\text{\TeX}$  the number of registers is 256 and  $\varepsilon\text{-}\text{\TeX}$  bumped that to 32K. One reason for a fixed number is that these registers are fast ways to store data and therefore are part of the main lookup table (used for data and pointers to data as well as save and restore housekeeping). In  $\text{\LuaTeX}$  the number was bumped to 64K but one can argue that less would also do. In order to keep the default memory footprint reasonable, in  $\text{\LuaMetaTeX}$  the number of languages, fonts and marks is limited. The size of some tables can be limited by configuration settings, so they can start out small and grow till configured maximum which is smaller than the absolute maximum.

Because we have additional ways to store integers, dimensions and glue, we might actually decide to decrease the maximum of the registers: if 64K is not enough, and you work around it, then likely 32K might do as well. Also, we have Lua to store massive amounts of data. One can argue that saving some 1.5MB memory (when we go halfway) is not worth the effort in a time when you have to close a browser in order to free the gigabytes it consumes, but there is no reason not to be lean and mean: a more conservative approach to start with creates headroom for going wild later.

## 5.3 Attributes

### 5.3.1 Nodes

When  $\text{\TeX}$  reads input it will interpret the stream according to the properties of the characters. Some signal a macro name and trigger expansion, others open and close groups, trigger math mode, etc. What's left over becomes the typeset text. Internally we get a linked list of nodes. Characters become glyph nodes that have for instance a font and char property and `\kern 10pt` becomes a kern node with a width property. Spaces are alien to  $\text{\TeX}$  as they are turned into glue nodes. So, a simple paragraph is mostly a mix of sequences of glyph nodes (words) and glue nodes (spaces). A node can have a subtype so that it can be recognized as for instance a space related glue.

The sequences of characters at some point are extended with disc nodes that relate to hyphenation. After that font logic can be applied and we get a list where some characters can be replaced, for instance multiple characters can become one ligature, and font kerns can be injected. This is driven by the font properties.

Boxes (like `\hbox` and `\vbox`) become `hlist` or `vlist` nodes with width, height, depth and shift properties and a pointer list to its actual content. Boxes can be constructed explicitly or can be the result of subprocesses. For instance, when lines are broken into paragraphs, the lines are a linked list of `hlist` nodes, possibly with glue and penalties in between.

Internally nodes have a number. This number is actually an index in the memory used to store nodes.



So, to summarize: all that you enter as content eventually becomes a node, often as part of a (nested) list structure. They have a relative small memory footprint and carry only the minimal amount of information needed. In traditional  $\text{\TeX}$  a character node only held the font and slot number, in  $\text{\LuaTeX}$  we also store some language related information, the expansion factor, etc. Now that we have access to these nodes from Lua it makes sense to be able to carry more information with a node and this is where attributes kick in.

It is important to keep in mind that there are situations where nodes get created in the current context. For instance, when  $\text{\TeX}$  builds a paragraph or page or constructs math formulas, it does add nodes and giving these the current attributes makes no sense and can even give weird side effects. In these cases, the attributes are inherited from neighbouring nodes.

### 5.3.2 Attribute registers

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

```
 $\backslash attribute$  <16-bit number> <optional equals> <32-bit number>  
 $\backslash attributedef$  <cname> <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:  $-7\text{FFFFFF}$  in hexadecimal, a.k.a.  $-2147483647$  in decimal. It follows that the value  $-7\text{FFFFFF}$  cannot be used as a legal attribute value, but you *can* assign  $-7\text{FFFFFF}$  to ‘unset’ an attribute. All attributes start out in this ‘unset’ state in  $\text{\initex}$ .

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 10.

Attributes are stored in a sorted (sparse) linked list that are shared when possible. This permits efficient testing and updating. You can define many thousands of attributes but normally such a large number makes no sense and is also not that efficient because each node carries a (possibly shared) link to a list of currently set attributes. But they are a convenient extension and one of the first extensions we implemented in  $\text{\LuaTeX}$ .

In  $\text{\LuaMetaTeX}$  we try to minimize the memory footprint and creation of these attribute lists more aggressive sharing them. This feature is still somewhat experimental.

### 5.3.3 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  $\backslash par$  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 LuaT<sub>E</sub>X 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`. The `attr` keyword(s) should come before a `to` or `spread`, if that is also specified. An example is:

```
\attribute997=123
\attribute998=456
\setbox0=\hbox {Hello}
\setbox2=\hbox attr 999 = 789 attr 998 = -"7FFFFFFF{Hello}
```

Box 0 now has attributes 997 and 998 set while box 2 has attributes 997 and 999 set while the nodes inside that box will all have attributes 997 and 998 set. Assigning the maximum negative value causes an attribute to be ignored.

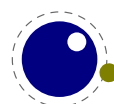
To give you an idea of what this means at the Lua end, take the following code:

```
for b=0,2,2 do
  for a=997, 999 do
    tex.sprint("box ", b, " : attr ",a," : ",tostring(tex.box[b]    [a]))
    tex.sprint("\quad\quad")
    tex.sprint("list ",b, " : attr ",a," : ",tostring(tex.box[b].list[a]))
    tex.sprint("\par")
  end
end
```

Later we will see that you can access properties of a node. The boxes here are so called `hlist` nodes that have a field `list` that points to the content. Because the attributes are a list themselves you can access them by indexing the node (here we do that with `[a]`). Running this snippet gives:

```
box 0 : attr 997 : 123    list 0 : attr 997 : 123
box 0 : attr 998 : 456    list 0 : attr 998 : 456
box 0 : attr 999 : nil    list 0 : attr 999 : nil
box 2 : attr 997 : 123    list 2 : attr 997 : 123
box 2 : attr 998 : nil    list 2 : attr 998 : 456
box 2 : attr 999 : 789    list 2 : attr 999 : nil
```

Because some values are not set we need to apply the `tostring` function here so that we get the word `nil`.



A special kind of box is `\vcenter`. This one also can have attributes. When one or more are set these plus the currently set attributes are bound to the resulting box. In regular  $\text{T}_{\text{E}}\text{X}$  these centered boxes are only permitted in math mode, but in  $\text{LuaMetaT}_{\text{E}}\text{X}$  there is no error message and the box the height and depth are equally divided. Of course in text mode there is no math axis related offset applied.

It is possible to change or add to the attributes assigned to a box with `\boxattribute`:

```
\boxattribute 0 123 456
```

You can set attributes of the current paragraph specification node with `\parattribute`:

```
\parattribute 123 456
```

## 5.4 Lua related primitives

### 5.4.1 `\directlua`

In order to merge Lua code with  $\text{T}_{\text{E}}\text{X}$  input, a few new primitives are needed. The primitive `\directlua` is used to execute Lua code immediately. The syntax is

```
\directlua <general text>
```

The `<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  $\text{T}_{\text{E}}\text{X}$ -style line comments (starting with `%`), or change the  $\text{T}_{\text{E}}\text{X}$  category codes locally. Another possibility is to say:

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

Then Lua line comments can be used, since  $\text{T}_{\text{E}}\text{X}$  does not replace line endings with spaces. Of course such an approach depends on the macro package that you use.

The `\directlua` command is expandable. Since it passes Lua code to the Lua interpreter its expansion from the  $\text{T}_{\text{E}}\text{X}$  viewpoint is usually empty. However, there are some Lua functions that produce material to be read by  $\text{T}_{\text{E}}\text{X}$ , 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  $\text{T}_{\text{E}}\text{X}$  input buffer, that is, ‘before  $\text{T}_{\text{E}}\text{X}$ ’s eyes’ to be read by  $\text{T}_{\text{E}}\text{X}$  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

$\pi = 3.1415926535898$

Note that the expansion of `\directlua` is a sequence of characters, not of tokens, contrary to all  $\text{\TeX}$  commands. So formally speaking its expansion is null, but it collects material in a new level on the input stack to be immediately read by  $\text{\TeX}$  after the Lua call as finished. It is a bit like  $\varepsilon\text{-}\text{\TeX}$ 's `\scantokens`, which now uses the same mechanism. For a description of print functions look at section 12.3.13.

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 often suboptimal because it can come from deep down, and  $\text{\TeX}$  has no knowledge about what you do in Lua. 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  $\text{\LaTeX}$  pretty bad. If you are not careful while working with the node list interface, you may even end up with errors or even crashes from within the  $\text{\TeX}$  portion of the executable.

### 5.4.2 `\luaescapestring`

This primitive converts a  $\text{\TeX}$  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.

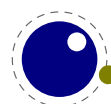
```
\luaescapestring <general text>
```

Most often, this command is not actually the best way to deal with the differences between  $\text{\TeX}$  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") }
```

### 5.4.3 `\luafunction`, `\luafunctioncall` and `\luadef`

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 have millions of calls it can have some impact. 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
\uafunction2
```

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
}
```

The `\luafunctioncall` primitive does the same but is unexpandable, for instance in an `\edef`. In addition LuaTeX provides a definer:

```
\luadef\MyFunctionA 1
\global\luadef\MyFunctionB 2
\protected\global\luadef\MyFunctionC 3
```

You should really use these commands with care. Some references get stored in tokens and assume that the function is available when that token expands. On the other hand, as we have tested this functionality in relative complex situations normal usage should not give problems.

*It makes sense to delegate the implementation of the primitives to Lua.*

#### 5.4.4 `\luabytecode` and `\luabytecodecall`

Analogue to the function callers discussed in the previous section we have byte code callers. Again the call variant is unexpandable.

```
\directlua {
  lua.bytecode[9998] = function(s)
    tex.sprint(s*token.scan_int())
  end
  lua.bytecode[5555] = function(s)
    tex.sprint(s*token.scan_dimen())
  end
}
```

This works with:

```
\luabytecode 9998 5 \luabytecode 5555 5sp
```



```
\luabytecodecall9998 5 \luabytecodecall5555 5sp
```

The variable `s` in the code is the number of the byte code register that can be used for diagnostic purposes. The advantage of bytecode registers over function calls is that they are stored in the format (but without upvalues).

*It makes sense to delegate the implementation of the primitives to Lua.*

## 5.5 Catcode tables

### 5.5.1 Catcodes

Catcode tables are a new feature that allows you to switch to a predefined catcode regime in a single statement. You can have lots of different tables, but if you need a dozen you might wonder what you're doing. This subsystem is backward compatible: if you never use the following commands, your document will not notice any difference in behaviour compared to traditional  $\text{\TeX}$ . The contents of each catcode table is independent from any other catcode table, and its contents is stored and retrieved from the format file.

### 5.5.2 `\catcodetable`

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 `ini $\text{\TeX}$` .

```
\catcodetable <15-bit number>
```

### 5.5.3 `\initcatcodetable`

```
\initcatcodetable <15-bit number>
```

The primitive `\initcatcodetable` creates a new table with catcodes identical to those defined by `ini $\text{\TeX}$` . 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. The initial values are:

CATCODE	CHARACTER	EQUIVALENT	CATEGORY
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



### 5.5.4 `\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. Again, 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.

## 5.6 Tokens, commands and strings

### 5.6.1 `\scantextokens`, `\tokenized` and `\retokenized`

The syntax of `\scantextokens` is identical to `\scantokens`. This primitive is a slightly adapted version of  $\varepsilon$ -T<sub>E</sub>X'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.
- ▶ There are no ‘... while end of file ...’ error tests executed. This allows the expansion to end on a different grouping level or while a conditional is still incomplete.

The implementation in LuaMetaT<sub>E</sub>X is different in the sense that it uses the same methods as printing from Lua to T<sub>E</sub>X does. Therefore, in addition to the two commands we also have this expandable command:

`\tokenized ... \tokenized catcodetable <number> ...`

The `\retokenized` variant differs in that it doesn't check for a keyword and just used the current catcode regime.

The  $\varepsilon$ -T<sub>E</sub>X command `\tracingscantokens` has been dropped in the process as that was interwoven with the old code.

### 5.6.2 `\toksapp`, `\tokspre`, `\etoksapp`, `\etokspre`, `\gtoksapp`, `\gtokspre`, `\xtoksapp`, `\xtokspre`

Instead of:

`\toks0\expandafter{\the\toks0 foo}`

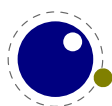
you can use:

`\etoksapp0{foo}`

The pre variants prepend instead of append, and the e variants expand the passed general text. The g and x variants are global.

### 5.6.3 `\csstring`, `\beginsname` and `\lastnamedcs`

These are somewhat special. The `\csstring` primitive is like `\string` but it omits the leading escape character. This can be somewhat more efficient than stripping it afterwards.





The `\begincsname` primitive is like `\csname` but doesn't create a relaxed equivalent when there is no such name. It is equivalent to

```
\ifcsname foo\endcsname
  \csname foo\endcsname
\fi
```

The advantage is that it saves a lookup (don't expect much speedup) but more important is that it avoids using the `\if` test. The `\lastnamedcs` is one that should be used with care. The above example could be written as:

```
\ifcsname foo\endcsname
  \lastnamedcs
\fi
```

This is slightly more efficient than constructing the string twice (deep down in LuaTeX this also involves some utf8 juggling), but probably more relevant is that it saves a few tokens and can make code a bit more readable.

#### 5.6.4 `\clearmarks`, `\flushmarks`, `\currentmarks`

The `\clearmarks` primitive complements the  $\epsilon$ -TeX mark primitives and clears a mark class completely, resetting all three connected mark texts to empty. It is an immediate command (no synchronization node is used).

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

The `\flushmarks` variant is delayed but puts a (mark) node in the list as signal (we could have gone for a keyword to `\marks` instead).

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

In addition to the three mark fetch commands, we also have access to the last set mark in the given class.

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

Marks can be traced with `\tracingmarks`. When set to 1 the page builder shows the set values, and when set to a higher value details about collecting them are shown.

#### 5.6.5 `\alignmark`, `\aligntab`, `\aligncontent`, `\tabsize` and `\everytab`

The primitive `\alignmark` duplicates the functionality of `#` inside alignment preambles, while `\aligntab` duplicates the functionality of `&`. The `\aligncontent` primitive directly refers to an entry so that one does not get repeated.

Alignments can be traced with `\tracingalignments`. When set to 1 basics usage is shown, for instance of `\noalign` but more interesting is 2 or more: you then get the preambles reported.

The `\halign` (tested) and `\valign` (yet untested) primitives accept a few keywords in addition to `to` and `spread`:



KEYWORD	EXPLANATION
<code>attr</code>	set the given attribute to the given value
<code>callback</code>	trigger the <code>alignment_filter</code> callback
<code>discard</code>	discard zero <code>\tabskip</code> 's
<code>noskip</code>	don't even process zero <code>\tabskip</code> 's
<code>reverse</code>	reverse the final rows

In the preamble the `\tabsize` primitive can be used to set the width of a column. By doing so one can avoid using a box in the preamble which, combined with the sparse `\tabskip` features, is a bit easier on memory when you produce tables that span hundreds of pages and have a dozen columns.

The `\everytab` complements the `\everycr` token register but is sort of experimental as it might become more selective and powerful some day.

### 5.6.6 `\latcharcode`

This primitive can be used to assign a meaning to an active character, as in:

```
\def\foo{bar} \latcharcode123=\foo
```

This can be a bit nicer than using the uppercase tricks (using the property of `\uppercase` that it treats active characters special).

### 5.6.7 `\lettonothing` and `\glettonothing`

This primitive is equivalent to:

```
\protected\def\lettonothing#1{\def#1{}}
```

and although it might feel faster (only measurable with millions of calls) it's mostly there because it is easier on tracing (less clutter). An advantage over letting to an empty predefined macro is also that in tracing we keep seeing the name (relaxing would show the `relax` equivalent).

### 5.6.8 `\glet`

This primitive is similar to:

```
\protected\def\glet{\global\let}
```

but faster (only measurable with millions of calls) and probably more convenient (after all we also have `\gdef`).

### 5.6.9 `\defcsname`, `\edefcsname`, `\edefcsname` and `\xdefcsname`

Although we can implement these primitives easily using macros it makes sense, given the popularity of `\csname` to have these as primitives. It also saves some `\expandafter` usage and it looks a bit better in the source.



```
\def\gdefcsname foo\endcsname{oof}
```

### 5.6.10 `\expanded`

The `\expanded` primitive takes a token list and expands its content which can come in handy: it avoids a tricky mix of `\expandafter` and `\noexpand`. You can compare it with what happens inside the body of an `\edef`. The `\immediateassignment` and `\immediateassigned` commands are gone because we have the more powerful local control commands. They are a tad slower but this mechanism isn't used that much anyway. Inside an `\edef` you can use the `\immediate` prefix anyway, so if you really want these primitives back you can say:

```
\let\immediateassignment\immediate  
\let\immediateassigned \localcontrolled
```

### 5.6.11 `\ignorepars`

This primitive is like `\ignorespaces` but also skips paragraph ending commands (normally `\par` and empty lines).

### 5.6.12 `\futureexpand`, `\futureexpandis`, `\futureexpandisap`

These commands are used as:

```
\futureexpand\sometoken\whenfound\whennotfound
```

When there is no match and a space was gobbled a space will be put back. The `is` variant doesn't do that while the `isap` even skips `\pars`, These characters stand for 'ignorespaces' and 'ignorespacesandpars'.

### 5.6.13 `\aftergrouped`

There is a new experimental feature that can inject multiple tokens to after the group ends. An example demonstrate its use:

```
{  
  \aftergroup A \aftergroup B \aftergroup C  
test 1 : }
```

```
{  
  \aftergrouped{What comes next 1}  
  \aftergrouped{What comes next 2}  
  \aftergrouped{What comes next 3}  
test 2 : }
```

```
{  
  \aftergroup A \aftergrouped{What comes next 1}
```



```

\aftergroup B \aftergrouped{What comes next 2}
\aftergroup C \aftergrouped{What comes next 3}
test 3 : }

{
  \aftergrouped{What comes next 1} \aftergroup A
  \aftergrouped{What comes next 2} \aftergroup B
  \aftergrouped{What comes next 3} \aftergroup C
test 4 : }

```

This gives:

```

test 1 : ABC
test 2 : What comes next 1What comes next 2What comes next 3
test 3 : AWhat comes next 1BWhat comes next 2CWhat comes next 3
test 4 : What comes next 1AWhat comes next 2BWhat comes next 3C

```

## 5.7 Conditions

### 5.7.1 `\ifabsnum` and `\ifabsdim`

There are two tests that we took from pdfTeX:

```

\ifabsnum -10 = 10
  the same number
\fi
\ifabsdim -10pt = 10pt
  the same dimension
\fi

```

This gives

```

the same number the same dimension

```

### 5.7.2 `\ifcmpnum`, `\ifcmpdim`, `\ifnumval`, `\ifdimval`, `\ifchknum` and `\ifchkdim`

New are the ones that compare two numbers or dimensions:

```

\ifcmpnum 5 8 less \or equal \else more \fi
\ifcmpnum 5 5 less \or equal \else more \fi
\ifcmpnum 8 5 less \or equal \else more \fi

```

```

less equal more

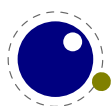
```

and

```

\ifcmpdim 5pt 8pt less \or equal \else more \fi

```



```
\ifcmpdim 5pt 5pt less \or equal \else more \fi
\ifcmpdim 8pt 5pt less \or equal \else more \fi
```

less equal more

There are also some number and dimension tests. All four expose the `\else` branch when there is an error, but two also report if the number is less, equal or more than zero.

```
\ifnumval -123 \or < \or = \or > \or ! \else ? \fi
\ifnumval 0 \or < \or = \or > \or ! \else ? \fi
\ifnumval 123 \or < \or = \or > \or ! \else ? \fi
\ifnumval abc \or < \or = \or > \or ! \else ? \fi
```

```
\ifdimval -123pt \or < \or = \or > \or ! \else ? \fi
\ifdimval 0pt \or < \or = \or > \or ! \else ? \fi
\ifdimval 123pt \or < \or = \or > \or ! \else ? \fi
\ifdimval abcpt \or < \or = \or > \or ! \else ? \fi
```

< = > !

< = > !

```
\ifchknum -123 \or okay \else bad \fi
\ifchknum 0 \or okay \else bad \fi
\ifchknum 123 \or okay \else bad \fi
\ifchknum abc \or okay \else bad \fi
```

```
\ifchkdim -123pt \or okay \else bad \fi
\ifchkdim 0pt \or okay \else bad \fi
\ifchkdim 123pt \or okay \else bad \fi
\ifchkdim abcpt \or okay \else bad \fi
```

okay okay okay bad

okay okay okay bad

The last checked values are available in `\lastchknum` and `\lastchkdim`. These don't obey grouping.

### 5.7.3 `\ifmathstyle` and `\ifmathparameter`

These two are variants on `\ifcase` where the first one operates with values in ranging from zero (display style) to seven (cramped script script style) and the second one can have three values: a parameter is zero, has a value or is unset. The `\ifmathparameter` primitive takes a proper parameter name and a valid style identifier (a primitive identifier or number). The `\ifmathstyle` primitive is equivalent to `\ifcase\mathstyle`.

### 5.7.4 `\ifempty`

This primitive tests for the following token (control sequence) having no content. Assuming that `\empty` is indeed empty, the following two are equivalent:



```
\ifempty\whatever
\ifx\whatever\empty
```

There is no real performance gain here, it's more one of these extensions that lead to less clutter in tracing.

### 5.7.5 `\ifrelax`

This primitive complements `\ifdefined`, `\ifempty` and `\ifcstype` so that we have all reasonable tests directly available.

### 5.7.6 `\ifboolean`

This primitive tests for non-zero, so the next variants are similar

```
\ifcase <integer>.F.\else .T.\fi
\unless\ifcase <integer>.T.\else .F.\fi
\ifboolean<integer>.T.\else .F.\fi
```

### 5.7.7 `\iftok` and `\ifcstok`

Comparing tokens and macros can be done with `\ifx`. Two extra test are provided in LuaMetaTeX:

```
\def\ABC{abc} \def\DEF{def} \def\PQR{abc} \newtoks\XYZ \XYZ {abc}
```

```
\iftok{abc}{def}\relax (same) \else [different] \fi
\iftok{abc}{abc}\relax [same] \else (different) \fi
\iftok\XYZ {abc}\relax [same] \else (different) \fi
```

```
\ifcstok\ABC \DEF\relax (same) \else [different] \fi
\ifcstok\ABC \PQR\relax [same] \else (different) \fi
\ifcstok{abc}\ABC\relax [same] \else (different) \fi
```

```
[different][same][same]
[different][same][same]
```

You can check if a macro is defined as protected with `\ifprotected` while frozen macros can be tested with `\iffrozen`. A provisional `\ifusercmd` tests will check if a command is defined at the user level (and this one might evolve).

### 5.7.8 `\ifarguments`, `\ifparameters` and `\ifparameter`

These are part of the extended macro argument parsing features. The `\ifarguments` condition is like an `\ifcase` where the number is the picked up number of arguments. The number reflects the *last* count, so successive macro expansions will adapt the value. The `\ifparameters` counts till the first empty parameter and the `\ifparameter` (singular) takes a parameter reference (like



#2) and again is an `\ifcase` where zero means a bad reference, one a non-empty argument and two an empty one. A typical usage is:

```
\def\foo#1#2%
  {\ifparameter#1\or one\fi
   \ifparameter#2\or two\fi}
```

No expansion of arguments takes place here but you can use a test like this:

```
\def\foo#1#2%
  {\iftok{#1}{}\else one\fi
   \iftok{#2}{}\else two\fi}
```

### 5.7.9 `\ifcondition`

This is a somewhat special one. When you write macros conditions need to be properly balanced in order to let  $\text{\TeX}$ 's fast branch skipping work well. This new primitive is basically a no-op flagged as a condition so that the scanner can recognize it as an if-test. However, when a real test takes place the work is done by what follows, in the next example `\something`.

```
\unexpanded\def\something#1#2%
  {\edef\tempa{#1}%
   \edef\tempb{#2}
   \ifx\tempa\tempb}

\ifcondition\something{a}{b}%
  \ifcondition\something{a}{a}%
    true 1
  \else
    false 1
  \fi
\else
  \ifcondition\something{a}{a}%
    true 2
  \else
    false 2
  \fi
\fi
```

If you are familiar with MetaPost, this is a bit like `vardef` where the macro has a return value. Here the return value is a test.

Experiments with something `\ifdef` actually worked ok but were rejected because in the end it gave no advantage so this generic one has to do. The `\ifcondition` test is basically is a no-op except when branches are skipped. However, when a test is expected, the scanner gobbles it and the next test result is used. Here is an other example:

```
\def\mytest#1%
```



```
{\ifabsdim#1>0pt\else
  \expandafter \unless
\fi
\iftrue}
```

```
\ifcondition\mytest{10pt}\relax non-zero \else zero \fi
\ifcondition\mytest {0pt}\relax non-zero \else zero \fi
```

non-zero zero

The last expansion in a macro like `\mytest` has to be a condition and here we use `\unless` to negate the result.

### 5.7.10 `\orelse` and `\orunless`

Sometimes you have successive tests that, when laid out in the source lead to deep trees. The `\ifcase` test is an exception. Experiments with `\ifcasex` worked out fine but eventually were rejected because we have many tests so it would add a lot. As `LuaMetaTeX` permitted more experiments, eventually an alternative was cooked up, one that has some restrictions but is relative lightweight. It goes like this:

```
\ifnum\count0<10
  less
\orelse\ifnum\count0=10
  equal
\else
  more
\fi
```

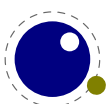
The `\orelse` has to be followed by one of the if test commands, except `\ifcondition`, and there can be an `\unless` in front of such a command. These restrictions make it possible to stay in the current condition (read: at the same level). If you need something more complex, using `\orelse` is probably unwise anyway. In case you wonder about performance, there is a little more checking needed when skipping branches but that can be neglected. There is some gain due to staying at the same level but that is only measurable when you runs tens of millions of complex tests and in that case it is very likely to drown in the real action. It's a convenience mechanism, in the sense that it can make your code look a bit easier to follow.

There is a nice side effect of this mechanism. When you define:

```
\def\quitcondition{\orelse\iffalse}
```

you can do this:

```
\ifnum\count0<10
  less
\orelse\ifnum\count0=10
  equal
  \quitcondition
```





```

    indeed
\else
    more
\fi

```

Of course it is only useful at the right level, so you might end up with cases like

```

\ifnum\count0<10
    less
\orelse\ifnum\count0=10
    equal
    \ifnum\count2=30
        \expandafter\quitcondition
    \fi
    indeed
\else
    more
\fi

```

The `\orunless` variant negates the next test, just like `\unless`. In some cases these commands look at the next token to see if it is an if-test so a following negation will not work (read: making that work would complicate the code and hurt efficiency too). Side note: interesting is that in ConT<sub>E</sub>Xt we hardly use this kind of negation.

### 5.7.11 `\ifflags`

This checker deal with control sequences. You can check if a command is a protected one, that is, defined with the `\protected` prefix. A command is frozen when it has been defined with the `\frozen` prefix. Beware: only macros can be frozen. A user command is a command that is not part of the predefined set of commands. This is an experimental command. The flag values can be queried with `tex.getflagvalues`.

## 5.8 Boxes, rules and leaders

### 5.8.1 `\outputbox`


This 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.

```
\outputbox = 12345
```

### 5.8.2 `\hrule`, `\vrule`, `\nohrule` and `\novrule`

Both rule drawing commands take an optional `xoffset` and `yoffset` parameter. The displacement is virtual and not taken into account when the dimensions are calculated. A rule is specified in the usual way:






There is however a catch. The keyword scanners in LuaMetaT<sub>E</sub>X are implemented slightly different. When T<sub>E</sub>X scans a keyword it will (case insensitive) scan for a whole keyword. So, it scans for height and when it doesn't find it it will scan for depth etc. When it does find a keyword in this case it expects a dimension next. When that criterium is not met it will issue an error message.

In order to avoid look ahead failures like that it is recommended to end the specification with `\relax`. A glue specification is an other example where a `\relax` makes sense when look ahead issues are expected and actually there in traditional scanning the order of keywords can also matter. In any case, when no valid keyword is seen the characters scanned so far are pushed back in the input.

The main reason for using an adapted scanner is that we always permit repetition (consistency) and accept an arbitrary order. Because we have more keywords to process the scanner quits at a partial failure. This prevents some push back and also gives an earlier warning. Interesting is that some ConT<sub>E</sub>Xt users ran into error messages due to a missing `\relax` and found out that their style has a potential flaw with respect to look ahead. One can be lucky for years.

Back to rules, there are some extra keywords, two deal with an offset, and four provide margins. The margins are a bit special because left and top are the same as are right and bottom. They influence the edges and these depend on it being a horizontal or vertical rule.



Two new primitives were introduced: `\nohrule` and `\novrule`. These can be used to reserve space. This is often more efficient than creating an empty box with fake dimensions. Of course this assumes that the backend implements them being invisible but still taking space.

### 5.8.3 `\vsplit`

The `\vsplit` primitive has to be followed by a specification of the required height. As alternative for the `to` keyword you can use `upto` to get a split of the given size but result has the natural dimensions then.

```
\vsplit 123 to 10cm % final box has the required height
\vsplit 123 upto 10cm % final box has its natural height
```

### 5.8.4 Images and reused box objects

In original T<sub>E</sub>X image support is dealt with via specials. It's not a native feature of the engine. All that T<sub>E</sub>X cares about is dimensions, so in practice that meant: using a box with known dimensions that wraps a special that instructs the backend to include an image. The wrapping is needed because a special itself is a whatsit and as such has no dimensions.

In pdfT<sub>E</sub>X a special whatsit for images was introduced and that one *has* dimensions. As a consequence, in several places where the engine deals with the dimensions of nodes, it now has to check the details of whatsits. By inheriting code from pdfT<sub>E</sub>X, the LuaT<sub>E</sub>X engine also had that property. However, at some point this approach was abandoned and a more natural trick



was used: images (and box resources) became a special kind of rules, and as rules already have dimensions, the code could be simplified.

When direction nodes and (formerly local) par nodes also became first class nodes, whatsits again became just that: nodes representing whatever you want, but without dimensions, and therefore they could again be ignored when dimensions mattered. And, because images were disguised as rules, as mentioned, their dimensions automatically were taken into account. This separation between front and backend cleaned up the code base already quite a bit.

In LuaMetaT<sub>E</sub>X we still have the image specific subtypes for rules, but the engine never looks at subtypes of rules. That was up to the backend. This means that image support is not present in LuaMetaT<sub>E</sub>X. When an image specification was parsed the special properties, like the filename, or additional attributes, were stored in the backend and all that LuaT<sub>E</sub>X does is registering a reference to an image's specification in the rule node. But, having no backend means nothing is stored, which in turn would make the image inclusion primitives kind of weird.

Therefore you need to realize that contrary to LuaT<sub>E</sub>X, *in LuaMetaT<sub>E</sub>X support for images and box reuse is not built in!* However, we can assume that an implementation uses rules in a similar fashion as LuaT<sub>E</sub>X does. So, you can still consider images and box reuse to be core concepts. Here we just mention the primitives that LuaT<sub>E</sub>X provides. They are not available in the engine but can of course be implemented in Lua.

COMMAND	EXPLANATION
<code>\saveboxresource</code>	save the box as an object to be included later
<code>\saveimageresource</code>	save the image as an object to be included later
<code>\useboxresource</code>	include the saved box object here (by index)
<code>\useimageresource</code>	include the saved image object here (by index)
<code>\lastsavedboxresourceindex</code>	the index of the last saved box object
<code>\lastsavedimageresourceindex</code>	the index of the last saved image object
<code>\lastsavedimageresourcepages</code>	the number of pages in the last saved image object

An implementation probably should accept the usual optional dimension parameters for `\use...resource` in the same format as for rules. With images, these dimensions are then used instead of the ones given to `\useimageresource` but the original dimensions are not overwritten, so that a `\useimageresource` without dimensions still provides the image with dimensions defined by `\saveimageresource`. These optional parameters are not implemented for `\saveboxresource`.

```
\useimageresource width 20mm height 10mm depth 5mm \lastsavedimageresourceindex
\useboxresource   width 20mm height 10mm depth 5mm \lastsavedboxresourceindex
```

Examples or optional entries are `attr` and `resources` that accept a token list, and the `type` key. When set to non-zero the `/Type` entry is omitted. A value of 1 or 3 still writes a `/BBox`, while 2 or 3 will write a `/Matrix`. But, as said: this is entirely up to the backend. Generic macro packages (like `tikz`) can use these assumed primitives so one can best provide them. It is probably, for historic reasons, the only more or less standardized image inclusion interface one can expect to work in all macro packages.



### 5.8.5 `\hpack`, `\vpack` and `\tpack`

These three primitives are the equivalents of `\hbox`, `\vbox` and `\vtop` but they don't trigger the packaging related callbacks. Of course one never know if content needs a treatment so using them should be done with care. Apart from accepting more keywords (and therefore options) the normal box behave the same as before. The `\vcenter` builder also works in text mode.

### 5.8.6 `\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.

## 5.9 Languages

### 5.9.1 `\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.

### 5.9.2 `\boundary`, `\noboundary`, `\protrusionboundary` and `\wordboundary`

The `\noboundary` command is used to inject a `whatsit` node but now injects a normal node with type `boundary` and subtype 0. In addition you can say:

```
x\boundary 123\relax y
```

This has the same effect but the subtype is now 1 and the value 123 is stored. The traditional ligature builder still sees this as a cancel boundary directive but at the Lua end you can implement different behaviour. The added benefit of passing this value is a side effect of the generalization. The subtypes 2 and 3 are used to control protrusion and word boundaries in hyphenation and have related primitives.

## 5.10 Control and debugging

### 5.10.1 Tracing

If `\tracingonline` is larger than 2, the node list display will also print the node number of the nodes as well as set attributes (these can be made verbose by a callback). We have only a generic `whatsit` but again a callback can be used to provide detail. So, when a box is shown in `ConTEXt` you will see quite a lot more than in other engines. Because nodes have more fields, more is shown anyway, and for nodes that have sublists (like `discretionaries`) these are also shown. All that could have been delegated to Lua but it felt wrong to not made that a core engine feature.



When bit 1 of `\tracinglevels` is set the current level is prepended to tracing lines in the log and when bit 2 is set the input level is prepended. You can set both bits and get both numbers prepended. In ConT<sub>E</sub>Xt we default to the value 3, so you get prefixes like 3:4: followed by a space.

When `\tracingcommands` is larger than 3 the mode switch will be not be prefixed to the {command} but get its own [line].

When `\tracinglevels` variable is set to 3 the group and input level are shown, a value of 1 or 2 shows only one of them (in ConT<sub>E</sub>Xt we default to 3).

When `\tracinghyphenation` is set to 1 duplicate patterns are reported (in ConT<sub>E</sub>Xt we default to that) and higher values will also show details about the Lua hyphenation (exception) feedback loop discussed elsewhere.

When set to 1 the `\tracingmath` variable triggers the reporting of the mode (inline or display) an mlist is processed. Other new tracing commands are discussed where the mechanisms that they relate to are introduced.

Because in LuaT<sub>E</sub>X the saving and restoring of locally redefined macros and set variables is optimized a bit in order to prevent redundant stack usage, there will be less tracing visible.

Also, because we have a more extensive macro argument parser, a fast path (and less storage demands) for macros with no arguments, and flags that can be set for macros the way macros are traced can be different in details (we therefore have for instance `\meaningfull` (double l's indeed) and `\meaningless` as variants of `\meaning` as well as `\meaningasis` for more literal alternative).

### 5.10.2 `\lastnodetype`, `\lastnodesubtype`, `\currentifttype`

The  $\varepsilon$ -T<sub>E</sub>X command `\lastnodetype` returns the node codes as used in the engine. You can query the numbers at the Lua end if you need the actual values. The parameter `\internalcodesmode` is no longer provided as compatibility switch because LuaT<sub>E</sub>X has more cq. some different nodes and it makes no sense to be incompatible with the Lua end of the engine. The same is true for `\currentifttype`, as we have more conditionals and also use a different order. The `\lastnodesubtype` is a bonus and again reports the codes used internally. During development these might occasionally change, but eventually they will be stable.

## 5.11 Files

### 5.11.1 File syntax

LuaMetaT<sub>E</sub>X 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. Keep in mind that as side effect of delegating io to Lua the `\openin`



primitive is not provided by the engine and has to be implemented by the macro package. This also means that the limit on the number of open files is not enforced by the engine.

The `\tracingfonts` primitive that has been inherited from pdf $\TeX$  has been adapted to support variants in reporting the font. The reason for this extension is that a `csname` not always makes sense. The zero case is the default.

VALUE	REPORTED
0	<code>\foo xyz</code>
1	<code>\foo (bar)</code>
2	<code>&lt;bar&gt; xyz</code>
3	<code>&lt;bar @ ..pt&gt; xyz</code>
4	<code>&lt;id&gt;</code>
5	<code>&lt;id: bar&gt;</code>
6	<code>&lt;id: bar @ ..pt&gt; xyz</code>

### 5.11.2 Writing to file

Writing to a file in  $\TeX$  has two forms: delayed and immediate. Delayed writing means that the to be written text is anchored in the node list and flushed by the backend. As all io is delegated to Lua, this also means that it has to deal with distinction. In Lua $\TeX$  the number of open files was already bumped to 127, but in LuaMeta $\TeX$  it depends on the macro package. The special meaning of channel 18 was already dropped in Lua $\TeX$  because we have `os.execute`.

## 5.12 Math

We will cover math extensions in its own chapter because not only the font subsystem and spacing model have been enhanced (thereby introducing many new primitives) but also because some more control has been added to existing functionality. Much of this relates to the different approaches of traditional  $\TeX$  fonts and OpenType math.

## 5.13 Fonts

Like math, we will cover fonts extensions in its own chapter. Here we stick to mentioning that loading fonts is different in LuaMeta $\TeX$ . As in Lua $\TeX$  we have the extra primitives `\fontid` and `\setfontid`, `\noligs` and `\nokerns`, and `\nospaces`. The other new primitives in Lua $\TeX$  have been dropped.

## 5.14 Directions

### 5.14.1 Two directions

The directional model in LuaMeta $\TeX$  is a simplified version the the model used in Lua $\TeX$ . In fact, not much is happening at all: we only register a change in direction.



### 5.14.2 How it works

The approach is that we try to make node lists balanced but also try to avoid some side effects. What happens is quite intuitive if we forget about spaces (turned into glue) but even there what happens makes sense if you look at it in detail. However that logic makes in-group switching kind of useless when no properly nested grouping is used: switching from right to left several times nested, results in spacing ending up after each other due to nested mirroring. Of course a sane macro package will manage this for the user but here we are discussing the low level injection of directional information.

This is what happens:

```
\textdirection 1 nur {\textdirection 0 run \textdirection 1 NUR} nur
```

This becomes stepwise:

```
injected: [push 1]nur {[push 0]run [push 1]NUR} nur
balanced: [push 1]nur {[push 0]run [pop 0][push 1]NUR[pop 1]} nur[pop 0]
result   : run {RUNrun } run
```

And this:

```
\textdirection 1 nur {nur \textdirection 0 run \textdirection 1 NUR} nur
```

becomes:

```
injected: [+TRT]nur {nur [+TLT]run [+TRT]NUR} nur
balanced: [+TRT]nur {nur [+TLT]run [-TLT][+TRT]NUR[-TRT]} nur[-TRT]
result   : run {run RUNrun } run
```

Now, in the following examples watch where we put the braces:

```
\textdirection 1 nur {\{\textdirection 0 run} {\textdirection 1 NUR}} nur
```

This becomes:

```
run RUNrun run
```

Compare this to:

```
\textdirection 1 nur {\{\textdirection 0 run }{\textdirection 1 NUR}} nur
```

Which renders as:

```
run RUNrun run
```

So how do we deal with the next?

```
\def\ltr{\textdirection 0\relax}
\def\rtl{\textdirection 1\relax}
```

```
run {\rtl nur {\ltr run \rtl NUR \ltr run \rtl NUR} nur}
run {\ltr run {\rtl nur \ltr RUN \rtl nur \ltr RUN} run}
```



It gets typeset as:

```
run run RUNrun RUNrun run
run run runRUN runRUN run
```

We could define the two helpers to look back, pick up a skip, remove it and inject it after the dir node. But that way we loose the subtype information that for some applications can be handy to be kept as-is. This is why we now have a variant of `\textdirection` which injects the balanced node before the skip. Instead of the previous definition we can use:

```
\def\ltr{\linedirection 0\relax}
\def\rtl{\linedirection 1\relax}
```

and this time:

```
run {\rtl nur {\ltr run \rtl NUR \ltr run \rtl NUR} nur}
run {\ltr run {\rtl nur \ltr RUN \rtl nur \ltr RUN} run}
```

comes out as a properly spaced:

```
run run RUN run RUN run run
run run run RUN run RUN run
```

Anything more complex than this, like combination of skips and penalties, or kerns, should be handled in the input or macro package because there is no way we can predict the expected behaviour. In fact, the `\linedirection` is just a convenience extra which could also have been implemented using node list parsing.

Directions are complicated by the fact that they often need to work over groups so a separate grouping related stack is used. A side effect is that there can be paragraphs with only a local par node followed by direction synchronization nodes. Paragraphs like that are seen as empty paragraphs and therefore ignored. Because `\noindent` doesn't inject anything but a `\indent` injects an box, paragraphs with only an indent and directions are handles and paragraphs with content. When indentation is normalized a paragraph with an indentation skip is seen as content.

### 5.14.3 Normalizing lines

The original  $\TeX$  machinery was never meant to be opened up. As a consequence a constructed line can have different layouts. There can be left- and/or right skips and hanging indentation or parshape can result in a shift and adapted width. In  $\text{Lua}\TeX$  glue got subtypes so we can recognize the left-, right and parfill skips, but still there is no hundred percent certainty about the shape.

In  $\text{LuaMeta}\TeX$  lines can be normalized. This is optional because we want to preserve the original (for comparison) and is controlled by `\normalizelinemode`. That variable actually drives some more. An earlier version provided a few more granular options (for instance: does a leftskip comes before or after a left hanging indentation) but in the end that was dropped. Because this normalization only is seen at the Lua end there is no need to go into much detail here.





At this moment a line has this pattern: left parfill, left hang, left skip, indentation, content, right hang, right skip, right parfill. Of course the indentation and fill skips are not present in every line.

Control over normalization happens via the mentioned mode variable and here is what the engine provides right now. We use a bitmap:

VALUE	REPORTED
0x0001	normalize line as described above
0x0002	use a skip for parindent instead of a box
0x0004	swap hangindent in l2r mode
0x0008	swap parshape in l2r mode
0x0010	put breaks after dir in l2r mode
0x0020	remove margin kerns (pdfTeX left-over)
0x0040	if needed clip width and use correction kern

Setting the bit enables the related normalization. More features might be added in future releases.

#### 5.14.4 Orientations

As mentioned, the difference with LuaTeX is that we only have numeric directions and that there are only two: left-to-right (0) and right-to-left (1). The direction of a box is set with `direction`.

In addition to that boxes can now have an `orientation` keyword followed by optional `xoffset` and/or `yoffset` keywords. The offsets don't have consequences for the dimensions. The alternatives `xmove` and `ymove` on the contrary are reflected in the dimensions. Just play with them. The offsets and moves only are accepted when there is also an orientation, so no time is wasted on testing for these rarely used keywords. There are related primitives `\box...` that set these properties.

As these are experimental it will not be explained here (yet). They are covered in the descriptions of the development of LuaMetaTeX: articles and/or documents in the ConTeXt distribution. For now it is enough to know that the orientation can be up, down, left or right (rotated) and that it has some anchoring variants. Combined with the offsets this permits macro writers to provide solutions for top-down and bottom-up writing directions, something that is rather macro package specific and used for scripts that need manipulations anyway. The 'old' vertical directions were never okay and therefore not used.

There are a couple of properties in boxes that you can set and query but that only really take effect when the backend supports them. When usage on ConTeXt shows that isn't okay, they will become official, so we just mention them: `\boxdirection`, `\boxattr`, `\boxorientation`, `\boxxoffset`, `\boxyoffset`, `\boxxmove`, `\boxymove` and `\boxtotal`.

*This is still somewhat experimental and will be documented in more detail when I've used it more in ConTeXt and the specification is frozen. This might take some time (and user input).*

### 5.15 Keywords

Some primitives accept one or more keywords and LuaMetaTeX adds some more. In order to deal with this efficiently the keyword scanner has been optimized, where even the context was



taken into account. As a result the scanner was quite a bit faster. This kind of optimization was a graduate process the eventually ended up in what we have now. In traditional  $\text{\TeX}$  (and also  $\text{\LuaTeX}$ ) the order of keywords is sometimes mixed and sometimes prescribed. In most cases only one occurrence is permitted. So, for instance, this is valid in  $\text{\LuaTeX}$ :

```
\hbox attr 123 456 attr 123 456 spread 10cm { }  
\hrule width 10cm depth 3mm  
\hskip 3pt plus 2pt minus 1pt
```

The `attr` comes before the `spread`, rules can have multiple mixed dimension specifiers, and in glue the optional `minus` part always comes last. The last two commands are famous for look ahead side effects which is why macro packages will end them with something not keyword, like `\relax`, when needed.

In  $\text{\LuaMetaTeX}$  the following is okay. Watch the few more keywords in box and rule specifications.

```
\hbox reverse to 10cm attr 123 456 orientation 4 xoffset 10pt spread 10cm { }  
\hrule xoffset 10pt width 10cm depth 3mm  
\hskip 3pt minus 1pt plus 2pt
```

Here the order is not prescribed and, as demonstrated with the box specifier, for instance dimensions (specified by `to` or `spread` can be overloaded by later settings. In case you wonder if that breaks compatibility: in some way it does but bad or sloppy keyword usage breaks a run anyway. For instance `minuscule` results in `minus` with no dimension being seen. So, in the end the user should not noticed it and when a user does, the macro package already had an issue that had to be fixed.

## 5.16 Expressions and `\numERICSCALE`

The `*expr` parsers now accept `:` as operator for integer division (the `/` operators does rounding. This can be used for division compatible with `\divide`. I'm still wondering if adding a couple of bit operators makes sense (for integers).

The `\numERICSCALE` parser is kind of special (and might evolve). For now it converts a following number in a scale value as often used in  $\text{\TeX}$ , where 1000 means scaling by 1.0. The trick is in the presence of a digit (or comma): 1.234 becomes 1234 but 1234 stays 1234 and from this you can deduce that 12.34 becomes 123400. Internally  $\text{\TeX}$  calculates with integers, but this permits the macro package to provide an efficient mix.

## 5.17 Macro arguments

Again this is experimental and (used and) discussed in document that come with the  $\text{\ConTeXt}$  distribution. When defining a macro you can do this:

```
\def\foo(#1)#2{...}
```

Here the first argument between parentheses is mandate. But the magic prefix `\tolerant` makes that limitation go away:



```
\tolerant\def\foo(#1)#2{...}
```

A variant is this:

```
\tolerant\def\foo(#1)*(#2){...}
```

Here we have two optional arguments, possibly be separated by spaces. There are more parsing options:

+	keep the braces
-	discard and don't count the argument
/	remove leading an trailing spaces and pars
=	braces are mandate
_	braces are mandate and kept
^	keep leading spaces
<hr/>	
1-9	an argument
0	discard but count the argument
<hr/>	
*	ignore spaces
.	ignore pars and spaces
,	push back space when no match
<hr/>	
:	pick up scanning here
;	quit scanning
<hr/>	

For the moment we leave it to your fantasy what these options do. Most probably only make sense when you write a bit more complex macros. Just try to imagine what this does:

```
\permanent\tolerant\global\protected\def\foo(#1)*#;[#2]#:#3{...}
```

Of course complex combinations can be confusing because after all T<sub>E</sub>X is parsing for (multi-token) delimiters and will happily gobble the whole file if you are not careful. You can quit scanning if you want:

```
\mymacro 123\ignorearguments
```

which of course only makes sense when used in a nested call where an already picked up arguments is processed further. A not (yet) discussed feature of the parser is that it will happily skip tokens that have the (probably seldom used) ignored characters property.

When you use tracing or see error messages arguments defined using for instance `#=` will have their usual number in the macro body, so you need to keep track of the numbers.

All this is rather easy on the engine and although it might have a little impact on performance this has been compensated by some more efficiency in the macro parser and engine in general and of course you can gain back some by using these features.

## 5.18 Overload protection

There is an experimental overload protection mechanism that we will test for a while before declaring it stable. The reason for that is that we need to adapt the ConT<sub>E</sub>Xt code base in



order to test its usefulness. Protection is achieved via prefixes. Depending on the value of the `\overloadmode` variable warnings or errors will be triggered. Examples of usage can be found in some documents that come with ConT<sub>E</sub>Xt, so here we just stick to the basics.

```
\mutable \def\foo{...}
\immutable\def\foo{...}
\permanent\def\foo{...}
\frozen \def\foo{...}
\aliased \def\foo{...}
```

A `\mutable` macro can always be changed contrary to an `\immutable` one. For instance a macro that acts as a variable is normally `\mutable`, while a constant can best be `\immutable`. It makes sense to define a public core macro as `\permanent`. Primitives start out as `\permanent` ones but with a primitive property instead.

```
\let\relaxone \relax 1: \meaningfull\relaxone
\aliased \let\relaxtwo \relax 2: \meaningfull\relaxtwo
\permanent\let\relaxthree\relax 3: \meaningfull\relaxthree
```

The `\meaningfull` primitive is like `\meaning` but report the properties too. The `\meaningless` companion reports the body of a macro. Anyway, this typesets:

```
1: \relax
2: primitive \relax
3: permanent \relax
```

So, the `\aliased` prefix copies the properties. Keep in mind that a macro package can redefine primitives, but `\relax` is an unlikely candidate.

There is an extra prefix `\noaligned` that flags a macro as being valid for `\noalign` compatible usage (which means that the body must contain that one). The idea is that we then can do this:

```
\permanent\protected\noaligned\def\foo{\noalign{...}} % \foo is unexpandable
```

that is: we can have protected macros that don't trigger an error in the parser where there is a look ahead for `\noalign` which is why normally protection doesn't work well. So: we have macro flagged as permanent (overload protection), being protected (that is, not expandable by default) and a valid equivalent of the `\noalign` primitive. Of course we can also apply the `\global` and `\tolerant` prefixes here. The complete repertoire of extra prefixes is:

frozen	a macro that has to be redefined in a managed way
permanent	a macro that had better not be redefined
primitive	a primitive that normally will not be adapted
immutable	a macro or quantity that cannot be changed, it is a constant
mutable	a macro that can be changed no matter how well protected it is
instance	a macro marked as (for instance) be generated by an interface
noaligned	the macro becomes acceptable as <code>\noalign</code> alias
overloaded	when permitted the flags will be adapted



enforced	all is permitted (but only in zero mode or ini mode)
aliased	the macro gets the same flags as the original
untraced	the macro gets a different treatment in tracing

The not yet discussed `\instance` is just a flag with no special meaning which can be used as classifier. The `\frozen` also protects against overload which brings amount of blockers to four.

To what extent the engine will complain when a property is changed in a way that violates the flags depends on the parameter `\overloadmode`. When this parameter is set to zero no checking takes place. More interesting are values larger than zero. If that is the case, when a control sequence is flagged as mutable, it is always permitted to change. When it is set to immutable one can never change it. The other flags determine the kind of checking done. Currently the following overload values are used:

	<b>immutable</b>	<b>permanent</b>	<b>primitive</b>	<b>frozen</b>	<b>instance</b>
1 warning	*	*	*		
2 error	*	*	*		
3 warning	*	*	*	*	
4 error	*	*	*	*	
5 warning	*	*	*	*	*
6 error	*	*	*	*	*

The even values (except zero) will abort the run. A value of 255 will freeze this parameter. At level five and above the `\instance` flag is also checked but no drastic action takes place. We use this to signal to the user that a specific instance is redefined (of course the definition macros can check for that too).

The `\overloaded` prefix can be used to overload a frozen macro. The `\enforced` is more powerful and forces an overload but that prefix is only effective in ini mode or when it's embedded in the body of a macro or token list at ini time unless of course at runtime the mode is zero.

So far for a short explanation. More details can be found in the ConT<sub>E</sub>Xt documentation where we can discuss it in a more relevant perspective. It must be noted that this feature only makes sense a controlled situation, that is: user modules or macros of unpredictable origin will probably suffer from warnings and errors when de mode is set to non zero. In ConT<sub>E</sub>Xt we're okay unless of course users redefine instances but there a warning or error is kind of welcome.

There is an extra prefix `\untraced` that will suppress the meaning when tracing so that the macro looks more like a primitive. It is still somewhat experimental so what gets displayed might change.

## 5.19 Constants with `\integerdef` and `\dimensiondef`

It is rather common to store constant values in a register or character definition.

```
\newcount\MyConstantA \MyConstantA 123
\newdimen\MyConstantB \MyConstantB 123pt
\chardef \MyConstantC \MyConstantC 123
```

But in LuaMetaT<sub>E</sub>X we also can do this:



```
\integerdef \MyConstantC 456
\dimensiondef\MyConstantD 456pt
```

These two are stored as efficient as a register but don't occupy a register slot. They can be set as above, need \the for serializations and are seen as valid number or dimension when needed.

Experiments with constant strings made the engine source more complex than I wanted so that features was rejected. Of course we can use the prefixes mentioned in a previous section.

## 5.20 Serialization with \todimension, \toscaled and \tointeger

These three serializers take a verbose or symbolic quantity:

```
\todimension 10pt    \todimension \scratchdimen    % with unit
\toscaled     10pt    \toscaled     \scratchdimen    % without unit
\tointeger    10      \tointeger    \scratchcounter
```

This is particularly handy in cases where you don't know what you deal with, for instance when a value is stored in a macro. Using \the could fail there while:

```
\the\dimexpr10pt\relax
```

is often overkill and gives more noise in a trace.

## 5.21 Expressions with \numexpression

The  $\varepsilon$ -TeX expression primitives are handy but have some limitations. Although the parsers have been rewritten in LuaMetaTeX and somewhat more efficient the only extension we have is support for an integer division with :. After experimenting for a while and pondering how to make \dimexpr and \numexpr more powerful I decided to come up with alternatives in order not to introduce incompatibilities.

The \numexpression and \dimexpression primitives are equivalent but offer more. The first one operates in the integer domain and the second one assumes scaled values. Often the second one can act like the first when serialized with \number in front. This is because when TeX sees a symbolic reference to an integer or dimension it can treat them as it likes.

The set of operators that we have to support is the following. Most have alternatives so that we can get around catcode issues.

ACTION	SYMBOL	KEYWORD
add	+	
subtract	-	
multiply	*	
divide	/ :	
mod	%	mod
band	&	band



bxor	^	bxor
bor	v	bor
and	&&	and
or		or
setbit	<undecided>	bset
resetbit	<undecided>	breset
left	<<	
right	>>	
less	<	
lessequal	<=	
equal	= ==	
moreequal	>=	
more	>	
unequal	<> != ~=	
not	! ~	not

---

Here are some things that `\numexpr` is not suitable for:

```
\scratchcounter = \numexpression
    "00000 bor "00001 bor "00020 bor "00400 bor "08000 bor "F0000
\relax

\ifcase \numexpression
    (\scratchcounterone > 5) && (\scratchcountertwo > 5)
\relax yes\else nop\fi
```

You can get an idea what the engines sees by setting `\tracingexpressions` to a value larger than zero. It shows the expression in `rpn` form.

```
\dimexpression 4pt * 2 + 6pt \relax
\dimexpression 2 * 4pt + 6pt \relax
\dimexpression 4pt * 2.5 + 6pt \relax
\dimexpression 2.5 * 4pt + 6pt \relax
\numexpression 2 * 4 + 6 \relax
\numexpression (1 + 2) * (3 + 4) \relax
```

The `\relax` is mandate simply because there are keywords involved so the parser needs to know where to stop scanning. It made no sense to be more clever and introduce fuzziness (so there is no room for exposing in-depth  $\text{\TeX}$  insight and expertise here). In case you wonder: the difference in performance between the  $\varepsilon\text{-TeX}$  expression mechanism and the more extended variant will normally not be noticed, probably because they both use a different approach and because the  $\varepsilon\text{-TeX}$  variant also has been optimized.<sup>3</sup>

The if-test shown before can be done using the new primitives `\ifdimexpression` and `\ifnumexpression` which are boolean tests with zero being false.

<sup>3</sup> I might add some features in the future.



## 5.22 Nodes

The  $\varepsilon$ -TeX primitive `\lastnodetype` is not honest in reporting the internal numbers as it uses its own values. But you can set `\internalcodesmode` to a non-zero value to get the real id's instead. In addition there is `\lastnodesubtype`.

Another last one is `\lastnamedcs` which holds the last match but this one should be used with care because one never knows if in the meantime something else ‘last’ has been seen.





# 6 Fonts

## 6.1 Introduction

The traditional T<sub>E</sub>X ligature and kerning routines are build in but anything more (like OpenType rendering) has to be implemented in Lua. In ConT<sub>E</sub>Xt we call the former base mode and the later node mode (we have some more modes). This conforms to the LuaT<sub>E</sub>X philosophy. When you pass a font to the frontend only the dimensions matter, as these are used in typesetting, and optionally ligatures and kerns when you rely on the built-in font handler. For math some extra data is needed, like information about extensibles and next in size glyphs. You can of course put more information in your Lua tables because when such a table is passed to T<sub>E</sub>X only that what is needed is filtered from it.

Because there is no built-in backend, virtual font information is not used. If you want to be compatible you'd better make sure that your tables are okay, and in that case you can best consult the LuaT<sub>E</sub>X manual. For instance, parameters like `extend` are backend related and the standard LuaT<sub>E</sub>X backend sets the standard here.

## 6.2 Defining fonts

All T<sub>E</sub>X 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 passed to `font.define`. When the callback is set, which is needed for `\font` to work, its function gets the name and size passed, and it has to return a valid font identifier (a positive number).

For the engine to work well, the following information has to be present at the font level:

KEY	VALUE TYPE	DESCRIPTION
<code>name</code>	string	metric (file) name
<code>original</code>	string	the name used in logging and feedback
<code>designsize</code>	number	expected size (default: 655360 == 10pt)
<code>size</code>	number	the required scaling (by default the same as <code>designsize</code> )
<code>characters</code>	table	the defined glyphs of this font
<code>fonts</code>	table	locally used fonts
<code>parameters</code>	hash	default: 7 parameters, all zero
<code>stretch</code>	number	the 'stretch'
<code>shrink</code>	number	the 'shrink'
<code>step</code>	number	the 'step'
<code>textcontrol</code>	bitset	this controls various code paths in the text engine
<code>hyphenchar</code>	number	default: T <sub>E</sub> X's <code>\hyphenchar</code>
<code>skewchar</code>	number	default: T <sub>E</sub> X's <code>\skewchar</code>
<code>nomath</code>	boolean	this key allows a minor speedup for text fonts; if it is present and true, then LuaT <sub>E</sub> X will not check the character entries for math-specific keys



oldmath	boolean	this key flags a font as representing an old school T <sub>E</sub> X math font and disables the OpenType code path
mathcontrol	bitset	this controls various code paths in the math engine, like enforcing the traditional code path
compactmath	boolean	experimental: use the smaller chain to locate a character
textscale	number	scale applied to math text
scriptscale	number	scale applied to math script
scriptscriptscale	number	scale applied to math script script

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	REMAPPING
slant	1
space	2
spacestretch	3
spaceshrink	4
xheight	5
quad	6
extraspace	7

The characters table is a Lua hash table where the keys are integers. When a character in the input is turned into a glyph node, it gets a character code that normally refers to an entry in that table. For proper paragraph building and math rendering the following fields can be present in an entry in the characters table. You can of course add all kind of extra fields. The engine only uses those that it needs for typesetting a paragraph or formula. The subtables that define ligatures and kerns are also hashes with integer keys, and these indices should point to entries in the main characters table.

Providing ligatures and kerns this way permits T<sub>E</sub>X to construct ligatures and add inter-character kerning. However, normally you will use an OpenType font in combination with Lua code that does this. In ConT<sub>E</sub>Xt we have base mode that uses the engine, and node mode that uses Lua. A monospaced font normally has no ligatures and kerns and is normally not processed at all.

KEY	TYPE	DESCRIPTION
width	number	width in sp (default 0)
height	number	height in sp (default 0)
depth	number	depth in sp (default 0)
italic	number	italic correction in sp (default 0)
topaccent	number	top accent alignment place in sp (default zero)
botaccent	number	bottom accent alignment place, in sp (default zero)
leftprotruding	number	left protruding factor (\lrcode)
rightprotruding	number	right protruding factor (\rrcode)
expansion	number	expansion factor (\efcode)
next	number	'next larger' character index



extensible	table	constituent parts of an extensible (traditional) recipe
vparts	table	constituent parts of a vertical (OpenType) recipe
hparts	table	constituent parts of a horizontal (OpenType) recipe
kerns	table	kerning information
ligatures	table	ligaturing information
mathkern	table	math cut-in specifications
smaller	number	the next smaller math size character

---

For example, here is the character ‘f’ (decimal 102) in the font `cmr10` at 10pt. The numbers that represent dimensions are in scaled 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 }
  }
}
```

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

The values of `topaccent`, `botaccent` and `mathkern` are used only for math accent and superscript placement, see page 103 in this manual for details. The values of `leftprotrusion` and `rightprotrusion` are used only when `\protrudechars` is non-zero. Whether or not expansion is used depends on the font's global expansion settings, as well as on the value of `\adjustspacing`.

A math character can have a `next` field that points to a next larger shape. However, the presence of `extensible` will overrule `next`, if that is also present. The `extensible` field in turn can be overruled by `vparts`, the OpenType version. 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 `hparts` and `vparts` are arrays of components. Each of those components is itself a hash of up to five keys:

KEY	TYPE	EXPLANATION
<code>glyph</code>	number	The character index. Note that this is an encoding number, not a name.
<code>extender</code>	number	One (1) if this part is repeatable, zero (0) otherwise.
<code>start</code>	number	The maximum overlap at the starting side (in scaled points).
<code>end</code>	number	The maximum overlap at the ending side (in scaled points).
<code>advance</code>	number	The total advance width of this item. It 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 `rightboundary`), with the values of 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 `rightboundary`), with the values being yet another small hash, with two fields:

KEY	TYPE	DESCRIPTION
<code>type</code>	number	the type of this ligature command, default 0
<code>char</code>	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  $\text{T}_{\text{E}}\text{X}$ . When  $\text{T}_{\text{E}}\text{X}$  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
<code>l + r =: n</code>	0	<code>=:</code>	<code> n</code>
<code>l + r =:   n</code>	1	<code>=:  </code>	<code> nr</code>
<code>l + r  =: n</code>	2	<code> =:</code>	<code> ln</code>
<code>l + r  =:   n</code>	3	<code> =:  </code>	<code> lnr</code>
<code>l + r =:  &gt; n</code>	5	<code>=:  &gt;</code>	<code>n r</code>
<code>l + r  =: &gt; n</code>	6	<code> =: &gt;</code>	<code>l n</code>
<code>l + r  =:  &gt; n</code>	7	<code> =:  &gt;</code>	<code>l nr</code>
<code>l + r  =:  &gt;&gt; n</code>	11	<code> =:  &gt;&gt;</code>	<code>ln r</code>

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 `mathcontrol` bitset is mostly there for experimental purposes. Because there is inconsistency in the OpenType math fonts with respect to for instance glyph dimensions, it is possible to force the traditional code path. We just mention the possible flags:

VALUE	EFFECT
0x0001	<code>usefontcontrol</code>



0x0002	overrule
0x0004	underrule
0x0008	radicalrule
0x0010	fractionrule
0x0020	accentskewhalf
0x0040	accentskewapply
0x0080	checkligatureandkern
0x0100	applyverticalitalickern
0x0200	applyordinaryitalickern
0x0400	applycharitalickern
0x0800	reboxcharitalickern
0x1000	applyboxeditalickern
0x2000	staircasekern
0x4000	applytextitalickern
0x8000	checktextitalickern
0x10000	checkspaceitalickern
0x20000	applyscriptitalickern
0x40000	italicshapekern

---

Compact math is an experimental feature. The smaller field in a character definition of a text character can point to a script character that itself can point to a scriptscript one. When set the `textscale`, `scriptscale` and `scriptscriptscale` is applied to those.

The `textcontrol` field is used to control some aspects of text processing. More options might be added in the future.

VALUE	EFFECT
0x0001	collapsehyphens

---

In ConT<sub>E</sub>Xt these are interfaced via pseudo features. The math control flags of a font can be overloaded by `\mathcontrolmode` on the spot and the set controls of a font can be queried by `\fontmathcontrol`. The text control flags in a font always win over the ones set by other parameters, like `\hyphenationmode`. They can be queried with `\fonttextcontrol`.

## 6.3 Virtual fonts

Virtual fonts have been introduced to overcome limitations of good old T<sub>E</sub>X. They were mostly used for providing a direct mapping from for instance accented characters onto a glyph. The backend was responsible for turning a reference to a character slot into a real glyph, possibly constructed from other glyphs. In our case there is no backend so there is also no need to pass this information through T<sub>E</sub>X. But it can of course be part of the font information and because it is a kind of standard, we describe it here.

A character is virtual when it has a `commands` array as part of the data. A virtual character can itself point to virtual characters but be careful with nesting as you can create loops and overflow the stack (which often indicates an error anyway).

At the font level there can be a an (indexed) `fonts` table. 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 in for instance a virtual font, you can use the `slot` command with a zero font reference, which indicates that the font itself is used. So, a table looks like this:

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

The first referenced font (at index 1) in this virtual font is `ptmr8a` loaded at 10pt, and the second is `psyr` loaded at a little over 9pt. The third one is a previously defined font that is known to LuaTeX as font id 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 frontend is only interested in the dimensions, ligatures and kerns of a font, which is the reason why the TeX engine didn't have to be extended when virtual fonts showed up: dealing with it is up to the driver that comes after the backend. In pdfTeX and LuaTeX that driver is integrated so there the backend also deals with virtual fonts. The first block in the next table is what the standard mentions. The `special` command is indeed special because it is an extension container. The mentioned engines only support pseudo standards where the content starts with `pdf:`. The last block is LuaTeX specific and will not be found in native fonts. These entries can be used in virtual fonts that are constructed in Lua.

But ... in LuaMetaTeX there is no backend built in but we might assume that the one provided deals with these entries. However, a provided backend can provide more and that is indeed what happens in ConTeXt. There, because we no longer have compacting (of passed tables) and unpacking (when embedding) of these tables going on we stay in the Lua domain. None of the virtual specification is ever seen in the engine.

COMMAND	ARGUMENTS	TYPE	DESCRIPTION
font	1	number	select a new font from the local <code>fonts</code> table
char	1	number	typeset this character number from the current font, and move right by the character's width
push	0		save current position
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 driver directive
nop	0		do nothing
slot	2	2 numbers	a shortcut for the combination of a font and char command
node	1	node	output this node (list), and move right by the width of this list



pdf	2	2 strings	output a pdf literal, the first string is one of origin, page, text, font, direct or raw; if you have one string only origin is assumed
lua	1	string, function	execute a Lua script when the glyph is embedded; in case of a function it gets the font id and character code passed
image	1	image	depends on the backend
comment	any	any	the arguments of this command are ignored

When a font id is set to 0 then it will be replaced by the currently assigned font id. This prevents the need for hackery with future id's.

The pdf option also accepts a mode keyword in which case the third argument sets the mode. That option will change the mode in an efficient way (passing an empty string would result in an extra empty lines in the pdf file. This option only makes sense for virtual fonts. The font mode only makes sense in virtual fonts. Modes are somewhat fuzzy and partially inherited from pdfTeX.

MODE	DESCRIPTION
origin	enter page mode and set the position
page	enter page mode
text	enter text mode
font	enter font mode (kind of text mode, only in virtual fonts)
always	finish the current string and force a transform if needed
raw	finish the current string

You always need to check what pdf code is generated because there can be all kind of interferences with optimization in the backend and fonts are complicated anyway. Here is a rather elaborate glyph commands example using such keys:

```
...
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')
  -- { "slot", 2, 97 },      -- an alternative for the previous two
  { "pop" },                 -- go all the way back
  { "down", -200000 },       -- move upwards by about 3pt
  { "special", "pdf: 1 0 0 rg" } -- switch to red color
  -- { "pdf", "origin", "1 0 0 rg" } -- switch to red color (alternative)
  { "rule", 500000, 20000 }  -- draw a bar
  { "special", "pdf: 0 g" }  -- back to black
  -- { "pdf", "origin", "0 g" }  -- back to black (alternative)
}
...
```



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 created 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.

The special can have a `pdf:`, `pdf:origin:`, `pdf:page:`, `pdf:direct:` or `pdf:raw:` prefix. When you have to concatenate strings using the `pdf` command might be more efficient.

*For the record: in ConT<sub>E</sub>Xt LMTX we no longer support the `pdf`, `image` and `special` keywords.*

## 6.4 Additional T<sub>E</sub>X commands

### 6.4.1 Font syntax

LuaT<sub>E</sub>X 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.

### 6.4.2 \fontid and \setfontid

```
\fontid\font
```

This primitive expands into a number. The currently used font id is 13. Here are some more:<sup>4</sup>

STYLE	COMMAND	FONT ID
normal	<code>\tf</code>	13
bold	<code>\bf</code>	<b>17</b>
italic	<code>\it</code>	22
bold italic	<code>\bi</code>	<b>24</b>

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 ConT<sub>E</sub>Xt 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.

The primitive `\setfontid` can be used to enable a font with the given id, which of course needs to be a valid one.

<sup>4</sup> Contrary to LuaT<sub>E</sub>X this is now a number so you need to use `\number` or `\the`. The same is true for some other numbers and dimensions that for some reason ended up in the serializer that produced a sequence of tokens.





### 6.4.3 `\glyphoptions`

In LuaTeX the `\noligs` and `\nokerns` primitives suppress these features but in LuaMetaTeX these primitives are gone. They are replaced by a more generic control primitive `\glyphoptions`. This numerical parameter is a bitset with the following fields:

VALUE	EFFECT
0x01	prevent left ligature
0x02	prevent right ligature
0x04	block left kern
0x08	block right kern
0x10	don't apply expansion
0x20	don't apply protrusion
0x40	apply xoffset to width
0x80	apply yoffset to height and depth

The effects speak for themselves. They provide detailed control over individual glyph, this because the current value of this option is stored with glyphs.

### 6.4.4 `\glyphscale`, `\glyphxscale`, `\glyphyscale` and `\scaledfontdimen`

The three scale parameters control the current scaling. They are traditional TeX integer parameters that operate independent of each other. The scaling is reflected in the dimensions of glyphs as well as in the related font dimensions, which means that units like `ex` and `em` work as expected. If you query a font dimensions with `\fontdimen` you get the raw value but with `\scaledfontdimen` you get the useable value.

### 6.4.5 `\fontspecdef`, `\fontspecid`, `\fontspecscale`, `\fontspecxscale`, `\fontspecyscale`

Because we have three scale related primitives `\glyphscale`, `\glyphxscale` and `\glyphyscale`, we also have a way to quickly set them all.

```
\fontspecdef \MyFontA 2 all 1000
\fontspecdef \MyFontB \MyFontA xscale 1200
```

The defined control sequence will set the font id (which is 2 in the case of `\MyFontA`) as well as the scale(s). Possible keywords are `scale`, `xscale`, `yscale` and `all`. By default the values are 1000. Instead of an id an already defined specification can be given in which case we start from a copy. This mechanism is still somewhat experimental and might evolve. The main reason for introducing it is that it gives less tracing.

Say that we have:

```
\fontspecdef\MyFoo\font xscale 1200 \relax
```

The four properties of such a specification can then be queried as follows:



```
[\the\fontspecid \MyFoo]
[\the\fontspecscale \MyFoo]
[\the\fontspecxscale\MyFoo]
[\the\fontspecyscale\MyFoo]
```

```
[13] [1000] [1200] [1000]
```

A font specification obeys grouping but is not a register. Like `\integerdef` and `\dimendef` it is just a control sequence with a special meaning.

### 6.4.6 `\glyphxoffset`, `\glyphyoffset`

These two parameters control the horizontal and vertical shift of glyphs with, when applied to a stretch of them, the horizontal offset probably being the least useful.

### 6.4.7 `\glyph`

This command is a variation in `\char` that takes keywords:

KEYWORD	EFFECT	type
<code>xoffset</code>	(virtual) horizontal shift	dimension
<code>yoffset</code>	(virtual) vertical shift	dimension
<code>xscale</code>	horizontal scaling	integer
<code>yscale</code>	vertical scaling	integer
<code>options</code>	glyph options	bitset
<code>font</code>	font	identifier
<code>id</code>	font	integer

The values default to the currently set values. Here is a ConTeXt example:

```
\ruledhbox{
  \ruledhbox{\glyph yoffset 1ex options 0 123}
  \ruledhbox{\glyph xoffset .5em yoffset 1ex options "C0 125}
  \ruledhbox{baseline\glyphyoffset 1ex \glyphxscale 800 \glyphyscale\glyphxs-
cale raised}
}
```

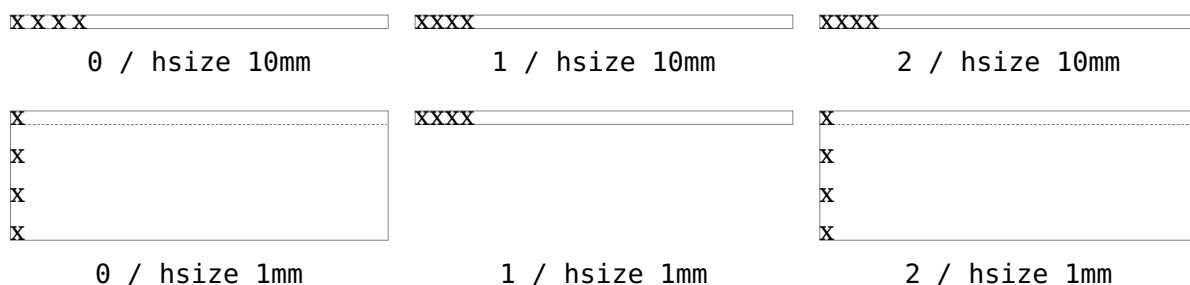
Visualized:



### 6.4.8 `\nospaces`

This new primitive can be used to overrule the usual `\spaceskip` related heuristics when a space character is seen in a text flow. The value 1 triggers no injection while 2 results in injection of a zero skip. In figure 6.1 we see the results for four characters separated by a space.





**Figure 6.1** The `\nospaces` options.

### 6.4.9 `\protrusionboundary`

The protrusion detection mechanism is enhanced a bit to enable a bit more complex situations. When protrusion characters are identified some nodes are skipped:

- ▶ zero glue
- ▶ penalties
- ▶ empty discretionaries
- ▶ normal zero kerns
- ▶ rules with zero dimensions
- ▶ math nodes with a surround of zero
- ▶ dir nodes
- ▶ empty horizontal lists
- ▶ local par nodes
- ▶ inserts, marks and adjusts
- ▶ boundaries
- ▶ whatsits

Because this can not be enough, you can also use a protrusion boundary node to make the next node being ignored. When the value is 1 or 3, the next node will be ignored in the test when locating a left boundary condition. When the value is 2 or 3, the previous node will be ignored when locating a right boundary condition (the search goes from right to left). This permits protrusion combined with for instance content moved into the margin:

```
\protrusionboundary1\llap{!\quad}«Who needs protrusion?»
```

## 6.5 The Lua font library

### 6.5.1 Introduction

The Lua font library is reduced to a few commands. Contrary to Lua<sub>T</sub><sub>E</sub>X there is no loading of `tfm` or `vf` files. The explanation of the following commands is in the Lua<sub>T</sub><sub>E</sub>X manual.

FUNCTION	DESCRIPTION
<code>current</code>	returns the id of the currently active font
<code>max</code>	returns the last assigned font identifier
<code>setfont</code>	enables a font setfont (sets the current font id)
<code>addcharacters</code>	adds characters to a font
<code>define</code>	defined a font
<code>id</code>	returns the id that relates to a command name

For practical reasons the management of font identifiers is still done by  $\text{T}_{\text{E}}\text{X}$  but it can become an experiment to delegate that to Lua as well.



### 6.5.2 Defining a font with `define`, `addcharacters` and `setfont`

Normally you will use a callback to define a font but there's also a Lua function that does the job.

```
id = font.define(<table> f)
```

Within reasonable bounds you can extend a font after it has been defined. Because some properties are best left unchanged this is limited to adding characters.

```
font.addcharacters(<number n>, <table> f)
```

The table passed can have the fields `characters` which is a (sub)table like the one used in `define`, and for virtual fonts a `fonts` table can be added. The characters defined in the `characters` table are added (when not yet present) or replace an existing entry. Keep in mind that replacing can have side effects because a character already can have been used. Instead of posing restrictions we expect the user to be careful. The `setfont` helper is a more drastic replacer and only works when a font has not been used yet.

### 6.5.3 Font ids: `id`, `max` and `current`

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

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

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

This is the largest used index so far. The currently active font id can be queried or set with:

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

### 6.5.4 Glyph data: `\glyphdatafield`, `\glyphscriptfield`, `\glyphstatefield`

These primitives can be used to set an additional glyph properties. Of course it's very macro package dependant what is done with that. It started with just the first one as experiment, simply because we had some room left in the glyph data structure. It's basically an single attribute. Then, when we got rid of the ligature pointer we could either drop it or use that extra field for some more, and because ConT<sub>E</sub>Xt already used the data field, that is what happened. The script and state fields are shorts, that is, they run from zero to `0xFFFF` where we assume that zero means 'unset'. Although they can be used for whatever purpose their use in ConT<sub>E</sub>Xt is fixed.



# 7 Languages, characters, fonts and glyphs

## 7.1 Introduction

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 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.

Those 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 at all. Instead, language information is passed along using language whatsit nodes 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 and a reference to a character in that 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.

## 7.2 Characters, glyphs and discretionaries

TeX82 (including pdfTeX) differentiates between char nodes and lig nodes. 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 LuaMetaTeX we no longer keep the list of components with the glyph node.



In LuaT<sub>E</sub>X, these two types are merged into one, somewhat larger structure called a glyph node. Besides having the old character, font, and component fields there are a few more, like ‘attr’ that we will see in section 10.2.12, these nodes also contain a subtype, that codes four main types and two additional ghost types. For ligatures, multiple bits can be set at the same time (in case of a single-glyph word).

- ▶ 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 constructed ligatures bit 1 is set.

The glyph nodes 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, LuaT<sub>E</sub>X allows 16383 separate languages, and words can be 256 characters long. The language is stored with each character. You can set `\firstvalidlanguage` to for instance 1 and make thereby language 0 an ignored hyphenation language.

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

Because the `\uchyph` value is saved in the actual nodes, its handling is subtly different from T<sub>E</sub>X82: 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 T<sub>E</sub>X82, 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 LuaT<sub>E</sub>X, all language variables are already frozen.

In traditional T<sub>E</sub>X the process of hyphenation is driven by `lccodes`. In LuaT<sub>E</sub>X we made this dependency less strong. There are several strategies possible. When you do nothing, the currently used `lccodes` are used, when loading patterns, setting exceptions or hyphenating a list.

When you set `\savingshyphcodes` to a value greater than zero the current set of `lccodes` will be saved with the language. In that case changing a `lccode` afterwards has no effect. However, you can adapt the set with:

```
\hjcode`a=`a
```

This change is global which makes sense if you keep in mind that the moment that hyphenation happens is (normally) when the paragraph or a horizontal box is constructed. When `\savingshyphcodes` was zero when the language got initialized you start out with nothing, otherwise you already have a set.

When a `\hjcode` is greater than 0 but less than 32 it indicates the to be used length. In the following example we map a character (x) onto another one in the patterns and tell the engine that `æ` counts as two characters. Because traditionally zero itself is reserved for inhibiting hyphenation, a value of 32 counts as zero.

Here are some examples (we assume that French patterns are used):



	foobar	foo-bar
\hjcode `x=`o	fxxbar	fxx-bar
\lefthyphenmin 3	ædipus	ædi-pus
\lefthyphenmin 4	ædipus	ædipus
\hjcode `æ=2	ædipus	ædi-pus
\hjcode `i=32 \hjcode `d=32	ædipus	ædipus

Carrying all this information with each glyph would give too much overhead and also make the process of setting up these codes more complex. A solution with hjcode sets was considered but rejected because in practice the current approach is sufficient and it would not be compatible anyway.

Beware: the values are always saved in the format, independent of the setting of \savingshyphcodes at the moment the format is dumped.

A boundary node normally would mark the end of a word which interferes with for instance discretionary injection. For this you can use the \wordboundary as a trigger. Here are a few examples of usage:

discrete---discrete

dis-  
crete—  
dis-  
crete

discrete\discretionary{}{}{---}discrete

discrete  
discrete

discrete\wordboundary\discretionary{}{}{---}discrete

dis-  
crete  
discrete

discrete\wordboundary\discretionary{}{}{---}\wordboundary discrete

dis-  
crete  
dis-  
crete

discrete\wordboundary\discretionary{---}{}{}\wordboundary discrete

dis-  
crete—  
dis-  
crete



We only accept an explicit hyphen when there is a preceding glyph and we skip a sequence of explicit hyphens since that normally indicates a -- or --- ligature in which case we can in a worse case usage get bad node lists later on due to messed up ligature building as these dashes are ligatures in base fonts. This is a side effect of separating the hyphenation, ligaturing and kerning steps.

The start and end of a sequence of characters is signalled by a glue, penalty, kern or boundary node. But by default also a hlist, vlist, rule, dir, whatsit, insert, and adjust node indicate a start or end. You can omit the last set from the test by setting flags in \hyphenationmode:

VALUE	BEHAVIOUR
	not strict
64	strict start
128	strict end
192	strict start and strict end

The word start is determined as follows:

NODE	BEHAVIOUR
<b>boundary</b>	yes when wordboundary
<b>hlist</b>	when the start bit is set
<b>vlist</b>	when the start bit is set
<b>rule</b>	when the start bit is set
<b>dir</b>	when the start bit is set
<b>whatsit</b>	when the start bit is set
<b>glue</b>	yes
<b>math</b>	skipped
<b>glyph</b>	exhyphenchar (one only) : yes (so no —)
<b>otherwise</b>	yes

The word end is determined as follows:

NODE	BEHAVIOUR
<b>boundary</b>	yes
<b>glyph</b>	yes when different language
<b>glue</b>	yes
<b>penalty</b>	yes
<b>kern</b>	yes when not italic (for some historic reason)
<b>hlist</b>	when the end bit is set
<b>vlist</b>	when the end bit is set
<b>rule</b>	when the end bit is set
<b>dir</b>	when the end bit is set
<b>whatsit</b>	when the end bit is set
<b>ins</b>	when the end bit is set
<b>adjust</b>	when the end bit is set

Figures 7.1 upto 7.5 show some examples. In all cases we set the min values to 1 and make sure that the words hyphenate at each character.





o-	o-	o-	o-
n-	n-	n-	n-
e	e	e	e
0	64	128	192

**Figure 7.1** one

o-	o-	onet-	onet-
n-	n-	w-	w-
et-	et-	o	o
w-	w-		
o	o		
0	64	128	192

**Figure 7.2** one\null two

o-	o-	onet-	onet-
n-	n-	w-	w-
et-	et-	o	o
w-	w-		
o	o		
0	64	128	192

**Figure 7.3** \null one\null two

o-	o-	onetwo	onetwo
n-	n-		
et-	et-		
w-	w-		
o	o		
0	64	128	192

**Figure 7.4** one\null two\null

In traditional T<sub>E</sub>X ligature building and hyphenation are interwoven with the line break mechanism. In LuaT<sub>E</sub>X these phases are isolated. As a consequence we deal differently with (a sequence of) explicit hyphens. We already have added some control over aspects of the hyphenation and yet another one concerns automatic hyphens (e.g. - characters in the input).

Hyphenation and discretionary injection is driven by a mode parameter which is a bitset made from the following values, some of which we saw in the previous examples.

- 1 honour (normal) \discretionary's
- 2 turn - into (automatic) discretionaries
- 4 turn \- into (explicit) discretionaries
- 8 hyphenate (syllable) according to language
- 16 hyphenate uppercase characters too (replaces \uchyph
- 32 permit break at an explicit hyphen (border cases)
- 64 traditional T<sub>E</sub>X compatibility wrt the start of a word
- 128 traditional T<sub>E</sub>X compatibility wrt the end of a word
- 256 use \automatichyphenpenalty



o-	o-	onetwo	onetwo
n-	n-		
et-	et-		
w-	w-		
o	o		
0	64	128	192

**Figure 7.5** `\null one\null two\null`

```

512  use \explicithyphenpenalty
1024  turn glue in discretionaries into kerns
2048  okay, let's be even more tolerant in discretionaries
4096  and again we're more permissive
16384 controls how successive explicit discretionaries are handled in base mode
8192  treat all discretionaries equal when breaking lines (in all three passes)
32768 kick in the handler (experiment)
65536 feedback compound snippets

```

Some of these options are still experimental, simply because not all aspects and side effects have been explored. You can find some experimental use cases in `ConTEXt`.

## 7.3 The main control loop

In Lua<sub>T<sub>E</sub></sub>X'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.

1. The `\accent` primitive 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 T<sub>E</sub>X82 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 T<sub>E</sub>X82, `\accent` prohibits hyphenation of the current word. Since in Lua<sub>T<sub>E</sub></sub>X hyphenation only takes place on ‘character’ nodes, it is possible to achieve the same effect. Of course, modern Unicode aware macro packages will not use the `\accent` primitive at all but try to map directly on composed characters. This change of meaning did happen with `\char`, that now generates ‘glyph’ nodes with a character subtype. In traditional T<sub>E</sub>X there was a strong relationship between the 8-bit input encoding, hyphenation and glyphs taken from a font. In Lua<sub>T<sub>E</sub></sub>X 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 always use Lua to do so, because fonts are available as Lua table.
2. All the results of processing in math mode eventually become nodes with ‘glyph’ subtypes. In fact, the result of processing math is just a regular list of glyphs, kerns, glue, penalties, boxes etc.
3. Automatic discretionaries are handled differently. T<sub>E</sub>X82 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.<sup>5</sup>

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. Initially TeX sets the `\exhyphenchar=-1`. 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 behaviour 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.

4. The `\setlanguage` command 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`.
5. The `\noboundary` command (that prohibits word boundary processing where that would normally take place) now does create nodes. These nodes 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.
6. 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.

Because we tend to make hard coded behaviour configurable a few new primitives have been added:

```
\hyphenpenaltymode
\automatichyphenpenalty
\explicithyphenpenalty
```

The usage of these penalties is controlled by the `\hyphenationmode` flags 256 and 512 and when these are not set `\exhyphenpenalty` is used.

You can use the `\tracinghyphenation` variable to get a bit more information about what happens.

VALUE	EFFECT
-------	--------

1	report redundant pattern (happens by default in LuaTeX)
---	---

<sup>5</sup> When TeX showed up we didn't have Unicode yet and being limited to eight bits meant that one sometimes had to compromise between supporting character input, glyph rendering, hyphenation.



- 2 report words that reach the hyphenator and got treated
  - 3 show the result of a hyphenated word (a node list)
- 

## 7.4 Loading patterns and exceptions

Although we keep the traditional approach towards hyphenation (which is still superior) the implementation of the hyphenation algorithm in LuaTeX is quite different from the one in T<sub>E</sub>X82.

After expansion, the argument for `\patterns` has to be proper utf8 with individual patterns separated by spaces, no `\char` or `\chardef` commands are allowed. The current implementation is quite strict and will reject all non-Unicode characters. Likewise, the expanded argument for `\hyphenation` also has to be proper utf8, but here a bit of extra syntax is provided:

1. Three sets of arguments in curly braces (`{-}{-}{-}`) indicate 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 T<sub>E</sub>X-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 language library. This loading method is quite a bit faster than going through the T<sub>E</sub>X 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 inbetween ‘boun’ and ‘daries’:

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

The motivation behind the  $\varepsilon$ -T<sub>E</sub>X extension `\savingshyphcodes` was that hyphenation heavily depended on font encodings. This is no longer true in LuaTeX, and the corresponding primitive is



basically ignored. Because we now have \hj code, the case related codes can be used exclusively for \uppercase and \lowercase.

The three curly brace pair pattern in an exception can be somewhat unexpected so we will try to explain it by example. The pattern `foo{}{}{x}bar` pattern creates a lookup `fooxbar` and the pattern `foo{}{}{}bar` creates `foobar`. Then, when a hit happens there is a replacement text (x) or none. Because we introduced penalties in discretionary nodes, the exception syntax now also can take a penalty specification. The value between square brackets is a multiplier for \exceptionpenalty. Here we have set it to 10000 so effectively we get 30000 in the example.

x{a-}{}{-b}{}{x}{a-}{}{-b}{}{x}{a-}{}{-b}{}{x}{a-}{}{-b}{}{xx}			
10em	3em	0em	6em
123 xxxxxx 123	123 xxa- -bxa- -bxa- -bxx 123	123 xa- -bxa- -bxa- -bxa- -bxx 123	123 xxxxxx xxxxxx xxa- -bxxxx xxa- -bxxxx 123

x{a-}{}{-b}{}{x}{a-}{}{-b}{}[3]{x}{a-}{}{-b}{}[1]{x}{a-}{}{-b}{}{xx}			
10em	3em	0em	6em
123 xxxxxx 123	123 xa- -bxxxa- -bxx 123	123 xa- -bxxxa- -bxx 123	123 xxxxa- -bxx xxxxxx xxxxxx xa- -bxxxxx 123

z{a-}{}{-b}{}{z}{a-}{}{-b}{}{z}{a-}{}{-b}{}{z}{a-}{}{-b}{}{z}z			
10em	3em	0em	6em
123 zzzzzz 123	123 za- -bza- -bza- -b 123	123 za- -bza- -bza- -b a- -b23	123 zzzzzz zzzzzz zzza- -bzz zzzzzz 123

z{a-}{}{-b}{}{z}{a-}{}{-b}{}{z}[3]{a-}{}{-b}{}{z}[1]{a-}{}{-b}{}{z}z			
10em	3em	0em	6em
123 zzzzzz 123	123 za- -bzzzz 123	123 za- -bzzzz a- -b23	123 zzzzzz zzzzzz za- -bzzzz a- -bzzzzz 123



## 7.5 Applying hyphenation

The internal structures LuaTeX uses for the insertion of discretionary 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 pattern files generated by Patgen, 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.

There are a few differences between LuaTeX and TeX82 that are a direct result of the implementation:

- ▶ 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 `language.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 discretionary, instead of TeX82’s `\hyphenchar`, and the values of the language-specific variables `\preexhyphenchar` and `\postexhyphenchar` for explicit discretionary (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 `language.hyphenate` function the current values are used.
- ▶ The hyphenation exception dictionary is maintained as key-value hash, and that is also dynamic, so the `hyph_size` setting is not used either.

Because we store penalties in the disc node the `\discretionary` command has been extended to accept an optional penalty specification, so you can do the following:

```
\hsizelmm
1:foo{\hyphenpenalty 10000\discretionary{}{}{}}bar\par
2:foo\discretionary penalty 10000 {}{}{}}bar\par
3:foo\discretionary{}{}{}}bar\par
```

This results in:

```
1:foobar
2:foobar
```



3:foo  
bar

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 behaviour is compatible with `\patterns` and `\hyphenation`.

Lua $\text{\TeX}$  also hyphenates the first word in a paragraph. Words can be up to 256 characters long (up from 64 in  $\text{\TeX}$ 82). Longer words are ignored 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  $\text{\TeX}$ 82, but there the behaviour cannot be controlled).

If you are using the Lua function `language.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.

## 7.6 Applying ligatures and kerning

After all possible hyphenation points have been inserted in the list, Lua $\text{\TeX}$  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 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 Lua $\text{\TeX}$  itself in order to make sure that the other, non-overruled, routine continues to function properly.

Although we could improve the situation the reality is that in modern OpenType fonts ligatures can be constructed in many ways: by replacing a sequence of characters by one glyph, or by selectively replacing individual glyphs, or by kerning, or any combination of this. Add to that contextual analysis and it will be clear that we have to let Lua do that job instead. The generic font handler that we provide (which is part of Con $\text{\TeX}$ t) distinguishes between base mode (which essentially is what we describe here and which delegates the task to  $\text{\TeX}$ ) and node mode (which deals with more complex fonts).

In so called base mode, where  $\text{\TeX}$  does the work, the ligature construction (normally) goes in small steps. An f followed by an f becomes an ff ligatures and that one followed by an i



can become a ffi ligature. The situation can be complicated by hyphenation points between these characters. When there are several in a ligature collapsing happens. Flag "4000 in the \hyphenationmode variable determines if this happens lazy or greedy, i.e. the first hyphen wins or the last one does. In practice a ConT<sub>E</sub>Xt user won't have to deal with this because most fonts are processed in node mode.

## 7.7 Breaking paragraphs into lines

This code is almost unchanged, but because of the above-mentioned changes with respect to discretionaries and ligatures, line breaking will potentially be different from traditional T<sub>E</sub>X. The actual line breaking code is still based on the T<sub>E</sub>X82 algorithms, and there can be no discretionaries inside of discretionaries. But, as patterns evolve and font handling can influence discretionaries, you need to be aware of the fact that long term consistency is not an engine matter only.

But that situation is now fairly common in LuaT<sub>E</sub>X, due to the changes to the ligaturing mechanism. And also, the LuaT<sub>E</sub>X discretionary nodes are implemented slightly different from the T<sub>E</sub>X82 nodes: the no\_break text is now embedded inside the disc node, where previously these nodes kept their place in the horizontal list. In traditional T<sub>E</sub>X the discretionary node contains a counter indicating how many nodes to skip, but in LuaT<sub>E</sub>X we store the pre, post and replace text in the discretionary node.

The combined effect of these two differences is that LuaT<sub>E</sub>X does not always use all of the potential breakpoints in a paragraph, especially when fonts with many ligatures are used. Of course kerning also complicates matters here.

## 7.8 The language library

### 7.8.1 new and id

This library provides the interface to LuaT<sub>E</sub>X's structure representing a language, and the associated functions.

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

The number returned is the internal \language id number this object refers to.

### 7.8.2 hyphenation

You can load exceptions with:





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

When no string is given (the first example) a string with all exceptions is returned.

### 7.8.3 clearhyphenation and clean

This either returns the current hyphenation exceptions for this language, or adds new ones. The syntax of the string is explained in section 7.4.

```
language.clearhyphenation(<language> l)
```

This call clears the exception dictionary (string) for this language.

```
<string> n = language.clean(<language> l, <string> o)
<string> n = language.clean(<string> o)
```

This function creates a hyphenation key from the supplied hyphenation value. The syntax of the argument string is explained in section 7.4. This function is useful if you want to do something else based on the words in a dictionary file, like spell-checking.

### 7.8.4 patterns and clearpatterns

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

This adds additional patterns for this language object, or returns the current set. The syntax of this string is explained in section 7.4.

```
language.clearpatterns(<language> l)
```

This can be used to clear the pattern dictionary for a language.

### 7.8.5 hyphenationmin

This function sets (or gets) the value of the TeX parameter \hyphenationmin.

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

### 7.8.6 [pre|post][ex|]hyphenchar

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

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



These two are used to get or set the ‘pre-break’ and ‘post-break’ hyphen characters for implicit hyphenation in this language. The initial values are decimal 45 (hyphen) and decimal 0 (indicating emptiness).

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

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

These gets or set the ‘pre-break’ and ‘post-break’ hyphen characters for explicit hyphenation in this language. Both are initially decimal 0 (indicating emptiness).

### 7.8.7 hyphenate

The next call 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.

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

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 7.2 for more details.

### 7.8.8 [set|get]hjcode

The following two commands can be used to set or query hj codes:

```
language.sethjcode(<language> l, <number> char, <number> usedchar)
<number> usedchar = language.gethjcode(<language> l, <number> char)
```

When you set a hjcode the current sets get initialized unless the set was already initialized due to \savingsphcodes being larger than zero.



# 8 Math

## 8.1 Traditional alongside OpenType

End 2021 this chapter started with this:

“At this point there is no difference between LuaMetaTeX and LuaTeX with respect to math.<sup>6</sup> 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 or considered in the past that are now added to the engine.

You might be surprised that we don’t use all these new control features in ConTeXt LMTX but who knows what might happen because users drive it. The main reason for adding so much is that I decided it made more sense to be complete now than gradually add more and more. At some point we should be able to say ‘This is it’. Also, when looking at these features, you need to keep in mind that when it comes to math, L<sup>A</sup>TeX is the dominant macro package and it never needed these engine features, so most are probably just here for exploration purposes.”

Although we still process math as TeX does, there have been some fundamental changes to the machinery. Most of that is discussed in documents that come with ConTeXt and in Mikael Sundqvist math manual. Together we explored some new ways to deal with math spacing, penalties, fencing, operators, fractions, atoms and other features of the TeX engine. We started from the way ConTeXt used the already present functionality combine with sometimes somewhat dirty (but on the average working well) tricks.

It will take a while before this chapter is updated. If you find errors or things missing, let me know. A lot of pairwise spacing primitives were dropped but also quite a bit of new ones introduced to control matters. Much in LuaMetaTeX math handling is about micro-typography and for us the results are quite visible. But, as far as we know, there have never been complaints or demands in the direction of the features discussed here. Also, TeX math usage outside ConTeXt is rather chiselled in stone (already for nearly three decades) so we don’t expect other macro packages to use the new features anyway.

## 8.2 Unicode math characters

For various reasons we need to encode a math character in a 32 bit number and because we often also need to keep track of families and classes the range of characters is limited to 20 bits. There are upto 64 classes (more than in LuaTeX) and 64 families (less than in LuaTeX). The upper limit of characters is less than what Unicode offers but for math we’re okay. If needed we can provide less families.

<sup>6</sup> This might no longer be true because we have more control options that define default behavior and also have a more extensive scaling model. Anyway, it should not look worse, and maybe even a bit better.



The math primitives from T<sub>E</sub>X are kept as they are, except for the ones that convert from input to math commands: `mathcode`, and `delcode`. These two now allow for the larger character codes argument on the left hand side of the equals sign. The number variants of some primitives might be dropped in favor of the primitives that read more than one separate value (class, family and code), for instance:

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

The altered T<sub>E</sub>X82 primitives are:

PRIMITIVE	MIN	MAX		MIN	MAX
<code>\mathcode</code>	0	10FFFF	=	0	8000
<code>\delcode</code>	0	10FFFF	=	0	FFFFFF

The unaltered ones are:

PRIMITIVE	MIN	MAX
<code>\mathchardef</code>	0	8000
<code>\mathchar</code>	0	7FFF
<code>\mathaccent</code>	0	7FFF
<code>\delimiter</code>	0	7FFFFFFF
<code>\radical</code>	0	7FFFFFFF

In LuaT<sub>E</sub>X we support the single number primitives \*with num in their name) conforming the X<sub>Y</sub>T<sub>E</sub>X method. For the moment that still works but you need to figure out the number yourself. The split number variants are more future safe with respect to classes and families. We don't document `\Umathcharnumdef`, `\Umathcharnum`, `\Umathcodenum` and `\Udelcodenum` here any longer.

PRIMITIVE		CLASS	FAMILY	CHARACTER
<code>\Umathchardef</code>	csname	"40	"40	"FFFFF
<code>\Umathcode</code>		"40	"40	"FFFFF
<code>\Udelcode</code>	"FFFFF	"40	"40	"FFFFF
<code>\Umathchar</code>		"40	"40	"FFFFF
<code>\Umathaccent</code>		"40	"40	"FFFFF
<code>\Udelimiter</code>		"40	"40	"FFFFF
<code>\Uradical</code>		"40	"40	"FFFFF

So, there are upto 64 classes of which at this moment about 20 are predefined so, taking some future usage by the engine into account, you can assume 32 upto 63 to be available for any purpose. The number of families has been reduced from 256 to 64 which is plenty for daily use in an OpenType setup. If we ever need to expand the Unicode range there will be less families. The values of begin and end classes and the number of classes can be fetched from the Lua status table.

Given the above, specifications typically look like:

```
\Umathchardef \xx = "1 "0 "456
```



`\Umathcode` 123 = "1 "0 "789

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 `\Umathoperator-size` parameter. Old school fonts can still be handled but you need to set up the engine to do that; this can be done per font. In principle we assume that OpenType fonts are used, which is no big deal because loading fonts is already under Lua control. At that moment the distinction between small and large delimiters will be gone. Of course an alternative is to support a specific large size but that is unlikely to happen.

The `\Umathaccent` command accepts optional keywords to control various details regarding math accents. See section 8.7.2 below for details.

There are more new primitives and all of these will be explained in following sections. For instance these are variants of radicals and delimiters all are set the same::

PRIMITIVE	CLASS	FAMILY	character
<code>\Uroot</code>	"40	"40	"FFFFF
<code>\Uoverdelimiter</code>	"40	"40	"FFFFF
<code>\Uunderdelimiter</code>	"40	"40	"FFFFF
<code>\Udelimiterover</code>	"40	"40	"FFFFF
<code>\Udelimiterunder</code>	"40	"40	"FFFFF

In addition there are `\UVextensible` and `\Uoperator` and extended versions of fenced: `\Uleft`, `\Uright` and `\Umiddle`. There is also `\Uover` and similar primitives that expect the numerator and denominator after the primitive. In addition to regular scripts there are prescripts and a dedicated prime script. Many of these U primitives can be controlled by options and keywords.

## 8.3 Setting up the engine

Processing math is controlled by `\mathfontcontrol`, a numeric bitset parameter. The recommended bits are marked with a star but it really depends on the macro package to set up the machinery well. Of course one can just enable all and see what happens.

BIT	NAME	
0x00001	usefontcontrol	*
0x00002	overrule	*
0x00004	underrule	*
0x00008	radicalrule	*
0x00010	fractionrule	*
0x00020	accentskewhalf	*
0x00040	accentskewapply	*
0x00080	checkligatureandkern	
0x00100	applyverticalitalickern	*
0x00200	applyordinaryitalickern	*
0x00400	applycharitalickern	
0x00800	reboxcharitalickern	
0x01000	applyboxeditalickern	*



0x02000	staircasekern	*
0x04000	applytextitalickern	*
0x08000	checktextitalickern	*
0x10000	checkspaceitalickern	
0x20000	applyscriptitalickern	*
0x40000	italicshapekern	*

---

One reason for this approach is that traditional and OpenType fonts have different approaches (especially when it comes to dealing with the width and italic corrections) and is even more complicated by the fact that the fonts are often inconsistent (within and between). In ConT<sub>E</sub>Xt we deal with this by runtime fixes to fonts. In any case the Cambria font is taken as reference.

## 8.4 Math styles

### 8.4.1 `\mathstyle`

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, LuaT<sub>E</sub>X adds the new primitive: `\mathstyle`. This is a ‘convert command’ like e.g. `\romannumeral`: its value can only be read, not set. Beware that contrary to LuaT<sub>E</sub>X this is now a proper number so you need to use `\number` or `\the` in order to serialize it.

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 they 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
```

Sometimes you won't get what you expect so a bit of explanation might help to understand what happens. When math is parsed and expanded it gets turned into a linked list. In a second pass the formula will be build. This has to do with the fact that in order to determine the automatically chosen sizes (in for instance fractions) following content can influence preceding sizes. A side effect of this is for instance that one cannot change the definition of a font family (and thereby reusing numbers) because the number that got used is stored and used in the second pass (so changing `\fam 12` mid-formula spoils over to preceding use of that family).

The style switching primitives like `\textstyle` are turned into nodes so the styles set there are frozen. The `\mathchoice` primitive results in four lists being constructed of which one is used in the second pass. The fact that some automatic styles are not yet known also means that the `\mathstyle` primitive expands to the current style which can of course be different from the one really used. It's a snapshot of the first pass state. As a consequence in the following example you get a style number (first pass) typeset that can actually differ from the used style (second pass). In the case of a math choice used ungrouped, the chosen style is used after the choice too, unless you group.



```

[a:\number\mathstyle]\quad
\bgroup
\mathchoice
  {\bf \scriptstyle      (x:d :\number\mathstyle)}
  {\bf \scriptscriptstyle (x:t :\number\mathstyle)}
  {\bf \scriptscriptstyle (x:s :\number\mathstyle)}
  {\bf \scriptscriptstyle (x:ss:\number\mathstyle)}
\egroup
\quad[b:\number\mathstyle]\quad
\mathchoice
  {\bf \scriptstyle      (y:d :\number\mathstyle)}
  {\bf \scriptscriptstyle (y:t :\number\mathstyle)}
  {\bf \scriptscriptstyle (y:s :\number\mathstyle)}
  {\bf \scriptscriptstyle (y:ss:\number\mathstyle)}
\quad[c:\number\mathstyle]\quad
\bgroup
\mathchoice
  {\bf \scriptstyle      (z:d :\number\mathstyle)}
  {\bf \scriptscriptstyle (z:t :\number\mathstyle)}
  {\bf \scriptscriptstyle (z:s :\number\mathstyle)}
  {\bf \scriptscriptstyle (z:ss:\number\mathstyle)}
\egroup
\quad[d:\number\mathstyle]

```

This gives:

$$[a:0] \quad (\mathbf{x:d:4}) \quad [b:0] \quad (\mathbf{y:d:4}) \quad [c:0] \quad (\mathbf{z:s:6}) \quad [d:0]$$

$$[a:2] \quad (\mathbf{x:t:6}) \quad [b:2] \quad (\mathbf{y:t:6}) \quad [c:2] \quad (\mathbf{z:ss:6}) \quad [d:2]$$

Using `\begingroup ... \endgroup` instead gives:

$$[a:0] \quad (\mathbf{x:d:4}) \quad [b:0] \quad (\mathbf{y:s:6}) \quad [c:0] \quad (\mathbf{z:ss:6}) \quad [d:0]$$

$$[a:2] \quad (\mathbf{x:t:6}) \quad [b:2] \quad (\mathbf{y:ss:6}) \quad [c:2] \quad (\mathbf{z:ss:6}) \quad [d:2]$$

This might look wrong but it's just a side effect of `\mathstyle` expanding to the current (first pass) style and the number being injected in the list that gets converted in the second pass. It all makes sense and it illustrates the importance of grouping. In fact, the math choice style being effective afterwards has advantages. It would be hard to get it otherwise.

### 8.4.2 `\Ustack`

There are a few math commands in  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  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,  $\mathrm{LuaT}_{\mathrm{E}}\mathrm{X}$  introduces a special prefix command: `\Ustack`:

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


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

### 8.4.3 The new `\cramped ...style` commands

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.

In Eijkhouts “TeX by Topic” the rules for handling styles in scripts are described as follows:

- ▶ In any style superscripts and subscripts are taken from the next smaller style. Exception: in display style they are in script style.
- ▶ Subscripts are always in the cramped variant of the style; superscripts are only cramped if the original style was cramped.
- ▶ In an `... \over ...` formula in any style the numerator and denominator are taken from the next smaller style.
- ▶ The denominator is always in cramped style; the numerator is only in cramped style if the original style was cramped.
- ▶ Formulas under a `\sqrt` or `\overline` are in cramped style.

In LuaTeX one can set the styles in more detail which means that you sometimes have to set both normal and cramped styles to get the effect you want. (Even) if we force styles in the script using `\scriptstyle` and `\crampedscriptstyle` we get this:

STYLE	EXAMPLE
default	$b^{x=xx}_{x=xx}$
script	$b^{x=xx}_{x=xx}$
crampedscript	$b^{x=xx}_{x=xx}$

Now we set the following parameters

```
\setmathspacing 0 3 \scriptstyle = 30mu
\setmathspacing 0 3 \scriptstyle = 30mu
```

This gives a different result:

STYLE	EXAMPLE
default	$b^x \quad =xx$ $\quad \quad x=xx$
script	$b^x \quad =xx$ $\quad \quad x=xx$
crampedscript	$b^{x=xx}$ $\quad \quad x=xx$

But, as this is not what is expected (visually) we should say:





```

\setmathspacing 0 3 \scriptstyle      = 30mu
\setmathspacing 0 3 \scriptstyle      = 30mu
\setmathspacing 0 3 \crampedscriptstyle = 30mu
\setmathspacing 0 3 \crampedscriptstyle = 30mu

```

Now we get:

STYLE	EXAMPLE
default	$b_x^x =_{xx}$
script	$b_x^x =_{xx}$
crampedscript	$b_x^x =_{xx}$

## 8.5 Math parameter settings

### 8.5.1 Many new `\Umath*` primitives

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 turning some hard codes properties into parameters.

PRIMITIVE NAME	DESCRIPTION
<code>\Umathquad</code>	the width of 18 mu's
<code>\Umathaxis</code>	height of the vertical center axis of the math formula above the baseline
<code>\Umathoperatorsize</code>	minimum size of large operators in display mode
<code>\Umathoverbarkern</code>	vertical clearance above the rule
<code>\Umathoverbarrule</code>	the width of the rule
<code>\Umathoverbarvgap</code>	vertical clearance below the rule
<code>\Umathunderbarkern</code>	vertical clearance below the rule
<code>\Umathunderbarrule</code>	the width of the rule
<code>\Umathunderbarvgap</code>	vertical clearance above the rule
<code>\Umathradicalkern</code>	vertical clearance above the rule
<code>\Umathradicalrule</code>	the width of the rule
<code>\Umathradicalvgap</code>	vertical clearance below the rule
<code>\Umathradicaldegreebefore</code>	the forward kern that takes place before placement of the radical degree
<code>\Umathradicaldegreeafter</code>	the backward kern that takes place after placement of the radical degree
<code>\Umathradicaldegreeraise</code>	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
<code>\Umathstackvgap</code>	vertical clearance between the two elements in an <code>\atop</code> stack
<code>\Umathstacknumup</code>	numerator shift upward in <code>\atop</code> stack
<code>\Umathstackdenomdown</code>	denominator shift downward in <code>\atop</code> stack
<code>\Umathfractionrule</code>	the width of the rule in a <code>\over</code>



<code>\Umathfractionnumvgap</code>	vertical clearance between the numerator and the rule
<code>\Umathfractionnumup</code>	numerator shift upward in <code>\over</code>
<code>\Umathfractiondenomvgap</code>	vertical clearance between the denominator and the rule
<code>\Umathfractiondenomdown</code>	denominator shift downward in <code>\over</code>
<code>\Umathfractiondelsize</code>	minimum delimiter size for <code>\dotswithdelims</code>
<code>\Umathlimitabovevgap</code>	vertical clearance for limits above operators
<code>\Umathlimitabovebgap</code>	vertical baseline clearance for limits above operators
<code>\Umathlimitabovekern</code>	space reserved at the top of the limit
<code>\Umathlimitbelowvgap</code>	vertical clearance for limits below operators
<code>\Umathlimitbelowbgap</code>	vertical baseline clearance for limits below operators
<code>\Umathlimitbelowkern</code>	space reserved at the bottom of the limit
<code>\Umathoverdelimitervgap</code>	vertical clearance for limits above delimiters
<code>\Umathoverdelimiterbgap</code>	vertical baseline clearance for limits above delimiters
<code>\Umathunderdelimitervgap</code>	vertical clearance for limits below delimiters
<code>\Umathunderdelimiterbgap</code>	vertical baseline clearance for limits below delimiters
<code>\Umathsubshiftdrop</code>	subscript drop for boxes and subformulas
<code>\Umathsubshiftdown</code>	subscript drop for characters
<code>\Umathsupshiftdrop</code>	superscript drop (raise, actually) for boxes and subformulas
<code>\Umathsupshiftup</code>	superscript raise for characters
<code>\Umathsubsupshiftdown</code>	subscript drop in the presence of a superscript
<code>\Umathsubtopmax</code>	the top of standalone subscripts cannot be higher than this above the baseline
<code>\Umathsupbottommin</code>	the bottom of standalone superscripts cannot be less than this above the baseline
<code>\Umathsupsubbottommax</code>	the bottom of the superscript of a combined super- and subscript be at least as high as this above the baseline
<code>\Umathsubsupvgap</code>	vertical clearance between super- and subscript
<code>\Umathspaceafterscript</code>	additional space added after a super- or subscript
<code>\Umathconnectoroverlapmin</code>	minimum overlap between parts in an extensible recipe

In addition to the above official OpenType font parameters we have these:

PRIMITIVE NAME	DESCRIPTION
<code>\Umathprimeraisepercent</code>	the percentage that the vertical position is scaled
<code>\Umathprimeshiftup</code>	the prime variant of <code>\SuperscriptShiftUp</code>
<code>\Umathprimebaselinedropmax</code>	the prime variant of <code>\SuperscriptBaselineDropMax</code>
<code>\Umathprimeshiftupcramped</code>	the prime variant of <code>\SuperscriptShiftUpCramped</code>
<code>\Umathprimespaceafter</code>	the prime variant of <code>\UmathSpaceAfterScript</code>
<code>\Umathprimewidthpercent</code>	the percentage of width that gets added
<code>\Umathspacebeforescript</code>	the prescript variant of <code>\UmathSpaceAfterScript</code>
<code>\Umathnolimitsupfactor</code>	a multiplier for the way limits are shifted up and down
<code>\Umathnolimitsubfactor</code>	a multiplier for the way limits are shifted up and down
<code>\Umathaccenttopshiftup</code>	the amount that a top accent is shifted up
<code>\Umathaccentbottomshiftdown</code>	the amount that a bottom accent is shifted down



<code>\Umathflattenedaccenttopshiftup</code>	the amount that a wide top accent is shifted up
<code>\Umathflattenedaccentbottomshiftdown</code>	the amount that a wide bottom accent is shifted down
<code>\Umathaccentbasedepth</code>	the complement of <code>\UmathAccentBaseHeight</code>
<code>\Umathaccentflattenedbasedepth</code>	the complement of <code>\UmathFlattenedAccentBaseHeight</code>

These parameters not only provide a bit more control over rendering, they also can be used in compensating issues in font, because no font is perfect. Some are the side effects of experiments and they have CamelCase companions in the MathConstants table.

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.

There are quite some parameters that can be set and there are eight styles, which means a lot of keying in. For that reason it is possible to set parameters groupwise

PRIMITIVE NAME	DESCRIPTION
<code>\alldisplaystyles</code>	set both display styles
<code>\alltextstyles</code>	set both text styles
<code>\allscriptstyles</code>	set both script styles
<code>\allscriptscriptstyles</code>	set both scriptscript styles
<code>\allmathstyles</code>	set all eight styles
<code>\allsplitstyles</code>	set all display and text styles, and reset all script(script) styles
<code>\alluncrampedstyles</code>	set all four uncramped styles
<code>\allcrampedstyles</code>	set all four cramped styles

These groups are especially handy when you set up inter atom spacing, pre- and post atom penalties and atom rules.

## 8.5.2 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, Lua $\TeX$  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 used in the  $\TeX$ book. Assignments to `\textfont` set the values for the cramped and uncramped display and text styles, `\scriptfont` sets the script styles, and `\scriptscriptfont` sets the scriptscript styles, so we have 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	TFM / OPENTYPE
<b>\Umathaxis</b>	axis_height AxisHeight
<sup>6</sup> <b>\Umathoperatorsize</b> D, D'	— DisplayOperatorMinHeight
<sup>9</sup> <b>\Umathfractiondelsize</b> D, D'	delim1 FractionDelimiterDisplayStyleSize
<sup>9</sup> <b>\Umathfractiondelsize</b> T, T', S, S', SS, SS'	delim2 FractionDelimiterSize
<b>\Umathfractiondenomdown</b> D, D'	denom1 FractionDenominatorDisplayStyleShiftDown
<b>\Umathfractiondenomdown</b> T, T', S, S', SS, SS'	denom2 FractionDenominatorShiftDown
<b>\Umathfractiondenomvgap</b> D, D'	3*default_rule_thickness FractionDenominatorDisplayStyleGapMin
<b>\Umathfractiondenomvgap</b> T, T', S, S', SS, SS'	default_rule_thickness FractionDenominatorGapMin
<b>\Umathfractionnumup</b> D, D'	num1 FractionNumeratorDisplayStyleShiftUp
<b>\Umathfractionnumup</b> T, T', S, S', SS, SS'	num2 FractionNumeratorShiftUp
<b>\Umathfractionnumvgap</b> D, D'	3*default_rule_thickness FractionNumeratorDisplayStyleGapMin
<b>\Umathfractionnumvgap</b> T, T', S, S', SS, SS'	default_rule_thickness FractionNumeratorGapMin
<b>\Umathfractionrule</b>	default_rule_thickness FractionRuleThickness
<b>\Umathskewedfractionhgap</b>	math_quad/2 SkewedFractionHorizontalGap
<b>\Umathskewedfractionvgap</b>	math_x_height SkewedFractionVerticalGap
<b>\Umathlimitabovebgap</b>	big_op_spacing3 UpperLimitBaselineRiseMin
<sup>1</sup> <b>\Umathlimitabovekern</b>	big_op_spacing5 0
<b>\Umathlimitabovevgap</b>	big_op_spacing1 UpperLimitGapMin
<b>\Umathlimitbelowbgap</b>	big_op_spacing4



	LowerLimitBaselineDropMin
<sup>1</sup> <b>\Umathlimitbelowkern</b>	big_op_spacing5 0
<b>\Umathlimitbelowvgap</b>	big_op_spacing2 LowerLimitGapMin
<b>\Umathoverdelimitervgap</b>	big_op_spacing1 StretchStackGapBelowMin
<b>\Umathoverdelimiterbgap</b>	big_op_spacing3 StretchStackTopShiftUp
<b>\Umathunderdelimitervgap</b>	big_op_spacing2 StretchStackGapAboveMin
<b>\Umathunderdelimiterbgap</b>	big_op_spacing4 StretchStackBottomShiftDown
<b>\Umathoverbarkern</b>	default_rule_thickness OverbarExtraAscender
<b>\Umathoverbarrule</b>	default_rule_thickness OverbarRuleThickness
<b>\Umathoverbarvgap</b>	3*default_rule_thickness OverbarVerticalGap
<sup>1</sup> <b>\Umathquad</b>	math_quad <font_size(f)>
<b>\Umathradicalkern</b>	default_rule_thickness RadicalExtraAscender
<sup>2</sup> <b>\Umathradicalrule</b>	<not set> RadicalRuleThickness
<sup>3</sup> <b>\Umathradicalvgap</b> D, D'	default_rule_thickness+abs(math_x_height)/4 RadicalDisplayStyleVerticalGap
<sup>3</sup> <b>\Umathradicalvgap</b> T, T', S, S', SS, SS'	default_rule_thickness+abs(default_rule_thickness)/4 RadicalVerticalGap
<sup>2</sup> <b>\Umathradicaldegreebefore</b>	<not set> RadicalKernBeforeDegree
<sup>2</sup> <b>\Umathradicaldegreeafter</b>	<not set> RadicalKernAfterDegree
<sup>2,7</sup> <b>\Umathradicaldegreeraise</b>	<not set> RadicalDegreeBottomRaisePercent
<sup>4</sup> <b>\Umathspaceafterscript</b>	script_space SpaceAfterScript
<b>\Umathstackdenomdown</b> D, D'	denom1 StackBottomDisplayStyleShiftDown
<b>\Umathstackdenomdown</b> T, T', S, S', SS, SS'	denom2 StackBottomShiftDown
<b>\Umathstacknumup</b>	num1



D, D'	StackTopDisplayStyleShiftUp
<b>\Umathstacknumup</b>	num3
T, T', S, S', SS, SS'	StackTopShiftUp
<b>\Umathstackvgap</b>	7*default_rule_thickness
D, D'	StackDisplayStyleGapMin
<b>\Umathstackvgap</b>	3*default_rule_thickness
T, T', S, S', SS, SS'	StackGapMin
<b>\Umathsubshiftdown</b>	sub1
	SubscriptShiftDown
<b>\Umathsubshiftdrop</b>	sub_drop
	SubscriptBaselineDropMin
<sup>8</sup> <b>\Umathsubsupshiftdown</b>	—
	SubscriptShiftDownWithSuperscript
<b>\Umathsubtopmax</b>	abs(math_x_height*4)/5
	SubscriptTopMax
<b>\Umathsubsupvgap</b>	4*default_rule_thickness
	SubSuperscriptGapMin
<b>\Umathsupbottommin</b>	abs(math_x_height/4)
	SuperscriptBottomMin
<b>\Umathsupshiftdrop</b>	sup_drop
	SuperscriptBaselineDropMax
<b>\Umathsupshiftup</b>	sup1
D	SuperscriptShiftUp
<b>\Umathsupshiftup</b>	sup2
T, S, SS,	SuperscriptShiftUp
<b>\Umathsupshiftup</b>	sup3
D', T', S', SS'	SuperscriptShiftUpCramped
<b>\Umathsupsubbottommax</b>	abs(math_x_height*4)/5
	SuperscriptBottomMaxWithSubscript
<b>\Umathunderbarkern</b>	default_rule_thickness
	UnderbarExtraDescender
<b>\Umathunderbarrule</b>	default_rule_thickness
	UnderbarRuleThickness
<b>\Umathunderbarvgap</b>	3*default_rule_thickness
	UnderbarVerticalGap
<sup>5</sup> <b>\Umathconnectoroverlapmin</b>	0
	MinConnectorOverlap

---

*Todo: add the extra ones.*

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: Traditional tfm fonts do not set `\Umathradicalrule` because T<sub>E</sub>X82 uses the height of the radical instead. When this parameter is indeed not set when LuaT<sub>E</sub>X has to typeset a radi-



cal, 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  $5/18\text{quad}$ ,  $-10/18\text{quad}$ , and  $60$ .

Note 3: If tfm fonts are used, then the `\Umathradicalvgap` is not set until the first time Lua $\TeX$  has to typeset a formula because this needs parameters from both family 2 and family 3. This provides a partial backward compatibility with  $\TeX$ 82, but that compatibility is only partial: once the `\Umathradicalvgap` is set, it will not be recalculated any more.

Note 4: When tfm fonts are used a similar situation arises with respect to `\Umathspaceafter-script`: it is not set until the first time Lua $\TeX$  has to typeset a formula. This provides some backward compatibility with  $\TeX$ 82. But once the `\Umathspaceafterscript` is set, `\script-space` will never be looked at again.

Note 5: Traditional tfm fonts set `\Umathconnectoroverlapmin` to zero because  $\TeX$ 82 always stacks extensibles without any overlap.

Note 6: The `\Umathoperatorsizes` 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  $\TeX$ 82.

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

## 8.6 Math spacing

### 8.6.1 Setting inline surrounding space with `\mathsurround` and `\mathsurroundskip`

Inline math is surrounded by (optional) `\mathsurround` spacing but that is a fixed dimension. There is now an additional parameter `\mathsurroundskip`. When set to a non-zero value (or zero with some stretch or shrink) this parameter will replace `\mathsurround`. By using an additional parameter instead of changing the nature of `\mathsurround`, we can remain compatible. In the meantime a bit more control has been added via `\mathsurroundmode`. This directive can take 6 values with zero being the default behaviour.

```
\mathsurround      10pt
\mathsurroundskip20pt
```

MODE	$X\$X\$X$		$X \$X\$ X$		EFFECT
0	x	x	x	x	obey <code>\mathsurround</code> when <code>\mathsurroundskip</code> is 0pt
1	x	xx	x	x x	only add skip to the left
2	xx	x	x x	x	only add skip to the right



3	x	x	x	x	x	x	add skip to the left and right
4		x	x	x		x	ignore the skip setting, obey <code>\mathsurround</code>
5		xxx			x	x	disable all spacing around math
6	x		x	x		x	only apply <code>\mathsurroundskip</code> when also spacing
7		x	x	x		x	only apply <code>\mathsurroundskip</code> when no spacing

---

Anything more fancy, like checking the beginning or end of a paragraph (or edges of a box) would not be robust anyway. If you want that you can write a callback that runs over a list and analyzes a paragraph. Actually, in that case you could also inject glue (or set the properties of a math node) explicitly. So, these modes are in practice mostly useful for special purposes and experiments (they originate in a tracker item). Keep in mind that this glue is part of the math node and not always treated as normal glue: it travels with the begin and end math nodes. Also, method 6 and 7 will zero the skip related fields in a node when applicable in the first occasion that checks them (linebreaking or packaging).

## 8.6.2 Pairwise spacing

Besides the parameters mentioned in the previous sections, there are also primitives to control the math spacing table (as explained in Chapter 18 of the `TEXbook`). This happens per class pair. Because we have many possible classes, we no longer hand the many primitives that `LuaTEX` has but you can define then using the generic `\setmathspacing` primitive:

```
\def\Umathordordspacing {\setmathspacing 0 0 }
\def\Umathordordopspacing {\setmathspacing 0 4 }
```

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

```
\setmathspacing 1 0 \displaystyle=4mu plus 2mu % op ord Umathopordspacing
```

The atom pairs known by the engine are all initialized by `initex` to the values mentioned in the table in Chapter 18 of the `TEXbook`.

For ease of use as well as for backward compatibility, `\thinmuskip`, `\medmuskip` and `\thickmuskip` are treated specially. 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. As a bonus we also introduced a `\tinymuskip` primitive.

In `LuaMetaTEX` we go a bit further. Any named dimension, glue and mu glue register as well as the constants with these properties can be bound to a pair by prefixing `\setmathspacing` by `\inherited`.

Careful readers will realize that there are also primitives for the items marked \* in the `TEXbook`. These will actually be used because we pose no restrictions. However, you can enforce the remapping rules to conform to the rules of `TEX` (or yourself).

*Todo: explain rules.*

*Todo: explain penalties.*

*Todo: explain let, set and copy primitives.*





### 8.6.3 Local \frozen settings with

Math is processed in two passes. The first pass is needed to intercept for instance `\over`, one of the few  $\TeX$  commands that actually has a preceding argument. There are often lots of curly braces used in math and these can result in a nested run of the math sub engine. However, you need to be aware of the fact that some properties are kind of global to a formula and the last setting (for instance a family switch) wins. This also means that a change (or again, the last one) in math parameters affects the whole formula. In  $\text{LuaMeta}\TeX$  we have changed this model a bit. One can argue that this introduces an incompatibility but it's hard to imagine a reason for setting the parameters at the end of a formula run and assume that they also influence what goes in front.

```
$
                                x \Usubscript {-}
    \frozen\Umathsubshiftdown\textstyle 0pt x \Usubscript {0}
    {\frozen\Umathsubshiftdown\textstyle 5pt x \Usubscript {5}}
                                x \Usubscript {0}
    {\frozen\Umathsubshiftdown\textstyle 15pt x \Usubscript {15}}
                                x \Usubscript {0}
    {\frozen\Umathsubshiftdown\textstyle 20pt x \Usubscript {20}}
                                x \Usubscript {0}
    \frozen\Umathsubshiftdown\textstyle 10pt x \Usubscript {10}
                                x \Usubscript {0}
$
```

The `\frozen` prefix does the magic: it injects information in the math list about the set parameter. In  $\text{Lua}\TeX$  1.10+ the last setting, the 10pt drop wins, but in  $\text{LuaMeta}\TeX$  you will see each local setting taking effect. The implementation uses a new node type, parameters nodes, so you might encounter these in an unprocessed math list. The result looks as follows:

```
x_x0x_5x0x_x0x_x0x_x
      15      20      10 0
```

### 8.6.4 Checking a state with \ifmathparameter

When you adapt math parameters it might make sense to see if they are set at all. When a parameter is unset its value has the maximum dimension value and you might for instance mistakenly multiply that value to open up things a bit, which gives unexpected side effects. For that reason there is a convenient checker: `\ifmathparameter`. This test primitive behaves like an `\ifcase`, with:

VALUE	MEANING
0	the parameter value is zero
1	the parameter is set
2	the parameter is unset



### 8.6.5 Skips around display math and `\mathdisplayskipmode`

The injection of `\abovedisplayskip` and `\belowdisplayskip` is not symmetrical. An above one is always inserted, also when zero, but the below is only inserted when larger than zero. Especially the latter makes it sometimes hard to fully control spacing. Therefore LuaT<sub>E</sub>X comes with a new directive: `\mathdisplayskipmode`. The following values apply:

VALUE	MEANING
0	normal T <sub>E</sub> X behaviour
1	always (same as 0)
2	only when not zero
3	never, not even when not zero

### 8.6.6 Nolimit correction with `\mathnolimitsmode`

There are two extra math parameters `\Umathnolimitsupfactor` and `\Umathnolimitssubfactor` that were added to provide some control over how limits are spaced (for example the position of super and subscripts after integral operators). They relate to an extra parameter `\mathnolimitsmode`. The half corrections are what happens when scripts are placed above and below. The problem with italic corrections is that officially that correction italic is used for above/below placement while advanced kerns are used for placement at the right end. The question is: how often is this implemented, and if so, do the kerns assume correction too. Anyway, with this parameter one can control it.

	$\int_1^0$	$\int_1^0$	$\int_1^0$	$\int_1^0$	$\int_1^0$	$\int_1^0$
<b>mode</b>	0	1	2	3	4	8000
<b>superscript</b>	0	font	0	0	+ic/2	0
<b>subscript</b>	-ic	font	0	-ic/2	-ic/2	8000ic/1000

When the mode is set to one, the math parameters are used. This way a macro package writer can decide what looks best. Given the current state of fonts in ConT<sub>E</sub>Xt we currently use mode 1 with factor 0 for the superscript and 750 for the subscripts. Positive values are used for both parameters but the subscript shifts to the left. A `\mathnolimitsmode` larger than 15 is considered to be a factor for the subscript correction. This feature can be handy when experimenting.

### 8.6.7 Influencing script kerning with `\mathscriptboxmode`

If you want to typeset text in math macro packages often provide something `\text` which obeys the script sizes. As the definition can be anything there is a good chance that the kerning doesn't come out well when used in a script. Given that the first glyph ends up in an `\hbox` we have some control over this. And, as a bonus we also added control over the normal sublist kerning. The `\mathscriptboxmode` parameter defaults to 1.

VALUE	MEANING
0	forget about kerning



- 1 kern math sub lists with a valid glyph
- 2 also kern math sub boxes that have a valid glyph
- 3 only kern math sub boxes with a boundary node present

Here we show some examples. Of course this doesn't solve all our problems, if only because some fonts have characters with bounding boxes that compensate for italics, while other fonts can lack kerns.

	$\$T_{\{\backslash tf\ fluff\}}\$$	$\$T_{\{\backslash tf\ fluff\}}\$$	$\$T_{\{\backslash text\{fluff\}\}}\$$	$\$T_{\{\backslash text\{fluff\}\}}\$$	$\$T_{\{\backslash text\{boundary1\ fluff\}\}}\$$
	mode 0	mode 1	mode 1	mode 2	mode 3
modern	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$
lucidaot	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$
pagella	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$
cambría	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$
dejavu	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$	$T_{fluff}$

Kerning between a character subscript is controlled by `\mathscriptcharmode` which also defaults to 1.

Here is another example. Internally we tag kerns as italic kerns or font kerns where font kerns result from the staircase kern tables. In 2018 fonts like Latin Modern and Pagella rely on cheats with the boundingbox, Cambria uses staircase kerns and Lucida a mixture. Depending on how fonts evolve we might add some more control over what one can turn on and off.

normal	modern	$T_f$	$\gamma_e$	$\gamma_{ee}$	$T_{fluff}$
	pagella	$T_f$	$\gamma_e$	$\gamma_{ee}$	$T_{fluff}$
	cambría	$T_f$	$\gamma_e$	$\gamma_{ee}$	$T_{fluff}$
	lucidaot	$T_f$	$\gamma_e$	$\gamma_{ee}$	$T_{fluff}$
bold	modern	$T_f$	$\gamma_e$	$\gamma_{ee}$	$T_{fluff}$
	pagella	$T_f$	$\gamma_e$	$\gamma_{ee}$	$T_{fluff}$
	cambría	$T_f$	$\gamma_e$	$\gamma_{ee}$	$T_{fluff}$
	lucidaot	$T_f$	$\gamma_e$	$\gamma_{ee}$	$T_{fluff}$

### 8.6.8 Forcing fixed scripts with `\mathscriptsmode`

We have three parameters that are used for this fixed anchoring:

PARAMETER	REGISTER
$d$	<code>\Umathsubshiftdown</code>
$u$	<code>\Umathsupshiftup</code>
$s$	<code>\Umathsubsupshiftdown</code>

When we set `\mathscriptsmode` to a value other than zero these are used for calculating fixed positions. This is something that is needed for instance for chemistry. You can manipulate the mentioned variables to achieve different effects.



MODE	DOWN	UP	EXAMPLE
0	dynamic	dynamic	$\text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^2$
1	$d$	$u$	$\text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^2$
2	$s$	$u$	$\text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^2$
3	$s$	$u + s - d$	$\text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^2$
4	$d + (s - d)/2$	$u + (s - d)/2$	$\text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^2$
5	$d$	$u + s - d$	$\text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^2$

The value of this parameter obeys grouping but applies to the whole current formula.

### 8.6.9 Penalties: `\mathpenaltiesmode`

Only in inline math penalties will be added in a math list. You can force penalties (also in display math) by setting:

```
\mathpenaltiesmode = 1
```

This primitive is not really needed in LuaTeX because you can use the callback `mlist_to_hlist` to force penalties by just calling the regular routine with forced penalties. However, as part of opening up and control this primitive makes sense. As a bonus we also provide two extra penalties:

```
\prebinoppenalty = -100 % example value
\prerelpenalty    =  900 % example value
```

They default to `inifinite` which signals that they don't need to be inserted. When set they are injected before a binop or rel node. This is an experimental feature.

### 8.6.10 Equation spacing: `\matheqnogapstep`

By default TeX will add one quad between the equation and the number. This is hard coded. 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 mimick floating point factors in TeX.

















## 8.7 Math constructs

### 8.7.1 Unscaled fences and `\mathdelimitersmode`

The `\mathdelimitersmode` primitive is experimental and deals with the following (potential) problems. Three bits can be set. The first bit prevents an unwanted shift when the fence symbol is not scaled (a cambria side effect). The second bit forces italic correction between a preceding



character ordinal and the fenced subformula, while the third bit turns that subformula into an ordinary so that the same spacing applies as with unfenced variants.

$\backslash\mathdelimitersmode = 0$     
 $\backslash\mathdelimitersmode = 1$     
 $\backslash\mathdelimitersmode = 2$     
 $\backslash\mathdelimitersmode = 3$     
 $\backslash\mathdelimitersmode = 4$     
 $\backslash\mathdelimitersmode = 5$     
 $\backslash\mathdelimitersmode = 6$     
 $\backslash\mathdelimitersmode = 7$   

So, when set to 7 fenced subformulas with unscaled delimiters come out the same as unfenced ones. This can be handy for cases where one is forced to use `\left` and `\right` always because of unpredictable content. As said, it's an experimental feature (which somehow fits in the exceptional way fences are dealt with in the engine). The full list of flags is given in the next table:

VALUE	MEANING
"01	don't apply the usual shift
"02	apply italic correction when possible
"04	force an ordinary subformula
"08	no shift when a base character
"10	only shift when an extensible

The effect can depend on the font (and for Cambria one can use for instance "16).

Sometimes you might want to act upon the size of a delimiter, something that is not really possible because of the fact that they are calculated *after* most has been typeset already. For this we have two keyword: `phantom` and `void`. In both cases the symbol is replaced by an empty rule, in the first case all three dimensions are preserved in the last case only the height and depth.

```

\startformula
  x\mathlimop{\Uvextensible           \Udelimiter 5 0 "222B}_1^2 x
\stopformula
\vskip-9ex
\startformula \red
  x\mathlimop{\Uvextensible phantom \Udelimiter 5 0 "222B}_1^2 x
\stopformula
\vskip-9ex
\startformula \blue
  x\mathlimop{\Uvextensible void      \Udelimiter 5 0 "222B}_1^2 x
\stopformula

```

In typeset form this looks like:

$$x \int_1^2 x$$



### 8.7.2 Accent handling with `\Umathaccent`

Lua $\TeX$  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  $\TeX$ 82.

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  $\TeX$ book), but if a value turns out to be zero and the font had a `MathConstants` table, then `AccentBaseHeight` is used instead.

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

Possible locations are `top`, `bottom`, `both` and `center`. When no location is given `top` is assumed. An additional parameter `fraction` can be specified followed by a number; a value of for instance 1200 means that the criterium is 1.2 times the width of the nucleus. The fraction only applies to the stepwise selected shapes and is mostly meant for the `overlay` location. It also works for the other locations but then it concerns the width.

### 8.7.3 Building radicals with `\Uradical` and `\Uroot`

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`.



### 8.7.4 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, Lua $\TeX$  checks whether the super- or subscript and the nucleus are both simple character items. If they are, and if the fonts of both character items are OpenType fonts (as opposed to legacy  $\TeX$  fonts), then Lua $\TeX$  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 math kern 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 math kern 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 math kern 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 math kern pairs at all).

### 8.7.5 Scripts on extensibles: `\Uunderdelimit`, `\Uoverdelimit`, `\Udelimitover`, `\Udelimitunder` and `\Uhextensible`

The primitives `\Uunderdelimit` and `\Uoverdelimit` allow the placement of a subscript or superscript on an automatically extensible item and `\Udelimitover` and `\Udelimitunder` allow the placement of an automatically extensible item as a subscript or superscript on a nucleus. The input:

```
\Uoverdelimit 0 "2194 {\hbox{\strut overdelimit}}$  
\Uunderdelimit 0 "2194 {\hbox{\strut underdelimit}}$  
\Udelimitover 0 "2194 {\hbox{\strut delimitover}}$  
\Udelimitunder 0 "2194 {\hbox{\strut delimitunder}}$
```

will render this:

overdelimit	delimitover	delimitunder
underdelimit		



The vertical placements are controlled by `\Umathunderdelimterbgap`, `\Umathunderdelimtervgap`, `\Umathoverdelimterbgap`, and `\Umathoverdelimtervgap` in a similar way as limit placements on large operators. The superscript in `\Uoverdelimter` is typeset in a suitable scripted style, the subscript in `\Underdelimter` is cramped as well.

These primitives accept an optional width specification. When used the also optional keywords `left`, `middle` and `right` will determine what happens when a requested size can't be met (which can happen when we step to successive larger variants).

An extra primitive `\Uhexensible` is available that can be used like this:

```
$\Uhexensible width 10cm 0 "2194$
```

This will render this:



Here you can also pass options, like:

```
$\Uhexensible width 1pt middle 0 "2194$
```

This gives:



LuaTeX internally uses a structure that supports OpenType ‘MathVariants’ as well as tfm ‘extensible recipes’. In most cases where font metrics are involved we have a different code path for traditional fonts and OpenType fonts.

### 8.7.6 Fractions and the new `\Uskewed` and `\Uskewedwithdelims`

The `\abovewithdelims` command accepts a keyword `exact`. When issued the extra space relative to the rule thickness is not added. One can of course use the `\Umathfraction..gap` commands to influence the spacing. Also the rule is still positioned around the math axis.

```
$$ { {a} \abovewithdelims() exact 4pt {b} }$$
```

The math parameter table contains some parameters that specify a horizontal and vertical gap for skewed fractions. Of course some guessing is needed in order to implement something that uses them. And so we now provide a primitive similar to the other fraction related ones but with a few options so that one can influence the rendering. Of course a user can also mess around a bit with the parameters `\Umathskewedfractionhgap` and `\Umathskewedfractionvgap`.

The syntax used here is:

```
{ {1} \Uskewed / <options> {2} }
{ {1} \Uskewedwithdelims / ( ) <options> {2} }
```

where the options can be `noaxis` and `exact`. By default we add half the axis to the shifts and by default we zero the width of the middle character. For Latin Modern the result looks as follows:

	$x + \frac{a}{b} + x$	$x + \frac{1}{2} + x$	$x + (\frac{a}{b}) + x$	$x + (\frac{1}{2}) + x$
exact	$x + \frac{a}{b} + x$	$x + \frac{1}{2} + x$	$x + (\frac{a}{b}) + x$	$x + (\frac{1}{2}) + x$





noaxis             $x + a/b + x$     $x + 1/2 + x$     $x + (a/b) + x$     $x + (1/2) + x$   
exact noaxis    $x + a/b + x$     $x + 1/2 + x$     $x + (a/b) + x$     $x + (1/2) + x$

The `\over` and related primitives have the form:

`{{top}\over{bottom}}`

For convenience, which also avoids some of the trickery that makes this ‘looking back’ possible, the LuaMetaTeX also provides this variant:

`\Uover{top}{bottom}`

The optional arguments are also supported but we have one extra option: `style`. The style is applied to the numerator and denominator.

`\Uover style \scriptstyle {top} {bottom}`

The complete list of these commands is: `\Uabove`, `\Uatop`, `\Uover`, `\Uabovewithdelims`, `\Uatopwithdelims`, `\Uoverwithdelims`, `\UUskewed`, `\UUskewedwithdelims`. As with other extensions we use a leading U and because we already had extra skew related primitives we end up with a UU there. This obscurity is not that big an issue because normally such primitives are wrapped in a macro. Here are a few examples:

`$$\Uover {1234} {5678} $\quad`  
`$$\Uover {\textstyle 1234} {\textstyle 5678} $\quad`  
`$$\Uover {\scriptstyle 1234} {\scriptstyle 5678} $\quad`  
`$$\Uover {\scriptscriptstyle 1234} {\scriptscriptstyle 5678} $\blank`

`$$\Uover {1234} {5678} $\quad`  
`$$\Uover style \textstyle {1234} {5678} $\quad`  
`$$\Uover style \scriptstyle {1234} {5678} $\quad`  
`$$\Uover style \scriptscriptstyle {1234} {5678} $\blank`

These render as:  $\frac{1234}{5678}$   $\frac{1234}{5678}$   $\frac{1234}{5678}$   $\frac{1234}{5678}$

$\frac{1234}{5678}$   $\frac{1234}{5678}$   $\frac{1234}{5678}$   $\frac{1234}{5678}$

### 8.7.7 Math styles: `\Ustyle`

This primitive accepts a style identifier:

`\Ustyle \displaystyle`

This in itself is not spectacular because it is equivalent to

`\displaystyle`

Both commands inject a style node and change the current style. However, as in other places where LuaMetaTeX expects a style you can also pass a number in the range zero upto seven (like the ones reported by the primitive `\mathstyle`). So, the next few lines give identical results:



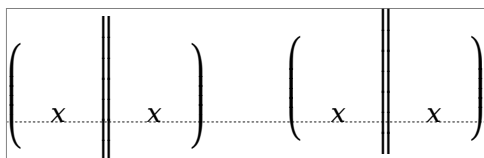
Like: 07 07 07. Values outside the valid range are ignored.

There is an extra option `norule` that can be used to suppress the rule while keeping the spacing compatible.

### 8.7.8 Delimiters: `\Uleft`, `\Umiddle` and `\Uright`

Normally you will force delimiters to certain sizes by putting an empty box or rule next to it. The resulting delimiter will either be a character from the stepwise size range or an extensible. The latter can be quite differently positioned than the characters as it depends on the fit as well as the fact whether the used characters in the font have depth or height. Commands like (plain  $\TeX$ s) `\big` need to use this feature. In  $\text{Lua}\TeX$  we provide a bit more control by three variants that support optional parameters `height`, `depth` and `axis`. The following example uses this:

```
\Uleft   height 30pt depth 10pt      \Udelimiter "0 "0 "000028
\quad x\quad
\Umiddle height 40pt depth 15pt      \Udelimiter "0 "0 "002016
\quad x\quad
\Uright  height 30pt depth 10pt      \Udelimiter "0 "0 "000029
\quad \quad \quad
\Uleft   height 30pt depth 10pt axis \Udelimiter "0 "0 "000028
\quad x\quad
\Umiddle height 40pt depth 15pt axis \Udelimiter "0 "0 "002016
\quad x\quad
\Uright  height 30pt depth 10pt axis \Udelimiter "0 "0 "000029
```



The keyword `exact` can be used as directive that the real dimensions should be applied when the criteria can't be met which can happen when we're still stepping through the successively larger variants. When no dimensions are given the `noaxis` command can be used to prevent shifting over the axis.

You can influence the final class with the keyword `class` which will influence the spacing. The numbers are the same as for character classes.

### 8.7.9 Accents: `\mathlimitsmode`

When you use `\limits` or `\nolimits` without scripts spacing might get messed up. This can be prevented by setting `\mathlimitsmode` to a non-zero value.



## 8.8 Extracting values

### 8.8.1 Codes and using `\Umathcode`, `\Umathcharclass`, `\Umathcharfam` and `\Umathcharslot`

You can extract the components of a math character. Say that we have defined:

```
\Umathcode 1 2 3 4
```

then

```
[\Umathcharclass1] [\Umathcharfam1] [\Umathcharslot1]
```

will return:

```
[2] [3] [4]
```

These commands are provided as convenience. Before they come available you could do the following:

```
\def\Umathcharclass{\numexpr  
  \directlua{tex.print(tex.getmathcode(token.scan_int())[1])}  
\relax}  
\def\Umathcharfam{\numexpr  
  \directlua{tex.print(tex.getmathcode(token.scan_int())[2])}  
\relax}  
\def\Umathcharslot{\numexpr  
  \directlua{tex.print(tex.getmathcode(token.scan_int())[3])}  
\relax}
```

### 8.8.2 Last lines and `\predisplaygapfactor`

There is a new primitive to control the overshoot in the calculation of the previous line in mid-paragraph display math. The default value is 2 times the em width of the current font:

```
\predisplaygapfactor=2000
```

If you want to have the length of the last line independent of math i.e. you don't want to revert to a hack where you insert a fake display math formula in order to get the length of the last line, the following will often work too:

```
\def\lastlinelength{\dimexpr  
  \directlua {tex.sprint (  
    (nodes.dimensions(node.tail(tex.lists.page_head).list))  
  )}sp  
\relax}
```



## 8.9 Math mode

### 8.9.1 Verbose versions of single-character math commands like `\Usuperscript` and `\Usubscript`

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

PRIMITIVE	EXPLANATION
<code>\Usuperscript</code>	duplicates the functionality of <code>^</code>
<code>\Usubscript</code>	duplicates the functionality of <code>_</code>
<code>\Ustartmath</code>	duplicates the functionality of <code>\$</code> , when used in non-math mode.
<code>\Ustopmath</code>	duplicates the functionality of <code>\$</code> , when used in inline math mode.
<code>\Ustartdisplaymath</code>	duplicates the functionality of <code>\$\$</code> , when used in non-math mode.
<code>\Ustopdisplaymath</code>	duplicates the functionality of <code>\$\$</code> , 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).

### 8.9.2 Script commands `\Unosuperscript` and `\Unosubscript`

These two commands result in super- and subscripts but with the current style (at the time of rendering). So,

```
$
x\Usuperscript {1}\Usubscript {2} =
x\Unosuperscript{1}\Unosubscript{2} =
x\Usuperscript {1}\Unosubscript{2} =
x\Unosuperscript{1}\Usubscript {2}
$
```

results in  $x_2^1 = x_2^1 = x_2^1 = x_2^1$ .

### 8.9.3 Injecting primes with `Uprimescript`

*Todo: explain this one.*

### 8.9.4 Prescripts with `\Usuperprescript` and `Usubprescript`

```
\hbox{$
{\tf X}^1_2^{^3}_4 \quad
{\tf X}^1_{}^{^3}_{} \quad
{\tf X}^{}_1^{^3}_{} \quad
{\tf X}^{}_{}^{^3}_{} \quad
{\tf X}^{}_{}^{^3}_4 \quad
```



```
{\tf X}^^3 ____4
$}
```

The question is: are these double super and subscript triggers the way to go? Anyway, you need to have them either being active (which in ConT<sub>E</sub>Xt then boils down to them being other characters), or say `\supmarkmode = 1` to disable the normal multiple `^` treatment (a value larger than 1 will also disable that in text mode).

```
3X1 3X1 4X1 3X4 3X4
```

The more explicit commands are:

```
\hbox{$
{\tf X}\Usuperscript{1} \quad
{\tf X} \Usubscript{2} \quad
{\tf X}\Usuperscript{1}\Usubscript{2} \quad
{\tf X}\Usuperscript{1} \Usuperprescript{3} \quad
{\tf X} \Usubscript{2} \Usubprescript{4} \quad
{\tf X}\Usuperscript{1}\Usubscript{2}\Usuperprescript{3}\Usubprescript{4} \quad
{\tf X} \Usuperprescript{3} \quad
{\tf X} \Usubprescript{4} \quad
{\tf X} \Usuperprescript{3}\Usubprescript{4}
$}
```

These more verbose triggers can be used to build interfaces:

```
X1 X2 X21 3X1 4X2 3X41 3X4 3X4
```

## 8.9.5 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.

## 8.10 Goodies

### 8.10.1 Less Tracing

Because there are quite some math related parameters and values, it is possible to limit tracing. Only when `tracingassigns` and/or `tracingrestores` are set to 2 or more they will be traced.

## 8.11 Experiments

There are a couple of experimental features. They will stay but details might change, for instance more control over spacing. We just show some examples and let your imagination work it out.



### 8.11.1 Forcing classes with `\Umathclass`

You can change the class of a math character on the fly:

```
$x\mathopen  {\!}+123+\mathclose  {\!}x$  
$x\Umathclass4  ! +123+\Umathclass5  ! x$  
$x              ! +123+                ! x$  
$x\mathclose  {\!}+123+\mathopen  {\!}x$  
$x\Umathclass5  ! +123+\Umathclass4  ! x$
```

Watch how the spacing changes:

```
x!+123+!x  
x!+123+!x  
x! + 123 + ! x  
x! + 123 + ! x  
x! + 123 + ! x
```

### 8.11.2 Scaling spacing with `\Umathxscale` and `\Umathyscale`

These two primitives scale the horizontal and vertical scaling related parameters. They are set by style. There is no combined scaling primitive.

```
 $\Umathxscale\textstyle 800 a + b + x + d + e = f $\par  
 $\Umathxscale\textstyle 1000 a + b + x + d + e = f $\par  
 $\Umathxscale\textstyle 1200 a + b + x + d + e = f $\blank
```

```
 $\Umathyscale\textstyle 800 \sqrt{2}\{x+1\}\quad  
 $\Umathyscale\textstyle 1000 \sqrt{2}\{x+1\}\quad  
 $\Umathyscale\textstyle 1200 \sqrt{2}\{x+1\}\quad
```

Normally only small deviations from 1000 make sense but here we want to show the effect and use a 20% scaling:

$$a + b + x + d + e = f$$

$$a + b + x + d + e = f$$

$$a + b + x + d + e = f$$

$$\sqrt[2]{x+1} \quad \sqrt[2]{x+1} \quad \sqrt[2]{x+1}$$



# 9 Building paragraphs and pages

## 9.1 Introduction

There are some enhancements that relate to the way paragraphs and pages are built. In this chapter we will cover those. There can be a bit of overlap with other chapters. These enhancements are still somewhat experimental.

## 9.2 Paragraphs

*This section will describe `\autoparagraphmode`, `\shapingpenaltiesmode`, `\shapingpenalty-mode`, `\everybeforepar`, `\snapshotpar`, `\wrapuppar`, `\orphanpenalties`, `\orphanpenalty`, enz. For the moment the manuals that come with ConT<sub>E</sub>Xt have to do.*

## 9.3 Inserts

Inserts are tightly integrated into the page builder. Depending on penalties and available space they end up on the same page as were they got injected or they move to following pages, either or not split.

In traditional T<sub>E</sub>X inserts are controlled by registers. A quadruple of box, skip, dimen and count registers with the same number acts as an insert class. Details can be found in the T<sub>E</sub>Xbook. A side effect of this is that we only have these four properties bound to class, other properties of inserts are driven by shared parameters. Another side effect is that register management has to make sure that these foursome get ‘allocates’ as set and not clashes with other register allocations.

In LuaMetaT<sub>E</sub>X you can set the `\insertmode` to a non zero value in which case inserts are not using the register pool but have their own (global) resources. For now this is mode driven (for compatibility reasons) and once set or when an insert has been accessed, this mode is frozen, so this parameter can be set very early in the macro package loading process.

PRIMITIVE	TRADITIONAL	EXPLANATION
<code>\insertdistance</code>	<code>skip</code>	the space before the first instance (on a page)
<code>\insertmultiplier</code>	<code>count</code>	a factor that is used to calculate the height used
<code>\insertlimit</code>	<code>dimen</code>	the maximum amount of space on a page to be taken
<code>\insertpenalty</code>	<code>\insertpenalties</code>	the floating penalty (used when set)
<code>\insertmaxdepth</code>	<code>\maxdepth</code>	the maximum split depth (used when set)
<code>\insertstorage</code>		signals that the insert has to be stored for later
<code>\insertheight</code>	<code>\ht box</code>	the accumulated height of the inserts so far
<code>\insertdepth</code>	<code>\dp box</code>	the current depth of the inserts so far
<code>\insertwidth</code>	<code>\wd box</code>	the width of the inserts

These primitives takes an insert class number. The `\insertpenalties` primitives is unchanged, as is the LuaT<sub>E</sub>X `\insertheights` one. When `\insertstoring` is set 1, all inserts that have their



storage flag set will be saved. Think of a multi column setup where inserts have to end up in the last column. If there are three columns, the first two will store inserts. Then when the last column is dealt with `\insertstoring` can be set to 2 and that will signal the builder that we will inject the inserts. In both cases, the value of this register will be set to zero so that it doesn't influence further processing. You can use `\ifinsert` to check if an insert box is void. More details about these (probably experimental for a while) features can be found in documents that come with ConT<sub>E</sub>Xt.

A limitation of inserts is that when they are buried too deep, a property they share with inserts, they become invisible. This can be dealt with by the migration feature described in an upcoming section.

The LuaMetaT<sub>E</sub>X engine has some tracing built in that is enabled by setting `\tracinginserts` to a positive value.

## 9.4 Marks

Marks are kind of signal nodes in the list that refer to stored token lists. When a page has been split off and is handed over to the output routine these signals are resolved into first, top and bottom mark references that can (for instance) be used for running headers.

In  $\varepsilon$ -T<sub>E</sub>X the standard T<sub>E</sub>X primitives `\mark`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark` and `\splitbotmark` have been extended with plural forms that accent a number before the token list. That number indicates a mark class.

A problem with marks is that one cannot really reset them. Mark states are kept in the node lists and only periodically the state is snapshot into the global state variables. The LuaT<sub>E</sub>X engine can reset these global states with `\clearmarks` but that's only half a solution. In LuaMetaT<sub>E</sub>X we have `\flushmarks` which, like `\marks`, puts a node in the list that does a reset. This permits implementing controlled resets of specific marks at the cost of a possible interfering mode, but that can normally be dealt with rather well.

Another problem with marks is that when they are buried too deep, a property they share with inserts, they become invisible. This can be dealt with by the migration feature described in the next section.

The LuaMetaT<sub>E</sub>X engine has some tracing built in that is enabled by setting `\tracingmarks` to a positive value.

## 9.5 Adjusts

The `\vadjust` primitive injects something in the vertical list after the line where it ends up. In pdfT<sub>E</sub>X the `pre` keyword was added so that one could force something before a previous line (actually this was something that we needed in ConT<sub>E</sub>Xt MkII). The LuaMetaT<sub>E</sub>X engine also supports the `post` keyword.

We support a few more keywords: `before` will prepend the adjustment to the already given one, and `after` will append it. The `index` keyword expects an integer and relates that to the current adjustment. This index is passed to an (optional) callback when the adjustment is finally moved to the vertical list. That move is actually delayed because like inserts and marks these (vertical) adjustments can migrate to the 'outer' vertical level.





The main reason for the index having no influence on the order is that this primitive already could be used multiple times and order is determined by usage.<sup>7</sup>

The LuaMetaTeX engine has some tracing built in that is enabled by setting `\tracingadjusts` to a positive value. Currently there is not that much tracing which is why the value has to be at least 2 in order to be compatible with other (detailed) tracers.

## 9.6 Migration

A new primitive `\automigrationmode` can be used to let deeply burried marks and inserts bubble up to the outer level.

VALUE	EXPLANATION
1	migrate marks in the par builder
2	migrate inserts in the par builder
4	migrate adjusts in the par builder
8	migrate prebox material in the page builder
16	migrate postbox material in the page builder

If you want to migrate marks and inserts you need to set all these flags. Migrated marks and inserts end up as post-box properties and will be handled in the page builder as such. At the Lua end you can add pre- and post-box material too.

## 9.7 Pages

The page builder can be triggered by (for instance) a penalty but you can also use `\pagebound-ary`. This will trigger the page builder but not leave anything behind.

## 9.8 Local boxes

As far as I know the Omega/Aleph local box mechanism is mostly in those engines in order to support repetitive quotes. In LuaTeX this mechanism has been made more robust and in LuaMetaTeX it became more tightly integrated in the paragraph properties. In order for it to be more generic and useful, it got more features. For instance it is a bit painful to manage with respect to grouping (which is a reason why it's not that much used). The most interesting property is that the dimensions are taken into account when a paragraph is broken into lines.

There are three commands: `\localleftbox`, `\localrightbox` and the LuaMetaTeX specific `\localmiddlebox` which is basically a right box but when we pass these boxes to a callback they can be distinguished (we could have used the index but this was a cheap extra signal so we keep it).

These commands take optional keywords. The `index` keyword has to be followed by an integer. This index determines the order which doesn't introduce a significant compatibility issue: local boxes are hardly used and originally had only one instance.

<sup>7</sup> Under consideration is to let the callback mess with the flushing order.



The `par` keyword forces the box to be added to the current paragraph head. This permits setting them when a paragraph has already started. The implementation of these boxes is done via so called (local) paragraph nodes and there is one at the start of each paragraph.

The `local` keyword tells this mechanism not to update the registers that keep these boxes. In that case a next paragraph will start fresh. The `keep` option will do the opposite and retain the box after a group ends.

The commands: `\localleftboxbox`, `\localrightboxbox` and `\localmiddleboxbox` return a copy of the current related register content.



# 10 Nodes

## 10.1 Lua node representation

TeX's nodes are represented in Lua as user data objects with a variable set of fields or by a numeric identifier when requested. When you print a node user data object you will see these numbers. In the following syntax tables the type of such a user data object is represented as `<node>`.

**The return values of `node.types` are:** `hlist` (0), `vlist` (1), `rule` (2), `insert` (3), `mark` (4), `adjust` (5), `boundary` (6), `disc` (7), `whatsit` (8), `par` (9), `dir` (10), `math` (11), `glue` (12), `kern` (13), `penalty` (14), `style` (15), `choice` (16), `parameter` (17), `noad` (18), `radical` (19), `fraction` (20), `accent` (21), `fence` (22), `math_char` (23), `math_text_char` (24), `sub_box` (25), `sub_mlist` (26), `delimiter` (27), `glyph` (28), `unset` (29), `align_record` (31), `attribute` (32), `glue_spec` (33), `temp` (34) and `split` (35)

You can ask for a list of fields with `node.fields` and for valid subtypes with `node.subtypes`. The `node.values` function reports some used values. Valid arguments are `glue`, `style` and `math`. Keep in mind that the setters normally expect a number, but this helper gives you a list of what numbers matter. For practical reason the `pagestate` values are also reported with this helper, but they are backend specific.

**The return values of `node.values("glue")` are:** `normal` (0), `fi` (1), `fil` (2), `fill` (3) and `filll` (4)

**The return values of `node.values("style")` are:** `display` (0), `crampeddisplay` (1), `text` (2), `crampedtext` (3), `script` (4), `crampedscript` (5), `scriptscript` (6) and `crampedscriptscript` (7)

**The return values of `node.values("math")` are:** `quad` (0), `axis` (1), `accentbaseheight` (2), `accentbasedepth` (3), `flattenedaccentbaseheight` (4), `flattenedaccentbasedepth` (5), `xscale` (6), `yscale` (7), `operatorsize` (8), `overbarkern` (9), `overbarrule` (10), `overbarvgap` (11), `underbarkern` (12), `underbarrule` (13), `underbarvgap` (14), `radicalkern` (15), `radicalrule` (16), `radicalvgap` (17), `radicaldegreebefore` (18), `radicaldegreeafter` (19), `radicaldegreerise` (20), `stackvgap` (21), `stacknumup` (22), `stackdenomdown` (23), `fractionrule` (24), `fractionnumvgap` (25), `fractionnumup` (26), `fractiondenomvgap` (27), `fractiondenomdown` (28), `fractiondelsize` (29), `skewedfractionhgap` (30), `skewedfractionvgap` (31), `limitabovevgap` (32), `limitabovebgap` (33), `limitabovekern` (34), `limitbelowvgap` (35), `limitbelowbgap` (36), `limitbelowkern` (37), `nolimitsubfactor` (38), `nolimitsupfactor` (39), `underdelimitervgap` (40), `underdelimiterbgap` (41), `overdelimitervgap` (42), `overdelimiterbgap` (43), `subshiftdrop` (44), `supshiftdrop` (45), `subshiftdown` (46), `subsupshiftdown` (47), `subtopmax` (48), `supshiftdown` (49), `supbottommin` (50), `subsubbottommax` (51), `subsupvgap` (52), `spacebefore-script` (53), `spaceafterscript` (54), `connectoroverlapmin` (55), `extrasuperscriptshift` (56), `extrasubscriptshift` (57), `extrasuperprescriptshift` (58), `extrasubprescriptshift` (59), `primeraise` (60), `primeshiftup` (61), `primeshiftdrop` (62), `primespaceafter` (63), `primewidth`



(64), ruleheight (65), ruledepth (66), superscriptshiftdistance (67), subscriptshiftdistance (68), presuperscriptshiftdistance (69), presubscriptshiftdistance (70), extrasuperscriptspace (71), extrasubscriptspace (72), extrasuperprescriptspace (73), extrasubprescriptspace (74), skeweddelimiter tolerance (75), accenttopshiftup (76), accentbottomshiftdown (77), flattenedaccenttopshiftup (78), flattenedaccentbottomshiftdown (79), overlinevariant (80), underlinevariant (81), overdelimitervariant (82), underdelimitervariant (83), delimiterovervariant (84), delimiterundervariant (85), hextensiblevvariant (86), vextensiblevvariant (87), fractionvariant (88), radicalvariant (89), accentvariant (90), degreevariant (91), topaccentvariant (92), botaccentvariant (93), overlayaccentvariant (94), numeratorvariant (95), denominatorvariant (96), superscriptvariant (97), subscriptvariant (98), primevariant (99) and stackvariant (100)

The return values of `node.values("pagestate")` are:

## 10.2 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 dummy entry because 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 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. By now most of  $\text{\TeX}$ 's node processing makes sure that the `prev` nodes are valid but there can be exceptions, especially when the internal magic uses a leading temp nodes to temporarily store a state.

The LuaMeta $\text{\TeX}$  engine provides a lot of freedom and it is up to the user to make sure that the node lists remain sane. There are some safeguards but there can be cases where the engine just quits out of frustration. And, of course you can make the engine crash.

### 10.2.1 hlist and vlist nodes

These lists share fields and subtypes although some subtypes can only occur in horizontal lists while others are unique for vertical lists. The possible fields are `attr`, `class`, `depth`, `direction`, `doffset`, `glueorder`, `glueset`, `gluesign`, `height`, `hoffset`, `list`, `orientation`, `shift`, `source`, `state`, `target`, `width`, `woffset`, `xoffset` and `yoffset`.

FIELD	TYPE	EXPLANATION
subtype	number	accent, alignment, box, cell, container, degree, denominator, equation, equationnumber, fence, fraction, hdelimiter, hextensible, in-



		dent, insert, left, line, local, math, mathchar, middle, modifier, nucleus, numerator, over, overdelimater, radical, right, scripts, sub, sup, under, underdelimater, unknown, vdelimater and vextensible
attr	node	list of attributes
width	number	the width of the box
height	number	the height of the box
depth	number	the depth of the box
direction	number	the direction of this box, see 10.2.15
shift	number	a displacement perpendicular to the character (hlist) or line (vlist) progression direction
glueorder	number	a number in the range [0, 4], indicating the glue order
glueset	number	the calculated glue ratio
gluesign	number	0 = normal, 1 = stretching, 2 = shrinking
list	node	the first node of the body of this list

The orientation, woffset, hoffset, doffset, xoffset and yoffset fields are special. They can be used to make the backend rotate and shift boxes which can be handy in for instance vertical typesetting. Because they relate to (and depend on the) the backend they are not discussed here (yet).

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

Note: the field name head and list are both valid. Sometimes it makes more sense to refer to a list by head, sometimes list makes more sense.

## 10.2.2 rule nodes

Contrary to traditional T<sub>E</sub>X, LuaT<sub>E</sub>X has more \hrule and \vrule subtypes because we also use rules to store reuseable objects and images. User nodes are invisible and can be intercepted by a callback. The supported fields are attr, char, data, depth, font, height, left, right, width, xoffset and yoffset.

FIELD	TYPE	EXPLANATION
subtype	number	box, empty, fraction, image, normal, outline, over, radical, strut, under and user
attr	node	list of attributes
width	number	the width of the rule where 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)
left	number	shift at the left end (also subtracted from width)
right	number	(subtracted from width)
dir	string	the direction of this rule, see 10.2.15
index	number	an optional index that can be referred to
transform	number	an private variable (also used to specify outline width)

The left and type right keys are somewhat special (and experimental). When rules are auto adapting to the surrounding box width you can enforce a shift to the right by setting left. The



value is also subtracted from the width which can be a value set by the engine itself and is not entirely under user control. The `right` is also subtracted from the width. It all happens in the backend so these are not affecting the calculations in the frontend (actually the auto settings also happen in the backend). For a vertical rule `left` affects the height and `right` affects the depth. There is no matching interface at the  $\text{T}_{\text{E}}\text{X}$  end (although we can have more keywords for rules it would complicate matters and introduce a speed penalty.) However, you can just construct a rule node with Lua and write it to the  $\text{T}_{\text{E}}\text{X}$  input. The `outline` subtype is just a convenient variant and the `transform` field specifies the width of the outline.

The `xoffset` and `yoffset` fields are special. They can be used to shift rules. Because they relate to (and depend on the) the backend they are not discussed here (yet).

### 10.2.3 insert nodes

This node relates to the `\insert` primitive and support the fields: `attr`, `cost`, `depth`, `height`, `list` and `spec`.

FIELD	TYPE	EXPLANATION
<code>subtype</code>	number	the insertion class
<code>attr</code>	node	list of attributes
<code>cost</code>	number	the penalty associated with this insert
<code>height</code>	number	height of the insert
<code>depth</code>	number	depth of the insert
<code>list</code>	node	the first node of the body of this insert

There is a set of extra fields that concern the associated glue: `width`, `stretch`, `stretchorder`, `shrink` and `shrinkorder`. These are all numbers.

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. You can use `list` instead (often in functions you want to use local variable with similar names and both names are equally sensible).

### 10.2.4 mark nodes

This one relates to the `\mark` primitive and only has a few fields: `attr`, `class` and `mark`.

FIELD	TYPE	EXPLANATION
<code>subtype</code>	number	unused
<code>attr</code>	node	list of attributes
<code>class</code>	number	the mark class
<code>mark</code>	table	a table representing a token list

### 10.2.5 adjust nodes

This node comes from `\vadjust` primitive and has fields: `attr` and `list`.

FIELD	TYPE	EXPLANATION
<code>subtype</code>	number	<code>local</code> , <code>post</code> and <code>pre</code>



attr	node	list of attributes
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 the result.

### 10.2.6 disc nodes

The `\discretionary` and `\-`, the `-` character but also the hyphenation mechanism produces these nodes. The available fields are: `attr`, `options`, `penalty`, `post`, `pre` and `replace`.

FIELD	TYPE	EXPLANATION
subtype	number	automatic, discretionary, explicit and regular
attr	node	list of attributes
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 <code>\hyphenpenalty</code> or <code>\exhyphenpenalty</code>

The subtype numbers 4 and 5 belong to the ‘of-fice’ explanation given elsewhere. These disc nodes are kind of special as at some point they also keep information about breakpoints and nested ligatures.

The `pre`, `post` and `replace` fields at the Lua end are in fact indirectly accessed and have a `prev` pointer that is not `nil`. This means that when you mess around with the head of these (three) lists, you also need to reassign them because that will restore the proper `prev` pointer, so:

```
pre = d.pre
-- change the list starting with pre
d.pre = pre
```

Otherwise you can end up with an invalid internal perception of reality and `LuaMetaTeX` might even decide to crash on you. It also means that running forward over for instance `pre` is ok but backward you need to stop at `pre`. And you definitely must not mess with the node that `prev` points to, if only because it is not really a node but part of the disc data structure (so freeing it again might crash `LuaMetaTeX`).

### 10.2.7 math nodes

Math nodes represent the boundaries of a math formula, normally wrapped into `$` signs. The following fields are available: `attr`, `penalty`, `shrink`, `shrinkorder`, `stretch`, `stretchorder`, `surround` and `width`.

FIELD	TYPE	EXPLANATION
subtype	number	<code>beginmath</code> and <code>endmath</code>
attr	node	list of attributes



surround	number	width of the <code>\mathsurround</code> kern
width	number	the horizontal or vertical displacement
stretch	number	extra (positive) displacement or stretch amount
stretchorder	number	factor applied to stretch amount
shrink	number	extra (negative) displacement or shrink amount
shrinkorder	number	factor applied to shrink amount

---

The glue fields only kick in when the surround fields is zero.

### 10.2.8 glue nodes

Skips are about the only type of data objects in traditional  $\text{T}_{\text{E}}\text{X}$  that are not a simple value. They are inserted when  $\text{T}_{\text{E}}\text{X}$  sees a space in the text flow but also by `\hskip` and `\vskip`. The structure that represents the glue components of a skip internally is called a `glue_spec`. In  $\text{LuaMetaT}_{\text{E}}\text{X}$  we don't use the spec itself but just its values. A glue node has the fields: `attr`, `font`, `leader`, `shrink`, `shrinkorder`, `stretch`, `stretchorder` and `width`.

FIELD	TYPE	EXPLANATION
subtype	number	<code>abovedisplayshortskip</code> , <code>abovedisplayskip</code> , <code>baselineskip</code> , <code>belowdisplayshortskip</code> , <code>belowdisplayskip</code> , <code>cleaders</code> , <code>conditionalmathskip</code> , <code>correctionskip</code> , <code>gladers</code> , <code>ignored</code> , <code>indentskip</code> , <code>intermathskip</code> , <code>leaders</code> , <code>lefthangskip</code> , <code>leftskip</code> , <code>lineskip</code> , <code>mathskip</code> , <code>medmuskip</code> , <code>muglue</code> , <code>page</code> , <code>parfillsleftskip</code> , <code>parfillskip</code> , <code>parskip</code> , <code>righthangskip</code> , <code>rightskip</code> , <code>spaceskip</code> , <code>splittopskip</code> , <code>tabskip</code> , <code>thickmuskip</code> , <code>thinmuskip</code> , <code>topskip</code> , <code>user-skip</code> , <code>xleaders</code> , <code>xspaceskip</code> and <code>zerospaceskip</code>
<code>attr</code>	node	list of attributes
<code>leader</code>	node	pointer to a box or rule for leaders
<code>width</code>	number	the horizontal or vertical displacement
<code>stretch</code>	number	extra (positive) displacement or stretch amount
<code>stretchorder</code>	number	factor applied to stretch amount
<code>shrink</code>	number	extra (negative) displacement or shrink amount
<code>shrinkorder</code>	number	factor applied to shrink amount

---

Note that we use the key `width` in both horizontal and vertical glue. This suits the  $\text{T}_{\text{E}}\text{X}$  internals well so we decided to stick to that naming.

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 or shrink to make the content fit in the available space. The `effectiveglue` function that takes a glue node and a parent (`hlist` or `vlist`) returns the effective width of that glue item. When you pass `true` as third argument the value will be rounded.

### 10.2.9 glue\_spec nodes

Internally  $\text{LuaMetaT}_{\text{E}}\text{X}$  (like its ancestors) also uses nodes to store data that is not seen in node lists. For instance the state of expression scanning (`\dimexpr` etc.) and conditionals (`\ifcase`





etc.) is also kept in lists of nodes. A glue, which has five components, is stored in a node as well, so, where most registers store just a number, a skip register (of internal quantity) uses a pointer to a glue spec node. It has similar fields as glue nodes: `shrink`, `shrinkorder`, `stretch`, `stretchorder` and `width`, which is not surprising because in the past (and other engines than LuaTeX) a glue node also has its values stored in a glue spec. This has some advantages because often the values are the same, so for instance spacing related skips were not resolved immediately but pointed to the current value of a space related internal register (like `\spaceskip`). But, in LuaTeX we do resolve these quantities immediately and we put the current values in the glue nodes.

FIELD	TYPE	EXPLANATION
<code>width</code>	number	the horizontal or vertical displacement
<code>stretch</code>	number	extra (positive) displacement or stretch amount
<code>stretchorder</code>	number	factor applied to stretch amount
<code>shrink</code>	number	extra (negative) displacement or shrink amount
<code>shrinkorder</code>	number	factor applied to shrink amount

You will only find these nodes in a few places, for instance when you query an internal quantity. In principle we could do without them as we have interfaces that use the five numbers instead. For compatibility reasons we keep glue spec nodes exposed but this might change in the future.

### 10.2.10 kern nodes

The `\kern` command creates such nodes but for instance the font and math machinery can also add them. There are not that many fields: `attr`, `expansion` and `kern`.

FIELD	TYPE	EXPLANATION
<code>subtype</code>	number	<code>accentkern</code> , <code>fontkern</code> , <code>horizontalmathkern</code> , <code>italiccorrection</code> , <code>left-marginkern</code> , <code>mathkern</code> , <code>mathshapekern</code> , <code>rightmarginkern</code> , <code>userkern</code> and <code>verticalmathkern</code>
<code>attr</code>	node	list of attributes
<code>kern</code>	number	fixed horizontal or vertical advance
<code>expansion</code>	number	multiplier related to <code>hz</code> for font kerns

### 10.2.11 penalty nodes

The `\penalty` command is one that generates these nodes. It is one of the type of nodes often found in vertical lists. It has the fields: `attr` and `penalty`.

FIELD	TYPE	EXPLANATION
<code>subtype</code>	number	<code>afterdisplaypenalty</code> , <code>beforedisplaypenalty</code> , <code>equationnumberpenalty</code> , <code>finalpenalty</code> , <code>linebreakpenalty</code> , <code>linepenalty</code> , <code>mathpostpenalty</code> , <code>math-prepenalty</code> , <code>orphanpenalty</code> , <code>userpenalty</code> and <code>wordpenalty</code>
<code>attr</code>	node	list of attributes
<code>penalty</code>	number	the penalty value



The subtypes are just informative and  $\text{\TeX}$  itself doesn't use them. When you run into an `linebreakpenalty` you need to keep in mind that it's an accumulation of `club`, `widow` and other relevant penalties.

### 10.2.12 glyph nodes

These are probably the mostly used nodes and although you can push them in the current list with for instance `\char`  $\text{\TeX}$  will normally do it for you when it considers some input to be text. Glyph nodes are relatively large and have many fields: `attr`, `char`, `data`, `depth`, `expansion`, `font`, `height`, `hyphenate`, `language`, `left`, `lmin`, `options`, `rhmin`, `right`, `script`, `state`, `total`, `uchyph`, `width`, `xoffset`, `xscale`, `yoffset` and `yscale`.

FIELD	TYPE	EXPLANATION
subtype	number	bit field
attr	node	list of attributes
char	number	the character index in the font
font	number	the font identifier
language	number	the language identifier
left	number	the frozen <code>\lefthyphenmmin</code> value
right	number	the frozen <code>\righthyphenmmin</code> value
uchyph	boolean	the frozen <code>\uchyph</code> value
state	number	a user field (replaces the component list)
xoffset	number	a virtual displacement in horizontal direction
yoffset	number	a virtual displacement in vertical direction
width	number	the (original) width of the character
height	number	the (original) height of the character
depth	number	the (original) depth of the character
expansion	number	the to be applied expansion factor
data	number	a general purpose field for users (we had room for it)

The width, height and depth values are read-only. The expansion is assigned in the par builder and used in the backend. Valid bits for the subtype field are:

BIT	MEANING
0	character
1	ligature
2	ghost
3	left
4	right

The expansion has been introduced as part of the separation between front- and backend. It is the result of extensive experiments with a more efficient implementation of expansion. Early versions of  $\text{\LuaTeX}$  already replaced multiple instances of fonts in the backend by scaling but contrary to  $\text{\pdfTeX}$  in  $\text{\LuaTeX}$  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 `ischar` function checks if a node is a glyph node with a subtype still less than 256. This function can be used to determine if applying font logic to a glyph node makes sense. The value `nil` gets returned when the node is not a glyph, a character number is returned if the node is still tagged as character and `false` gets returned otherwise. When `nil` is returned, the `id` is also returned. The `isglyph` variant doesn't check for a subtype being less than 256, so it returns either the character value or `nil` plus the `id`. These helpers are not always faster than separate calls but they sometimes permit making more readable tests. The `usesfont` helpers takes a node and font `id` and returns `true` when a glyph or disc node references that font.

The `isnextchar` and `isprevchar` return a next node, a character code (or `false`) and an node `id` or next character code. The four `is` checkers take a node and optionally a font, data, state, scale, `xscale` and `yscale` value that are then checked.

### 10.2.13 boundary nodes

This node relates to the `\noboundary`, `\boundary`, `\protrusionboundary` and `\wordboundary` primitives. These are small nodes: `attr` and `data` are the only fields.

FIELD	TYPE	EXPLANATION
subtype	number	cancel, protrusion, user and word
attr	node	list of attributes
data	number	values 0-255 are reserved

### 10.2.14 par nodes

This node is inserted at the start of a paragraph. You should not mess too much with this one. Valid fields are: `attr`, `brokenpenalty`, `dir`, `interlinepenalty`, `leftbox`, `leftboxwidth`, `middlebox`, `rightbox` and `rightboxwidth`.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
interlinepenalty	number	local interline penalty (from <code>\localinterlinepenalty</code> )
brokenpenalty	number	local broken penalty (from <code>\localbrokenpenalty</code> )
dir	string	the direction of this par. see 10.2.15
leftbox	node	the <code>\localleftbox</code>
leftboxwidth	number	width of the <code>\localleftbox</code>
rightbox	node	the <code>\localrightbox</code>
rightboxwidth	number	width of the <code>\localrightbox</code>
middlebox	node	the <code>\localmiddlebox</code> (zero width)

A warning: never assign a node list to one of the box fields unless you are sure its internal link structure is correct, otherwise an error may result.

### 10.2.15 dir nodes

Direction nodes mark parts of the running text that need a change of direction and the `\textdirection` command generates them. Again this is a small node, we just have `attr`, `dir` and `level`.



FIELD	TYPE	EXPLANATION
subtype	number	cancel and normal
attr	node	list of attributes
dir	string	the direction (0 = l2r, 1 = r2l)
level	number	nesting level of this direction

There are only two directions: left-to-right (0) and right-to-left (1). This is different from LuaTeX that has four directions.

## 10.2.16 Whatsits

A whatsit node is a real simple one and it only has a subtype. It is even less than a user node (which it actually could be) and uses hardly any memory. What you do with it it entirely up to you: it's is real minimalistic. You can assign a subtype and it has attributes. It is all up to the user how they are handled.

## 10.2.17 Math noads

### 10.2.17.1 The concept

These are the so-called ‘noad’s and the nodes that are specifically associated with math processing. When you enter a formula, TeX creates a node list with regular nodes and noads. Then it hands over the list the math processing engine. The result of that is a nodelist without noads. Most of the noads contain subnodes so that the list of possible fields is actually quite small. Math formulas are both a linked list and a tree. For instance in  $e = mc^2$  there is a linked list  $e = m \ c$  but the  $c$  has a superscript branch that itself can be a list with branches.

First, there are the objects (the TeXbook calls them ‘atoms’) that are associated with the simple math objects: ord, op, bin, rel, open, close, punct, inner, over, under, vcenter. These all have the same fields, and they are combined into a single node type with separate subtypes for differentiation: attr, nucleus, options, prime, sub, subpre, sup and suppre.

Many object fields in math mode are either simple characters in a specific family or math lists or node lists: math\_char, math\_text\_char, sub\_box and sub\_mlist and delimiter. These are endpoints and therefore the next and prev fields of these these subnodes are unused.

Some of the more elaborate noads have an option field. The values in this bitset are common:

MEANING	BITS
set	0x08
internal	0x00 + 0x08
internal	0x01 + 0x08
axis	0x02 + 0x08
no axis	0x04 + 0x08
exact	0x10 + 0x08
left	0x11 + 0x08
middle	0x12 + 0x08
right	0x14 + 0x08



no subscript	0x21 + 0x08
no superscript	0x22 + 0x08
no script	0x23 + 0x08

### 10.2.17.2 `math_char` and `math_text_char` subnodes

These are the most common ones, as they represent characters, and they both have the same fields: `attr`, `char`, `fam` and `options`.

FIELD	TYPE	EXPLANATION
<code>attr</code>	node	list of attributes
<code>char</code>	number	the character index
<code>fam</code>	number	the family number

The `math_char` is the simplest subnode field, it contains the character and family for a single glyph object. The family eventually resolves on a reference to a font. 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).

### 10.2.17.3 `sub_box` and `sub_mlist` subnodes

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

FIELD	TYPE	EXPLANATION
<code>attr</code>	node	list of attributes
<code>list</code>	node	list of nodes

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

### 10.2.17.4 `delimiter` subnodes

There is a fifth subnode type that is used exclusively for delimiter fields. As before, the `next` and `prev` fields are unused, but we do have: `attr`, `largechar`, `largefamily`, `smallchar` and `smallfamily`.

FIELD	TYPE	EXPLANATION
<code>attr</code>	node	list of attributes
<code>smallchar</code>	number	character index of base character
<code>smallfamily</code>	number	family number of base character
<code>largechar</code>	number	character index of next larger character
<code>largefamily</code>	number	family number of next larger character

The fields `largechar` and `largefamily` can be zero, in that case the font that is set for the `smallfamily` is expected to provide the large version as an extension to the `smallchar`.



### 10.2.17.5 simple noad nodes

In these noads, the nucleus, sub and sup fields can branch of. Its fields are: attr, nucleus, options, prime, sub, subpre, sup and suppre.

FIELD	TYPE	EXPLANATION
subtype	number	accent, active, bin, close, fenced, fraction, ghost, inner, middle, op, open, ord, over, punct, radical, rel, under, variable and vcenter
attr	node	list of attributes
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
options	number	bitset of rendering options

### 10.2.17.6 accent nodes

Accent nodes deal with stuff on top or below a math constructs. They support: accent, attr, botaccent, fraction, nucleus, overlayaccent, sub, sup and topaccent.

FIELD	TYPE	EXPLANATION
subtype	number	bothflexible, fixedboth, fixedbottom and fixedtop
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
topaccent	kernel node	top accent
botaccent	kernel node	bottom accent
fraction	number	larger step criterium (divided by 1000)

### 10.2.17.7 style nodes

These nodes are signals to switch to another math style. They are quite simple: attr and style. Currently the subtype is actually used to store the style but don't rely on that for the future. Fields are: attr and style.

FIELD	TYPE	EXPLANATION
style	string	contains the style

Valid styles are: display (0), crampeddisplay (1), text (2), crampedtext (3), script (4), crampedscript (5), scriptscript (6) and crampedscriptscript (7).

### 10.2.17.8 parameter nodes

These nodes are used to (locally) set math parameters: list, name, style and value. Fields are: list, name, style and value.

FIELD	TYPE	EXPLANATION
style	string	contains the style
name	string	defines the parameter
value	number	holds the value, in case of a muglue multiple



### 10.2.17.9 choice nodes

Of its fields `attr`, `display`, `script`, `scriptscript` and `text` most are lists. Warning: never assign a node list unless you are sure its internal link structure is correct, otherwise an error can occur.

FIELD	TYPE	EXPLANATION
<code>attr</code>	node	list of attributes
<code>display</code>	node	list of display size alternatives
<code>text</code>	node	list of text size alternatives
<code>script</code>	node	list of scriptsize alternatives
<code>scriptscript</code>	node	list of scriptscriptsize alternatives

### 10.2.17.10 radical nodes

Radical nodes are the most complex as they deal with scripts as well as constructed large symbols. Many fields: `attr`, `degree`, `left`, `nucleus`, `options`, `presub`, `presup`, `prime`, `sub`, `sup` and `width`. 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 can be triggered.

FIELD	TYPE	EXPLANATION
<code>subtype</code>	number	<code>delimiterover</code> , <code>delimiterunder</code> , <code>hextensible</code> , <code>normal</code> , <code>overdelimiter</code> , <code>radical</code> , <code>root</code> and <code>underdelimiter</code>
<code>attr</code>	node	list of attributes
<code>nucleus</code>	kernel node	base
<code>sub</code>	kernel node	subscript
<code>sup</code>	kernel node	superscript
<code>left</code>	delimiter node	
<code>degree</code>	kernel node	only set by <code>\Uroot</code>
<code>width</code>	number	required width
<code>options</code>	number	bitset of rendering options

### 10.2.17.11 fraction nodes

Fraction nodes are also used for delimited cases, hence the `left` and `right` fields among: `attr`, `denom`, `fam`, `left`, `middle`, `num`, `options`, `right` and `width`.

FIELD	TYPE	EXPLANATION
<code>attr</code>	node	list of attributes
<code>width</code>	number	(optional) width of the fraction
<code>num</code>	kernel node	numerator
<code>denom</code>	kernel node	denominator
<code>left</code>	delimiter node	left side symbol
<code>right</code>	delimiter node	right side symbol
<code>middle</code>	delimiter node	middle symbol
<code>options</code>	number	bitset of rendering options

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 can result.



#### 10.2.17.12 fence nodes

Fence nodes come in pairs but either one can be a dummy (this period driven empty fence). Fields are: attr, bottom, class, delimiter, depth, height, italic, options, source and top. Some of these fields are used by the renderer and might get adapted in the process.

FIELD	TYPE	EXPLANATION
subtype	number	left, middle, no, operator, right and unset
attr	node	list of attributes
delimiter	delimiter node	delimiter specification
italic	number	italic correction
height	number	required height
depth	number	required depth
options	number	bitset of rendering options
class	number	spacing related class

## 10.3 The node library

### 10.3.1 Introduction

The node library provides methods that facilitate dealing with (lists of) nodes and their values. They allow you to create, alter, copy, delete, and insert node, the core objects within the typesetter. Nodes are represented in Lua as user data. 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 other available fields depend on the id.

- ▶ The next field returns the user data object for the next node in a linked list of nodes, or nil, if there is no next node.
- ▶ The id indicates T<sub>E</sub>X'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.

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. The general approach to a node related callback is as follows:

- ▶ Assume that the node list that you get is okay and properly double linked. If for some reason the links are not right, you can apply node.slide to the list.
- ▶ When you insert a node, make sure you use a previously removed one, a new one or a copy. Don't simply inject the same node twice.
- ▶ When you remove a node, make sure that when this is permanent, you also free the node or list.





- ▶ Although you can fool the system, normally you will trigger an error when you try to copy a nonexistent node, or free an already freed node. There is some overhead involved in this checking but the current compromise is acceptable.
- ▶ When you're done, pass back (if needed) the result. It's your responsibility to make sure that the list is properly linked (you can play safe and again apply `node.slide`. In principle you can put nodes in a list that are not acceptable in the following up actions. Some nodes get ignored, others will trigger an error, and sometimes the engine will just crash.

So, from the above it will be clear then memory management of nodes has to 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. Normally the setters and getters handle this for you.

A good example are discretionary nodes that themselves have three sublists. Internally they use special pointers, but the user never sees them because when you query them or set fields, this property is hidden and taken care of. You just see a list. But, when you mess with these sub lists it is your responsibility that it only contains nodes that are permitted in a discretionary.

There are statistics available with regards to the allocated node memory, which can be handy for tracing. Normally the amount of used nodes is not that large. Typesetting a page can involve thousands of them but most are freed when the page has been shipped out. Compared to other programs, node memory usage is not that excessive. So, if for some reason your application leaks nodes, if at the end of your run you lost as few hundred it's not a real problem. In fact, if you created boxes and made copies but not flushed them for good reason, your run will for sure end with used nodes and the statistics will mention that. The same is true for attributes and skips (glue spec nodes): keeping the current state involves using nodes.

## 10.3.2 Housekeeping

### 10.3.2.1 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.

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

When we issue this command, we get a table. The currently visible types are { [0] = "hlist", "vlist", "rule", "insert", "mark", "adjust", "boundary", "disc", "whatsit", "par", "dir", "math", "glue", "kern", "penalty", "style", "choice", "parameter", "noad", "radical", "fraction", "accent", "fence", "math\_char", "math\_text\_char", "sub\_box", "sub\_mlist", "delimiter", "glyph", "unset", [31] = "align\_record", [32] = "attribute", [33] = "glue\_spec", [34] = "temp", [35] = "split", } where the numbers are the internal identifiers. Only those nodes are reported that make sense to users so there can be gaps in the range of numbers.



### 10.3.2.2 id and type

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

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

The `node.id("glyph")` command returns the number 28 and `node.id("hlist")` returns 0 where the numbers don't relate to importance or some ordering; they just appear in the order that is handy for the engine. Commands like this are rather optimized so performance should be ok but you can of course always store the id in a Lua number.

The reverse operation is: `node.type`. If the argument is a number, then the next 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.

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

The `node.type(4)` command returns the string `mark` and `node.id(99)` returns `nil` because there is no node with that id.

### 10.3.2.3 fields and hasfield

This function returns an indexed table with valid field names for a particular type of node.

```
<table> t = node.fields(<number|string> id)
```

The function accepts a string or number, so `node.fields("glyph")` returns `{ [-1] = "prev", [0] = "next", "id", "subtype", "attr", "char", "font", "language", "lhmmin", "rhmin", "uchyph", "state", "left", "right", "xoffset", "yoffset", "xscale", "yscale", "width", "height", "depth", "total", "expansion", "data", "script", "hyphenate", "options", }` and `node.fields(12)` gives `{ [-1] = "prev", [0] = "next", "id", "subtype", "attr", "leader", "width", "stretch", "shrink", "stretchorder", "shrinkorder", "font", }`.

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

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

This function probably is not that useful but some nodes don't have a subtype, attr or prev field and this is a way to test for that.

### 10.3.2.4 isnode

```
<boolean|integer> t = node.isnode(<any> item)
```

This function returns a number (the internal index of the node) if the argument is a user data object of type `<node>` and false when no node is passed.

### 10.3.2.5 new

The `new` function creates a new node. All its fields are initialized to either zero or `nil` except for id and subtype. Instead of numbers you can also use strings (names). If you pass a second argument the subtype will be set too.



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

As already has been mentioned, you are responsible for making sure that nodes created this way are used only once, and are freed when you don't pass them back somehow.

#### 10.3.2.6 free, flushnode and flushlist

The next one frees node *n* from T<sub>E</sub>X'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. Fields that point to nodes or lists are flushed too. So, when you used their content for something else you need to set them to nil first.

```
<node> next = node.free(<node> n)
flushnode(<node> n)
```

The `free` function returns the next field of the freed node, while the `flushnode` alternative returns nothing.

A list starting with node *n* can be flushed from T<sub>E</sub>X's memory too. 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.

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

When you free for instance a discretionary node, `flushlist` is applied to the pre, post, replace so you don't need to do that yourself. Assigning them nil won't free those lists!

#### 10.3.2.7 copy and copylist

This 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.

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

A deep copy of the node list that starts at *n* can be created too. If *m* is also given, the copy stops just before node *m*.

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

Note that you cannot copy attribute lists this way. 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. When you change a value of an attribute *in* a list, it will affect all the nodes that share that list.

#### 10.3.2.8 write

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

This function will append a node list to T<sub>E</sub>X's 'current list'. The node list is not deep-copied! There is no error checking either! You might need to enforce horizontal mode in order for this to work as expected.



## 10.3.3 Manipulating lists

### 10.3.3.1 slide

This helper makes sure that the node list is double linked and returns the found tail node.

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

In most cases  $\text{\TeX}$  itself only uses next pointers but your other callbacks might expect proper prev pointers too. So, when you run into issues or are in doubt, apply the slide function before you return the list.

### 10.3.3.2 tail

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

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

### 10.3.3.3 length and count

```
<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.

```
<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.

### 10.3.3.4 remove

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

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. When the third argument is passed, the node is freed.

### 10.3.3.5 insertbefore

```
<node> head, new = node.insertbefore(<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`.

#### 10.3.3.6 insertafter

```
<node> head, new = node.insertafter(<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`.

#### 10.3.3.7 lastnode

```
<node> n = node.lastnode()
```

This function pops the last node from `TeX`'s 'current list'. It returns that node, or `nil` if the current list is empty.

#### 10.3.3.8 traverse

```
<node> t, id, subtype = 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.

#### 10.3.3.9 `traverseid`

```
<node> t, subtype = node.traverseid(<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
```

#### 10.3.3.10 `traversechar` and `traverseglyph`

The `traversechar` iterator loops over the glyph nodes in a list. Only nodes with a subtype less than 256 are seen.

```
<direct> n, font, char = node.direct.traversechar(<direct> n)
```

The `traverseglyph` iterator loops over a list and returns the list and filters all glyphs:

```
<direct> n, font, char = node.traverseglyph(<direct> n)
```

These functions are only available for direct nodes.

#### 10.3.3.11 `traverselist`

This iterator loops over the `hlist` and `vlist` nodes in a list.

```
<direct> n, id, subtype, list = node.traverselist(<direct> n)
```

The four return values can save some time compared to fetching these fields but in practice you seldom need them all. This function is only available for direct nodes.



### 10.3.3.12 traversecontent

This iterator loops over nodes that have content: hlist, vlist, glue with leaders, glyphs, disc and rules nodes.

```
<direct> n, id, subtype[, list|leader] = node.traverselist(<direct> n)
```

The four return values can save some time compared to fetching these fields but in practice you seldom need them all. This function is only available for direct nodes.

### 10.3.3.13 Reverse traversing

The traversers also support backward traversal. An optional extra boolean triggers this. Yet another optional boolean will automatically start at the end of the given list.

```
\setbox0\hbox{1 2 3 4 5}
```

```
local l = tex.box[0].list
for n in node.traverse(l) do
    print("1>",n)
end
for n in node.traverse(l,true) do
    print("2>",n)
end
for n in node.traverse(l,true,true) do
    print("3>",n)
end
for n in node.traverseid(nodes.nodecodes.glyph,l) do
    print("4>",n)
end
for n in node.traverseid(nodes.nodecodes.glyph,l,true) do
    print("5>",n)
end
for n in node.traverseid(nodes.nodecodes.glyph,l,true,true) do
    print("6>",n)
end
```

This produces something similar to this (the glyph subtype indicates that it has been processed by the font handlers):

```
1>    <node :    nil <=    1112 =>    590 : glyph 32768>
1>    <node :    1112 <=    590 =>    1120 : glue spaceskip>
1>    <node :    590 <=    1120 =>    849 : glyph 32768>
1>    <node :    1120 <=    849 =>    1128 : glue spaceskip>
1>    <node :    849 <=    1128 =>    880 : glyph 32768>
1>    <node :    1128 <=    880 =>    1136 : glue spaceskip>
1>    <node :    880 <=    1136 =>    1020 : glyph 32768>
1>    <node :    1136 <=    1020 =>    1144 : glue spaceskip>
1>    <node :    1020 <=    1144 =>    nil : glyph 32768>
```



```

2> <node : nil <= 1112 => 590 : glyph 32768>
3> <node : 1020 <= 1144 => nil : glyph 32768>
3> <node : 1136 <= 1020 => 1144 : glue spaceskip>
3> <node : 880 <= 1136 => 1020 : glyph 32768>
3> <node : 1128 <= 880 => 1136 : glue spaceskip>
3> <node : 849 <= 1128 => 880 : glyph 32768>
3> <node : 1120 <= 849 => 1128 : glue spaceskip>
3> <node : 590 <= 1120 => 849 : glyph 32768>
3> <node : 1112 <= 590 => 1120 : glue spaceskip>
3> <node : nil <= 1112 => 590 : glyph 32768>
4> <node : nil <= 1112 => 590 : glyph 32768>
4> <node : 590 <= 1120 => 849 : glyph 32768>
4> <node : 849 <= 1128 => 880 : glyph 32768>
4> <node : 880 <= 1136 => 1020 : glyph 32768>
4> <node : 1020 <= 1144 => nil : glyph 32768>
5> <node : nil <= 1112 => 590 : glyph 32768>
6> <node : 1020 <= 1144 => nil : glyph 32768>
6> <node : 880 <= 1136 => 1020 : glyph 32768>
6> <node : 849 <= 1128 => 880 : glyph 32768>
6> <node : 590 <= 1120 => 849 : glyph 32768>
6> <node : nil <= 1112 => 590 : glyph 32768>

```

#### 10.3.3.14 findnode

This helper returns the location of the first match at or after node n:

```

<node> n = node.findnode(<node> n, <integer> subtype)
<node> n, subtype = node.findnode(<node> n)

```

### 10.3.4 Glue handling

#### 10.3.4.1 setglue

You can set the five properties of a glue in one go. If a non-numeric value is passed the property becomes zero.

```

node.setglue(<node> n)
node.setglue(<node> n,width,stretch,shrink,stretchorder,shrinkorder)

```

When you pass values, only arguments that are numbers are assigned so

```
node.setglue(n,655360,false,65536)
```

will only adapt the width and shrink.

When a list node is passed, you set the glue, order and sign instead.

#### 10.3.4.2 getglue

The next call will return 5 values or nothing when no glue is passed.





```
<integer> width, <integer> stretch, <integer> shrink, <integer> stretchorder,  
    <integer> shrinkorder = node.getglue(<node> n)
```

When the second argument is false, only the width is returned (this is consistent with `tex.get`).  
When a list node is passed, you get back the glue that is set, the order of that glue and the sign.

### 10.3.4.3 iszeroglue

This function returns true when the width, stretch and shrink properties are zero.

```
<boolean> isglue = node.iszeroglue(<node> n)
```

## 10.3.5 Attribute handling

### 10.3.5.1 Attributes

Assignments to attributes registers result in assigning lists with set attributes to nodes and the implementation is non-trivial because the value that is attached to a node is essentially a (sorted) 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.

### 10.3.5.2 attribute nodes

An attribute comes in two variants, indicated by subtype. Because attributes are stored in a sorted linked list, and because they are shared, the first node is a list reference node and the following ones are value nodes. So, most attribute nodes are value nodes. These are forward linked lists. The reference node has fields:

FIELD	TYPE	EXPLANATION
next	node	pointer to the first attribute
count	number	the reference count

Value nodes have these:

FIELD	TYPE	EXPLANATION
next	node	pointer to the next attribute
index	number	the attribute index
value	number	the attribute value

Because there are assumptions to how these list are build you should rely on the helpers, also because details might change.

### 10.3.5.3 currentattr

This returns the currently active list of attributes, if there is one.

```
<node> m = node.currentattr()
```

The intended usage of `currentattr` is as follows:



```
local x1 = node.new("glyph")
x1.attr = node.currentattr()
local x2 = node.new("glyph")
x2.attr = node.currentattr()
```

or:

```
local x1 = node.new("glyph")
local x2 = node.new("glyph")
local ca = node.currentattr()
x1.attr = ca
x2.attr = ca
```

The attribute lists are reference counted and the assignment takes care of incrementing the count. 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`.

#### 10.3.5.4 **hasattribute**

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

#### 10.3.5.5 **getattribute**

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

Tests if a node has an attribute with number `id` set. It returns the value, or, if no match is found, `nil`. If no `id` is given then the zero attributes is assumed.

#### 10.3.5.6 **findattribute**

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

Finds the first node that has attribute with number `id` set. It returns the value and the node if there is a match and otherwise nothing.

#### 10.3.5.7 **setattribute**

```
node.setattribute(<node> n, <number> id, <number> val)
```

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

#### 10.3.5.8 **unsetattribute**

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

## 10.3.6 Glyph handling

### 10.3.6.1 firstglyph

```
<node> n = node.firstglyph(<node> n)
<node> n = node.firstglyph(<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.

### 10.3.6.2 ischar and isglyph

The subtype of a glyph node signals if the glyph is already turned into a character reference or not.

```
<boolean> b = node.ischar(<node> n)
<boolean> b = node.isglyph(<node> n)
```

### 10.3.6.3 hasglyph

This function returns the first glyph or disc node in the given list:

```
<node> n = node.hasglyph(<node> n)
```

### 10.3.6.4 ligaturing

```
<node> h, <node> t, <boolean> success = node.ligaturing(<node> n)
<node> h, <node> t, <boolean> success = node.ligaturing(<node> n, <node> m)
```

Apply T<sub>E</sub>X-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).

### 10.3.6.5 kerning

```
<node> h, <node> t, <boolean> success = node.kerning(<node> n)
<node> h, <node> t, <boolean> success = node.kerning(<node> n, <node> m)
```

Apply T<sub>E</sub>X-style kerning to the specified node list. 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).

### 10.3.6.6 unprotectglyph[s]

```
node.unprotectglyph(<node> n)
node.unprotectglyphs(<node> n, [<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. The second argument is optional and indicates the end of a range.

#### 10.3.6.7 **protectglyph[s]**

```
node.protectglyph(<node> n)
node.protectglyphs(<node> n, [<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. A single character can be marked by the singular call. The second argument is optional and indicates the end of a range.

#### 10.3.6.8 **protrusionskipable**

```
<boolean> skipable = node.protrusionskipable(<node> n)
```

Returns `true` if, for the purpose of line boundary discovery when character protrusion is active, this node can be skipped.

#### 10.3.6.9 **checkdiscretionary, checkdiscretionaries**

When you fool around with disc nodes you need to be aware of the fact that they have a special internal data structure. As long as you reassign the fields when you have extended the lists it's ok because then the tail pointers get updated, but when you add to list without reassigning you might end up in trouble when the linebreak routine kicks in. You can call this function to check the list for issues with disc nodes.

```
node.checkdiscretionary(<node> n)
node.checkdiscretionaries(<node> head)
```

The plural variant runs over all disc nodes in a list, the singular variant checks one node only (it also checks if the node is a disc node).

#### 10.3.6.10 **flattendiscretionaries**

This function will remove the discretionaries in the list and inject the replace field when set.

```
<node> head, count = node.flattendiscretionaries(<node> n)
```

### 10.3.7 **Packaging**

#### 10.3.7.1 **hpack**

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.



```

<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)

```

Caveat: 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!

### 10.3.7.2 vpack

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.

```

<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)

```

The second return value is the badness of the generated box. See the description of `hpack` for a few memory allocation caveats.

### 10.3.7.3 dimensions, rangedimensions, naturalwidth

```

<number> w, <number> h, <number> d =
    node.dimensions(<node> n)
<number> w, <number> h, <number> d =
    node.dimensions(<node> n, <node> t)

```

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> glueset, <number> gluesign, <number> glueorder,
        <node> n)
<number> w, <number> h, <number> d =
    node.dimensions(<number> glueset, <number> gluesign, <number> glueorder,
        <node> n, <node> t)

```

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 in between 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].glueset,
    tex.box[0].gluesign,
    tex.box[0].glueorder,
    tex.box[0].head.next,
    node.tail(tex.box[0].head)
)) }

```

You need to keep in mind that this is one of the few places in  $\TeX$  where floats are used, which means that you can get small differences in rounding when you compare the width reported by `hpack` with `dimensions`.

The second alternative saves a few lookups and can be more convenient in some cases:

```

<number> w, <number> h, <number> d =
    node.rangedimensions(<node> parent, <node> first)
<number> w, <number> h, <number> d =
    node.rangedimensions(<node> parent, <node> first, <node> last)

```

A simple and somewhat more efficient variant is this:

```

<number> w =
    node.naturalwidth(<node> start, <node> stop)

```

## 10.3.8 Math

### 10.3.8.1 `mlisttohlist`

```

<node> h =
    node.mlisttohlist(<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 `mlisttohlist`.

### 10.3.8.2 `endofmath`

```

<node> t = node.endofmath(<node> start)

```

Looks for and returns the next `math_node` following the `start`. If the given node is a math end node this helper returns that node, else it follows the list and returns the next math endnote. If no such node is found `nil` is returned.

## 10.4 Two access models

Deep down in  $\TeX$  a node has a number which is a numeric 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 too. Each node gets a number that is in fact an index in the



memory table and that number often is reported when you print node related information. You go from user data nodes and there numeric references and back with:

```
<integer> d = node.todirect(<node> n)
<node> n = node.tonode(<integer> d)
```

The user data model is rather robust as it is a virtual interface with some additional checking while the more direct access which uses the node numbers directly. However, even with user data you can get into troubles when you free nodes that are no longer allocated or mess up lists. if you apply `tostring` to a node you see its internal (direct) number and id.

The first model provides key based access while the second always accesses fields via functions:

```
nodeobject.char
getfield(nodenum, "char")
```

If you use the direct model, even if you know that you deal with numbers, you should not depend on that property but treat it as 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, multiplying a node number makes no sense.

So our advice is: use the indexed (table) approach when possible and investigate the direct one when speed might be a real issue. For that reason LuaTeX 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 nodes are accessed 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
```

In the direct namespace there are more helpers and most of them are accompanied by setters. The getters and setters are clever enough to see what node is meant. We don't deal with whatsit nodes: their fields are always accessed by name. It doesn't make sense to add getters for all fields, we just identifier the most likely candidates. In complex documents, many node and fields types never get seen, or seen only a few times, but for instance glyphs are candidates for such optimization. The `node.direct` interface has some more helpers.<sup>8</sup>

The `setdisc` helper takes three (optional) arguments plus an optional fourth indicating the subtype. Its `getdisc` takes an optional boolean; when its value is `true` the tail nodes will also be returned. The `setfont` helper takes an optional second argument, it being the character. The `directnode` setter `setlink` takes a list of nodes and will link them, thereby ignoring `nil` entries. The first valid node is returned (beware: for good reason it assumes single nodes). For rarely used fields no helpers are provided and there are a few that probably are used seldom too but were added for consistency. You can of course always define additional accessors using `getfield` and `setfield` with little overhead. When the second argument of `setattributelist` is `true` the current attribute list is assumed.

The `reverse` function reverses a given list. The `exchange` function swaps two nodes; it takes upto three arguments: a head node, and one or two to be swapped nodes. When there is no third argument, it will assume that the node following node is to be used. So we have:

```
head = node.direct.reverse(head)
head = node.direct.exchange(head,first,[second])
```

In ConT<sub>E</sub>Xt some of the not performance-critical user data variants are emulated in Lua and not in the engine, so we retain downward compatibility.

FUNCTION	NODE	DIRECT	emulated
checkdiscretionaries	—	+	+
checkdiscretionary	—	+	+
copylist	+	+	
copy	+	+	
count	—	+	+
currentattributes	+	+	
dimensions	—	+	+
effectiveglue	—	+	+
endofmath	—	+	+
findattributerange	—	+	
findattribute	—	+	+
findnode	—	+	
firstglyph	—	+	+
flattendiscretionaries	—	+	+
flushlist	+	+	

<sup>8</sup> We can define the helpers in the node namespace with `getfield` which is about as efficient, so at some point we might provide that as module.





flushnode	+	+	
free	+	+	
getattributes	-	+	
getattribute	+	+	
getpropertystable	+	+	
getsynctexfields	-	+	
getattributelist	-	+	
getboth	-	+	
getbox	-	+	
getchar	-	+	
getdata	-	+	
getdepth	-	+	
getdirection	-	+	
getdisc	-	+	
getexpansion	-	+	
getfam	-	+	
getfield	+	+	
getfont	-	+	
getglue	-	+	+
getglyphdata	-	+	
getglyphdimensions	-	+	+
getglyphscript	-	+	
getglyphstate	-	+	
getheight	-	+	
getid	-	+	
getindex	-	+	
getkerndimension	-	+	+
getkern	-	+	
getlanguage	-	+	
getleader	-	+	
getlist	-	+	
getnext	-	+	
getnormalizedline	-	+	
getnucleus	-	+	
getoffsets	-	+	
getoptions	-	+	
getorientation	-	+	
getparstate	-	+	
getpenalty	-	+	
getpost	-	+	
getprev	-	+	
getpre	-	+	
getproperty	+	+	
getreplace	-	+	
getscales	-	+	
getscript	-	+	



getshift	-	+	
getstate	-	+	
getsubpre	-	+	
getsubtype	-	+	
getsub	-	+	
getsuppre	-	+	
getsup	-	+	
gettotal	+	+	
getwhd	-	+	
getwidth	-	+	
getxscale	-	+	
getxyscale	-	+	
getyscale	-	+	
hasattribute	+	+	
hasdimensions	-	+	
hasfield	+	+	
hasglyphoption	-	+	+
hasglyph	-	+	+
hpack	-	+	+
hyphenating	-	+	+
ignoremathskip	-	+	
insertafter	+	+	
insertbefore	+	+	
ischar	-	+	
isdirect	-	+	
isglyph	-	+	
isnextchar	-	+	
isnextglyph	-	+	
isnode	+	+	
isprevchar	-	+	
isprevglyph	-	+	
isvalid	-	+	
iszeroglue	-	+	+
kerning	-	+	+
lastnode	-	+	+
length	-	+	+
ligaturing	-	+	+
makeextensible	-	+	+
migrate	-	+	
mlisttohlist	-	+	+
naturalwidth	-	+	+
new	+	+	
protectglyphs	-	+	+
protectglyph	-	+	+
protrusionskipable	-	+	+
rangedimensions	-	+	+



remove	+	+
setattributes	-	+
setattribute	+	+
setsynctexfields	-	+
setattributelist	-	+
setboth	-	+
setbox	-	+
setchar	-	+
setdata	-	+
setdepth	-	+
setdirection	-	+
setdisc	-	+
setexpansion	-	+
setfam	-	+
setfield	+	+
setfont	-	+
setglue	+	+
setglyphdata	-	+
setglyphscript	-	+
setglyphstate	-	+
setheight	-	+
setindex	-	+
setkern	-	+
setlanguage	-	+
setleader	-	+
setlink	-	+
setlist	-	+
setnext	-	+
setnucleus	-	+
setoffsets	-	+
setoptions	-	+
setorientation	-	+
setpenalty	-	+
setpost	-	+
setprev	-	+
setpre	-	+
setproperty	+	+
setreplace	-	+
setscales	-	+
setscript	-	+
setshift	-	+
setsplit	-	+
setstate	-	+
setsubpre	-	+
setsubtype	-	+
setsub	-	+



setsuppre	–	+	
setup	–	+	
setwhd	–	+	
setwidth	–	+	
slide	–	+	+
startofpar	–	+	
subtype	–	–	
tail	+	+	
todirect	–	+	
tonode	–	+	
tostring	+	–	
total	–	+	
tovaliddirect	–	+	
traversechar	+	+	
traversecontent	+	+	
traverseglyph	+	+	
traverseid	+	+	
traverselist	+	+	
traverse	+	+	
type	+	–	
unprotectglyphs	–	+	+
unprotectglyph	–	+	+
unsetattributes	–	+	
unsetattribute	+	+	
usedlist	–	+	+
usesfont	–	+	+
verticalbreak	–	+	
vpack	–	+	+
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.

Some of the getters and setters handle multiple node types, given that the field is relevant. In that case, some field names are considered similar (like `kern` and `width`, or `data` and `value`). In retrospect we could have normalized field names better but we decided to stick to the original (internal) names as much as possible. After all, at the Lua end one can easily create synonyms.

Some nodes have indirect references. For instance a math character refers to a family instead of a font. In that case we provide a virtual font field as accessor. So, `getfont` and `.font` can be used on them. The same is true for the `width`, `height` and `depth` of glue nodes. These actually access the `spec` node properties, and here we can set as well as get the values.

You can set and query the SyncTeX fields, a file number aka tag and a line number, for a glue, kern, hlist, vlist, rule and math nodes as well as glyph nodes (although this last one is not used in native SyncTeX).



```
node.setsynctexfields(<integer> f, <integer> l)
<integer> f, <integer> l =
    node.getsynctexfields(<node> n)
```

Of course you need to know what you're doing as no checking on sane values takes place. Also, the synctex interpreter used in editors is rather peculiar and has some assumptions (heuristics).

## 10.5 Normalization

As an experiment the lines resulting from paragraph construction can be normalized. There are several modes, that can be set and queried with:

```
node.direct.setnormalize(<integer> n)
<integer> n = node.direct.getnormalize()
```

The state of a line (a hlist) can be queried with:

```
<integer> leftskip, <integer> rightskip,
    <integer> lefthangskip, <integer> righthangskip,
    <node> head, <node> tail,
    <integer> parindent, <integer> parfillskip = node.direct.getnormalized()
```

The modes accumulate, so mode 4 includes 1 upto 3:

VALUE	EXPLANATION
1	left and right skips and directions
2	indentation and parfill skip
3	hanging indentation and par shapes
4	idem but before left and right skips
5	inject compensation for overflow

This is experimental code and might take a while to become frozen.

## 10.6 Properties

Attributes are a convenient way to relate extra information to a node. You can assign them at the  $\text{T}_{\text{E}}\text{X}$  end as well as at the Lua end and consult them at the Lua end. One big advantage is that they obey grouping. They are linked lists and normally checking for them is pretty efficient, even if you use a lot of them. A macro package has to provide some way to manage these attributes at the  $\text{T}_{\text{E}}\text{X}$  end because otherwise clashes in their usage can occur.

Each node also can have a properties table and you can assign values to this table using the `setproperty` function and get properties using the `getproperty` function. Managing properties is way more demanding than managing attributes.

Take the following example:

```
\directlua {
    local n = node.new("glyph")
```



```

node.setProperty(n, "foo")
print(node.getProperty(n))

node.setProperty(n, "bar")
print(node.getProperty(n))

node.free(n)
}

```

This will print foo and bar which in itself is not that useful when multiple mechanisms want to use this feature. A variant is:

```

\directlua {
    local n = node.new("glyph")

    node.setProperty(n, { one = "foo", two = "bar" })
    print(node.getProperty(n).one)
    print(node.getProperty(n).two)

    node.free(n)
}

```

This time we store two properties with the node. It really makes sense to have a table as property because that way we can store more. But in order for that to work well you need to do it this way:

```

\directlua {
    local n = node.new("glyph")

    local t = node.getProperty(n)

    if not t then
        t = { }
        node.setProperty(n, t)
    end
    t.one = "foo"
    t.two = "bar"

    print(node.getProperty(n).one)
    print(node.getProperty(n).two)

    node.free(n)
}

```

Here our own properties will not overwrite other users properties unless of course they use the same keys. So, eventually you will end up with something:

```

\directlua {

```



```

local n = node.new("glyph")

local t = node.getproperty(n)

if not t then
    t = { }
    node.setproperty(n,t)
end
t.myself = { one = "foo", two = "bar" }

print(node.getproperty(n).myself.one)
print(node.getproperty(n).myself.two)

node.free(n)
}

```

This assumes that only you use `myself` as subtable. The possibilities are endless but care is needed. For instance, the generic font handler that ships with ConT<sub>E</sub>Xt uses the injections subtable and you should not mess with that one!

There are a few helper functions that you normally should not touch as user: `getpropertytable` and will give the table that stores properties (using direct entries) and you can best not mess too much with that one either because LuaT<sub>E</sub>X itself will make sure that entries related to nodes will get wiped when nodes get freed, so that the Lua garbage collector can do its job. In fact, the main reason why we have this mechanism is that it saves the user (or macro package) some work. One can easily write a property mechanism in Lua where after a shipout properties gets cleaned up but it's not entirely trivial to make sure that with each freed node also its properties get freed, due to the fact that there can be nodes left over for a next page. And having a callback bound to the node deallocator would add way to much overhead.

When we copy a node list that has a table as property, there are several possibilities: we do the same as a new node, we copy the entry to the table in properties (a reference), we do a deep copy of a table in the properties, we create a new table and give it the original one as a metatable. After some experiments (that also included timing) with these scenarios we decided that a deep copy made no sense, nor did nilling. In the end both the shallow copy and the metatable variant were both ok, although the second one is slower. The most important aspect to keep in mind is that references to other nodes in properties no longer can be valid for that copy. We could use two tables (one unique and one shared) or metatables but that only complicates matters.

When defining a new node, we could already allocate a table but it is rather easy to do that at the lua end e.g. using a metatable `__index` method. That way it is under macro package control. When deleting a node, we could keep the slot (e.g. setting it to false) but it could make memory consumption raise unneeded when we have temporary large node lists and after that only small lists. Both are not done because in the end this is what happens now: when a node is copied, and it has a table as property, the new node will share that table. The copy gets its own table with the original table as metatable.

A few more experiments were done. For instance: copy attributes to the properties so that we have fast access at the Lua end. In the end the overhead is not compensated by speed and



convenience, in fact, attributes are not that slow when it comes to accessing them. So this was rejected.

Another experiment concerned a bitset in the node but again the gain compared to attributes was neglectable and given the small amount of available bits it also demands a pretty strong agreement over what bit represents what, and this is unlikely to succeed in the T<sub>E</sub>X community. It doesn't pay off.

Just in case one wonders why properties make sense: it is not so much speed that we gain, but more convenience: storing all kinds of (temporary) data in attributes is no fun and this mechanism makes sure that properties are cleaned up when a node is freed. Also, the advantage of a more or less global properties table is that we stay at the Lua end. An alternative is to store a reference in the node itself but that is complicated by the fact that the register has some limitations (no numeric keys) and we also don't want to mess with it too much.





# 11 Lua callbacks

## 11.1 Registering callbacks

*The callbacks are a moving target. Don't bother me with questions about them.*

This library has functions that register, find and list callbacks. Callbacks are Lua functions that are called in well defined places. There are two kinds 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 functionality 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 = callback.register(<string> callback_name, <function> func)
id = callback.register(<string> callback_name, nil)
id = 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.

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!

```
<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, `find` returns `nil`. The known function can be used to check if a callback is supported.

```
if callback.known("foo") then ... end
```



## 11.2 File related callbacks

### 11.2.1 find\_format\_file and find\_log\_file

These callbacks are called as:

```
<string> actualname =  
    function (<string> askedname)
```

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

### 11.2.2 open\_data\_file

This callback function gets a filename passed:

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

The return value is either the boolean value false or a table with two functions. A mandate reader function will be called once for each new line to be read, the optional close function will be called once 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.

## 11.3 Data processing callbacks

### 11.3.1 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.

## 11.4 Node list processing callbacks

The description of nodes and node lists is in chapter 10.

### 11.4.1 contribute\_filter

This callback is called when LuaTeX adds contents to list:



```
function(<string> extrainfo)
end
```

The string reports the group code. From this you can deduce from what list you can give a treat.

VALUE	EXPLANATION
pre_box	interline material is being added
pre_adjust	\vadjust material is being added
box	a typeset box is being added (always called)
adjust	\vadjust material is being added

### 11.4.2 buildpage\_filter

This callback is called whenever Lua<sub>T</sub><sub>E</sub>X 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 T<sub>E</sub>X's state is with respect to the 'current page'. The possible values for the buildpage\_filter callback are:

VALUE	EXPLANATION
alignment	a (partial) alignment is being added
after_output	an output routine has just finished
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	Lua <sub>T</sub> <sub>E</sub> X is terminating (it's all over)

### 11.4.3 build\_page\_insert

This callback is called when the pagebuilder adds an insert. There is not much control over this mechanism but this callback permits some last minute manipulations of the spacing before an insert, something that might be handy when for instance multiple inserts (types) are appended in a row.

```
function(<number> n, <number> i)
    return <number> register
end
```

with



VALUE	EXPLANATION
n	the insert class
i	the order of the insert

The return value is a number indicating the skip register to use for the prepended spacing. This permits for instance a different top space (when `i` equals one) and intermediate space (when `i` is larger than one). Of course you can mess with the insert box but you need to make sure that Lua<sub>T</sub><sub>E</sub>X is happy afterwards.

#### 11.4.4 `pre_linebreak_filter`

This callback is called just before Lua<sub>T</sub><sub>E</sub>X starts converting a list of nodes into a stack of `\hboxes`, after the addition of `\parfillskip`.

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

The string called `groupcode` identifies the nodelist's context within T<sub>E</sub>X'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	<code>\hbox</code> in horizontal mode
adjusted_hbox	<code>\hbox</code> in vertical mode
vbox	<code>\vbox</code>
vtop	<code>\vtop</code>
align	<code>\halign</code> or <code>\valign</code>
disc	discretionaries
insert	packaging an insert
vcenter	<code>\vcenter</code>
local_box	<code>\localleftbox</code> or <code>\localrightbox</code>
split_off	top of a <code>\vsplit</code>
split_keep	remainder of a <code>\vsplit</code>
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 successful 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.



### 11.4.5 linebreak\_filter

This callback replaces LuaT<sub>E</sub>X'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>, LuaT<sub>E</sub>X 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 an \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'.

### 11.4.6 append\_to\_vlist\_filter

This callback is called whenever LuaT<sub>E</sub>X adds a box to a vertical list (the mirrored argument is obsolete):

```
function(<node> box, <string> locationcode, <number> prevdepth)
    return list [, prevdepth [, checkdepth ] ]
end
```

It is ok to return nothing or nil in which case you also need to flush the box or deal with it yourself. The prevdepth is also optional. Locations are box, alignment, equation, equation\_number and post\_linebreak. When the third argument returned is true the normal prevdepth correction will be applied, based on the first node.

### 11.4.7 post\_linebreak\_filter

This callback is called just after LuaT<sub>E</sub>X has converted a list of nodes into a stack of \hboxes.

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

This callback does not replace any internal code.

### 11.4.8 glyph\_run

When set this callback is triggered when T<sub>E</sub>X normally handles the ligaturing and kerning. In LuaT<sub>E</sub>X you use the hpack\_filter and per\_linebreak\_filter callbacks for that (where each passes different arguments). This callback doesn't get triggered when there are no glyphs (in LuaT<sub>E</sub>X this optimization is controlled by a variable).



```
function(<node> head, <string> groupcode, <number> direction))
    return <node> newhead
end
```

The traditional T<sub>E</sub>X font processing is bypassed so you need to take care of that with the helpers. (For the moment we keep the ligaturing and kerning callbacks but they are kind of obsolete.)

#### 11.4.9 hpack\_filter

This callback is called when T<sub>E</sub>X 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 [, <number> direction] [, <node> attributelist])
    return <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.

This callback does not replace any internal code.

#### 11.4.10 vpack\_filter

This callback is called when T<sub>E</sub>X 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 T<sub>E</sub>X's `\maxdepth` setting.

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

This callback does not replace any internal code.

#### 11.4.11 hpack\_quality

This callback can be used to intercept the overflow messages that can result from packing a horizontal list (as happens in the `par` builder). The function takes a few arguments:

```
function(<string> incident, <number> detail, <node> head, <number> first,
    <number> last)
    return <node> whatever
end
```

The incident is one of `overfull`, `underfull`, `loose` or `tight`. The detail is either the amount of overflow in case of `overfull`, or the badness otherwise. The head is the list that is constructed



(when protrusion or expansion is enabled, this is an intermediate list). Optionally you can return a node, for instance an overfull rule indicator. That node will be appended to the list (just like  $\TeX$ 's own rule would).

#### 11.4.12 vpack\_quality

This callback can be used to intercept the overfull messages that can result from packing a vertical list (as happens in the page builder). The function takes a few arguments:

```
function(<string> incident, <number> detail, <node> head, <number> first,  
        <number> last)  
end
```

The incident is one of `overfull`, `underfull`, `loose` or `tight`. The detail is either the amount of overflow in case of `overfull`, or the badness otherwise. The head is the list that is constructed.

#### 11.4.13 process\_rule

This is an experimental callback. It can be used with rules of subtype 4 (user). The callback gets three arguments: the node, the width and the height. The callback can use `pdf.print` to write code to the pdf file but beware of not messing up the final result. No checking is done.

#### 11.4.14 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 [, <number> direction])  
    return <node> newhead  
end
```

This callback does not replace any internal code.

#### 11.4.15 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.

#### 11.4.16 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.

You must not ruin the node list. For instance, the head normally is a local par node, and the tail a glue. Messing too much can push LuaTeX into panic mode.

#### 11.4.17 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.

You must not ruin the node list. For instance, the head normally is a local par node, and the tail a glue. Messing too much can push LuaTeX into panic mode.

#### 11.4.18 insert\_par

Each paragraph starts with a local par node that keeps track of for instance the direction. You can hook a callback into the creator:

```
function(<node> par, <string> location)
end
```

There is no return value and you should make sure that the node stays valid as otherwise TeX can get confused.

#### 11.4.19 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.

## 11.5 Information reporting callbacks

### 11.5.1 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.

### 11.5.2 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.

### 11.5.3 stop\_run

```
function()  
end
```

This callback replaces the code that prints LuaTeX's statistics and 'output written to' messages. The engine can still do housekeeping and therefore you should not rely on this hook for postprocessing the pdf or log file.

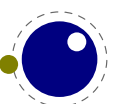
### 11.5.4 intercept\_tex\_error, intercept\_lua\_error

```
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. The TeX related callback gets two arguments: the current processing mode and a boolean indicating if there was a runaway.

### 11.5.5 show\_error\_message and show\_warning\_message

```
function()  
end
```



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

### 11.5.6 start\_file

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

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

VALUE	MEANING
1	a normal data file, like a T <sub>E</sub> X 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

### 11.5.7 stop\_file

```
function(category)
end
```

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

### 11.5.8 wrapup\_run

This callback is called after the pdf and log files are closed. Use it at your own risk.

## 11.6 Font-related callbacks

### 11.6.1 define\_font

```
function(<string> name, <number> size)
    return <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 design size of the font.



The font can be defined with `font.define` which returns a font identifier that can be returned in the callback. So, contrary to Lua<sub>T</sub><sub>E</sub>X, in LuaMeta<sub>T</sub><sub>E</sub>X we only accept a number.

The internal structure of the font table that is passed to `font.define` is explained in chapter 6. 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.

### 11.6.2 show\_whatsit

Because we only have a generic `whatsit` it is up to the macro package to provide details when tracing them.

```
function(<node> whatsit, <number> indentation,  
        <number> tracinglevel, <number> currentlevel, <number> inputlevel)  
    -- no return value  
end
```

The `indentation` tells how many periods are to be typeset if you want to be compatible with the rest of tracing. The `tracinglevels` indicates if the current level and/or input level are shown cf. `\tracinglevels`. Of course one is free to show whatever in whatever way suits the `whatsit` best.





# 12 The T<sub>E</sub>X related libraries

## 12.1 The lua library

### 12.1.1 Version information

This version of the used Lua interpreter (currently Lua 5.4) can be queried with:

```
<string> v = lua.getversion()
```

The name of used startup file, if at all, is returned by:

```
<string> s = lua.getstartupfile()
```

For this document the reported value is:

```
c:/data/develop/tex-context/tex/texmf-cache/luametatex-cache/context/764bd4e1ce0f004ab3cec90018f8b80a/formats/luametatex/cont-en.lui
```

### 12.1.2 Table allocators

Sometimes performance (and memory usage) can benefit a little from it preallocating a table with `newtable`:

```
<table> t = lua.newtable(100,5000)
```

This preallocates 100 hash entries and 5000 index entries. The `newindex` function create an indexed table with preset values:

```
<table> t = lua.newindex(2500,true)
```

### 12.1.3 Bytecode registers

Lua registers can be used to store Lua code 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  
<function> f = lua.bytecode[<number> n] % -- f()
```

The contents of the `lua.bytecode` array is stored inside the format file as actual Lua bytecode, so it can also be used to preload Lua code. The function must not contain any upvalues. The associated function calls are:

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



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.setbytecode(n,loadfile(filename))
```

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

```
lua.callbytecode(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`.

### 12.1.4 Introspection

The `getstacktop` function return a number indicating how full the Lua stack is. This function only makes sense as breakpoint when checking some mechanism going haywire.

There are four time related helpers. The `getruntime` function returns the time passed since startup. The `getcurrenttime` does what its name says. Just play with them to see how it pays off. The `getpreciseticks` returns a number that can be used later, after a similar call, to get a difference. The `getpreciseseconds` function gets such a tick (delta) as argument and returns the number of seconds. Ticks can differ per operating system, but one always creates a reference first and then deltas to this reference.

## 12.2 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. There are toplevel items and items that are tables with subentries. The current list is:

TOPLEVEL STATISTICS	
banner	This is LuaMetaT <sub>E</sub> X, Version 2.09.42
copyright	Taco Hoekwater, Hans Hagen & Wolfgang Schuster
development_id	20220308
filename	E:/context/manuals/mkiv/external/luametatex/luametatex-tex.tex
format_id	642
logfilename	luametatex.log
luatex_engine	luametatex
luatex_revision	42
luatex_verbos	2.09.42
luatex_version	209



permit_loadlib	false
run_state	1
used_compiler	gcc

#### **BUFFERSTATE.\***

all	1000000
ini	-1
max	100000000
mem	1000000
min	1000000
ptr	0
set	10000000
stp	1000000
top	858

#### **CALLBACKSTATE.\***

bytecode	601
count	208225
direct	1262
file	14786
function	53700
local	0
message	0
saved	133766
value	4110

#### **ENGINESTATE.\***

banner	This is LuaMetaTeX, Version 2.09.42
copyright	Taco Hoekwater, Hans Hagen & Wolfgang Schuster
development_id	20220308
format_id	642
logfile	luametateX.log
luatex_engine	luametateX
luatex_revision	42
luatex_verbos	2.09.42
luatex_version	209
permit_loadlib	false
run_state	1
tex_hash_size	131072
used_compiler	gcc

#### **ERRORLINESTATE.\***

max	255
min	132
set	250
top	0



**ERRORSTATE.\***

error	unset
errorcontext	unset
luaerror	unset

**EXPANDSTATE.\***

max	1000000
min	10000
set	10000
top	10

**EXTRASTATE.\***

all	0
ini	-1
max	-1
mem	0
min	-1
ptr	0
set	-1
stp	-1
top	0

**FILESTATE.\***

all	16000
ini	-1
max	2000
mem	500
min	500
ptr	6
set	2000
stp	250
top	11

**FONTSTATE.\***

all	8699096
ini	-1
max	100000
mem	8699096
min	250
ptr	28
set	100000
stp	250
top	250

**HALFERRORLINESTATE.\***

max	255
-----	-----





min	80
set	234
top	0

---

#### **HASHSTATE.\***

all	2400000
ini	0
max	2097152
mem	150000
min	150000
ptr	12059
set	250000
stp	100000
top	740702

---

#### **INPUTSTATE.\***

all	320000
ini	-1
max	100000
mem	10000
min	10000
ptr	7
set	100000
stp	10000
top	58

---

#### **INSERTSTATE.\***

all	10240
ini	-1
max	500
mem	320
min	10
ptr	7
set	250
stp	25
top	10

---

#### **LANGUAGESTATE.\***

all	96
ini	0
max	10000
mem	96
min	250
ptr	0
set	250



stp	250
top	250

#### LOOKUPSTATE.\*

all	-1
ini	45747
max	2097152
mem	-1
min	150000
ptr	55452
set	250000
stp	100000
top	131074

#### LUASTATE.\*

bytecodebytes	16064
bytecodes	1003
functionsizes	32768
propertyssizes	10000
statebytes	147562437
statebytesmax	162048213

#### MARKSTATE.\*

all	28800
ini	-1
max	10000
mem	1200
min	50
ptr	28
set	250
stp	50
top	50

#### NESTSTATE.\*

all	56000
ini	-1
max	10000
mem	1000
min	1000
ptr	0
set	10000
stp	1000
top	19

#### NODESTATE.\*

all	9000000
-----	---------



ini	0
max	100000000
mem	1000000
min	1000000
ptr	-172307
set	100000000
stp	500000
top	211928

---

#### PARAMETERSTATE.\*

all	80000
ini	-1
max	100000
mem	20000
min	20000
ptr	1
set	100000
stp	10000
top	54

---

#### POOLSTATE.\*

all	1080137
ini	863699
max	100000000
mem	1080137
min	10000000
ptr	-1
set	10000000
stp	1000000
top	-1

---

#### READSTATE.\*

filename	E:/context/manuals/mkiv/external/luametateX/luametateX-tex.tex
iocode	5
linenumber	210
skiplinenumber	163

---

#### SAVESTATE.\*

all	160000
ini	-1
max	500000
mem	10000
min	100000
ptr	162
set	500000



stp	10000
top	862

#### SPARSESTATE.\*

all	2042840
ini	-1
max	-1
mem	2042840
min	-1
ptr	-1
set	-1
stp	-1
top	-1

#### STRINGSTATE.\*

all	2400000
ini	2142917
max	2097152
mem	150000
min	150000
ptr	55470
set	500000
stp	100000
top	55470

#### TEXSTATE.\*

approximate	36293209
-------------	----------

#### TOKENSTATE.\*

all	8000000
ini	501468
max	10000000
mem	1000000
min	1000000
ptr	1275274
set	10000000
stp	250000
top	692643

#### WARNINGSTATE.\*

warning	unset
warningtag	unset

There are also getters for the subtables. The whole repertoire of functions in the status table is: `getbufferstate`, `getcallbackstate`, `getconstants`, `geterrorlinestate`,



geterrorstate, getexpandstate, getextrastate, getfilestate, getfontstate, gethalferrorlinestate, gethashstate, getinputstate, getinsertstate, getlanguagesstate, getlookupstate, getluastate, getmarkstate, getneststate, getnodestate, getparameterstate, getpoolstate, getreadstate, getsavestate, getsparsestate, getstringstate, gettexstate, gettokenstate, getwarningstate, iocodes, list, resetmessages. The error and warning messages can be wiped with the resetmessages function. The states in subtables relate to memory management and are mostly there for development purposes.

The getconstants query gives back a table with all kind of internal quantities and again these are only relevant for diagnostic and development purposes. Many are good old T<sub>E</sub>X constants that are describes in the original documentation of the source but some are definitely LuaMetaT<sub>E</sub>X specific.

#### CONSTANTS.\*

assumed_math_control	455550
awful_bad	1073741823
decent_criterium	12
default_catcode_table	-1
default_deadcycles	25
default_eqno_gap_step	1000
default_hangafter	1
default_output_box	255
default_pre_display_gap	2000
default_rule	26214
default_space_factor	1000
default_tolerance	10000
deplorable	100000
eject_penalty	-10000
ignore_depth	-65536000
infinite_bad	10000
infinite_penalty	10000
infinity	2147483647
large_width_excess	7230584
loose_criterium	99
math_begin_class	62
math_end_class	63
max_attribute_register_index	65535
max_box_register_index	65535
max_bytecode_index	65535
max_cardinal	4294967295
max_category_code	15
max_char_code	15
max_character_code	1114111
max_data_value	2097151
max_dimen	1073741823
max_dimen_register_index	65535



max_function_reference	2097151
max_glue_register_index	65535
max_half_value	32767
max_halfword	1073741823
max_int_register_index	65535
max_integer	2147483647
max_mark_index	9999
max_math_class_code	63
max_math_family_index	63
max_mu_glue_register_index	65535
max_n_of_bytecodes	65536
max_n_of_catcode_tables	256
max_n_of_fonts	100000
max_n_of_languages	10000
max_n_of_marks	10000
max_n_of_math_families	64
max_newline_character	127
max_quarterword	65535
max_size_of_word	1024
max_space_factor	32767
max_toks_register_index	65535
min_cardinal	0
min_data_value	0
min_dimen	-1073741823
min_halfword	-1073741823
min_infinity	-2147483647
min_integer	-2147483647
min_quarterword	0
min_space_factor	0
no_catcode_table	-2
null	0
null_flag	-1073741824
null_font	0
one_bp	65781
preset_rule_thickness	1073741824
running_rule	-1073741824
small_stretchability	1663497
tex_eqtb_size	590702
tex_hash_prime	131041
tex_hash_size	131072
two	131072
unity	65536
unused_attribute_value	-2147483647
unused_math_family	255
unused_math_style	255
unused_script_value	0



<code>unused_state_value</code>	0
<code>zero_glue</code>	0

---

Most variables speak for themselves, some are more obscure. For instance the `run_state` variable indicates what the engine is doing:

N	meaning	explanation
0	initializing	--ini mnode
1	updating	relates to <code>\overloadmode</code>
2	production	a regular (format driven) run

---

## 12.3 The tex library

### 12.3.1 Introduction

The `tex` table contains a large list of virtual internal  $\TeX$  parameters that are partially writable. The designation ‘virtual’ means that these items are not properly defined in Lua, but are only frontends that are handled by a metatable that operates on the actual  $\TeX$  values. As a result, most of the Lua table operators (like `pairs` and `#`) do not work on such items.

At the moment, it is possible to access almost every parameter that you can use after `\the`, is a single token or is sort of special in  $\TeX$ . This excludes parameters that need extra arguments, like `\the\scriptfont`. The subset comprising simple integer and dimension registers are writable as well as readable (like `\tracingcommands` and `\parindent`).

### 12.3.2 Internal parameter values, set and get

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 ([ "global", ] <string> n, ...)
... = tex.get (<string> n)
```

Glue is kind of special because there are five values involved. The return value is a `glue_spec` node but when you pass `false` as last argument to `tex.get` you get the width of the glue and when you pass `true` you get all five values. Otherwise you get a node which is a copy of the internal value so you are responsible for its freeing at the Lua end. When you set a glue quantity you can either pass a `glue_spec` or upto five numbers.

Beware: as with regular Lua tables you can add values to the `tex` table. So, the following is valid:

```
tex.foo = 123
```



When you access a  $\text{\TeX}$  parameter a look up takes place. For read-only variables that means that you will get something back, but when you set them you create a new entry in the table thereby making the original invisible.

There are a few special cases that we make an exception for: `prevdepth`, `prevgraf` and `spacefactor`. These normally are accessed via the `tex.nest` table:

```
tex.nest[tex.nest.ptr].prevdepth    = p
tex.nest[tex.nest.ptr].spacefactor = s
```

However, the following also works:

```
tex.prevdepth    = p
tex.spacefactor = s
```

Keep in mind that when you mess with node lists directly at the Lua end you might need to update the top of the nesting stack's `prevdepth` explicitly as there is no way  $\text{\LuaTeX}$  can guess your intentions. By using the accessor in the `tex` tables, you get and set the values at the top of the nesting stack.

### 12.3.2.1 Integer parameters

The integer parameters accept and return Lua integers. In some cases the values are checked, trigger other settings or result in some immediate change of behaviour: `adjdemerits`, `adjustspacing`, `adjustspacingshrink`, `adjustspacingstep`, `adjustspacingstretch`, `automatichyphenpenalty`, `automigrationmode`, `autoparagraphmode`, `binoppenalty`, `brokenpenalty`, `catcodetable`, `clubpenalty`, `day`, `defaultshyphenchar`, `defaultskewchar`, `delimiterfactor`, `displaywidowpenalty`, `doublehyphendemerits`, `endlinechar`, `errorcontextlines`, `escapechar`, `exceptionpenalty`, `exhyphenchar`, `exhyphenpenalty`, `explicitshyphenpenalty`, `fam`, `finalhyphendemerits`, `firstvalidlanguage`, `floatingpenalty`, `globaldefs`, `glyphdatafield`, `glyphoptions`, `glyphscale`, `glyphscriptfield`, `glyphscriptscale`, `glyphscriptscriptscale`, `glyphstatefield`, `glyphtextscale`, `glyphxscale`, `glyphyscale`, `hangafter`, `hbadness`, `holdinginserts`, `hyphenationmode`, `hyphenpenalty`, `interlinepenalty`, `language`, `lastlinefit`, `lefthyphenmin`, `linedirection`, `linepenalty`, `localbrokenpenalty`, `localinterlinepenalty`, `looseness`, `luacopyinputnodes`, `mathdelimitersmode`, `mathdirection`, `mathdisplayskipmode`, `matheqnogapstep`, `mathfencesmode`, `mathfontcontrol`, `mathlimitsmode`, `mathnolimitsmode`, `mathpenaltiesmode`, `mathrulesfam`, `mathrulesmode`, `mathscriptboxmode`, `mathscriptcharmode`, `mathscriptsmode`, `mathslackmode`, `mathspacingmode`, `mathsurroundmode`, `maxdeadcycles`, `month`, `newlinechar`, `normalizelinemode`, `nospaces`, `orphanpenalty`, `outputbox`, `outputpenalty`, `overloadmode`, `pageboundarypenalty`, `pardirection`, `pausing`, `postdisplaypenalty`, `postinlinepenalty`, `prebinoppenalty`, `predisplaydirection`, `predisplaygapfactor`, `predisplaypenalty`, `preinlinepenalty`, `prerelpenalty`, `pretolerance`, `protrudechars`, `relpenalty`, `rightshyphenmin`, `savinghyphcodes`, `savingvdiscards`, `setfontid`, `setlanguage`, `shapingpenaltiesmode`, `shapingpenalty`, `showboxbreadth`, `showboxdepth`, `shownodedetails`, `supmarkmode`, `textdirection`, `time`, `tolerance`, `tracingadjusts`, `tracingalignments`, `tracingassigns`, `tracingcommands`, `tracingexpressions`, `tracingfonts`, `tracingfullboxes`, `tracinggroups`, `tracinghyphenation`, `tracinggifts`, `tracinginserts`, `tracinglevels`, `tracinglostchars`, `tracingmacros`, `tracingmarks`, `tracingmath`,





tracingnesting, tracingnodes, tracingonline, tracingoutput, tracingpages, tracing-paragraphs, tracingrestores, tracingstats, uchyph, vbadness, widowpenalty, year.

Some integer parameters are read only, because they are actually referring not to some internal integer register but to an engine property: deadcycles, insertpenalties, parshape, interlinepenalties, clubpenalties, widowpenalties, displaywidowpenalties, prevgraf and spacefactor.

#### 12.3.2.2 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. These are read-write: box-maxdepth, delimitershortfall, displayindent, displaywidth, emergencystretch, glyphx-offset, glyphyoffset, hangindent, hfuzz, hsize, lineskiplimit, mathsurround, maxdepth, nulldelimiterspace, overfullrule, parindent, predisplaysize, pxdimen, scriptspace, splitmaxdepth, tabsize, vfuzz, vsize.

These are read-only: pagedepth, pagefilllstretch, pagefillstretch, pagefilstretch, pagegoal, pageshrink, pagestretch and pagetotal.

#### 12.3.2.3 Direction parameters

The direction states can be queried with: `gettextdir`, `getlinedir`, `getmathdir` and `getpardir`. You can set them with `settextdir`, `setlinedir`, `setmathdir` and `setpardir`, commands that accept a number. You can also set these parameters as table key/values: `textdirection`, `linedirection`, `mathdirection` and `pardirection`, so the next code sets the text direction to `r2l`:

```
tex.textdirection = 1
```

#### 12.3.2.4 Glue parameters

The internal glue parameters accept and return a userdata object that represents a `glue_spec` node: `abovedisplayshortskip`, `abovedisplayskip`, `baselineskip`, `belowdisplayshortskip`, `belowdisplayskip`, `leftskip`, `lineskip`, `mathsurroundskip`, `maththreshold`, `parfillleftskip`, `parfillskip`, `parskip`, `rightskip`, `spaceskip`, `splittopskip`, `tabskip`, `topskip`, `xspaceskip`.

#### 12.3.2.5 Muglue parameters

All muglue parameters are to be used read-only and return a Lua string `medmuskip`, `thickmuskip`, `thinmuskip`, `tinymuskip`.

#### 12.3.2.6 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. Here is the lot: `errhelp`, `everybeforepar`, `everycr`, `everydisplay`, `everyeof`, `everyhbox`, `everyjob`, `everymath`, `everypar`, `everytab`, `everyvbox`, `output`.



### 12.3.3 Convert commands

All ‘convert’ commands are read-only and return a Lua string. The supported commands at this moment are: `Uchar`, `csstring`, `directlua`, `expanded`, `fontname`, `fontspecifiedname`, `formatname`, `jobname`, `luabytecode`, `luaescapestring`, `luafunction`, `luatexbanner`, `meaning`, `meaningasis`, `meaningfull`, `meaningless`, `number`, `romannumeral`, `semiexpanded`, `string`, `todimension`, `tointeger`, `tomathstyle`, `toscaled`. You will get an error message if an operation is not (yet) permitted. Some take a string or number argument, just like at the  $\TeX$  end some extra input is expected.

### 12.3.4 Item commands

All so called ‘item’ commands are read-only and return a number. The complete list of these commands is: `Umathcharclass`, `Umathcharfam`, `Umathcharslot`, `badness`, `currentgrouplevel`, `currentgrouptype`, `currentifbranch`, `currentiflevel`, `currentifttype`, `currentloopiterator`, `currentloopnesting`, `dimexpr`, `dimexpression`, `fontchardp`, `fontcharht`, `fontcharic`, `fontcharwd`, `fontid`, `fontmathcontrol`, `fontspecid`, `fontspecifiedsize`, `fontspecscale`, `fontspecxscale`, `fontspecyscale`, `fonttextcontrol`, `glueexpr`, `glueshrink`, `glueshrinkorder`, `gluestretch`, `gluestretchorder`, `gluetomu`, `inputlineno`, `insertprogress`, `lastarguments`, `lastchkdim`, `lastchknum`, `lastkern`, `lastleftclass`, `lastloopiterator`, `lastnodesubtype`, `lastnodetype`, `lastparcontext`, `lastpenalty`, `lastrightclass`, `lastskip`, `leftmarginkern`, `luatexrevision`, `luatexversion`, `mathscale`, `mathstackstyle`, `mathstyle`, `mathstylefontid`, `muexpr`, `mutoglua`, `numericsscale`, `numexpr`, `numexpression`, `overshoot`, `parametercount`, `parshapedimen`, `parshapeindent`, `parshapelength`, `rightmarginkern`, `scaledemwidth`, `scaledexheight`, `scaledextraspaces`, `scaledinterwordshrink`, `scaledinterwordspace`, `scaledinterwordstretch`, `scaledslantperpoint`. No all are currently supported but eventually that might be the case. Like the lists in previous sections, there are differences between  $\text{Lua}\TeX$  and  $\text{LuaMeta}\TeX$ , where some commands are organized differently in order to provide a consistent Lua interface.

### 12.3.5 Accessing registers: `set*`, `get*` and `is*`

$\TeX$ 's attributes (`\attribute`), counters (`\count`), dimensions (`\dimen`), skips (`\skip`, `\muskip`) and token (`\toks`) registers can be accessed and written to using two times five virtual sub-tables of the `tex` table:

<code>tex.attribute</code>	<code>tex.skip</code>	<code>tex.muglua</code>
<code>tex.count</code>	<code>tex.glue</code>	<code>tex.toks</code>
<code>tex.dimen</code>	<code>tex.muskip</code>	

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, Lua<sub>T</sub><sub>E</sub>X 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 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).
- ▶ The glue registers are just skip registers but instead of userdata are verbose.
- ▶ Like the counts, the attribute registers accept and return Lua numbers.

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 ([ "global", ] <number> n, <node> s)
tex.setskip ([ "global", ] <string> s, <node> s)
<node> s = tex.getskip (<number> n)
<node> s = tex.getskip (<string> s)
```

We have similar setters for `count`, `dimen`, `muskip`, and `toks`. Counters and `dimen` are represented by numbers, `skips` and `muskip`s by nodes, and `toks` by strings.

Again the glue variants are not using the `glue-spec` userdata nodes. The `setglue` function accepts upto five arguments: `width`, `stretch`, `shrink`, `stretch_order` and `shrink_order`. Non-numeric values set the property to zero. The `getglue` function reports all five properties, unless the second argument is false in which case only the width is returned.

Here is an example using a threesome:

```
local d = tex.getdimen("foo")
if tex.isdimen("oof") then
    tex.setdimen("oof",d)
end
```

There are six extra skip (glue) related helpers:

```
tex.setglue ([ "global", ] <number> n,
    width, stretch, shrink, stretch_order, shrink_order)
tex.setglue ([ "global", ] <string> s,
    width, stretch, shrink, stretch_order, shrink_order)
width, stretch, shrink, stretch_order, shrink_order =
    tex.getglue (<number> n)
width, stretch, shrink, stretch_order, shrink_order =
    tex.getglue (<string> s)
```



The other two are `tex.setmuglue` and `tex.getmuglue`.

There are such helpers for `dimen`, `count`, `skip`, `muskip`, `box` and `attribute` registers but the glue ones are special because they have to deal with more properties.

As with the general `get` and `set` function discussed before, for the `skip` registers `getskip` returns a node and `getglue` returns numbers, while `setskip` accepts a node and `setglue` expects upto 5 numbers. Again, when you pass `false` as second argument to `getglue` you only get the width returned. The same is true for the `mu` variants `getmuskip`, `setmuskip`, `getmuskip` and `setmuskip`.

For `tokens` registers we have an alternative where a `catcode` table is specified:

```
tex.scantoks(0,3,"$e=mc^2$")
tex.scantoks("global",0,3,"$\int\limits^1_2$")
```

In the function-based interface, it is possible to define values globally by using the string `global` as the first function argument.

There is a dedicated getter for marks: `getmark` that takes two arguments. The first argument is one of `top`, `bottom`, `first`, `splitbottom` or `splitfirst`, and the second argument is a marks class number. When no arguments are given the current maximum number of classes is returned.

When `tex.gettoks` gets an extra argument `true` it will return a table with `userdata` tokens.

### 12.3.6 Character code registers: `[get|set]*code[s]`

$\TeX$ 's character code tables (`\lccode`, `\uccode`, `\sfcode`, `\catcode`, `\mathcode`, `\delcode`) can be accessed and written to using six virtual subtables of the `tex` table

<code>tex.lccode</code>	<code>tex.sfcode</code>	<code>tex.mathcode</code>
<code>tex.uccode</code>	<code>tex.catcode</code>	<code>tex.delcode</code>

The function call interfaces are roughly as above, but there are a few twists. `sfcodes` are the simple ones:

```
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> class,
    <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
}

```

You can also avoid the table:

```

tex.setmathcode (["global"], <number> n, <number> class,
    <number> family, <number> character)
class, family, char =
    tex.getmathcodes (<number> n)
tex.setdelcode (["global"], <number> n, <number> smallfam,
    <number> smallchar, <number> largefam, <number> largechar)
smallfam, smallchar, largefam, largechar =
    tex.getdelcodes (<number> n)

```

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`.

### 12.3.7 Box registers: `[get|set]box`

It is possible to set and query actual boxes, coming for instance from `\hbox`, `\vbox` or `\vtop`, using the node interface as defined in the node library:



`tex.box`

for array access, or

```
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])
```

### 12.3.8 triggerbuildpage

You should not expect too much from the `triggerbuildpage` helpers because often  $\TeX$  doesn't do much if it thinks nothing has to be done, but it might be useful for some applications. It just does as it says it calls the internal function that build a page, given that there is something to build.

### 12.3.9 splitbox

You can split a box:

```
local vlist = tex.splitbox(n,height,mode)
```

The remainder is kept in the original box and a packaged vlist is returned. This operation is comparable to the `\vsplit` operation. The mode can be `additional` or `exactly` and concerns the split off box.

### 12.3.10 Accessing math parameters: [get|set]math

It is possible to set and query the internal math parameters using:

```
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 are:

display	crampeddisplay
text	crampedtext
script	crampedscript
scriptscript	crampedscriptscript

The value is either a number (representing a dimension or number) or a glue spec node representing a muskip for `ordordspacing` and similar spacing parameters.

### 12.3.11 Special list heads: `[get|set]list`

The virtual table `tex.lists` contains the set of internal registers that keep track of building page lists.



FIELD	EXPLANATION
pageinserthead	circular list of pending insertions
contributehead	the recent contributions
pagehead	the current page content
holdhead	used for held-over items for next page
postadjusthead	head of the (pending) post adjustments
preadjusthead	head of the (pending) pre adjustments
postmigratehead	head of the (pending) post migrations
premigratehead	head of the (pending) pre migrations
pagediscardshead	head of the discarded items of a page break
splitdiscardshead	head of the discarded items in a vsplit

The getter and setter functions are `getlist` and `setlist`. You have to be careful with what you set as  $\TeX$  can have expectations with regards to how a list is constructed or in what state it is.

### 12.3.12 Semantic nest levels: `getnest` and `ptr`

The virtual table `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 `ptr`, which gives the highest available index. Neither the array items in `nest[]` nor `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, `nest[0]` the outermost (main vertical list) level. The getter function is `getnest`. You can pass a number (which gives you a list), nothing or `top`, which returns the topmost list, or the string `ptr` which gives you the index of the topmost list.

The known fields are:

KEY	TYPE	MODES	EXPLANATION
mode	number	all	the meaning of these numbers depends on the engine and sometimes even the version; you can use <code>tex.getmodevalues()</code> to get the mapping: positive values signal vertical, horizontal and math mode, while negative values indicate inner and inline variants
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
spacefactor	number	hmode	the current space factor
direction	node	hmode	stack used for temporary storage by the line break algorithm
noad	node	mmode	used for temporary storage of a pending fraction numerator, for <code>\over</code> etc.
delimiter	node	mmode	used for temporary storage of the previous math delimiter, for <code>\middle</code>





<code>mathdir</code>	boolean	mmode	true when during math processing the <code>\mathdirection</code> is not the same as the surrounding <code>\texdirection</code>
<code>mathstyle</code>	number	mmode	the current <code>\mathstyle</code>

---

When a second string argument is given to the `getnest`, the value with that name is returned. Of course the level must be valid. When `setnest` gets a third argument that value is assigned to the field given as second argument.

### 12.3.13 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.

#### 12.3.13.1 `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 `\newlinechar` appended, all others do.

#### 12.3.13.2 `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 `\newlinechar` 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(" in between")}after
```

the space before `in between` 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`. This influences the string arguments (or numbers turned into strings).

Although this needs to be used with care, you can also pass token or node userdata objects. These get injected into the stream. Tokens had best be valid tokens, while nodes need to be around when they get injected. Therefore it is important to realize the following:

- ▶ When you inject a token, you need to pass a valid token userdata object. This object will be collected by Lua when it no longer is referenced. When it gets printed to  $\TeX$  the token itself gets copied so there is no interference with the Lua garbage collection. You manage the object yourself. Because tokens are actually just numbers, there is no real extra overhead at the  $\TeX$  end.
- ▶ When you inject a node, you need to pass a valid node userdata object. The node related to the object will not be collected by Lua when it no longer is referenced. It lives on at the  $\TeX$  end in its own memory space. When it gets printed to  $\TeX$  the node reference is used assuming that node stays around. There is no Lua garbage collection involved. Again, you manage the object yourself. The node itself is freed when  $\TeX$  is done with it.

If you consider the last remark you might realize that we have a problem when a printed mix of strings, tokens and nodes is reused. Inside  $\TeX$  the sequence becomes a linked list of input buffers. So, `"123"` or `"\foo{123}"` gets read and parsed on the fly, while `<token userdata>` already is tokenized and effectively is a token list now. A `<node userdata>` is also tokenized into a token list but it has a reference to a real node. Normally this goes fine. But now assume that you store the whole lot in a macro: in that case the tokenized node can be flushed many times. But, after the first such flush the node is used and its memory freed. You can prevent this by using copies which is controlled by setting `\luacopyinputnodes` to a non-zero value. This is one of these fuzzy areas you have to live with if you really mess with these low level issues.

### 12.3.13.3 `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.

### 12.3.13.4 `cprint`

This function takes a number indicating the to be used catcode, plus either a table of strings or an argument list of strings that will be pushed into the input stream.

```
tex.cprint( 1," 1: ${\\foo}") tex.print("\\par") -- a lot of \bgroup s
tex.cprint( 2," 2: ${\\foo}") tex.print("\\par") -- matching \egroup s
tex.cprint( 9," 9: ${\\foo}") tex.print("\\par") -- all get ignored
```



```
tex.cprint(10,"10: ${\\foo}") tex.print("\\par") -- all become spaces
tex.cprint(11,"11: ${\\foo}") tex.print("\\par") -- letters
tex.cprint(12,"12: ${\\foo}") tex.print("\\par") -- other characters
tex.cprint(14,"12: ${\\foo}") tex.print("\\par") -- comment triggers
```

### 12.3.13.5 write

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

Each string argument is treated by T<sub>E</sub>X 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).

## 12.3.14 Helper functions

### 12.3.14.1 round

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

Rounds Lua number `o`, and returns a number that is in the range of a valid T<sub>E</sub>X register value. If the number starts out of range, it generates a ‘number too big’ error as well.

### 12.3.14.2 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 T<sub>E</sub>X 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 too 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 LuaT<sub>E</sub>X's internal, 100% portable scale function will be added at a later date.

### 12.3.14.3 number and romannumeral

These are the companions to the primitives `\number` and `\romannumeral`. They can be used like:

```
tex.print(tex.romannumeral(123))
```

### 12.3.14.4 fontidentifier and fontname

The first one returns the name only, the second one reports the size too.



```
tex.print(tex.fontname(tex.fontname))
tex.print(tex.fontname(tex.fontidentidier))
```

#### 12.3.14.5 `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)

#### 12.3.14.6 `tex.getlinenumber` and `tex.setlinenumber`

You can mess with the current line number:

```
local n = tex.getlinenumber()
tex.setlinenumber(n+10)
```

which can be shortcut to:

```
tex.setlinenumber(10,true)
```

This might be handy when you have a callback that reads numbers from a file and combines them in one line (in which case an error message probably has to refer to the original line). Interference with TeX's internal handling of numbers is of course possible.

#### 12.3.14.7 `error`, `show_context` and `gethelptext`

```
tex.error(<string> s)
tex.error(<string> s, <table> help)
<string> s = tex.gethelptext()
```

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.

In case of an error the `show_context` function will show the current context where we're at (in the expansion).

#### 12.3.14.8 `getfamilyoffont`

When you pass a proper family identifier the next helper will return the font currently associated with it.



```
<integer> id = font.getfamilyoffont(<integer> fam)
```

#### 12.3.14.9 [set|get]interaction

The engine can be in one of four modes:

VALUE	mode	MEANING
0	batch	omits all stops and omits terminal output
1	nonstop	omits all stops
2	scroll	omits error stops
3	errorstop	stops at every opportunity to interact

The mode can be queried and set with:

```
<integer> i = tex.getinteraction()  
tex.setinteraction(<integer> i)
```

#### 12.3.14.10 runtoks and quittoks

Because of the fact that  $\text{\TeX}$  is in a complex dance of expanding, dealing with fonts, typesetting paragraphs, messing around with boxes, building pages, and so on, you cannot easily run a nested  $\text{\TeX}$  run (read nested main loop). However, there is an option to force a local run with `runtoks`. The content of the given token list register gets expanded locally after which we return to where we triggered this expansion, at the Lua end. Instead a function can get passed that does some work. You have to make sure that at the end  $\text{\TeX}$  is in a sane state and this is not always trivial. A more complex mechanism would complicate  $\text{\TeX}$  itself (and probably also harm performance) so this simple local expansion loop has to do.

```
tex.runtoks(<token register>)  
tex.runtoks(<lua function>)  
tex.runtoks(<macro name>)  
tex.runtoks(<register name>)
```

When the `\tracingnesting` parameter is set to a value larger than 2 some information is reported about the state of the local loop. The return value indicates an error:

VALUE	meaning
0	no error
1	bad register number
2	unknown macro or register name
3	macro is unsuitable for runtoks (has arguments)

This function has two optional arguments in case a token register is passed:

```
tex.runtoks(<token register>, force, grouped, obeymode)
```

Inside for instance an `\edef` the `runtoks` function behaves (at least tries to) like it were an `\the`. This prevents unwanted side effects: normally in such a definition tokens remain tokens and (for instance) characters don't become nodes. With the second argument you can force the local



main loop, no matter what. The third argument adds a level of grouping. The last argument tells the scanner to stay in the current mode.

You can quit the local loop with `\endlocalcontrol` or from the Lua end with `tex.quittoks`. In that case you end one level up! Of course in the end that can mean that you arrive at the main level in which case an extra end will trigger a redundancy warning (not an abort!).

#### 12.3.14.11 **forcehmode**

An example of a (possible error triggering) complication is that  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  expects to be in some state, say horizontal mode, and you have to make sure it is when you start feeding back something from Lua into  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ . Normally a user will not run into issues but when you start writing tokens or nodes or have a nested run there can be situations that you need to run `forcehmode`. There is no recipe for this and intercepting possible cases would weaken  $\mathrm{LuaT}_{\mathrm{E}}\mathrm{X}$ 's flexibility.

#### 12.3.14.12 **hashtokens**

```
for i,v in pairs (tex.hashtokens()) do ... end
```

Returns a list of names. 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. You can use `token.create` to inspect properties, for instance when the command key in a created table equals 123, you have the `cmdname` value `undefined_cs`.

#### 12.3.14.13 **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).

### 12.3.15 **Functions for dealing with primitives**

#### 12.3.15.1 **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  $\mathrm{T}_{\mathrm{E}}\mathrm{X}82$ , 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.

When Lua<sub>T</sub><sub>E</sub>X is run with `--ini` only the T<sub>E</sub>X82 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 T<sub>E</sub>X82-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.

### 12.3.15.2 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 given in the following table. In addition the somewhat special primitives `'\ '`, `'\/'` and `'-'` are defined.

NAME	VALUES
tex	above abovedisplayshortskip abovedisplayskip abovewithdelims accent adjde- merits advance afterassignment aftergroup aligncontent atop atopwithdelims badness baselineskip batchmode begingroup belowdisplayshortskip belowdis- playskip binoppenalty botmark box boxmaxdepth brokenpenalty catcode char chardef cleaders clubpenalty copy count countdef cr crcr csname day dead- cycles def defaultthyphenchar defaultskewchar delcode delimiter delimiter- factor delimitershortfall dimen dimendef discretionary displayindent dis- playlimits displaystyle displaywidowpenalty displaywidth divide double- hyphendemerits dp dump edef else emergencystretch end endcsname endgroup endinput endlineschar eqno errhelp errmessage errorcontextlines errorstop- mode escapechar everycr everydisplay everyhbox everyjob everymath everypar everyvbox exhyphenchar exhyphenpenalty expandafter fam fi finalhyphende- merits firstmark floatingpenalty font fontdimen fontname fontspecifiedname futurelet gdef global globaldefs glyph halign hangafter hangindent hbad- ness hbox hfil hfill hfilneg hfuzz holdinginserts hrule hsize hskip hss ht hyphenation hyphenchar hyphenpenalty if ifcase ifcat ifdim iffalse ifhbox ifhmode ifinner ifmmode ifnum ifodd iftrue ifvbox ifvmode ifvoid ifx ignore-



spaces indent input inputlineno insert insertpenalties interlinepenalty  
 jobname kern language lastbox lastkern lastpenalty lastskip lccode lead-  
 ers left lefthyphenmin leftskip leqno let limits linepenalty lineskip line-  
 skiplimit long looseness lower lowercase mark mathbin mathchar mathchardef  
 mathchoice mathclose mathcode mathinner mathop mathopen mathord mathpunct  
 mathrel mathsurround maxdeadcycles maxdepth meaning meaningasis meaning-  
 full meaningless medmuskip message middle mkern month moveleft moveright  
 mskip multiply muskip muskipdef newlinechar noalign noexpand noindent no-  
 limits nonscript nonstopmode nulldelimiterspace nullfont number omit or  
 outer output outputpenalty over overfullrule overline overshoot overwith-  
 delims pagedepth pagefillllstretch pagefillstretch pagefilstretch page-  
 goal pageshrink pagestretch pagetotal par parfillleftskip parfillskip  
 parindent parshape parskip patterns pausing penalty postdisplaypenalty pre-  
 displaypenalty predisplaysize pretolerance prevdepth prevgraf radical raise  
 relax relpenalty right righthyphenmin rightskip romannumeral scaledfontdi-  
 men scriptfont scriptscriptfont scriptscriptstyle scriptspace scriptstyle  
 scrollmode setbox setlanguage sfcode shipout show showbox showboxbreadth  
 showboxdepth showlists shownodedetails showthe skewchar skip skipdef space-  
 factor spaceskip span splitbotmark splitfirstmark splitmaxdepth splittop-  
 skip srule string tabskip textfont textstyle the thickmuskip thinmuskip time  
 tinymuskip toks toksdef tolerance topmark topskip tracingcommands tracin-  
 glostchars tracingmacros tracingonline tracingoutput tracingpages tracing-  
 paragraphs tracingrestores tracingstats uccode uchyph underline unhbox un-  
 hcopy unhpack unkern unpenalty unskip unvbox unvcopy unvpack uppercase vad-  
 just valign vbadness vbox vcenter vfil vfill vfilneg vfuzz vrule vsize vskip  
 vsplit vss vtop wd widowpenalty xdef xleaders xspaceskip year

core

etex

botmarks clubpenalties currentgrouplevel currentgrouptype currentifbranch  
 currentiflevel currentifttype detokenize dimexpr displaywidowpenalties  
 everyeof firstmarks fontchardp fontcharht fontcharic fontcharwd glueexpr  
 glueshrink glueshrinkorder gluestretch gluestretchorder gluetomu ifc-  
 sname ifdefined iffontchar interactionmode interlinepenalties lastline-  
 fit lastnodetype marks muexpr mutogluue numexpr pagediscards parshapedimen  
 parshapeindent parshapeheight predisplaydirection protected savinghyph-  
 codes savingvdiscards scantokens showgroups showifs showtokens splitbot-  
 marks splitdiscards splitfirstmarks topmarks tracingassigns tracinggroups  
 tracingifs tracinglevels tracingnesting unexpanded unless widowpenalties

luatex

Uabove Uabovewithdelims Uatop Uatopwithdelims Uchar Udelcode Udelcodenum  
 Udelimiter Udelimiterover Udelimiterunder Uhextensible Uleft Umathaccent  
 Umathaccentbasedepth Umathaccentbaseheight Umathaccentbottomshiftup  
 Umathaccenttopshiftup Umathaccentvariant Umathadapttoleft Umathadapt-  
 toright Umathaxis Umathbotaccentvariant Umathchar Umathcharclass Umath-  
 chardef Umathcharfam Umathcharnum Umathcharnumdef Umathcharslot Umathclass  
 Umathcode Umathcodenum Umathconnectoroverlapmin Umathdegreevariant Umath-  
 delimiterovervariant Umathdelimiterundervariant Umathdenominatorvariant  
 Umathextrasubpreshift Umathextrasubprespace Umathextrasubshift Umathextra-





subspace Umatheextrasuppresshift Umatheextrasupprespace Umatheextrasupshift  
 Umatheextrasupspace Umathflattenedaccentbasedepth Umathflattenedaccent-  
 baseheight Umathflattenedaccentbottomshiftedown Umathflattenedaccenttop-  
 shiftup Umathfractiondelsize Umathfractiondenomdown Umathfractiondenomvgap  
 Umathfractionnumup Umathfractionnumvgap Umathfractionrule Umathfraction-  
 variant Umathhextensiblevariant Umathlimitabovebgap Umathlimitabovekern  
 Umathlimitabovevgap Umathlimitbelowbgap Umathlimitbelowkern Umathlimitbe-  
 lowvgap Umathlimits Umathnoaxis Umathnolimits Umathnolimitsubfactor Umath-  
 nolimitsupfactor Umathnumeratorvariant Umathopenupdepth Umathopenupheight  
 Umathoperatorsize Umathoverbarkern Umathoverbarrule Umathoverbarvgap  
 Umathoverdelimiterbgap Umathoverdelimitervariant Umathoverdelimitervgap  
 Umathoverlayaccentvariant Umathoverlinevariant Umathphantom Umathpresub-  
 shiftdistance Umathpresupshiftdistance Umathprimeraise Umathprimeshiftedrop  
 Umathprimeshiftup Umathprimespaceafter Umathprimevariant Umathprimewidth  
 Umathquad Umathradicaldegreeafter Umathradicaldegreebefore Umathradicalde-  
 greeraise Umathradicalkern Umathradicalrule Umathradicalvariant Umath-  
 radicalvgap Umathruleddepth Umathruleheight Umathskeweddelimitertertolerance  
 Umathskewedfractionhgap Umathskewedfractionvgap Umathspaceafterscript  
 Umathspacebeforescript Umathstackdenomdown Umathstacknumup Umathstackvari-  
 ant Umathstackvgap Umathsubscriptvariant Umathsubshiftdistance Umathsub-  
 shiftdown Umathsubshiftdrop Umathsubsupshiftdown Umathsubsupvgap Umath-  
 subtopmax Umathsupbottommin Umathsuperscriptvariant Umathsupshiftdistance  
 Umathsupshiftdrop Umathsupshiftup Umathsupsubbottommax Umathtopaccentvari-  
 ant Umathunderbarkern Umathunderbarrule Umathunderbarvgap Umathunderde-  
 limiterbgap Umathunderdelimitervariant Umathunderdelimitervgap Umathun-  
 derlinevariant Umathuseaxis Umathvextensiblevariant Umathvoid Umathxscale  
 Umathyscale Umiddle Unosubprescript Unosubscript Unosuperprescript Unosu-  
 perscript Uoperator Uover Uoverdelimiter Uoverwithdelims Uprimescript Urad-  
 ical Uright Uroot Ushiftedsubprescript Ushiftedsubscript Ushiftedsuperpre-  
 script Ushiftedsuperscript Uskewed Uskewedwithdelims Ustack Ustartdisplay-  
 math Ustartmath Ustopdisplaymath Ustopmath Ustyle Usubprescript Usubscript  
 Usuperprescript Usuperscript Uunderdelimiter Uvextensible adjustspacing  
 adjustspacingshrink adjustspacingstep adjustspacingstretch afterassigned  
 aftergrouped aliased alignmark aligntab allcrampedstyles alldisplaystyles  
 allmathstyles allscriptscriptstyles allscriptstyles allsplitstyles all-  
 textstyles alluncrampedstyles atendofgroup atendofgrouped attribute at-  
 tributedef automaticdiscretionary automatichyphenpenalty automigration-  
 mode autoparagraphmode beginscname beginlocalcontrol beginmathgroup be-  
 ginsimplegroup boundary boxanchor boxanchors boxattribute boxdirection  
 boxgeometry boxorientation boxshift boxsource boxtarget boxtotal boxxmove  
 boxxoffset boxymove boxyoffset catcodetable clearmarks copymathatomrule  
 copymathparent copymathspacing crampeddisplaystyle crampedscriptscript-  
 style crampedscriptstyle crampedtextstyle csstring currentloopiterator cur-  
 rentloopnesting currentmarks defcsname dimensiondef dimexpression directlua  
 edefcsname efcodes endlocalcontrol endmathgroup endsimplegroup enforced  
 etoks etoksapp etokspre everybeforepar everytab exceptionpenalty expand ex-



pandafterpars expandafterspaces expandcstoken expanded expandedafter expandedloop expandtoken explicitdiscretionary explicithyphenpenalty first-validlanguage flushmarks fontid fontmathcontrol fontspecdef fontspecid fontspecifiedsize fontspecscale fontspecxscale fontspecyscale fonttextcontrol formatname frozen futurecsname futuredef futureexpand futureexpandis futureexpandisap gdefcsname gleaders glet gletcsname glettonothing glue-specdef glyphdatafield glyphoptions glyphscale glyphscriptfield glyphscriptscale glyphscriptscriptscale glyphstatefield glyphtextscale glyphx-offset glyphxscale glyphyoffset glyphyscale gtoksapp gtokspre hcode hj-code hpack hyphenationmin hyphenationmode ifabsdim ifabsnum ifarguments ifboolean ifchkdim ifchknum ifcmpdim ifcmpnum ifcondition ifcstok ifdimexpression ifdimval ifempty ifflags ifhaschar ifhastok ifhastoks ifhasxtoks ifincsname ifinsert ifmathparameter ifmathstyle ifnumexpression ifnumval ifparameter ifparameters ifrelax iftok ignorearguments ignorepars immediate immutable inherited initcatcodetable insertbox insertcopy insertdepth insertdistance insertheight insertheights insertlimit insertmaxdepth insertmode insertmultiplier insertpenalty insertprogress insertstorage insertstoring insertunbox insertuncopy insertwidth instance integerdef lastarguments lastchkdim lastchknum lastleftclass lastloopiterator lastnamedcs lastnodesubtype lastparcontext lastrightclass leftmarginkern letcharcode letcsname letfrozen letmathatomrule letmathparent letmathspacing letprotected lettonothing linedirection localbrokenpenalty localcontrol localcontrolled localcontrolledloop localinterlinepenalty localleftbox localleftboxbox localmiddlebox localmiddleboxbox localrightbox localrightboxbox lp-code luabytecode luabytecodecall luacopyinputnodes luaescapestring luafunction luafunctioncall luatexbanner luatexrevision luatexversion mathaccent mathatom mathatomskip mathbackwardpenalties mathdelimitersmode mathdirection mathdisplayskipmode matheqnogapstep mathfenced mathfencesmode mathfontcontrol mathforwardpenalties mathfrac mathghost mathlimitsmode mathmiddle mathnolimitsmode mathpenaltiesmode mathrad mathrulesfam mathrulesmode mathscale mathscriptboxmode mathscriptcharmode mathscriptsmode mathslackmode mathspacingmode mathstackstyle mathstyle mathstylefontid mathsurroundmode mathsurroundskip maththreshold mugluespecdef mutable noaligned noboundary nohrule norelax normalizelinemode nospaces novrule numericscale numexpression orelse orphanpenalties orphanpenalty orunless outputbox overloaded overloadmode pageboundary pageboundarypenalty pagevsize parametercount parametermark parattribute pardirection permanent postexhyphenchar posthyphenchar postinlinepenalty prebinoppenalty predisplaygapfactor preexhyphenchar prehyphenchar preinlinepenalty prerelpenalty protrudechars protrusionboundary pxdimen quitloop quitvmode resetmathspacing retokenized rightmarginkern rcode savecatcodetable scaledemwidth scaledex-height scaledextraspaces scaledinterwordshrink scaledinterwordspace scaledinterwordstretch scaledslantperpoint scantextokens semiexpanded semiprotected setdefaultmathcodes setfontid setmathatomrule setmathignore setmathoptions setmathpostpenalty setmathprepenalty setmathspacing shapingpenaltiesmode shapingpenalty skewed skewedwithdelims snapshotpar supmark-



```

mode swapcsvalues tabsize textdirection thewithoutunit todimension tointe-
ger tokenized toksapp tokspre tolerant tomathstyle toscaled tpack tracingad-
justs tracingalignments tracingexpressions tracingfonts tracingfullboxes
tracinghyphenation tracinginserts tracingmarks tracingmath tracingnodes
uleaders undent unexpandedloop unletfrozen unletprotected untraced vpack
wordboundary wrapuppar xdefcsname xtoks xtoksapp xtokspre

```

---

Note that `luatex` does not contain `directlua`, as that is considered to be a core primitive, along with all the  $\text{\TeX}$ 82 primitives, so it is part of the list that is returned from 'core'.

Running `tex.extraprimitives` will give you the complete list of primitives -ini startup. It is exactly equivalent to `tex.extraprimitives("etex","luatex")`.

### 12.3.15.3 primitives

```
<table> t = tex.primitives()
```

This function returns a list of all primitives that  $\text{\TeX}$  knows about.

## 12.3.16 Core functionality interfaces

### 12.3.16.1 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  $100(t/s)^3$ ;

### 12.3.16.2 tex.resetparagraph

This function resets the parameters that  $\text{\TeX}$  normally resets when a new paragraph is seen.

### 12.3.16.3 linebreak

```

local <node> nodelist, <table> info =
    tex.linebreak(<node> listhead, <table> parameters)

```

The understood parameters are as follows:

NAME	TYPE	EXPLANATION
<code>pardir</code>	string	
<code>pretolerance</code>	number	
<code>tracingparagraphs</code>	number	
<code>tolerance</code>	number	
<code>looseness</code>	number	
<code>hyphenpenalty</code>	number	
<code>exhyphenpenalty</code>	number	
<code>pdfadjustspacing</code>	number	



adjdemerits	number	
protrudechars	number	
linepenalty	number	
lastlinefit	number	
doublehyphendemerits	number	
finalhyphendemerits	number	
hangafter	number	
interlinepenalty	number or table	if a table, then it is an array like <code>\interlinepenalties</code>
clubpenalty	number or table	if a table, then it is an array like <code>\clubpenalties</code>
widowpenalty	number or table	if a table, then it is an array like <code>\widowpenalties</code>
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	
parshape	table	

Note that there is no interface for `\displaywidowpenalties`, you have to pass the right choice for `widowpenalties` yourself.

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 `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:

NAME	EXPLANATION
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.



#### 12.3.16.4 shipout

`tex.shipout(<number> n)`

Ships out box number `n` to the output file, and clears the box register.

#### 12.3.16.5 getpagestate

This helper reports the current page state: `empty`, `box_there` or `inserts_only` as integer value.

#### 12.3.16.6 getlocallevel

This integer reports the current level of the local loop. It's only useful for debugging and the (relative state) numbers can change with the implementation.

### 12.3.17 Functions related to synctex

The next helpers only make sense when you implement your own synctex logic. Keep in mind that the library used in editors assumes a certain logic and is geared for plain and L<sup>A</sup>T<sub>E</sub>X, so after a decade users expect a certain behaviour.

NAME	EXPLANATION
<code>setsynctexmode</code>	0 is the default and used normal synctex logic, 1 uses the values set by the next helpers while 2 also sets these for glyph nodes; 3 sets glyphs and glue and 4 sets only glyphs
<code>setsynctextag</code>	set the current tag (file) value (obeys save stack)
<code>setsynctexline</code>	set the current line value (obeys save stack)
<code>setsynctexnofiles</code>	disable synctex file logging
<code>getsynctexmode</code>	returns the current mode (for values see above)
<code>getsynctextag</code>	get the currently set value of tag (file)
<code>getsynctexline</code>	get the currently set value of line
<code>forcesynctextag</code>	overload the tag (file) value (0 resets)
<code>forcesynctexline</code>	overload the line value (0 resets)

The last one is somewhat special. Due to the way files are registered in SyncT<sub>E</sub>X we need to explicitly disable that feature if we provide our own alternative if we want to avoid that overhead. Passing a value of 1 disables registering.

## 12.4 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. Watch out: some keys are different from LuaT<sub>E</sub>X, which is a side effect of a more granular and dynamic memory management.

KEY	TYPE	DEFAULT	COMMENT
<code>bufferize</code>	number/table	1000000	input buffer bytes



filesize	number/table	1000	max number of open files
fontsize	number/table	250	number of permitted fonts
hashsize	number/table	150000	number of hash entries
inputsize	number/table	10000	maximum input stack
languagesize	number/table	250	number of permitted languages
marksize	number/table	50	number of mark classes
nestsize	number/table	1000	max depth of nesting
nodesize	number/table	1000000	max node memory (various size)
parametersize	number/table	20000	max size of parameter stack
poolsize	number/table	10000000	max number of string bytes
savesize	number/table	100000	mas size of save stack
stringsize	number/table	150000	max number of strings
tokensize	number/table	1000000	max token memory
expandsize	number/table	10000	max expansion nesting
proptiessize	number	0	initial size of node properties table
functionsize	number	0	initial size of Lua functions table
errorlinesize	number	79	how much or an error is shown
halferrorlinesize	number	50	idem
formatname	string		
jobname	string		
starttime	number		for testing only
useutctime	number		for testing only
permitloadlib	number		for testing only

If no format name or jobname is given on the command line, the related keys will be tested first instead of simply quitting. The statistics library has methods for tracking down how much memory is available and has been configured. The size parameters take a number (for the maximum allocated size) or a table with three possible keys: `size`, `plus` (for extra size) and `step` for the increment when more memory is needed. They all start out with a hard coded minimum and also have an hard coded maximum, the the configured size sits somewhere between these.

## 12.5 The texio library

This library takes care of the low-level I/O interface: writing to the log file and/or console.

### 12.5.1 write and writeselector

```
texio.write(<string> target, <string> s, ...)
texio.write(<string> s, ...)
texio.writeselector(<string> s, ...)
```

Without the `target` argument, writes all given strings to the same location(s)  $\text{\TeX}$  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 `terminal`, `logfile` or `terminal_and_logfile`.



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.

## 12.5.2 `writeln` and `writeselectornl`

```
texio.writeln(<string> target, <string> s, ...)
texio.writeln(<string> s, ...)
texio.writeselectornl(<string> target, ...)
```

This function behaves like `texio.write`, but makes 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.

The selector variants always expect a selector, so there is no misunderstanding if `logfile` is a string or selector.

## 12.5.3 `setescape`

You can disable ^^ escaping of control characters by passing a value of zero.

## 12.5.4 `closeinput`

This function should be used with care. It acts as `\endinput` but at the Lua end. You can use it to (sort of) force a jump back to `TEX`. Normally a Lua call will just collect prints and at the end bump an input level and flush these prints. This function can help you stay at the current level but you need to know what you're doing (or more precise: what `TEX` is doing with input).

# 12.6 The token library

## 12.6.1 The scanner

The token library provides means to intercept the input and deal with it at the Lua level. The 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 okay so 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 scanner functions are probably the most intriguing.

FUNCTION	ARGUMENT	RESULT
<code>scankeyword</code>	string	returns true if the given keyword is gobbled; as with the regular <code>T<sub>E</sub>X</code> keyword scanner this is case insensitive (and ascii based)
<code>scankeywordcs</code>	string	returns true if the given keyword is gobbled; this variant is case sensitive and also suitable for utf8



<code>scanint</code>		returns an integer
<code>scanreal</code>		returns a number from e.g. 1, 1.1, .1 with optional collapsed signs
<code>scanfloat</code>		returns a number from e.g. 1, 1.1, .1, 1.1E10, , .1e-10 with optional collapsed signs
<code>scandimen</code>	infinity, mu-units	returns a number representing a dimension or two numbers being the filler and order
<code>scanglue</code>	mu-units	returns a glue spec node
<code>scantoks</code>	definer, expand	returns a table of tokens
<code>scancode</code>	bitset	returns a character if its category is in the given bitset (representing catcodes)
<code>scanstring</code>		returns a string given between {}, as <code>\macro</code> or as sequence of characters with catcode 11 or 12
<code>scanargument</code>		this one is similar to <code>scanstring</code> but also accepts a <code>\cs</code> (which then get expanded)
<code>scanword</code>		returns a sequence of characters with catcode 11 or 12 as string
<code>scancname</code>		returns <code>foo</code> after scanning <code>\foo</code>
<code>scanlist</code>		picks up a box specification and returns a <code>[h v]list</code> node

The integer, dimension and glue scanners take an extra optional argument that signals that an optional equal is permitted.

The scanners can be considered stable apart from the one scanning for a token. The `scancode` function takes an optional number, the `scankeyword` 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 `scanargument` function expands the given argument. When a braced argument is scanned, expansion can be prohibited by passing `false` (default is `true`). In case of a control sequence passing `false` will result in a one-level expansion (the meaning of the macro).

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\oof{oof}
\def\foo{foo-\oof}
```

we get:

NAME	RESULT	
<code>\directlua{token.scanstring()}{foo}</code>	<b>foo</b>	full expansion
<code>\directlua{token.scanstring()}foo</code>	<b>foo</b>	letters and others
<code>\directlua{token.scanstring()}\foo</code>	<b>foo-oof</b>	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 since there we do expand.

The scanner can be used to implement for instance a number scanner. An optional boolean argument can signal that a trailing space or `\relax` should be gobbled:

```
function token.scannumber(base)
    return tonumber(token.scanword(),base)
end
```

This scanner accepts any valid Lua number so it is a way to pick up floats in the input.

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 = token.scandimen()
        ...
    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  $\text{\TeX}$  to Lua we let Lua fetch from the input stream.



In the first case the input is tokenized and then turned into a string, then it is 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 T<sub>E</sub>X 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).

### 12.6.2 Picking up one token

The scanners look for a sequence. When you want to pick up one token from the input you use `scannext`. This creates a token with the (low level) properties as discussed next. This token is just the next one. If you want to enforce expansion first you can use `scantoken` or the `_expanded` variants. Internally tokens are characterized by a number that packs a lot of information. In order to access the bits of information a token is wrapped in a `userdata` object.

The `expand` function will trigger expansion of the next token in the input. This can be quite unpredictable but when you call it you probably know enough about T<sub>E</sub>X not to be too worried about that. It basically is a call to the internal `expand` related function.

NAME	EXPLANATION
<code>scannext</code>	get the next token
<code>scannextexpanded</code>	get the next expanded token
<code>skipnext</code>	skip the next token
<code>skipnextexpanded</code>	skip the next expanded token
<code>peeknext</code>	get the next token and put it back in the input
<code>peeknextexpanded</code>	get the next expanded token and put it back in the input

The `peek` function accept a boolean argument that triggers skipping spaces and alike.

### 12.6.3 Creating tokens

The creator function can be used as follows:

```
local t = token.create("relax")
```

This gives back a token object that has the properties of the `\relax` primitive. The possible properties of tokens are:

NAME	EXPLANATION
<code>command</code>	a number representing the internal command number
<code>cmdname</code>	the type of the command (for instance the <code>catcode</code> in case of a character or the classifier that determines the internal treatment)
<code>csname</code>	the associated control sequence (if applicable)
<code>id</code>	the unique id of the token
<code>tok</code>	the full token number as stored in T <sub>E</sub> X
<code>active</code>	a boolean indicating the active state of the token



<code>expandable</code>	a boolean indicating if the token (macro) is expandable
<code>protected</code>	a boolean indicating if the token (macro) is protected
<code>frozen</code>	a boolean indicating if the token is a frozen command
<code>user</code>	a boolean indicating if the token is a user defined command
<code>index</code>	a number that indicated the subcommand; differs per command

---

Alternatively you can use a getter `get<fieldname>` to access a property of a token.

The numbers that represent a catcode are the same as in  $\text{\TeX}$  itself, so using this information assumes that you know a bit about  $\text{\TeX}$ '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.

You can ask for a list of commands:

```
local t = token.commands()
```

The id of a token class can be queried as follows:

```
local id = token.command_id("math_shift")
```

If you really know what you're doing you can create character tokens by not passing a string but a number:

```
local letter_x = token.create(string.byte("x"))
local other_x = token.create(string.byte("x"), 12)
```

Passing weird numbers can give side effects so don't expect too much help with that. As said, you need to know what you're doing. The best way to explore the way these internals work is to just look at how primitives or macros or `\chardef`'d commands are tokenized. Just create a known one and inspect its fields. A variant that ignores the current catcode table is:

```
local whatever = token.new(123, 12)
```

You can test if a control sequence is defined with `is_defined`, which accepts a string and returns a boolean:

```
local okay = token.is_defined("foo")
```

The largest character possible is returned by `biggest_char`, just in case you need to know that boundary condition.

## 12.6.4 Macros

The `set_macro` function can get upto 4 arguments:

```
set_macro("csname", "content")
set_macro("csname", "content", "global")
set_macro("csname")
```

You can pass a catcodetable identifier as first argument:



```
set_macro(catcodetable,"csname","content")
set_macro(catcodetable,"csname","content","global")
set_macro(catcodetable,"csname")
```

The results are like:

```
\def\csname{content}
\gdef\csname{content}
\def\csname{}
```

The `getmacro` function can be used to get the content of a macro while the `getmeaning` function gives the meaning including the argument specification (as usual in  $\text{\TeX}$  separated by `->`).

The `set_char` function can be used to do a `\chardef` at the Lua end, where invalid assignments are silently ignored:

```
set_char("csname",number)
set_char("csname",number,"global")
```

A special one is the following:

```
set_lua("mycode",id)
set_lua("mycode",id,"global","protected")
```

This creates a token that refers to a Lua function with an entry in the table that you can access with `lua.getfunctions_table`. It is the companion to `\luaodef`. When the first (and only) argument is true the size will preset to the value of `texconfig.function_size`.

The `pushmacro` and `popmacro` function are very experimental and can be used to get and set an existing macro. The push call returns a user data object and the pop takes such a userdata object. These object have no accessors and are to be seen as abstractions.

### 12.6.5 Pushing back

There is a (for now) experimental putter:

```
local t1 = token.scannext()
local t2 = token.scannext()
local t3 = token.scannext()
local t4 = token.scannext()
-- watch out, we flush in sequence
token.putnext { t1, t2 }
-- but this one gets pushed in front
token.putnext ( t3, t4 )
```

When we scan `wxyz!` we get `yzwx!` back. The argument is either a table with tokens or a list of tokens. The `token.expand` function will trigger expansion but what happens really depends on what you're doing where.

This putter is actually a bit more flexible because the following input also works out okay:



```

\def\foo#1{[#1]}

\directlua {
  local list = { 101, 102, 103, token.create("foo"), "{abracadabra}" }
  token.putnext("(the)")
  token.putnext(list)
  token.putnext("(order)")
  token.putnext(unpack(list))
  token.putnext("(is reversed)")
}

```

We get this:

```
(is reversed)efg[abracadabra](order)efg[abracadabra](the)
```

So, strings get converted to individual tokens according to the current catcode regime and numbers become characters also according to this regime.

### 12.6.6 Nota bene

When scanning for the next token you need to keep in mind that we're not scanning like  $\text{\TeX}$  does: expanding, changing modes and doing things as it goes. When we scan with Lua we just pick up tokens. Say that we have:

```
\oof
```

but `\oof` is undefined. Normally  $\text{\TeX}$  will then issue an error message. However, when we have:

```
\def\foo{\oof}
```

We get no error, unless we expand `\foo` while `\oof` is still undefined. What happens is that as soon as  $\text{\TeX}$  sees an undefined macro it will create a hash entry and when later it gets defined that entry will be reused. So, `\oof` really exists but can be in an undefined state.

```

oof : oof
foo : foo
myfirstoof :

```

This was entered as:

```

oof      : \directlua{tex.print(token.scansname())}\oof
foo      : \directlua{tex.print(token.scansname())}\foo
myfirstoof : \directlua{tex.print(token.scansname())}\myfirstoof

```

The reason that you see `oof` reported and not `myfirstoof` is that `\oof` was already used in a previous paragraph.

If we now say:

```
\def\foo{}
```



we get:

```
oof : oof
foo : foo
myfirstoof :
```

And if we say

```
\def\foo{\oof}
```

we get:

```
oof : oof
foo : foo
myfirstoof :
```

When scanning from Lua we are not in a mode that defines (undefined) macros at all. There we just get the real primitive undefined macro token.

```
689527 537473675
684090 536969024
689705 536985953
```

This was generated with:

```
\directlua{local t = token.scannext() tex.print(t.id.." "..t.tok)}\myfirstoof
\directlua{local t = token.scannext() tex.print(t.id.." "..t.tok)}\mysecondoof
\directlua{local t = token.scannext() tex.print(t.id.." "..t.tok)}\mythirdoof
```

So, we do get a unique token because after all we need some kind of Lua object that can be used and garbage collected, but it is basically the same one, representing an undefined control sequence.



# 13 The MetaPost library `mplib`

## 13.1 Introduction

The library used in LuaMetaTeX differs from the one used in LuaTeX. There are for instance no backends and the binary number model is not available. There is also no textual output. There are scanners and injectors that make it possible to enhance the language and efficiently feed back into MetaPost. File handling is now completely delegated to Lua, so there are more callbacks.

*Some functionality is experimental and therefore documentation is limited. Also, details are discussed in articles.*

## 13.2 Process management

The MetaPost library interface registers itself in the table `mplib`. It is based on `mplib` version 3.11 (LuaTeX used version 2+). Not all functionality is described here. Once we're out of the experimental stage some more information will be added. Using the library boils down to initializing an instance, executing statements and picking up assembled figures in the form of Lua user data objects (and from there on Lua variables like tables).

### 13.2.1 new

To create a new MetaPost instance, call

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

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

NAME	TYPE	DESCRIPTION	DEFAULT
<code>error_line</code>	number	error line width	79
<code>print_line</code>	number	line length in ps output	100
<code>random_seed</code>	number	the initial random seed	variable
<code>math_mode</code>	string	the number system to use: scaled, double or decimal	scaled
<code>interaction</code>	string	the interaction mode: batch, nonstop, scroll or errorstop	errorstop
<code>job_name</code>	string	a compatibility value	
<code>utf8_mode</code>	boolean	permit characters in the range 128 upto 255 to be part of names	false
<code>text_mode</code>	boolean	permit characters 2 and 3 as fencing string literals	false
<code>tolerance</code>	number	the value used as criterium for straight lines	131/65536







The `run_error` callback gets the error message, help text and current interaction mode passed. Normally it's best to just quit and let the user fix the code.

When you are processing a snippet of text starting with `btex` or `verbatimtex` and ending with `etex`, the MetaPost `texscriptmode` parameter controls how spaces and newlines get honoured. The default value is 1. Possible values are:

NAME	MEANING
0	no newlines
1	newlines in <code>verbatimtex</code>
2	newlines in <code>verbatimtex</code> and <code>etex</code>
3	no leading and trailing strip in <code>verbatimtex</code>
4	no leading and trailing strip in <code>verbatimtex</code> and <code>btex</code>

That way the Lua handler (assigned to `make_text`) can do what it likes. An `etex` has to be followed by a space or `;` or be at the end of a line and preceded by a space or at the beginning of a line. The `make_text` function can return a string that gets fed into `scantokens`.

### 13.2.2 `getstatistics`

You can request statistics with:

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

This function returns the vital statistics for an `mplib` instance. Some are useful, others make more sense when debugging.

FIELD	TYPE	EXPLANATION
memory	number	bytes of node memory
hash	number	size of the hash
parameters	number	allocated parameter stack
input	number	allocated input stack
tokens	number	number of token nodes
pairs	number	number of pair nodes
knots	number	number of knot nodes
nodes	number	number of value nodes
symbols	number	number of symbolic nodes
characters	number	number of string bytes
strings	number	number of strings
internals	number	number of internals

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.

### 13.2.3 `execute`

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



```
<table> rettable = execute(mp,"metapost code")
```

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 stand alone `mpost` command, there is *no* implied ‘input’ at the start of the first chunk. When no string is passed to the `execute` function, there will still be one triggered because it then expects input from the terminal and you can emulate that channel with the callback you provide.

### 13.2.4 finish

Once you create an instance it is likely that you will keep it open for successive processing, if only because you want to avoid loading a format each time. If for some reason you want to stop using an `mplib` instance while processing is not yet actually done, you can call `finish`.

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

Eventually, used memory will be freed and open files will be closed by the Lua garbage collector, but an explicit `finish` is the only way to capture the final part of the output streams.

### 13.2.5 settolerance and gettolerance

These two functions relate to the bend tolerance, a value that is used when the export determines if a path has straight lines (like a rectangle has).

### 13.2.6 Errors

In case of an error you can get the context where it happened with `showcontext`.

### 13.2.7 The scanner status

When processing a graphic an instance is in a specific state and again we have a getter for the (internal) values `mplib.getstates()`: 0: normal, 1: skipping, 2: flushing, 3: absorbing, 4: var\_defining, 5: op\_defining, 6: loop\_defining. The current status can be queried with `getstatus`.

### 13.2.8 The hash

Macro names and variable names are stored in a hash table. You can get a list with entries with `gethashentries`, which takes an instance as first argument. When the second argument is `true` more details will be provided. With `gethashentry` you get info about the given macro or variable.



### 13.2.9 Callbacks

Some statistics about the number of calls to the callbacks can be queried with `getcallback-state`. This function expects a valid instance.

## 13.3 The end result

### 13.3.1 The figure

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

FIELD	TYPE	EXPLANATION
<code>status</code>	number	the return value: 0 = good, 1 = warning, 2 = errors, 3 = fatal error
<code>fig</code>	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:

You can check if a figure uses stacking with the `stacking` function. When objects are fetched, memory gets freed so no information about stacking is available then. You can get the used bend tolerance of an object with `tolerance`.

FIELD	TYPE	EXPLANATION
<code>boundingbox</code>	function	returns the bounding box, as an array of 4 values
<code>objects</code>	function	returns the actual array of graphic objects in this <code>fig</code>
<code>filename</code>	function	the filename this <code>fig</code> 's PostScript output would have written to in stand alone mode
<code>width</code>	function	the <code>fontcharwd</code> value
<code>height</code>	function	the <code>fontcharht</code> value
<code>depth</code>	function	the <code>fontchardp</code> value
<code>italic</code>	function	the <code>fontcharit</code> value
<code>charcode</code>	function	the (rounded) <code>charcode</code> value
<code>stacking</code>	function	is there a non-zero stacking

Note: you can call `fig:objects()` only once for any one `fig` object! Some information, like stacking, can only be queried when the complete figure is still present and calling up objects will free elements in the original once they are transferred.

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: `fill`, `outline`, `text`, `start_clip`, `stop_clip`, `start_bounds`, `stop_bounds`, `start_group` and `stop_group`. Each type has a different list of accessible values.



There is a 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, numbers are PostScript points (base points in  $\text{\TeX}$  speak) represented as a floating point number, unless stated otherwise. Field values that are of type table are explained in the next section.

### 13.3.2 fill

FIELD	TYPE	EXPLANATION
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
stacking	number	the stacking (level)

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

### 13.3.3 outline

FIELD	TYPE	EXPLANATION
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
stacking	number	the stacking (level)

The entry `dash` is optional.

### 13.3.4 start\_bounds, start\_clip, start\_group

FIELD	TYPE	EXPLANATION
path	table	the list of knots
stacking	number	the stacking (level)



### 13.3.5 stop\_bounds, stop\_clip, stop\_group

Here we have only one key:

FIELD	TYPE	EXPLANATION
stacking	number	the stacking (level)

## 13.4 Subsidiary table formats

### 13.4.1 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.

FIELD	TYPE	EXPLANATION
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 key type with value `elliptical` besides the array part containing the knot list.

### 13.4.2 Colors

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

FIELD	TYPE	EXPLANATION
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.

### 13.4.3 Transforms

Each transform is a six-item array.



INDEX	TYPE	EXPLANATION
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.

#### 13.4.4 Dashes

Each dash is a hash with two items. We use 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.

FIELD	TYPE	EXPLANATION
<code>dashes</code>	hash	an array of on-off numbers
<code>offset</code>	number	the starting offset value

#### 13.4.5 Pens and peninfo

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

FIELD	TYPE	EXPLANATION
<code>width</code>	number	width of the pen
<code>sx</code>	number	x scale
<code>rx</code>	number	x y multiplier
<code>ry</code>	number	y x multiplier
<code>sy</code>	number	y scale
<code>tx</code>	number	x offset
<code>ty</code>	number	y offset

#### 13.4.6 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.

```
<number> w = char_width(mp,<string> fontname, <number> char)
<number> h = char_height(mp,<string> fontname, <number> char)
<number> d = char_depth(mp,<string> fontname, <number> char)
```



## 13.5 Scanners

After a relative long period of testing the scanners are now part of the interface. That doesn't mean that there will be no changes: depending on the needs and experiences details might evolve. The summary below is there still preliminary and mostly provided as reminder.

SCANNER	ARGUMENT	RETURNS
scannext	instance, keep	token, mode, type
scanexpression	instance, keep	type
scantoken	instance, keep	token, mode, kind
scansymbol	instance, keep, expand	string
scannumeric	instance, type	number
scaninteger	instance, type	integer
scanboolean	instance, type	boolean
scanstring	instance, type	string
scanpair	instance, hashed, type	table or two numbers
scancolor	instance, hashed, type	table or three numbers
scancmykcolor	instance, hashed, type	table or four numbers
scantransform	instance, hashed, type	table or six numbers
scanpath	instance, hashed, type	table with hashes or arrays
scanpen	instance, hashed, type	table with hashes or arrays
scanproperty	<i>todo</i>	
skiptoken	<i>todo</i>	

The types and token codes are numbers but they actually depend on the implementation (although changes are unlikely). The types of data structures can be queried with `mplib.gettypes()`: 0: undefined, 1: vacuous, 2: boolean, 3: unknownboolean, 4: string, 5: unknownstring, 6: pen, 7: unknownpen, 8: path, 9: unknownpath, 10: picture, 11: unknownpicture, 12: transform, 13: color, 14: cmykcolor, 15: pair, 16: numeric, 17: known, 18: dependent, 19: protodependent, 20: independent, 21: tokenlist, 22: structured, 23: unsuffixedmacro, 24: suffixedmacro, and command codes with `mplib.getcodes()`: 0: undefined, 1: btex, 2: etex, 3: if, 4: fiorelse, 5: input, 6: iteration, 7: repeatloop, 8: exittest, 9: relax, 10: scantokens, 11: runscript, 12: maketext, 13: expandafter, 14: definedmacro, 15: save, 16: interim, 17: let, 18: newinternal, 19: macrodef, 20: shipout, 21: addto, 22: setbounds, 23: protection, 24: property, 25: show, 26: mode, 27: randomseed, 28: message, 29: everyjob, 30: delimiters, 31: write, 32: typename, 33: leftdelimiter, 34: begingroup, 35: nullary, 36: unary, 37: str, 38: void, 39: cycle, 40: ofbinary, 41: capsule, 42: string, 43: internal, 44: tag, 45: numeric, 46: plusorminus, 47: secondarydef, 48: tertiarybinary, 49: leftbrace, 50: pathjoin, 51: ampersand, 52: tertiarydef, 53: primarybinary, 54: equals, 55: and, 56: primarydef, 57: slash, 58: secondarybinary, 59: parametertype, 60: controls, 61: tension, 62: atleast, 63: curl, 64: macrospecial, 65: rightdelimiter, 66: leftbracket, 67: rightbracket, 68: rightbrace, 69: with, 70: thingstoadd, 71: of, 72: to, 73: step, 74: until, 75: within, 76: assignment, 77: colon, 78: comma, 79: semicolon, 80: endgroup, 81: stop, 82: undefinedcs

Now, if you really want to use these, keep in mind that the internals of MetaPost are not trivial, especially because expression scanning can be complex. So you need to experiment a bit. In



ConT<sub>E</sub>Xt all is (and will be) hidden below an abstraction layer so users are not bothered by all these look-ahead and push-back issues that originate in the way MetaPost scans its input.

The supported color models are: `mplib.getcolormodels()`: 0: no, 1: grey, 2: rgb, 3: cmyk.

If you want the internal codes of the possible fields in a graphic object use `mplib.getobject-types()`: 0: , 1: fill, 2: outline, 3: start\_clip, 4: start\_group, 5: start\_bounds, 6: stop\_clip, 7: stop\_group, 8: stop\_bounds. You can query the id of a graphic object with the `gettype` function.

ID	OBJECT	FIELDS
1	fill	type path htap pen color linejoin miterlimit prescript postscript stacking
2	outline	type path pen color linejoin miterlimit linecap dash prescript postscript stacking
3	start_clip	type path prescript postscript stacking
4	start_group	type path prescript postscript stacking
5	start_bounds	type path prescript postscript stacking
6	stop_clip	type stacking
7	stop_group	type stacking
8	stop_bounds	type stacking

## 13.6 Injectors

It is important to know that piping code into the library is pretty fast and efficient. Most processing time relates to memory management, calculations and generation of output can not be neglected either. Out of curiosity I added some functions that directly push data into the library but the gain is not that large.<sup>9</sup>

SCANNER	ARGUMENT
<code>injectnumeric</code>	instance, number
<code>injectinteger</code>	instance, number
<code>injectboolean</code>	instance, boolean
<code>injectstring</code>	instance, string
<code>injectpair</code>	instance, (table with) two numbers
<code>injectcolor</code>	instance, (table with) three numbers
<code>injectcmykcolor</code>	instance, (table with) four numbers
<code>injecttransform</code>	instance, (table with) six numbers
<code>injectpath</code>	instance, table with hashes or arrays, cycle, variant
<code>injectwhatever</code>	instance, ont of the above depending on type and size

The path injector takes a table with subtables that are either hashed (like the path solver) or arrays with two, four or six entries. When the third argument has the value `true` the path is closed. When the fourth argument is `true` the path is constructed out of straight lines (as with `--`) by setting the `curl` values to 1 automatically.<sup>10</sup>

<sup>9</sup> The main motivation was checking of huge paths could be optimized. The other data structures were then added for completeness.

<sup>10</sup> This is all experimental so future versions might provide more control.





This is the simplest path definition:

```
{
  { x, y },
  ...,
  cycle = true
}
```

and this one also has the control points:

```
{
  { x0, y0, x1, y1, x2, y2 },
  ...,
  cycle = true
}
```

A very detailed specification is this but you have to make sure that the parameters make sense.

```
{
  {
    x_coord      = ...,
    y_coord      = ...,
    left_x       = ...,
    left_y       = ...,
    right_x      = ...,
    right_y      = ...,
    left_tension = ...,
    right_tension = ...,
    left_curl    = ...,
    right_curl   = ...,
    direction_x  = ...,
    direction_y  = ...,
    left_type    = ...,
    right_type   = ...,
  },
  ...,
  cycle = true
}
```

Instead of the optional keyword `cycle` you can use `close`.

## 13.7 To be checked

```
% solvepath
% expandtex
```





# 14 The pdf related libraries

## 14.1 The pdf library

### 14.1.1 Introduction

The pdf library replaces the epdf library and provides an interface to pdf files. It uses the same code as is used for pdf image inclusion. The pplib library by Paweł Jackowski replaces the poppler (derived from xpdf) library.

A pdf file is basically a tree of objects and one descends into the tree via dictionaries (key/value) and arrays (index/value). There are a few topmost dictionaries that start at root that are accessed more directly.

Although everything in pdf is basically an object we only wrap a few in so called userdata Lua objects.

TYPE	MAPPING
<b>pdf</b>	<b>Lua</b>
null	nil
boolean	boolean
integer	integer
float	number
name	string
string	string
array	array userdata
dictionary	dictionary userdata
stream	stream userdata (with related dictionary)
reference	reference userdata

The regular getters return these Lua data types but one can also get more detailed information.

### 14.1.2 open, openfile, new, getstatus, close, unencrypt

A document is loaded from a file (by name or handle) or string:

```
<pdf document> = pdf.open(filename)
<pdf document> = pdf.openfile(filehandle)
<pdf document> = pdf.new(somestring,somelength)
```

Such a document is closed with:

```
pdf.close(<pdf document>)
```

You can check if a document opened well by:

```
pdf.getstatus(<pdf document>)
```



The returned codes are:

VALUE	EXPLANATION
-2	the document failed to open
-1	the document is (still) protected
0	the document is not encrypted
2	the document has been unencrypted

An encrypted document can be unencrypted by the next command where instead of either password you can give nil:

```
pdfc.unencrypt(<pdfc document>,userpassword,ownerpassword)
```

### 14.1.3 getsize, getversion, getnofobjects, getnofpages

A successfully opened document can provide some information:

```
bytes = getsize(<pdfc document>)
major, minor = getversion(<pdfc document>)
n = getnofobjects(<pdfc document>)
n = getnofpages(<pdfc document>)
bytes, waste = getnofpages(<pdfc document>)
```

### 14.1.4 get[catalog|trailer|info]

For accessing the document structure you start with the so called catalog, a dictionary:

```
<pdfc dictionary> = pdfc.getcatalog(<pdfc document>)
```

The other two root dictionaries are accessed with:

```
<pdfc dictionary> = pdfc.gettrailer(<pdfc document>)
<pdfc dictionary> = pdfc.getinfo(<pdfc document>)
```

### 14.1.5 getpage, getbox

A specific page can conveniently be reached with the next command, which returns a dictionary.

```
<pdfc dictionary> = pdfc.getpage(<pdfc document>,pagenumber)
```

Another convenience command gives you the (bounding) box of a (normally page) which can be inherited from the document itself. An example of a valid box name is MediaBox.

```
pages = pdfc.getbox(<pdfc dictionary>,boxname)
```

### 14.1.6 get[string|integer|number|boolean|name]

Common values in dictionaries and arrays are strings, integers, floats, booleans and names (which are also strings) and these are also normal Lua objects:



```
s = getstring (<pdf array|dictionary>,index|key)
i = getinteger(<pdf array|dictionary>,index|key)
n = getnumber (<pdf array|dictionary>,index|key)
b = getboolean(<pdf array|dictionary>,index|key)
n = getname    (<pdf array|dictionary>,index|key)
```

The `getstring` function has two extra variants:

```
s, h = getstring (<pdf array|dictionary>,index|key,false)
s     = getstring (<pdf array|dictionary>,index|key,true)
```

The first call returns the original string plus a boolean indicating if the string is hex encoded. The second call returns the unencoded string.

### 14.1.7 `get[dictionary|array|stream]`

Normally you will use an index in an array and key in a dictionary but dictionaries also accept an index. The size of an array or dictionary is available with the usual `#` operator.

```
<pdf dictionary> = getdictionary(<pdf array|dictionary>,index|key)
<pdf array>      = getarray      (<pdf array|dictionary>,index|key)
<pdf stream>,
<pdf dictionary> = getstream     (<pdf array|dictionary>,index|key)
```

These commands return dictionaries, arrays and streams, which are dictionaries with a blob of data attached.

Before we come to an alternative access mode, we mention that the objects provide access in a different way too, for instance this is valid:

```
print(pdf.open("foo.pdf").Catalog.Type)
```

At the topmost level there are `Catalog`, `Info`, `Trailer` and `Pages`, so this is also okay:

```
print(pdf.open("foo.pdf").Pages[1])
```

### 14.1.8 `[open|close|readfrom|whole|]stream`

Streams are sort of special. When your index or key hits a stream you get back a stream object and dictionary object. The dictionary you can access in the usual way and for the stream there are the following methods:

```
okay    = openstream(<pdf stream>,[decode])
         closestream(<pdf stream>)
str, n = readfromstream(<pdf stream>)
str, n = readwholestream(<pdf stream>,[decode])
```

You either read in chunks, or you ask for the whole. When reading in chunks, you need to open and close the stream yourself. The `n` value indicates the length read. The `decode` parameter controls if the stream data gets uncompressed.



As with dictionaries, you can access fields in a stream dictionary in the usual Lua way too. You get the content when you 'call' the stream. You can pass a boolean that indicates if the stream has to be decompressed.

### 14.1.9 getfrom[dictionary|array]

In addition to the interface described before, there is also a bit lower level interface available.

```
key, type, value, detail = getfromdictionary(<pdf dictionary>, index)
type, value, detail = getfromarray(<pdf array>, index)
```

TYPE	MEANING	VALUE	DETAIL
0	none	nil	
1	null	nil	
2	boolean	boolean	
3	integer	integer	
4	number	float	
5	name	string	
6	string	string	hex
7	array	arrayobject	size
8	dictionary	dictionaryobject	size
9	stream	streamobject	dictionary size
10	reference	integer	

A hex string is (in the pdf file) surrounded by <> while plain strings are bounded by <>.

### 14.1.10 [dictionary|array]totable

All entries in a dictionary or table can be fetched with the following commands where the return values are a hashed or indexed table.

```
hash = dictionarytotable(<pdf dictionary>)
list = arraytotable(<pdf array>)
```

You can get a list of pages with:

```
{ { <pdf dictionary>, size, objnum }, ... } = pagestotable(<pdf document>)
```

### 14.1.11 getfromreference

Because you can have unresolved references, a reference object can be resolved with:

```
type, <pdf dictionary|array|stream>, detail = getfromreference(<pdf reference>)
```

So, as second value you get back a new pdf userdata object that you can query.



## 14.2 Memory streams

The `pdfe.new` function takes three arguments:

VALUE	EXPLANATION
<code>stream</code>	this is a (in low level Lua speak) light userdata object, i.e. a pointer to a sequence of bytes
<code>length</code>	this is the length of the stream in bytes (the stream can have embedded zeros)
<code>name</code>	optional, this is a unique identifier that is used for hashing the stream

The third argument is optional. When it is not given the function will return a `pdfe` document object as with a regular file, otherwise it will return a filename that can be used elsewhere (e.g. in the `image` library) to reference the stream as pseudo file.

Instead of a light userdata stream (which is actually fragile but handy when you come from a library) you can also pass a Lua string, in which case the given length is (at most) the string length.

The function returns a `pdfe` object and a string. The string can be used in the `img` library instead of a filename. You need to prevent garbage collection of the object when you use it as image (for instance by storing it somewhere).

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 `graphicmagick`, 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 and we might come up with a better implementation.

## 14.3 The pdfscanner library

This library is not available in LuaMetaTeX.







# 15 Extra libraries

## 15.1 Introduction

The libraries can be grouped in categories like fonts, languages, T<sub>E</sub>X, MetaPost, pdf, etc. There are however also some that are more general purpose and these are discussed here.

## 15.2 File and string readers: **fio** and **sio**

This library provides a set of functions for reading numbers from a file and in addition to the regular io library functions. The following work on normal Lua file handles.

NAME	ARGUMENTS	RESULTS
readcardinal1	(f)	a 1 byte unsigned integer
readcardinal2	(f)	a 2 byte unsigned integer
readcardinal3	(f)	a 3 byte unsigned integer
readcardinal4	(f)	a 4 byte unsigned integer
readcardinaltable	(f,n,b)	n cardinals of b bytes
readinteger1	(f)	a 1 byte signed integer
readinteger2	(f)	a 2 byte signed integer
readinteger3	(f)	a 3 byte signed integer
readinteger4	(f)	a 4 byte signed integer
readintegertable	(f,n,b)	n integers of b bytes
readfixed2	(f)	a float made from a 2 byte fixed format
readfixed4	(f)	a float made from a 4 byte fixed format
read2dot14	(f)	a float made from a 2 byte in 2dot4 format
setposition	(f,p)	goto position p
getposition	(f)	get the current position
skipposition	(f,n)	skip n positions
readbytes	(f,n)	n bytes
readbytetable	(f,n)	n bytes

When relevant there are also variants that end with `le` that do it the little endian way. The fixed and dot floating points formats are found in font files and return Lua doubles.

A similar set of function as in the `fio` library is available in the `sio` library: `sio.readcardinal1`, `sio.readcardinal2`, `sio.readcardinal3`, `sio.readcardinal4`, `sio.readcardinaltable`, `sio.readinteger1`, `sio.readinteger2`, `sio.readinteger3`, `sio.readinteger4`, `sio.readintegertable`, `sio.readfixed2`, `sio.readfixed4`, `sio.read2dot14`, `sio.setposition`, `sio.getposition`, `sio.skipposition`, `sio.readbytes` and `sio.readbytetable`. Here the first argument is a string instead of a file handle.

## 15.3 md5

NAME	ARGUMENTS	RESULTS
sum		



hex  
HEX

---

## 15.4 sha2

NAME	ARGUMENTS	RESULTS
digest256		
digest384		
digest512		

---

## 15.5 xzip

NAME	ARGUMENTS	RESULTS
compress		
decompress		
adler32		
crc32		

---

## 15.6 xmath

This library just opens up standard C math library and the main reason for it being there is that it permits advanced graphics in MetaPost (via the Lua interface). There are three constant values:

NAME	ARGUMENTS	RESULTS
inf	—	inf
nan	—	nan
pi	—	3.1415926535898

---

and a lot of functions:

NAME	ARGUMENTS	RESULTS
acos	(a)	
acosh	(a)	
asin	(a)	
asinh	(a)	
atan	(a[,b])	
atan2	(a[,b])	
atanh	(a)	
cbirt	(a)	
ceil	(a)	
copysign	(a,b)	
cos	(a)	
cosh	(a)	
deg	(a)	



erf	(a)
erfc	(a)
exp	(a)
exp2	(a)
expm1	(a)
fabs	(a)
fdim	(a,b)
floor	(a)
fma	(a,b,c)
fmax	(...)
fmin	(...)
fmod	(a,b)
frexp	(a,b)
gamma	(a)
hypot	(a,b)
isfinite	(a)
isinf	(a)
isnan	(a)
isnormal	(a)
j0	(a)
j1	(a)
jn	(a,b)
ldexp	(a,b)
lgamma	(a)
l0	(a)
l1	(a)
ln	(a,b)
log	(a[,b])
log10	(a)
log1p	(a)
log2	(a)
logb	(a)
modf	(a,b)
nearbyint	(a)
nextafter	(a,b)
pow	(a,b)
rad	(a)
remainder	(a,b)
remquo	(a,b)
round	(a)
scalbn	(a,b)
sin	(a)
sinh	(a)
sqrt	(a)
tan	(a)
tanh	(a)



tgamma	(a)
trunc	(a)
y0	(a)
y1	(a)
yn	(a)

---

## 15.7 xcomplex

LuaMetaT<sub>E</sub>X also provides a complex library xcomplex. The complex number is a userdata:

NAME	ARGUMENTS	RESULTS
new	(r,i)	a complex userdata type
tostring	(z)	a string representation
topair	(z)	two numbers

---

There is a bunch of functions that take a complex number:

NAME	ARGUMENTS	RESULTS
abs	(a)	
arg	(a)	
imag	(a)	
real	(a)	
onj	(a)	
proj	(a)	
exp"	(a)	
log	(a)	
sqrt	(a)	
pow	(a,b)	
sin	(a)	
cos	(a)	
tan	(a)	
asin	(a)	
acos	(a)	
atan	(a)	
sinh	(a)	
cosh	(a)	
tanh	(a)	
asinh	(a)	
acosh	(a)	
atanh	(a)	

---

These are accompanied by libcerf functions:

NAME	ARGUMENTS	RESULTS
erf	(a)	The complex error function erf(z)
erfc	(a)	The complex complementary error function erfc(z) = 1 - erf(z)



erfcx	(a)	The underflow-compensating function $\text{erfcx}(z) = \exp(z^2) \text{erfc}(z)$
erfi	(a)	The imaginary error function $\text{erfi}(z) = -i \text{erf}(iz)$
dawson	(a)	Dawson's integral $D(z) = \sqrt{\pi}/2 * \exp(-z^2) * \text{erfi}(z)$
voigt	(a,b,c)	The convolution of a Gaussian and a Lorentzian
voigt_hwhm	(a,b)	The half width at half maximum of the Voigt profile

## 15.8 xdecimal

As an experiment LuaMetaTeX provides an interface to the decNumber library that we have on board for MetaPost anyway. Apart from the usual support for operators there are some functions.

NAME	ARGUMENTS	RESULTS
abs	(a)	
new	([n or s])	
copy	(a)	
trim	(a)	
tostring	(a)	
tonumber	(a)	
setprecision	(n)	
getprecision	()	
conj	(a)	
abs	(a)	
pow	(a,b)	
sqrt	(a)	
ln	(a)	
log	(a)	
exp	(a)	
bor	(a,b)	
bxor	(a,b)	
band	(a,b)	
shift	(a,b)	
rotate	(a,b)	
minus	(a)	
plus	(a)	
min	(a,b)	
max	(a,b)	

## 15.9 lfs

The original lfs module has been adapted a bit to our needs but for practical reasons we kept the namespace. This module will probably evolve a bit over time.

NAME	ARGUMENTS	RESULTS
attributes	(name)	
chdir	(name)	



currentdir	()	
dir	(name)	name, mode, size and mtime
mkdir	(name)	
rmdir	(name)	
touch	(name)	
link	(name)	
symlinkattributes	(name)	
isdir	(name)	
isfile	(name)	
iswriteabledir	(name)	
iswriteablefile	(name)	
isreadabledir	(name)	
isreadablefile	(name)	

---

The `dir` function is a traverser which in addition to the name returns some more properties. Keep in mind that the traverser loops over a directory and that it doesn't run well when used nested. This is a side effect of the operating system. It is also the reason why we return some properties because querying them via attributes would interfere badly.

The following attributes are returned by attributes:

NAME	VALUE
mode	
size	
modification	
access	
change	
permissions	
nlink	

---

## 15.10 pngdecode

This module is experimental and used in image inclusion. It is not some general purpose module and is supposed to be used in a very controlled way. The interfaces might evolve.

NAME	ARGUMENTS	RESULTS
applyfilter	(str,nx,ny,slice)	string
splitmask	(str,nx,ny,bpp,bytes)	string
interlace	(str,nx,ny,slice,pass)	string
expand	(str,nx,ny,parts,xline,factor)	string

---

## 15.11 basexx

Some more experimental helpers:

NAME	ARGUMENTS	RESULTS
encode16	(str[,newline])	string



<code>decode16</code>	<code>(str)</code>	string
<code>encode64</code>	<code>(str[,newline])</code>	string
<code>decode64</code>	<code>(str)</code>	string
<code>encode85</code>	<code>(str[,newline])</code>	string
<code>decode85</code>	<code>(str)</code>	string
<code>encodeRL</code>	<code>(str)</code>	string
<code>decodeRL</code>	<code>(str)</code>	string
<code>encodeLZW</code>	<code>(str[,defaults])</code>	string
<code>decodeLZW</code>	<code>(str[,defaults])</code>	string

---

## 15.12 Multibyte string functions

The `string` library has a few extra functions, for example `string.explode`. This function takes upto two arguments: `string.explode(s[,m])` and 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  $\text{\TeX}$  macros.

The `string` library also has six extra iterators that return strings piecemeal: `string.utfvalues`, `string.utfcharacters`, `string.characters`, `string.characterpairs`, `string.bytes` and `string.bytepairs`.

- `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 utf16 encoded data into utf8.

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 `lua.jit`. This is typically a function that gets adapted as Lua itself progresses.

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 provides some native utf8 support but we have added a few similar helpers too: `string.utfvalue`, `string.utfcharacter` and `string.utflength`.

- ▶ `string.utfvalue(s)`: returns the codepoints of the characters in the given string
- ▶ `string.utfcharacter(c, ...)`: returns a string with the characters of the given code points
- ▶ `string.utflength(s)`: returns the length of the given string

These three functions are relative fast and don't do much checking. They can be used as building blocks for other helpers.

## 15.13 Extra os library functions

The `os` library has a few extra functions and variables: `os.selfdir`, `os.selfarg`, `os.setenv`, `os.env`, `os.gettimeofday`, `os.type`, `os.name` and `os.uname`, that we will discuss here. There are also some time related helpers in the `lua` namespace.

- ▶ `os.selfdir` is a variable that holds the directory path of the actual executable. For example: `\directlua{tex.sprint(os.selfdir)}`.
- ▶ `os.selfarg` is a table with the command line arguments.
- ▶ `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. Keep in mind that there might be platforms where this function is not available.
- ▶ `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`, `generic` (unknown). But ... we only provide LuaMetaTeX binaries for the mainstream variants.

Officially we only support mainstream systems: MS Windows, linux, FreeBSD and os-x. Of course one can build LuaMetaTeX for other systems, in which case one has to check the above.

- ▶ `os.uname` returns a table with specific operating system information acquired at runtime. The keys in the returned table are all string values, and their names are: `sysname`, `machine`, `release`, `version`, and `nodename`.

## 15.14 The lua library functions

The `lua` library provides some general helpers.

- ▶ The `newtable` and `newindex` functions can be used to create tables with space reserved beforehand for the given amount of entries.
- ▶ The `getstacktop` function returns a number that can be used for diagnostic purposes.





- ▶ The functions `getruntime`, `getcurrenttime`, `getpreciseticks` and `getprecisesseconds` return what their name suggests.
- ▶ On MS Windows the `getcodepage` function returns two numbers, one for the command handler and one for the graphical user interface.
- ▶ The name of the startup file is reported by `getstartupfile`.
- ▶ The Lua version is reported by `getversion`.
- ▶ The `lua.openfile` function can be used instead of `io.open`. On MS Windows it will convert the filename to a so called wide one which means that filenames in utf8 encoding will work ok. On the other hand, names given in the codepage won't.





**254 Extra libraries**

# Primitive codes

here follows a list with all primitives and their category is shown. When the engine starts up in ini mode all primitives get defined along with some properties that makes it possible to do a reverse lookup of a combination of command code and char code. But, a primitive, being also a regular command can be redefined later on. The table below shows the original pairs but in ConT<sub>E</sub>Xt some of these primitives are redefined. However, any macro that fits a command and char pair is (reported as) a primitive in logs and error messages. In the end all tokens are such a combination, The first 16 command codes are reserved for characters (the whole Unicode range can be used as char code) with specific catcodes and not mentioned in the list.

PRIMITIVE	COMMAND NAME	CMD	CHR	ORIGIN
\	explicit_space	74	0	tex
\-	discretionary	57	1	tex
\/	italic_correction	54	0	tex
\Uabove	math_fraction	62	8	luatex
\Uabovewithdelims	math_fraction	62	9	luatex
\Uatop	math_fraction	62	12	luatex
\Uatopwithdelims	math_fraction	62	13	luatex
\Uchar	convert	132	17	luatex
\Udelcode	define_char_code	103	9	luatex
\Udelcodenum	define_char_code	103	10	luatex
\Udelimiter	delimiter_number	24	1	luatex
\Udelimiterover	math_radical	76	6	luatex
\Udelimiterunder	math_radical	76	5	luatex
\Uhextensible	math_radical	76	7	luatex
\Uleft	math_fence	59	6	luatex
\Umathaccent	math_accent	56	1	luatex
\Umathaccentbasedepth	set_math_parameter	105	3	luatex
\Umathaccentbaseheight	set_math_parameter	105	2	luatex
\Umathaccentbottomshiftdown	set_math_parameter	105	77	luatex
\Umathaccenttopshiftup	set_math_parameter	105	76	luatex
\Umathaccentvariant	set_math_parameter	105	91	luatex
\Umathadapttolleft	math_modifier	61	3	luatex
\Umathadapttoright	math_modifier	61	4	luatex
\Umathaxis	set_math_parameter	105	1	luatex
\Umathbotaccentvariant	set_math_parameter	105	93	luatex
\Umathchar	math_char_number	26	1	luatex
\Umathcharclass	some_item	83	33	luatex
\Umathchardef	shorthand_def	118	2	luatex
\Umathcharfam	some_item	83	34	luatex
\Umathcharnum	math_char_number	26	2	luatex
\Umathcharnumdef	shorthand_def	118	3	luatex
\Umathcharslot	some_item	83	35	luatex
\Umathclass	math_char_number	26	3	luatex



<code>\Umathcode</code>	<code>define_char_code</code>	103	6	luatex
<code>\Umathcodenum</code>	<code>define_char_code</code>	103	7	luatex
<code>\Umathconnectoroverlapmin</code>	<code>set_math_parameter</code>	105	55	luatex
<code>\Umathdegreevariant</code>	<code>set_math_parameter</code>	105	90	luatex
<code>\Umathdelimiterovervariant</code>	<code>set_math_parameter</code>	105	84	luatex
<code>\Umathdelimiterundervariant</code>	<code>set_math_parameter</code>	105	85	luatex
<code>\Umathdenominatorvariant</code>	<code>set_math_parameter</code>	105	96	luatex
<code>\Umathextrasubpreshift</code>	<code>set_math_parameter</code>	105	59	luatex
<code>\Umathextrasubprespace</code>	<code>set_math_parameter</code>	105	74	luatex
<code>\Umathextrasubshift</code>	<code>set_math_parameter</code>	105	57	luatex
<code>\Umathextrasubspace</code>	<code>set_math_parameter</code>	105	72	luatex
<code>\Umathextrasuppreshift</code>	<code>set_math_parameter</code>	105	58	luatex
<code>\Umathextrasupprespace</code>	<code>set_math_parameter</code>	105	73	luatex
<code>\Umathextrasupshift</code>	<code>set_math_parameter</code>	105	56	luatex
<code>\Umathextrasupspace</code>	<code>set_math_parameter</code>	105	71	luatex
<code>\Umathflattenedaccentbasedepth</code>	<code>set_math_parameter</code>	105	5	luatex
<code>\Umathflattenedaccentbaseheight</code>	<code>set_math_parameter</code>	105	4	luatex
<code>\Umathflattenedaccentbottomshiftdown</code>	<code>set_math_parameter</code>	105	79	luatex
<code>\Umathflattenedaccenttopshiftup</code>	<code>set_math_parameter</code>	105	78	luatex
<code>\Umathfractiondelsize</code>	<code>set_math_parameter</code>	105	29	luatex
<code>\Umathfractiondenomdown</code>	<code>set_math_parameter</code>	105	28	luatex
<code>\Umathfractiondenomvgap</code>	<code>set_math_parameter</code>	105	27	luatex
<code>\Umathfractionnumup</code>	<code>set_math_parameter</code>	105	26	luatex
<code>\Umathfractionnumvgap</code>	<code>set_math_parameter</code>	105	25	luatex
<code>\Umathfractionrule</code>	<code>set_math_parameter</code>	105	24	luatex
<code>\Umathfractionvariant</code>	<code>set_math_parameter</code>	105	88	luatex
<code>\Umathhextensiblevariant</code>	<code>set_math_parameter</code>	105	86	luatex
<code>\Umathlimitabovebgap</code>	<code>set_math_parameter</code>	105	33	luatex
<code>\Umathlimitabovekern</code>	<code>set_math_parameter</code>	105	34	luatex
<code>\Umathlimitabovevgap</code>	<code>set_math_parameter</code>	105	32	luatex
<code>\Umathlimitbelowbgap</code>	<code>set_math_parameter</code>	105	36	luatex
<code>\Umathlimitbelowkern</code>	<code>set_math_parameter</code>	105	37	luatex
<code>\Umathlimitbelowvgap</code>	<code>set_math_parameter</code>	105	35	luatex
<code>\Umathlimits</code>	<code>math_modifier</code>	61	1	luatex
<code>\Umathnoaxis</code>	<code>math_modifier</code>	61	6	luatex
<code>\Umathnolimits</code>	<code>math_modifier</code>	61	2	luatex
<code>\Umathnolimitsubfactor</code>	<code>set_math_parameter</code>	105	38	luatex
<code>\Umathnolimitsupfactor</code>	<code>set_math_parameter</code>	105	39	luatex
<code>\Umathnumeratorvariant</code>	<code>set_math_parameter</code>	105	95	luatex
<code>\Umathopenupdepth</code>	<code>math_modifier</code>	61	10	luatex
<code>\Umathopenupheight</code>	<code>math_modifier</code>	61	9	luatex
<code>\Umathoperatorsize</code>	<code>set_math_parameter</code>	105	8	luatex
<code>\Umathoverbarkern</code>	<code>set_math_parameter</code>	105	9	luatex
<code>\Umathoverbarrule</code>	<code>set_math_parameter</code>	105	10	luatex
<code>\Umathoverbarvgap</code>	<code>set_math_parameter</code>	105	11	luatex
<code>\Umathoverdelimiterbgap</code>	<code>set_math_parameter</code>	105	43	luatex



<code>\Umathoverdelimitervariant</code>	<code>set_math_parameter</code>	105	82	luatex
<code>\Umathoverdelimitervgap</code>	<code>set_math_parameter</code>	105	42	luatex
<code>\Umathoverlayaccentvariant</code>	<code>set_math_parameter</code>	105	94	luatex
<code>\Umathoverlinevariant</code>	<code>set_math_parameter</code>	105	80	luatex
<code>\Umathphantom</code>	<code>math_modifier</code>	61	7	luatex
<code>\Umathpresubshiftdistance</code>	<code>set_math_parameter</code>	105	70	luatex
<code>\Umathpresupshiftdistance</code>	<code>set_math_parameter</code>	105	69	luatex
<code>\Umathprimeraise</code>	<code>set_math_parameter</code>	105	60	luatex
<code>\Umathprimeshiftedrop</code>	<code>set_math_parameter</code>	105	62	luatex
<code>\Umathprimeshiftup</code>	<code>set_math_parameter</code>	105	61	luatex
<code>\Umathprimespaceafter</code>	<code>set_math_parameter</code>	105	63	luatex
<code>\Umathprimevariant</code>	<code>set_math_parameter</code>	105	99	luatex
<code>\Umathprimewidth</code>	<code>set_math_parameter</code>	105	64	luatex
<code>\Umathquad</code>	<code>set_math_parameter</code>	105	0	luatex
<code>\Umathradicaldegreeafter</code>	<code>set_math_parameter</code>	105	19	luatex
<code>\Umathradicaldegreebefore</code>	<code>set_math_parameter</code>	105	18	luatex
<code>\Umathradicaldegreeraise</code>	<code>set_math_parameter</code>	105	20	luatex
<code>\Umathradicalkern</code>	<code>set_math_parameter</code>	105	15	luatex
<code>\Umathradicalrule</code>	<code>set_math_parameter</code>	105	16	luatex
<code>\Umathradicalvariant</code>	<code>set_math_parameter</code>	105	89	luatex
<code>\Umathradicalvgap</code>	<code>set_math_parameter</code>	105	17	luatex
<code>\Umathrulerdepth</code>	<code>set_math_parameter</code>	105	66	luatex
<code>\Umathruleheight</code>	<code>set_math_parameter</code>	105	65	luatex
<code>\Umathskeweddelimitertolerance</code>	<code>set_math_parameter</code>	105	75	luatex
<code>\Umathskewedfractionhgap</code>	<code>set_math_parameter</code>	105	30	luatex
<code>\Umathskewedfractionvgap</code>	<code>set_math_parameter</code>	105	31	luatex
<code>\Umathspaceafterscript</code>	<code>set_math_parameter</code>	105	54	luatex
<code>\Umathspacebeforescript</code>	<code>set_math_parameter</code>	105	53	luatex
<code>\Umathstackdenomdown</code>	<code>set_math_parameter</code>	105	23	luatex
<code>\Umathstacknumup</code>	<code>set_math_parameter</code>	105	22	luatex
<code>\Umathstackvariant</code>	<code>set_math_parameter</code>	105	100	luatex
<code>\Umathstackvgap</code>	<code>set_math_parameter</code>	105	21	luatex
<code>\Umathsubscriptvariant</code>	<code>set_math_parameter</code>	105	98	luatex
<code>\Umathsubshiftdistance</code>	<code>set_math_parameter</code>	105	68	luatex
<code>\Umathsubshiftdown</code>	<code>set_math_parameter</code>	105	46	luatex
<code>\Umathsubshiftdrop</code>	<code>set_math_parameter</code>	105	44	luatex
<code>\Umathsubsupshiftdown</code>	<code>set_math_parameter</code>	105	47	luatex
<code>\Umathsubsupvgap</code>	<code>set_math_parameter</code>	105	52	luatex
<code>\Umathsubtopmax</code>	<code>set_math_parameter</code>	105	48	luatex
<code>\Umathsupbottommin</code>	<code>set_math_parameter</code>	105	50	luatex
<code>\Umathsuperscriptvariant</code>	<code>set_math_parameter</code>	105	97	luatex
<code>\Umathsupshiftdistance</code>	<code>set_math_parameter</code>	105	67	luatex
<code>\Umathsupshiftdrop</code>	<code>set_math_parameter</code>	105	45	luatex
<code>\Umathsupshiftup</code>	<code>set_math_parameter</code>	105	49	luatex
<code>\Umathsupsubbottommax</code>	<code>set_math_parameter</code>	105	51	luatex
<code>\Umathtopaccentvariant</code>	<code>set_math_parameter</code>	105	92	luatex



<code>\Umathunderbarkern</code>	<code>set_math_parameter</code>	105	12	luatex
<code>\Umathunderbarrule</code>	<code>set_math_parameter</code>	105	13	luatex
<code>\Umathunderbarvgap</code>	<code>set_math_parameter</code>	105	14	luatex
<code>\Umathunderdelimitervgap</code>	<code>set_math_parameter</code>	105	41	luatex
<code>\Umathunderdelimitervariant</code>	<code>set_math_parameter</code>	105	83	luatex
<code>\Umathunderdelimitervgap</code>	<code>set_math_parameter</code>	105	40	luatex
<code>\Umathunderlinevariant</code>	<code>set_math_parameter</code>	105	81	luatex
<code>\Umathuseaxis</code>	<code>math_modifier</code>	61	5	luatex
<code>\Umathvextensiblevariant</code>	<code>set_math_parameter</code>	105	87	luatex
<code>\Umathvoid</code>	<code>math_modifier</code>	61	8	luatex
<code>\Umathxscale</code>	<code>set_math_parameter</code>	105	6	luatex
<code>\Umathyscale</code>	<code>set_math_parameter</code>	105	7	luatex
<code>\Umiddle</code>	<code>math_fence</code>	59	7	luatex
<code>\Unosubprescript</code>	<code>math_script</code>	77	7	luatex
<code>\Unosubscript</code>	<code>math_script</code>	77	5	luatex
<code>\Unosuperprescript</code>	<code>math_script</code>	77	8	luatex
<code>\Unosuperscript</code>	<code>math_script</code>	77	6	luatex
<code>\Uoperator</code>	<code>math_fence</code>	59	4	luatex
<code>\Uover</code>	<code>math_fraction</code>	62	10	luatex
<code>\Uoverdelimiter</code>	<code>math_radical</code>	76	4	luatex
<code>\Uoverwithdelims</code>	<code>math_fraction</code>	62	11	luatex
<code>\Uprimescript</code>	<code>math_script</code>	77	13	luatex
<code>\Uradical</code>	<code>math_radical</code>	76	1	luatex
<code>\Uright</code>	<code>math_fence</code>	59	8	luatex
<code>\Uroot</code>	<code>math_radical</code>	76	2	luatex
<code>\Ushiftedsubprescript</code>	<code>math_script</code>	77	11	luatex
<code>\Ushiftedsubscript</code>	<code>math_script</code>	77	9	luatex
<code>\Ushiftedsuperprescript</code>	<code>math_script</code>	77	12	luatex
<code>\Ushiftedsuperscript</code>	<code>math_script</code>	77	10	luatex
<code>\Uskewed</code>	<code>math_fraction</code>	62	14	luatex
<code>\Uskewedwithdelims</code>	<code>math_fraction</code>	62	15	luatex
<code>\Ustack</code>	<code>math_choice</code>	64	1	luatex
<code>\Ustartdisplaymath</code>	<code>math_shift_cs</code>	78	2	luatex
<code>\Ustartmath</code>	<code>math_shift_cs</code>	78	0	luatex
<code>\Ustopdisplaymath</code>	<code>math_shift_cs</code>	78	3	luatex
<code>\Ustopmath</code>	<code>math_shift_cs</code>	78	1	luatex
<code>\Ustyle</code>	<code>math_style</code>	63	8	luatex
<code>\Usubprescript</code>	<code>math_script</code>	77	4	luatex
<code>\Usubscript</code>	<code>math_script</code>	77	1	luatex
<code>\Usuperprescript</code>	<code>math_script</code>	77	3	luatex
<code>\Usuperscript</code>	<code>math_script</code>	77	2	luatex
<code>\Uunderdelimiter</code>	<code>math_radical</code>	76	3	luatex
<code>\Uvextensible</code>	<code>math_fence</code>	59	5	luatex
<code>\above</code>	<code>math_fraction</code>	62	0	tex
<code>\abovedisplayshortskip</code>	<code>internal_glue</code>	92	0	tex
<code>\abovedisplayskip</code>	<code>internal_glue</code>	92	0	tex



<code>\abovewithdelims</code>	<code>math_fraction</code>	62	1	tex
<code>\accent</code>	<code>accent</code>	55	0	tex
<code>\adjdemerits</code>	<code>internal_int</code>	86	0	tex
<code>\adjustspacing</code>	<code>internal_int</code>	86	0	luatex
<code>\adjustspacingshrink</code>	<code>internal_int</code>	86	0	luatex
<code>\adjustspacingstep</code>	<code>internal_int</code>	86	0	luatex
<code>\adjustspacingstretch</code>	<code>internal_int</code>	86	0	luatex
<code>\advance</code>	<code>arithmic</code>	115	0	tex
<code>\afterassigned</code>	<code>after_something</code>	51	4	luatex
<code>\afterassignment</code>	<code>after_something</code>	51	1	tex
<code>\aftergroup</code>	<code>after_something</code>	51	0	tex
<code>\aftergrouped</code>	<code>after_something</code>	51	3	luatex
<code>\aliased</code>	<code>prefix</code>	116	11	luatex
<code>\aligncontent</code>	<code>end_template</code>	18	3	tex
<code>\alignmark</code>	<code>parameter</code>	6	0	luatex
<code>\aligntab</code>	<code>alignment_tab</code>	4	0	luatex
<code>\allcrampedstyles</code>	<code>math_style</code>	63	16	luatex
<code>\alldisplaystyles</code>	<code>math_style</code>	63	9	luatex
<code>\allmathstyles</code>	<code>math_style</code>	63	13	luatex
<code>\allscriptscriptstyles</code>	<code>math_style</code>	63	12	luatex
<code>\allscriptstyles</code>	<code>math_style</code>	63	11	luatex
<code>\allsplitstyles</code>	<code>math_style</code>	63	14	luatex
<code>\alltextstyles</code>	<code>math_style</code>	63	10	luatex
<code>\alluncrampedstyles</code>	<code>math_style</code>	63	15	luatex
<code>\atendofgroup</code>	<code>after_something</code>	51	2	luatex
<code>\atendofgrouped</code>	<code>after_something</code>	51	5	luatex
<code>\atop</code>	<code>math_fraction</code>	62	4	tex
<code>\atopwithdelims</code>	<code>math_fraction</code>	62	5	tex
<code>\attribute</code>	<code>register</code>	113	1	luatex
<code>\attributedef</code>	<code>shorthand_def</code>	118	5	luatex
<code>\automaticdiscretionary</code>	<code>discretionary</code>	57	2	luatex
<code>\automatichyphenpenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\automigrationmode</code>	<code>internal_int</code>	86	0	luatex
<code>\autoparagraphmode</code>	<code>internal_int</code>	86	0	luatex
<code>\badness</code>	<code>some_item</code>	83	6	tex
<code>\baselineskip</code>	<code>internal_glue</code>	92	0	tex
<code>\batchmode</code>	<code>set_interaction</code>	122	0	tex
<code>\begincsname</code>	<code>cs_name</code>	131	2	luatex
<code>\begingroup</code>	<code>begin_group</code>	72	0	tex
<code>\beginlocalcontrol</code>	<code>begin_local</code>	129	0	luatex
<code>\beginmathgroup</code>	<code>begin_group</code>	72	2	luatex
<code>\beginsimplegroup</code>	<code>begin_group</code>	72	1	luatex
<code>\belowdisplayshortskip</code>	<code>internal_glue</code>	92	0	tex
<code>\belowdisplayskip</code>	<code>internal_glue</code>	92	0	tex
<code>\binoppenalty</code>	<code>internal_int</code>	86	0	tex
<code>\botmark</code>	<code>get_mark</code>	134	8	tex



\botmarks	get_mark	134	3	etex
\boundary	boundary	75	1	luatex
\box	make_box	30	0	tex
\boxanchor	set_box_property	101	6	luatex
\boxanchors	set_box_property	101	7	luatex
\boxattribute	set_box_property	101	16	luatex
\boxdirection	set_box_property	101	3	luatex
\boxgeometry	set_box_property	101	4	luatex
\boxmaxdepth	internal_dimen	90	0	tex
\boxorientation	set_box_property	101	5	luatex
\boxshift	set_box_property	101	15	luatex
\boxsource	set_box_property	101	8	luatex
\boxtarget	set_box_property	101	9	luatex
\boxtotal	set_box_property	101	14	luatex
\boxxmove	set_box_property	101	12	luatex
\boxxoffset	set_box_property	101	10	luatex
\boxymove	set_box_property	101	13	luatex
\boxyoffset	set_box_property	101	11	luatex
\brokenpenalty	internal_int	86	0	tex
\catcode	define_char_code	103	0	tex
\catcodetable	internal_int	86	0	luatex
\char	char_number	25	0	tex
\chardef	shorthand_def	118	0	tex
\cleaders	leader	41	1	tex
\clearmarks	set_mark	27	2	luatex
\clubpenalties	set_specification	102	0	etex
\clubpenalty	internal_int	86	0	tex
\copy	make_box	30	1	tex
\copymathatomrule	set_math_parameter	105	8456	luatex
\copymathparent	set_math_parameter	105	8458	luatex
\copymathspacing	set_math_parameter	105	8453	luatex
\count	register	113	0	tex
\countdef	shorthand_def	118	4	tex
\cr	end_template	18	5	tex
\crampeddisplaystyle	math_style	63	1	luatex
\crampedscriptscriptstyle	math_style	63	7	luatex
\crampedscriptstyle	math_style	63	5	luatex
\crampedtextstyle	math_style	63	3	luatex
\crrc	end_template	18	6	tex
\csname	cs_name	131	0	tex
\csstring	convert	132	11	luatex
\currentgrouplevel	some_item	83	10	etex
\currentgrouptype	some_item	83	11	etex
\currentifbranch	some_item	83	14	etex
\currentiflevel	some_item	83	12	etex
\currentiftype	some_item	83	13	etex





<code>\currentloopiterator</code>	<code>some_item</code>	83	66	luatex
<code>\currentloopnesting</code>	<code>some_item</code>	83	67	luatex
<code>\currentmarks</code>	<code>get_mark</code>	134	0	luatex
<code>\day</code>	<code>internal_int</code>	86	0	tex
<code>\deadcycles</code>	<code>set_page_property</code>	100	9	tex
<code>\def</code>	<code>def</code>	119	1	tex
<code>\defaultthyphenchar</code>	<code>internal_int</code>	86	0	tex
<code>\defaultskewchar</code>	<code>internal_int</code>	86	0	tex
<code>\defcsname</code>	<code>def</code>	119	5	luatex
<code>\delcode</code>	<code>define_char_code</code>	103	8	tex
<code>\delimiter</code>	<code>delimiter_number</code>	24	0	tex
<code>\delimiterfactor</code>	<code>internal_int</code>	86	0	tex
<code>\delimitershortfall</code>	<code>internal_dimen</code>	90	0	tex
<code>\detokenize</code>	<code>the</code>	133	2	etex
<code>\dimen</code>	<code>register</code>	113	2	tex
<code>\dimendef</code>	<code>shorthand_def</code>	118	6	tex
<code>\dimensiondef</code>	<code>shorthand_def</code>	118	12	luatex
<code>\dimexpr</code>	<code>some_item</code>	83	56	etex
<code>\dimexpression</code>	<code>some_item</code>	83	60	luatex
<code>\directlua</code>	<code>convert</code>	132	5	luatex
<code>\discretionary</code>	<code>discretionary</code>	57	0	tex
<code>\displayindent</code>	<code>internal_dimen</code>	90	0	tex
<code>\displaylimits</code>	<code>math_modifier</code>	61	0	tex
<code>\displaystyle</code>	<code>math_style</code>	63	0	tex
<code>\displaywidowpenalties</code>	<code>set_specification</code>	102	0	etex
<code>\displaywidowpenalty</code>	<code>internal_int</code>	86	0	tex
<code>\displaywidth</code>	<code>internal_dimen</code>	90	0	tex
<code>\divide</code>	<code>arithmic</code>	115	2	tex
<code>\doublehyphendemerits</code>	<code>internal_int</code>	86	0	tex
<code>\dp</code>	<code>set_box_property</code>	101	2	tex
<code>\dump</code>	<code>end_job</code>	23	1	tex
<code>\edef</code>	<code>def</code>	119	0	tex
<code>\edefcsname</code>	<code>def</code>	119	4	luatex
<code>\efcode</code>	<code>set_font_property</code>	98	4	luatex
<code>\else</code>	<code>if_test</code>	130	3	tex
<code>\emergencystretch</code>	<code>internal_dimen</code>	90	0	tex
<code>\end</code>	<code>end_job</code>	23	0	tex
<code>\endcsname</code>	<code>end_cs_name</code>	79	0	tex
<code>\endgroup</code>	<code>end_group</code>	73	0	tex
<code>\endinput</code>	<code>input</code>	126	1	tex
<code>\endlinechar</code>	<code>internal_int</code>	86	0	tex
<code>\endlocalcontrol</code>	<code>end_local</code>	69	0	luatex
<code>\endmathgroup</code>	<code>end_group</code>	73	2	luatex
<code>\endsimplegroup</code>	<code>end_group</code>	73	1	luatex
<code>\enforced</code>	<code>prefix</code>	116	14	luatex
<code>\eqno</code>	<code>equation_number</code>	58	1	tex



<code>\errhelp</code>	<code>internal_toks</code>	84	0	tex
<code>\errmessage</code>	<code>message</code>	67	1	tex
<code>\errorcontextlines</code>	<code>internal_int</code>	86	0	tex
<code>\errorstopmode</code>	<code>set_interaction</code>	122	3	tex
<code>\escapechar</code>	<code>internal_int</code>	86	0	tex
<code>\etoks</code>	<code>combine_toks</code>	114	0	luatex
<code>\etoksapp</code>	<code>combine_toks</code>	114	2	luatex
<code>\etokspre</code>	<code>combine_toks</code>	114	4	luatex
<code>\everybeforepar</code>	<code>internal_toks</code>	84	0	luatex
<code>\everycr</code>	<code>internal_toks</code>	84	0	tex
<code>\everydisplay</code>	<code>internal_toks</code>	84	0	tex
<code>\everyeof</code>	<code>internal_toks</code>	84	0	etex
<code>\everyhbox</code>	<code>internal_toks</code>	84	0	tex
<code>\everyjob</code>	<code>internal_toks</code>	84	0	tex
<code>\everymath</code>	<code>internal_toks</code>	84	0	tex
<code>\everypar</code>	<code>internal_toks</code>	84	0	tex
<code>\everytab</code>	<code>internal_toks</code>	84	0	luatex
<code>\everyvbox</code>	<code>internal_toks</code>	84	0	tex
<code>\exceptionpenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\exhyphenchar</code>	<code>internal_int</code>	86	0	tex
<code>\exhyphenpenalty</code>	<code>internal_int</code>	86	0	tex
<code>\expand</code>	<code>expand_after</code>	124	9	luatex
<code>\expandafter</code>	<code>expand_after</code>	124	0	tex
<code>\expandafterpars</code>	<code>expand_after</code>	124	6	luatex
<code>\expandafterspaces</code>	<code>expand_after</code>	124	5	luatex
<code>\expandcstoken</code>	<code>expand_after</code>	124	8	luatex
<code>\expanded</code>	<code>convert</code>	132	8	luatex
<code>\expandedafter</code>	<code>expand_after</code>	124	10	luatex
<code>\expandedloop</code>	<code>begin_local</code>	129	4	luatex
<code>\expandtoken</code>	<code>expand_after</code>	124	7	luatex
<code>\explicitdiscretionary</code>	<code>discretionary</code>	57	1	luatex
<code>\explicithyphenpenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\fam</code>	<code>internal_int</code>	86	0	tex
<code>\fi</code>	<code>if_test</code>	130	2	tex
<code>\finalhyphendemerits</code>	<code>internal_int</code>	86	0	tex
<code>\firstmark</code>	<code>get_mark</code>	134	7	tex
<code>\firstmarks</code>	<code>get_mark</code>	134	2	etex
<code>\firstvalidlanguage</code>	<code>internal_int</code>	86	0	luatex
<code>\floatingpenalty</code>	<code>internal_int</code>	86	0	tex
<code>\flushmarks</code>	<code>set_mark</code>	27	3	luatex
<code>\font</code>	<code>define_font</code>	107	0	tex
<code>\fontchardp</code>	<code>some_item</code>	83	20	etex
<code>\fontcharht</code>	<code>some_item</code>	83	19	etex
<code>\fontcharic</code>	<code>some_item</code>	83	21	etex
<code>\fontcharwd</code>	<code>some_item</code>	83	18	etex
<code>\fontdimen</code>	<code>set_font_property</code>	98	5	tex



\fontid	some_item	83	17	luatex
\fontmathcontrol	some_item	83	27	luatex
\fontname	convert	132	19	tex
\fontspecdef	shorthand_def	118	15	luatex
\fontspecid	some_item	83	22	luatex
\fontspecifiedname	convert	132	20	tex
\fontspecifiedsize	some_item	83	26	luatex
\fontspecscale	some_item	83	23	luatex
\fontspecxscale	some_item	83	24	luatex
\fontspecyscale	some_item	83	25	luatex
\fonttextcontrol	some_item	83	28	luatex
\formatname	convert	132	22	luatex
\frozen	prefix	116	0	luatex
\futurecsname	cs_name	131	3	luatex
\futuredef	let	117	3	luatex
\futureexpand	expand_after	124	2	luatex
\futureexpandis	expand_after	124	3	luatex
\futureexpandisap	expand_after	124	4	luatex
\futurelet	let	117	2	tex
\gdef	def	119	3	tex
\gdefcsname	def	119	7	luatex
\gleaders	leader	41	3	luatex
\glet	let	117	0	luatex
\gletcsname	let	117	11	luatex
\glettonothing	let	117	13	luatex
\global	prefix	116	7	tex
\globaldefs	internal_int	86	0	tex
\glueexpr	some_item	83	57	etex
\glueshrink	some_item	83	52	etex
\glueshrinkorder	some_item	83	16	etex
\gluespecdef	shorthand_def	118	13	luatex
\gluestretch	some_item	83	51	etex
\gluestretchorder	some_item	83	15	etex
\gluetomu	some_item	83	54	etex
\glyph	char_number	25	1	tex
\glyphdatafield	internal_int	86	0	luatex
\glyphoptions	internal_int	86	0	luatex
\glyphscale	internal_int	86	0	luatex
\glyphscriptfield	internal_int	86	0	luatex
\glyphscriptscale	internal_int	86	0	luatex
\glyphscriptscriptscale	internal_int	86	0	luatex
\glyphstatefield	internal_int	86	0	luatex
\glyphtextscale	internal_int	86	0	luatex
\glyphxoffset	internal_dimen	90	0	luatex
\glyphxscale	internal_int	86	0	luatex
\glyphyoffset	internal_dimen	90	0	luatex



<code>\glyphscale</code>	<code>internal_int</code>	86	0	luatex
<code>\gtoksapp</code>	<code>combine_toks</code>	114	6	luatex
<code>\gtokspre</code>	<code>combine_toks</code>	114	8	luatex
<code>\halign</code>	<code>halign</code>	44	0	tex
<code>\hangafter</code>	<code>internal_int</code>	86	0	tex
<code>\hangindent</code>	<code>internal_dimen</code>	90	0	tex
<code>\hbadness</code>	<code>internal_int</code>	86	0	tex
<code>\hbox</code>	<code>make_box</code>	30	10	tex
<code>\hccode</code>	<code>define_char_code</code>	103	4	luatex
<code>\hfil</code>	<code>hskip</code>	36	0	tex
<code>\hfill</code>	<code>hskip</code>	36	1	tex
<code>\hfilneg</code>	<code>hskip</code>	36	3	tex
<code>\hfuzz</code>	<code>internal_dimen</code>	90	0	tex
<code>\hjcode</code>	<code>hyphenation</code>	121	7	luatex
<code>\holdinginserts</code>	<code>internal_int</code>	86	0	tex
<code>\hpack</code>	<code>make_box</code>	30	7	luatex
<code>\hrule</code>	<code>hrule</code>	47	0	tex
<code>\hsize</code>	<code>internal_dimen</code>	90	0	tex
<code>\hskip</code>	<code>hskip</code>	36	4	tex
<code>\hss</code>	<code>hskip</code>	36	2	tex
<code>\ht</code>	<code>set_box_property</code>	101	1	tex
<code>\hyphenation</code>	<code>hyphenation</code>	121	0	tex
<code>\hyphenationmin</code>	<code>hyphenation</code>	121	6	luatex
<code>\hyphenationmode</code>	<code>internal_int</code>	86	0	luatex
<code>\hyphenchar</code>	<code>set_font_property</code>	98	0	tex
<code>\hyphenpenalty</code>	<code>internal_int</code>	86	0	tex
<code>\if</code>	<code>if_test</code>	130	7	tex
<code>\ifabsdim</code>	<code>if_test</code>	130	11	luatex
<code>\ifabsnum</code>	<code>if_test</code>	130	9	luatex
<code>\ifarguments</code>	<code>if_test</code>	130	46	luatex
<code>\ifboolean</code>	<code>if_test</code>	130	41	luatex
<code>\ifcase</code>	<code>if_test</code>	130	32	tex
<code>\ifcat</code>	<code>if_test</code>	130	8	tex
<code>\ifchkdim</code>	<code>if_test</code>	130	29	luatex
<code>\ifchknum</code>	<code>if_test</code>	130	26	luatex
<code>\ifcmpdim</code>	<code>if_test</code>	130	31	luatex
<code>\ifcmpnum</code>	<code>if_test</code>	130	28	luatex
<code>\ifcondition</code>	<code>if_test</code>	130	37	luatex
<code>\ifcsname</code>	<code>if_test</code>	130	34	etex
<code>\ifcstok</code>	<code>if_test</code>	130	22	luatex
<code>\ifdefined</code>	<code>if_test</code>	130	33	etex
<code>\ifdim</code>	<code>if_test</code>	130	12	tex
<code>\ifdimexpression</code>	<code>if_test</code>	130	43	luatex
<code>\ifdimval</code>	<code>if_test</code>	130	30	luatex
<code>\ifempty</code>	<code>if_test</code>	130	39	luatex
<code>\iffalse</code>	<code>if_test</code>	130	25	tex



<code>\iflags</code>	<code>if_test</code>	130	38	luatex
<code>\iffontchar</code>	<code>if_test</code>	130	36	etex
<code>\ifhaschar</code>	<code>if_test</code>	130	52	luatex
<code>\ifhastok</code>	<code>if_test</code>	130	49	luatex
<code>\ifhastoks</code>	<code>if_test</code>	130	50	luatex
<code>\ifhasxtoks</code>	<code>if_test</code>	130	51	luatex
<code>\ifhbox</code>	<code>if_test</code>	130	19	tex
<code>\ifhmode</code>	<code>if_test</code>	130	15	tex
<code>\ifincsname</code>	<code>if_test</code>	130	35	luatex
<code>\ifinner</code>	<code>if_test</code>	130	17	tex
<code>\ifinsert</code>	<code>if_test</code>	130	53	luatex
<code>\ifmathparameter</code>	<code>if_test</code>	130	44	luatex
<code>\ifmathstyle</code>	<code>if_test</code>	130	45	luatex
<code>\ifmmode</code>	<code>if_test</code>	130	16	tex
<code>\ifnum</code>	<code>if_test</code>	130	10	tex
<code>\ifnumexpression</code>	<code>if_test</code>	130	42	luatex
<code>\ifnumval</code>	<code>if_test</code>	130	27	luatex
<code>\ifodd</code>	<code>if_test</code>	130	13	tex
<code>\ifparameter</code>	<code>if_test</code>	130	48	luatex
<code>\ifparameters</code>	<code>if_test</code>	130	47	luatex
<code>\ifrelax</code>	<code>if_test</code>	130	40	luatex
<code>\iftok</code>	<code>if_test</code>	130	21	luatex
<code>\iftrue</code>	<code>if_test</code>	130	24	tex
<code>\ifvbox</code>	<code>if_test</code>	130	20	tex
<code>\ifvmode</code>	<code>if_test</code>	130	14	tex
<code>\ifvoid</code>	<code>if_test</code>	130	18	tex
<code>\ifx</code>	<code>if_test</code>	130	23	tex
<code>\ignorearguments</code>	<code>ignore_something</code>	50	2	luatex
<code>\ignorepars</code>	<code>ignore_something</code>	50	1	luatex
<code>\ignorespaces</code>	<code>ignore_something</code>	50	0	tex
<code>\immediate</code>	<code>prefix</code>	116	12	luatex
<code>\immutable</code>	<code>prefix</code>	116	2	luatex
<code>\indent</code>	<code>begin_paragraph</code>	53	1	tex
<code>\inherited</code>	<code>prefix</code>	116	16	luatex
<code>\initcatcodetable</code>	<code>catcode_table</code>	68	1	luatex
<code>\input</code>	<code>input</code>	126	0	tex
<code>\inputlineno</code>	<code>some_item</code>	83	5	tex
<code>\insert</code>	<code>insert</code>	48	0	tex
<code>\insertbox</code>	<code>make_box</code>	30	11	luatex
<code>\insertcopy</code>	<code>make_box</code>	30	12	luatex
<code>\insertdepth</code>	<code>set_page_property</code>	100	20	luatex
<code>\insertdistance</code>	<code>set_page_property</code>	100	13	luatex
<code>\insertheight</code>	<code>set_page_property</code>	100	19	luatex
<code>\insertheights</code>	<code>set_page_property</code>	100	11	luatex
<code>\insertlimit</code>	<code>set_page_property</code>	100	15	luatex
<code>\insertmaxdepth</code>	<code>set_page_property</code>	100	18	luatex



<code>\insertmode</code>	<code>set_auxiliary</code>	99	4	luatex
<code>\insertmultiplier</code>	<code>set_page_property</code>	100	14	luatex
<code>\insertpenalties</code>	<code>set_page_property</code>	100	10	tex
<code>\insertpenalty</code>	<code>set_page_property</code>	100	17	luatex
<code>\insertprogress</code>	<code>some_item</code>	83	45	luatex
<code>\insertstorage</code>	<code>set_page_property</code>	100	16	luatex
<code>\insertstoring</code>	<code>set_page_property</code>	100	12	luatex
<code>\insertunbox</code>	<code>un_vbox</code>	34	11	luatex
<code>\insertuncopy</code>	<code>un_vbox</code>	34	12	luatex
<code>\insertwidth</code>	<code>set_page_property</code>	100	21	luatex
<code>\instance</code>	<code>prefix</code>	116	5	luatex
<code>\integerdef</code>	<code>shorthand_def</code>	118	11	luatex
<code>\interactionmode</code>	<code>set_auxiliary</code>	99	3	etex
<code>\interlinepenalties</code>	<code>set_specification</code>	102	0	etex
<code>\interlinepenalty</code>	<code>internal_int</code>	86	0	tex
<code>\jobname</code>	<code>convert</code>	132	21	tex
<code>\kern</code>	<code>kern</code>	39	0	tex
<code>\language</code>	<code>internal_int</code>	86	0	tex
<code>\lastarguments</code>	<code>some_item</code>	83	43	luatex
<code>\lastbox</code>	<code>make_box</code>	30	3	tex
<code>\lastchkdim</code>	<code>some_item</code>	83	62	luatex
<code>\lastchknun</code>	<code>some_item</code>	83	61	luatex
<code>\lastkern</code>	<code>some_item</code>	83	1	tex
<code>\lastleftclass</code>	<code>some_item</code>	83	64	luatex
<code>\lastlinefit</code>	<code>internal_int</code>	86	0	etex
<code>\lastloopiterator</code>	<code>some_item</code>	83	68	luatex
<code>\lastnamedcs</code>	<code>cs_name</code>	131	1	luatex
<code>\lastnodesubtype</code>	<code>some_item</code>	83	4	luatex
<code>\lastnodetype</code>	<code>some_item</code>	83	3	etex
<code>\lastparcontext</code>	<code>some_item</code>	83	69	luatex
<code>\lastpenalty</code>	<code>some_item</code>	83	0	tex
<code>\lastrightclass</code>	<code>some_item</code>	83	65	luatex
<code>\lastskip</code>	<code>some_item</code>	83	2	tex
<code>\lccode</code>	<code>define_char_code</code>	103	1	tex
<code>\leaders</code>	<code>leader</code>	41	0	tex
<code>\left</code>	<code>math_fence</code>	59	1	tex
<code>\lefthyphenmin</code>	<code>internal_int</code>	86	0	tex
<code>\leftmarginkern</code>	<code>some_item</code>	83	46	luatex
<code>\leftskip</code>	<code>internal_glue</code>	92	0	tex
<code>\leqno</code>	<code>equation_number</code>	58	0	tex
<code>\let</code>	<code>let</code>	117	1	tex
<code>\letcharcode</code>	<code>let</code>	117	4	luatex
<code>\letcsname</code>	<code>let</code>	117	10	luatex
<code>\letfrozen</code>	<code>let</code>	117	8	luatex
<code>\letmathatomrule</code>	<code>set_math_parameter</code>	105	8455	luatex
<code>\letmathparent</code>	<code>set_math_parameter</code>	105	8457	luatex



<code>\letmathspacing</code>	<code>set_math_parameter</code>	105	8452	luatex
<code>\letprotected</code>	<code>let</code>	117	6	luatex
<code>\lettonothing</code>	<code>let</code>	117	12	luatex
<code>\limits</code>	<code>math_modifier</code>	61	1	tex
<code>\linedirection</code>	<code>internal_int</code>	86	0	luatex
<code>\linepenalty</code>	<code>internal_int</code>	86	0	tex
<code>\lineskip</code>	<code>internal_glue</code>	92	0	tex
<code>\lineskiplimit</code>	<code>internal_dimen</code>	90	0	tex
<code>\localbrokenpenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\localcontrol</code>	<code>begin_local</code>	129	1	luatex
<code>\localcontrolled</code>	<code>begin_local</code>	129	2	luatex
<code>\localcontrolledloop</code>	<code>begin_local</code>	129	3	luatex
<code>\localinterlinepenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\lcalleftbox</code>	<code>local_box</code>	43	0	luatex
<code>\lcalleftboxbox</code>	<code>make_box</code>	30	13	luatex
<code>\lcalmiddlebox</code>	<code>local_box</code>	43	2	luatex
<code>\lcalmiddleboxbox</code>	<code>make_box</code>	30	15	luatex
<code>\lcalrightbox</code>	<code>local_box</code>	43	1	luatex
<code>\lcalrightboxbox</code>	<code>make_box</code>	30	14	luatex
<code>\long</code>	<code>prefix</code>	116	17	tex
<code>\looseness</code>	<code>internal_int</code>	86	0	tex
<code>\lower</code>	<code>vmove</code>	32	0	tex
<code>\lowercase</code>	<code>case_shift</code>	66	0	tex
<code>\lpcode</code>	<code>set_font_property</code>	98	2	luatex
<code>\luabytecode</code>	<code>convert</code>	132	7	luatex
<code>\luabytecodecall</code>	<code>lua_function_call</code>	70	1	luatex
<code>\luacopyinputnodes</code>	<code>internal_int</code>	86	0	luatex
<code>\luadef</code>	<code>shorthand_def</code>	118	10	luatex
<code>\luaescapestring</code>	<code>convert</code>	132	18	luatex
<code>\luafunction</code>	<code>convert</code>	132	6	luatex
<code>\luafunctioncall</code>	<code>lua_function_call</code>	70	0	luatex
<code>\luatexbanner</code>	<code>convert</code>	132	23	luatex
<code>\luatexrevision</code>	<code>some_item</code>	83	9	luatex
<code>\luatexversion</code>	<code>some_item</code>	83	8	luatex
<code>\mark</code>	<code>set_mark</code>	27	0	tex
<code>\marks</code>	<code>set_mark</code>	27	1	etex
<code>\mathaccent</code>	<code>math_component</code>	60	14	luatex
<code>\mathatom</code>	<code>math_component</code>	60	17	luatex
<code>\mathatomskip</code>	<code>mskip</code>	38	1	luatex
<code>\mathbackwardpenalties</code>	<code>set_specification</code>	102	0	luatex
<code>\mathbin</code>	<code>math_component</code>	60	2	tex
<code>\mathchar</code>	<code>math_char_number</code>	26	0	tex
<code>\mathchardef</code>	<code>shorthand_def</code>	118	1	tex
<code>\mathchoice</code>	<code>math_choice</code>	64	0	tex
<code>\mathclose</code>	<code>math_component</code>	60	5	tex
<code>\mathcode</code>	<code>define_char_code</code>	103	5	tex



<code>\mathdelimitersmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathdirection</code>	<code>internal_int</code>	86	0	luatex
<code>\mathdisplayskipmode</code>	<code>internal_int</code>	86	0	luatex
<code>\matheqnogapstep</code>	<code>internal_int</code>	86	0	luatex
<code>\mathfenced</code>	<code>math_component</code>	60	15	luatex
<code>\mathfencesmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathfontcontrol</code>	<code>internal_int</code>	86	0	luatex
<code>\mathforwardpenalties</code>	<code>set_specification</code>	102	0	luatex
<code>\mathfrac</code>	<code>math_component</code>	60	11	luatex
<code>\mathghost</code>	<code>math_component</code>	60	16	luatex
<code>\mathinner</code>	<code>math_component</code>	60	8	tex
<code>\mathlimitsmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathmiddle</code>	<code>math_component</code>	60	13	luatex
<code>\mathnolimitsmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathop</code>	<code>math_component</code>	60	1	tex
<code>\mathopen</code>	<code>math_component</code>	60	4	tex
<code>\mathord</code>	<code>math_component</code>	60	0	tex
<code>\mathpenaltiesmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathpunct</code>	<code>math_component</code>	60	6	tex
<code>\mathrad</code>	<code>math_component</code>	60	12	luatex
<code>\mathrel</code>	<code>math_component</code>	60	3	tex
<code>\mathrulesfam</code>	<code>internal_int</code>	86	0	luatex
<code>\mathrulesmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathscale</code>	<code>some_item</code>	83	29	luatex
<code>\mathscriptboxmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathscriptcharmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathscriptsmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathslackmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathspacingmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathstackstyle</code>	<code>some_item</code>	83	32	luatex
<code>\mathstyle</code>	<code>some_item</code>	83	30	luatex
<code>\mathstylefontid</code>	<code>some_item</code>	83	31	luatex
<code>\mathsurround</code>	<code>internal_dimen</code>	90	0	tex
<code>\mathsurroundmode</code>	<code>internal_int</code>	86	0	luatex
<code>\mathsurroundskip</code>	<code>internal_glue</code>	92	0	luatex
<code>\maththreshold</code>	<code>internal_glue</code>	92	0	luatex
<code>\maxdeadcycles</code>	<code>internal_int</code>	86	0	tex
<code>\maxdepth</code>	<code>internal_dimen</code>	90	0	tex
<code>\meaning</code>	<code>convert</code>	132	13	tex
<code>\meaningasis</code>	<code>convert</code>	132	16	tex
<code>\meaningfull</code>	<code>convert</code>	132	14	tex
<code>\meaningless</code>	<code>convert</code>	132	15	tex
<code>\medmuskip</code>	<code>internal_mu_glue</code>	94	1	tex
<code>\message</code>	<code>message</code>	67	0	tex
<code>\middle</code>	<code>math_fence</code>	59	2	tex
<code>\mkern</code>	<code>mkern</code>	40	0	tex





<code>\month</code>	<code>internal_int</code>	86	0	tex
<code>\moveleft</code>	<code>hmove</code>	31	1	tex
<code>\moveright</code>	<code>hmove</code>	31	0	tex
<code>\mskip</code>	<code>mskip</code>	38	0	tex
<code>\muexpr</code>	<code>some_item</code>	83	58	etex
<code>\mugluespecdef</code>	<code>shorthand_def</code>	118	14	luatex
<code>\multiply</code>	<code>arithmic</code>	115	1	tex
<code>\muskip</code>	<code>register</code>	113	4	tex
<code>\muskipdef</code>	<code>shorthand_def</code>	118	8	tex
<code>\mutable</code>	<code>prefix</code>	116	3	luatex
<code>\mutoglua</code>	<code>some_item</code>	83	53	etex
<code>\newlinechar</code>	<code>internal_int</code>	86	0	tex
<code>\noalign</code>	<code>end_template</code>	18	4	tex
<code>\noaligned</code>	<code>prefix</code>	116	4	luatex
<code>\noboundary</code>	<code>boundary</code>	75	0	luatex
<code>\noexpand</code>	<code>no_expand</code>	125	0	tex
<code>\nohrule</code>	<code>hrule</code>	47	1	luatex
<code>\noindent</code>	<code>begin_paragraph</code>	53	0	tex
<code>\nolimits</code>	<code>math_modifier</code>	61	2	tex
<code>\nonscript</code>	<code>math_script</code>	77	0	tex
<code>\nonstopmode</code>	<code>set_interaction</code>	122	1	tex
<code>\norelax</code>	<code>relax</code>	16	1	luatex
<code>\normalizelinemode</code>	<code>internal_int</code>	86	0	luatex
<code>\nospaces</code>	<code>internal_int</code>	86	0	luatex
<code>\novrule</code>	<code>vrule</code>	46	1	luatex
<code>\nulldelimiterspace</code>	<code>internal_dimen</code>	90	0	tex
<code>\nullfont</code>	<code>set_font</code>	106	0	tex
<code>\number</code>	<code>convert</code>	132	0	tex
<code>\numericsscale</code>	<code>some_item</code>	83	63	luatex
<code>\numexpr</code>	<code>some_item</code>	83	55	etex
<code>\numexpression</code>	<code>some_item</code>	83	59	luatex
<code>\omit</code>	<code>end_template</code>	18	2	tex
<code>\or</code>	<code>if_test</code>	130	4	tex
<code>\orelse</code>	<code>if_test</code>	130	5	luatex
<code>\orphanpenalties</code>	<code>set_specification</code>	102	0	luatex
<code>\orphanpenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\orunless</code>	<code>if_test</code>	130	6	luatex
<code>\outer</code>	<code>prefix</code>	116	18	tex
<code>\output</code>	<code>internal_toks</code>	84	0	tex
<code>\outputbox</code>	<code>internal_int</code>	86	0	luatex
<code>\outputpenalty</code>	<code>internal_int</code>	86	0	tex
<code>\over</code>	<code>math_fraction</code>	62	2	tex
<code>\overfullrule</code>	<code>internal_dimen</code>	90	0	tex
<code>\overline</code>	<code>math_component</code>	60	10	tex
<code>\overloaded</code>	<code>prefix</code>	116	10	luatex
<code>\overloadmode</code>	<code>internal_int</code>	86	0	luatex



<code>\overshoot</code>	<code>some_item</code>	83	7	tex
<code>\overwithdelims</code>	<code>math_fraction</code>	62	3	tex
<code>\pageboundary</code>	<code>boundary</code>	75	4	luatex
<code>\pageboundarypenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\pagedepth</code>	<code>set_page_property</code>	100	7	tex
<code>\pagediscards</code>	<code>un_vbox</code>	34	3	etex
<code>\pagefillllstretch</code>	<code>set_page_property</code>	100	5	tex
<code>\pagefillstretch</code>	<code>set_page_property</code>	100	4	tex
<code>\pagefilstretch</code>	<code>set_page_property</code>	100	3	tex
<code>\pagegoal</code>	<code>set_page_property</code>	100	0	tex
<code>\pageshrink</code>	<code>set_page_property</code>	100	6	tex
<code>\pagestretch</code>	<code>set_page_property</code>	100	2	tex
<code>\pagetotal</code>	<code>set_page_property</code>	100	1	tex
<code>\pagevsize</code>	<code>set_page_property</code>	100	8	luatex
<code>\par</code>	<code>end_paragraph</code>	22	0	tex
<code>\parametercount</code>	<code>some_item</code>	83	44	luatex
<code>\parametermark</code>	<code>parameter</code>	6	0	luatex
<code>\parattribute</code>	<code>begin_paragraph</code>	53	5	luatex
<code>\pardirection</code>	<code>internal_int</code>	86	0	luatex
<code>\parfilllleftskip</code>	<code>internal_glue</code>	92	0	tex
<code>\parfillskip</code>	<code>internal_glue</code>	92	0	tex
<code>\parindent</code>	<code>internal_dimen</code>	90	0	tex
<code>\parshape</code>	<code>set_specification</code>	102	0	tex
<code>\parshapedimen</code>	<code>some_item</code>	83	50	etex
<code>\parshapeindent</code>	<code>some_item</code>	83	49	etex
<code>\parshapelength</code>	<code>some_item</code>	83	48	etex
<code>\parskip</code>	<code>internal_glue</code>	92	0	tex
<code>\patterns</code>	<code>hyphenation</code>	121	1	tex
<code>\pausing</code>	<code>internal_int</code>	86	0	tex
<code>\penalty</code>	<code>penalty</code>	52	0	tex
<code>\permanent</code>	<code>prefix</code>	116	1	luatex
<code>\postdisplaypenalty</code>	<code>internal_int</code>	86	0	tex
<code>\postexhyphenchar</code>	<code>hyphenation</code>	121	5	luatex
<code>\postthyphenchar</code>	<code>hyphenation</code>	121	3	luatex
<code>\postinlinepenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\prebinoppenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\predisplaydirection</code>	<code>internal_int</code>	86	0	etex
<code>\predisplaygapfactor</code>	<code>internal_int</code>	86	0	luatex
<code>\predisplaypenalty</code>	<code>internal_int</code>	86	0	tex
<code>\predisplaysize</code>	<code>internal_dimen</code>	90	0	tex
<code>\preexhyphenchar</code>	<code>hyphenation</code>	121	4	luatex
<code>\prehyphenchar</code>	<code>hyphenation</code>	121	2	luatex
<code>\preinlinepenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\prerelpenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\pretolerance</code>	<code>internal_int</code>	86	0	tex
<code>\prevdepth</code>	<code>set_auxiliary</code>	99	1	tex



<code>\prevgraf</code>	<code>set_auxiliary</code>	99	2	tex
<code>\protected</code>	<code>prefix</code>	116	9	etex
<code>\protrudechars</code>	<code>internal_int</code>	86	0	luatex
<code>\protrusionboundary</code>	<code>boundary</code>	75	2	luatex
<code>\pxdimen</code>	<code>internal_dimen</code>	90	0	luatex
<code>\quitloop</code>	<code>input</code>	126	6	luatex
<code>\quitvmode</code>	<code>begin_paragraph</code>	53	2	luatex
<code>\radical</code>	<code>math_radical</code>	76	0	tex
<code>\raise</code>	<code>vmove</code>	32	1	tex
<code>\relax</code>	<code>relax</code>	16	0	tex
<code>\relpenalty</code>	<code>internal_int</code>	86	0	tex
<code>\resetmathspacing</code>	<code>set_math_parameter</code>	105	8450	luatex
<code>\retokenized</code>	<code>input</code>	126	5	luatex
<code>\right</code>	<code>math_fence</code>	59	3	tex
<code>\righthyphenmin</code>	<code>internal_int</code>	86	0	tex
<code>\rightmarginkern</code>	<code>some_item</code>	83	47	luatex
<code>\rightskip</code>	<code>internal_glue</code>	92	0	tex
<code>\romannumeral</code>	<code>convert</code>	132	12	tex
<code>\rptcode</code>	<code>set_font_property</code>	98	3	luatex
<code>\savecatcodetable</code>	<code>catcode_table</code>	68	0	luatex
<code>\savingshyphcodes</code>	<code>internal_int</code>	86	0	etex
<code>\savingsvdiscards</code>	<code>internal_int</code>	86	0	etex
<code>\scaledemwidth</code>	<code>some_item</code>	83	41	luatex
<code>\scaledexheight</code>	<code>some_item</code>	83	40	luatex
<code>\scaledextraspaces</code>	<code>some_item</code>	83	42	luatex
<code>\scaledfontdimen</code>	<code>set_font_property</code>	98	6	tex
<code>\scaledinterwordshrink</code>	<code>some_item</code>	83	39	luatex
<code>\scaledinterwordspace</code>	<code>some_item</code>	83	37	luatex
<code>\scaledinterwordstretch</code>	<code>some_item</code>	83	38	luatex
<code>\scaledslantperpoint</code>	<code>some_item</code>	83	36	luatex
<code>\scantexttokens</code>	<code>input</code>	126	3	luatex
<code>\scantokens</code>	<code>input</code>	126	2	etex
<code>\scriptfont</code>	<code>define_family</code>	104	1	tex
<code>\scriptscriptfont</code>	<code>define_family</code>	104	2	tex
<code>\scriptscriptstyle</code>	<code>math_style</code>	63	6	tex
<code>\scriptspace</code>	<code>internal_dimen</code>	90	0	tex
<code>\scriptstyle</code>	<code>math_style</code>	63	4	tex
<code>\scrollmode</code>	<code>set_interaction</code>	122	2	tex
<code>\semiexpanded</code>	<code>convert</code>	132	9	luatex
<code>\semiprotected</code>	<code>prefix</code>	116	13	luatex
<code>\setbox</code>	<code>set_box</code>	120	0	tex
<code>\setdefaultmathcodes</code>	<code>set_math_parameter</code>	105	8463	luatex
<code>\setfontid</code>	<code>internal_int</code>	86	0	luatex
<code>\setlanguage</code>	<code>internal_int</code>	86	0	tex
<code>\setmathatomrule</code>	<code>set_math_parameter</code>	105	8454	luatex
<code>\setmathignore</code>	<code>set_math_parameter</code>	105	8461	luatex



<code>\setmathoptions</code>	<code>set_math_parameter</code>	105	8462	luatex
<code>\setmathpostpenalty</code>	<code>set_math_parameter</code>	105	8460	luatex
<code>\setmathprepenalty</code>	<code>set_math_parameter</code>	105	8459	luatex
<code>\setmathspacing</code>	<code>set_math_parameter</code>	105	8451	luatex
<code>\sfcode</code>	<code>define_char_code</code>	103	3	tex
<code>\shapingpenaltiesmode</code>	<code>internal_int</code>	86	0	luatex
<code>\shapingpenalty</code>	<code>internal_int</code>	86	0	luatex
<code>\shipout</code>	<code>legacy</code>	42	0	tex
<code>\show</code>	<code>xray</code>	29	0	tex
<code>\showbox</code>	<code>xray</code>	29	1	tex
<code>\showboxbreadth</code>	<code>internal_int</code>	86	0	tex
<code>\showboxdepth</code>	<code>internal_int</code>	86	0	tex
<code>\showgroups</code>	<code>xray</code>	29	4	etex
<code>\showifs</code>	<code>xray</code>	29	6	etex
<code>\showlists</code>	<code>xray</code>	29	3	tex
<code>\shownodedetails</code>	<code>internal_int</code>	86	0	tex
<code>\showthe</code>	<code>xray</code>	29	2	tex
<code>\showtokens</code>	<code>xray</code>	29	5	etex
<code>\skewchar</code>	<code>set_font_property</code>	98	1	tex
<code>\skewed</code>	<code>math_fraction</code>	62	6	luatex
<code>\skewedwithdelims</code>	<code>math_fraction</code>	62	7	luatex
<code>\skip</code>	<code>register</code>	113	3	tex
<code>\skipdef</code>	<code>shorthand_def</code>	118	7	tex
<code>\snapshotpar</code>	<code>begin_paragraph</code>	53	4	luatex
<code>\spacefactor</code>	<code>set_auxiliary</code>	99	0	tex
<code>\spaceskip</code>	<code>internal_glue</code>	92	0	tex
<code>\span</code>	<code>end_template</code>	18	1	tex
<code>\splitbotmark</code>	<code>get_mark</code>	134	10	tex
<code>\splitbotmarks</code>	<code>get_mark</code>	134	5	etex
<code>\splitdiscards</code>	<code>un_vbox</code>	34	4	etex
<code>\splitfirstmark</code>	<code>get_mark</code>	134	9	tex
<code>\splitfirstmarks</code>	<code>get_mark</code>	134	4	etex
<code>\splitmaxdepth</code>	<code>internal_dimen</code>	90	0	tex
<code>\splittopskip</code>	<code>internal_glue</code>	92	0	tex
<code>\srule</code>	<code>vrule</code>	46	2	tex
<code>\string</code>	<code>convert</code>	132	10	tex
<code>\supmarkmode</code>	<code>internal_int</code>	86	0	luatex
<code>\swapcsvalues</code>	<code>let</code>	117	5	luatex
<code>\tabsize</code>	<code>internal_dimen</code>	90	0	luatex
<code>\tabskip</code>	<code>internal_glue</code>	92	0	tex
<code>\textdirection</code>	<code>internal_int</code>	86	0	luatex
<code>\textfont</code>	<code>define_family</code>	104	0	tex
<code>\textstyle</code>	<code>math_style</code>	63	2	tex
<code>\the</code>	<code>the</code>	133	0	tex
<code>\thewithoutunit</code>	<code>the</code>	133	1	luatex
<code>\thickmuskip</code>	<code>internal_mu_glue</code>	94	1	tex



<code>\thinmuskip</code>	<code>internal_mu_glue</code>	94	1	tex
<code>\time</code>	<code>internal_int</code>	86	0	tex
<code>\tinymuskip</code>	<code>internal_mu_glue</code>	94	1	tex
<code>\todimension</code>	<code>convert</code>	132	3	luatex
<code>\tointeger</code>	<code>convert</code>	132	1	luatex
<code>\tokenized</code>	<code>input</code>	126	4	luatex
<code>\toks</code>	<code>register</code>	113	5	tex
<code>\toksapp</code>	<code>combine_toks</code>	114	1	luatex
<code>\toksdef</code>	<code>shorthand_def</code>	118	9	tex
<code>\tokspre</code>	<code>combine_toks</code>	114	3	luatex
<code>\tolerance</code>	<code>internal_int</code>	86	0	tex
<code>\tolerant</code>	<code>prefix</code>	116	8	luatex
<code>\tomathstyle</code>	<code>convert</code>	132	4	luatex
<code>\topmark</code>	<code>get_mark</code>	134	6	tex
<code>\topmarks</code>	<code>get_mark</code>	134	1	etex
<code>\topskip</code>	<code>internal_glue</code>	92	0	tex
<code>\toscaled</code>	<code>convert</code>	132	2	luatex
<code>\tpack</code>	<code>make_box</code>	30	5	luatex
<code>\tracingadjusts</code>	<code>internal_int</code>	86	0	luatex
<code>\tracingalignments</code>	<code>internal_int</code>	86	0	luatex
<code>\tracingassigns</code>	<code>internal_int</code>	86	0	etex
<code>\tracingcommands</code>	<code>internal_int</code>	86	0	tex
<code>\tracingexpressions</code>	<code>internal_int</code>	86	0	luatex
<code>\tracingfonts</code>	<code>internal_int</code>	86	0	luatex
<code>\tracingfullboxes</code>	<code>internal_int</code>	86	0	luatex
<code>\tracinggroups</code>	<code>internal_int</code>	86	0	etex
<code>\tracinghyphenation</code>	<code>internal_int</code>	86	0	luatex
<code>\tracingifs</code>	<code>internal_int</code>	86	0	etex
<code>\tracinginserts</code>	<code>internal_int</code>	86	0	luatex
<code>\tracinglevels</code>	<code>internal_int</code>	86	0	etex
<code>\tracinglostchars</code>	<code>internal_int</code>	86	0	tex
<code>\tracingmacros</code>	<code>internal_int</code>	86	0	tex
<code>\tracingmarks</code>	<code>internal_int</code>	86	0	luatex
<code>\tracingmath</code>	<code>internal_int</code>	86	0	luatex
<code>\tracingnesting</code>	<code>internal_int</code>	86	0	etex
<code>\tracingnodes</code>	<code>internal_int</code>	86	0	luatex
<code>\tracingonline</code>	<code>internal_int</code>	86	0	tex
<code>\tracingoutput</code>	<code>internal_int</code>	86	0	tex
<code>\tracingpages</code>	<code>internal_int</code>	86	0	tex
<code>\tracingparagraphs</code>	<code>internal_int</code>	86	0	tex
<code>\tracingrestores</code>	<code>internal_int</code>	86	0	tex
<code>\tracingstats</code>	<code>internal_int</code>	86	0	tex
<code>\uccode</code>	<code>define_char_code</code>	103	2	tex
<code>\uchyph</code>	<code>internal_int</code>	86	0	tex
<code>\uleaders</code>	<code>leader</code>	41	4	luatex
<code>\undent</code>	<code>begin_paragraph</code>	53	3	luatex



<code>\underline</code>	<code>math_component</code>	60	9	tex
<code>\unexpanded</code>	<code>the</code>	133	3	etex
<code>\unexpandedloop</code>	<code>begin_local</code>	129	5	luatex
<code>\unhbox</code>	<code>un_hbox</code>	33	0	tex
<code>\unhcopy</code>	<code>un_hbox</code>	33	1	tex
<code>\unhpack</code>	<code>un_hbox</code>	33	2	tex
<code>\unkern</code>	<code>remove_item</code>	35	0	tex
<code>\unless</code>	<code>expand_after</code>	124	1	etex
<code>\unletfrozen</code>	<code>let</code>	117	9	luatex
<code>\unletprotected</code>	<code>let</code>	117	7	luatex
<code>\unpenalty</code>	<code>remove_item</code>	35	1	tex
<code>\unskip</code>	<code>remove_item</code>	35	2	tex
<code>\untraced</code>	<code>prefix</code>	116	6	luatex
<code>\unvbox</code>	<code>un_vbox</code>	34	0	tex
<code>\unvcopy</code>	<code>un_vbox</code>	34	1	tex
<code>\unvpack</code>	<code>un_vbox</code>	34	2	tex
<code>\uppercase</code>	<code>case_shift</code>	66	1	tex
<code>\vadjust</code>	<code>vadjust</code>	49	0	tex
<code>\valign</code>	<code>valign</code>	45	0	tex
<code>\vbadness</code>	<code>internal_int</code>	86	0	tex
<code>\vbox</code>	<code>make_box</code>	30	9	tex
<code>\vcenter</code>	<code>vcenter</code>	65	0	tex
<code>\vfil</code>	<code>vskip</code>	37	0	tex
<code>\vfill</code>	<code>vskip</code>	37	1	tex
<code>\vfilneg</code>	<code>vskip</code>	37	3	tex
<code>\vfuzz</code>	<code>internal_dimen</code>	90	0	tex
<code>\vpack</code>	<code>make_box</code>	30	6	luatex
<code>\vrule</code>	<code>vrule</code>	46	0	tex
<code>\vsize</code>	<code>internal_dimen</code>	90	0	tex
<code>\vskip</code>	<code>vskip</code>	37	4	tex
<code>\vsplit</code>	<code>make_box</code>	30	4	tex
<code>\vss</code>	<code>vskip</code>	37	2	tex
<code>\vtop</code>	<code>make_box</code>	30	8	tex
<code>\wd</code>	<code>set_box_property</code>	101	0	tex
<code>\widowpenalties</code>	<code>set_specification</code>	102	0	etex
<code>\widowpenalty</code>	<code>internal_int</code>	86	0	tex
<code>\wordboundary</code>	<code>boundary</code>	75	3	luatex
<code>\wrapuppar</code>	<code>begin_paragraph</code>	53	6	luatex
<code>\xdef</code>	<code>def</code>	119	2	tex
<code>\xdefcsname</code>	<code>def</code>	119	6	luatex
<code>\xleaders</code>	<code>leader</code>	41	2	tex
<code>\xspaceskip</code>	<code>internal_glue</code>	92	0	tex
<code>\xtoks</code>	<code>combine_toks</code>	114	5	luatex
<code>\xtoksapp</code>	<code>combine_toks</code>	114	7	luatex
<code>\xtokspre</code>	<code>combine_toks</code>	114	9	luatex
<code>\year</code>	<code>internal_int</code>	86	0	tex

---



# Topics

## **a**

Aleph 32, 66  
adjust 138  
attributes 46, 157, 198

## **b**

banner 42  
boundaries 87  
boundary 143  
boxes 13, 46, 201  
    split 202  
building 131  
bytecodes 185

## **c**

callbacks 173  
    building pages 175  
    contributions 174, 177  
    dump 181  
    errors 181  
    files 182  
    fonts 177, 182  
    format file 174  
    hyphenation 179  
    inserts 175  
    job run 181  
    jobname 174  
    kerning 180  
    ligature building 179  
    linebreaks 176, 177  
    log file 174  
    math 180  
    opening files 174  
    output 179  
    packing 178, 179  
    rules 179  
    warnings 181  
    whatsits 183  
    wrapping up 182  
catcodes 51  
characters 89  
    codes 200  
command line 37

conditions 59

    dimensions 56  
    numbers 56  
    tokens 58

configuration 217

convert commands 198

csnames 33

## **d**

dimensions 56  
direct nodes 162  
directions 66, 143  
discretionaries 98, 100, 139

## **e**

$\varepsilon$ -TeX 31  
engines 29  
errors 208  
escaping 49  
exceptions 96  
expansion 55  
    suppress 85

## **f**

files  
    binary 33  
    names 65  
    writing 66  
fonts 66, 84  
    current 88  
    define 88  
    defining 210  
    extend 88  
    id 88  
    used 297

## **g**

getstartupfile 185  
getversion 185  
glue 140  
gluespec 140  
glyphs 89, 142



## **h**

hash 210  
helpers 207  
history 29  
hyphenation 64, 89, 94, 96  
    discretionaries 98  
    exceptions 96  
    how it works 98  
    patterns 96  
    tracing 94

## **i**

io 218  
images  
    MetaPost 227  
    mplib 227  
initialization 37, 210  
insertions 138  
inserts 131

## **k**

kerning 99  
kerns 141  
    suppress 85  
keywords 35

## **l**

Lua 13  
    extensions 39  
    interpreter 37  
    libraries 39  
    modules 39  
languages 64, 89  
    library 100  
last items 198  
leaders 64  
libraries  
    lua 185  
    status 186  
    tex 195  
    texconfig 217  
    texio 218  
    token 219  
ligatures 99  
    suppress 85  
linebreaks 100, 215

lists 136, 203

## **m**

MetaPost 227  
    mplib 227  
macros 223  
main loop 94  
marks 53, 131, 138  
math 66, 103  
    accents 122, 126  
    codes 127  
    control 105  
    cramped 108  
    delimiters 123, 126  
    extensibles 123  
    fences 120  
    fractions 124  
    kerning 118  
    last line 127  
    limits 118  
    nodes 139, 144  
    parameters 109, 111, 202  
    penalties 120  
    radicals 122  
    scripts 118, 123, 128  
    spacing 108, 115, 116, 118  
    stacks 107  
    styles 106, 108, 128  
    text 129  
    tracing 129  
    Unicode 103  
memory 33

## **n**

nesting 204, 217  
newline 34  
nodes 13, 45, 135  
    adjust 138  
    attributes 157  
    boundary 143  
    direct 162  
    direction 143  
    discretionaries 139  
    functions 150  
    glue 140  
    glyph 142





- insertions 138
  - kerns 141
  - lists 136
  - marks 138
  - math 139, 144
  - paragraphs 143
  - penalty 141
  - properties 169
  - rules 137
  - text 136
- numbers 56
- o**
- Omega 66
- output 61
- p**
- pdf
  - analyze 239
  - memory streams 243
  - objects 239
  - pdfe 239
- pdfTeX 31
- pages 131, 202, 217
- paragraphs 100, 131, 143
  - reset 215
- parameters
  - internal 195
  - math 202
- patterns 96
- penalty 141
- primitives 210
- printing 205
- properties 169
- protrusion 87
  - suppress 85
- r**
- registers 198, 201
  - bytecodes 185
- rules 61, 137
- s**
- shipout 217
- space 34
- spaces
  - suppress 86
- splitting 62
- synctex 217
- t**
- TeX 29
- tables 185
- testing 40
- text
  - math 129
- tokens 58, 219
  - scanning 52
- tracing 64
- u**
- Unicode 44
  - math 103
- v**
- vcentering 46
- version 42
- w**
- web2c 33





**278 Topics**

# Primitives

This register contains the primitives that are mentioned in the manual. There are of course many more primitives. The  $\varepsilon$ -TeX and LuaTeX primitives are typeset in bold (some originate in pdfTeX).

<code>\abovedisplayskip</code>	118	<code>\clearmarks</code>	53, 132
<code>\abovewithdelims</code>	124	<code>\clubpenalties</code>	216
<code>\accent</code>	94	<code>\copy</code>	45
<code>\adjustspacing</code>	31, 32, 79	<code>\count</code>	39, 44, 46, 198
<code>\adjustspacingshrink</code>	32	<code>\countdef</code>	44, 198, 199
<code>\adjustspacingstep</code>	32	<code>\crampedscriptstyle</code>	108
<code>\adjustspacingstretch</code>	32	<code>\csname</code>	53, 54
<code>\aftergrouped</code>	55	<code>\csstring</code>	52
<code>\aliased</code>	72	<code>\currentifttype</code>	65
<code>\aligncontent</code>	53	<code>\currentmarks</code>	53
<code>\alignmark</code>	53	<code>\defcsname</code>	54
<code>\aligntab</code>	53	<code>\delcode</code>	33, 104, 200, 201
<code>\allcrampedstyles</code>	111	<code>\delimiter</code>	104
<code>\alldisplaystyles</code>	111	<code>\detokenize</code>	221
<code>\allmathstyles</code>	111	<code>\dimen</code>	39, 44, 198
<code>\allscriptscriptstyles</code>	111	<code>\dimendef</code>	44, 86, 198, 199
<code>\allscriptstyles</code>	111	<code>\dimensiondef</code>	73
<code>\allsplitstyles</code>	111	<code>\dimexpr</code>	74
<code>\alltextstyles</code>	111	<code>\dimexpression</code>	74
<code>\alluncrampedstyles</code>	111	<code>\directlua</code>	13, 41, 48, 49, 205, 210, 211
<code>\atop</code>	107, 109	<code>\discretionary</code>	16, 93, 96, 98, 139
<code>\atopwithdelims</code>	107	<code>\displaystyle</code>	115
<code>\attribute</code>	198	<code>\displaywidowpenalties</code>	216
<code>\attributedef</code>	198, 199	<code>\dp</code>	45, 131
<code>\automatichyphenpenalty</code>	93	<code>\edef</code>	50, 55, 221
<code>\automigrationmode</code>	133	<code>\edefcsname</code>	54
<code>\autoparagraphmode</code>	131	<code>\efcode</code>	31, 44, 78
<code>\batchmode</code>	218	<code>\endgroup</code>	107
<code>\beginscename</code>	52, 53	<code>\endinput</code>	219
<code>\begingroup</code>	107	<code>\endlinechar</code>	30, 52, 205, 207
<code>\belowdisplayskip</code>	118	<code>\enforced</code>	73
<code>\botmark</code>	132	<code>\errhelp</code>	208
<code>\boundary</code>	64, 143	<code>\errmessage</code>	208
<code>\box</code>	45	<code>\etoksapp</code>	52
<code>\boxattribute</code>	48	<code>\etokspre</code>	52
<code>\catcode</code>	33, 41, 44, 200	<code>\everybeforepar</code>	131
<code>\catcodetable</code>	51, 205	<code>\everycr</code>	54
<code>\char</code>	16, 44, 86, 94, 96, 142	<code>\everyeof</code>	52
<code>\chardef</code>	44, 96, 223, 224	<code>\everyjob</code>	38



<b>\everytab</b>	53, 54	<b>\ht</b>	45, 131
<b>\exceptionpenalty</b>	97	<b>\hyphenation</b>	96, 98, 99
<b>\exhyphenchar</b>	95	<b>\hyphenationmin</b>	64, 90
<b>\exhyphenpenalty</b>	95, 98, 139	<b>\hyphenationmode</b>	81, 92, 95, 100
<b>\expandafter</b>	54, 55	<b>\hyphenchar</b>	77, 94, 95, 98
<b>\expanded</b>	31, 55	<b>\hyphenpenalty</b>	98, 139
<b>\explicitthyphenpenalty</b>	94	<b>\if</b>	53
<b>\firstmark</b>	132	<b>\ifabsdim</b>	31, 56
<b>\firstvalidlanguage</b>	90	<b>\ifabsnum</b>	31, 56
<b>\flushmarks</b>	53, 132	<b>\ifarguments</b>	58
<b>\fontdimen</b>	85	<b>\ifboolean</b>	58
<b>\fontid</b>	84	<b>\ifcase</b>	57, 117
<b>\fontmathcontrol</b>	81	<b>\ifchkdir</b>	56
<b>\fontspecdef</b>	85	<b>\ifchcknum</b>	56
<b>\fontspecid</b>	85	<b>\ifcmpdim</b>	56
<b>\fontspecscale</b>	85	<b>\ifcmpnum</b>	56
<b>\fontspecxscale</b>	85	<b>\ifcondition</b>	59
<b>\fontspecyscale</b>	85	<b>\ifcstok</b>	58
<b>\fonttextcontrol</b>	81	<b>\ifdimexpression</b>	75
<b>\formatname</b>	211	<b>\ifdimval</b>	56
<b>\frozen</b>	73, 117	<b>\ifempty</b>	57
<b>\futureexpand</b>	55	<b>\ifflags</b>	61
<b>\futureexpandis</b>	55	<b>\ifincsname</b>	31
<b>\futureexpandisap</b>	55	<b>\ifinsert</b>	132
<b>\gleaders</b>	64	<b>\ifmathparameter</b>	57, 117
<b>\glet</b>	54	<b>\ifmathstyle</b>	57
<b>\glettonothing</b>	54	<b>\ifnumexpression</b>	75
<b>\global</b>	72	<b>\ifnumval</b>	56
<b>\glyph</b>	86	<b>\ifparameter</b>	58
<b>\glyphdatafield</b>	88	<b>\ifparameters</b>	58
<b>\glyphoptions</b>	85	<b>\ifrelax</b>	58
<b>\glyphscale</b>	85	<b>\iftok</b>	58
<b>\glyphscriptfield</b>	88	<b>\ignorepars</b>	55
<b>\glyphstatefield</b>	88	<b>\ignorespaces</b>	55
<b>\glyphxoffset</b>	86	<b>\immutable</b>	72
<b>\glyphxscale</b>	85	<b>\inherited</b>	116
<b>\glyphyoffset</b>	86	<b>\initcatcodetable</b>	51
<b>\glyphyscale</b>	85	<b>\insert</b>	44, 138
<b>\gtoksapp</b>	52	<b>\insertdepth</b>	131
<b>\gtokspre</b>	52	<b>\insertdistance</b>	131
<b>\halign</b>	53, 176	<b>\insertheight</b>	131
<b>\hbox</b>	15, 45, 64, 118, 176, 177, 201	<b>\insertheights</b>	131
<b>\hjcode</b>	33, 44, 90, 97	<b>\insertlimit</b>	131
<b>\hpack</b>	64	<b>\insertmaxdepth</b>	131
<b>\hrule</b>	16, 61, 137	<b>\insertmode</b>	131
<b>\hskip</b>	16, 140	<b>\insertmultiplier</b>	131



<code>\insertpenalties</code>	131
<code>\insertpenalty</code>	131
<code>\insertstorage</code>	131
<code>\insertstoring</code>	131, 132
<code>\insertwidth</code>	131
<code>\instance</code>	73
<code>\integerdef</code>	73, 86
<code>\interlinepenalties</code>	216
<code>\jobname</code>	38, 174
<code>\kern</code>	16, 141
<code>\language</code>	95, 96, 99, 100
<code>\lastchkdim</code>	57
<code>\lastchknun</code>	57
<code>\lastnamedcs</code>	52, 53
<code>\lastnodesubtype</code>	65
<code>\lastnodetype</code>	65
<code>\lccode</code>	33, 44, 200
<code>\leaders</code>	64
<code>\left</code>	121
<code>\lefthyphenmin</code>	64, 90
<code>\leftmarginkern</code>	31
<code>\letcharcode</code>	54
<code>\lettonothing</code>	54
<code>\linedirection</code>	68
<code>\localbrokenpenalty</code>	143
<code>\localinterlinepenalty</code>	143
<code>\localleftbox</code>	133, 143, 176
<code>\localleftboxbox</code>	134
<code>\localmiddlebox</code>	133, 143
<code>\localmiddleboxbox</code>	134
<code>\localrightbox</code>	133, 143, 176
<code>\localrightboxbox</code>	134
<code>\lowercase</code>	97
<code>\lpcode</code>	31, 44, 78
<code>\luabytecode</code>	50
<code>\luabytecodecall</code>	50
<code>\luacopyinputnodes</code>	206
<code>\luaedef</code>	49, 224
<code>\luaescapestring</code>	49
<code>\luafunction</code>	49
<code>\luafunctioncall</code>	49, 50
<code>\luatexbanner</code>	43
<code>\luatexrevision</code>	43
<code>\luatexversion</code>	43
<code>\mark</code>	132, 138
<code>\marks</code>	44, 53, 132, 161
<code>\mathaccent</code>	104
<code>\mathchar</code>	104, 129
<code>\mathchardef</code>	104, 129
<code>\mathchoice</code>	106
<code>\mathcode</code>	33, 104, 200
<code>\mathcontrolmode</code>	81
<code>\mathdelimitersmode</code>	120
<code>\mathdirection</code>	205
<code>\mathdisplayskipmode</code>	118
<code>\matheqnogapstep</code>	120
<code>\mathfontcontrol</code>	105
<code>\mathnolimitsmode</code>	118
<code>\mathpenaltiesmode</code>	120
<code>\mathscriptboxmode</code>	118
<code>\mathscriptcharmode</code>	119
<code>\mathscriptsmode</code>	119
<code>\mathstyle</code>	106, 107, 125, 205
<code>\mathsurround</code>	115, 116, 140
<code>\mathsurroundmode</code>	115
<code>\mathsurroundskip</code>	115, 116
<code>\maxdepth</code>	131, 178
<code>\meaning</code>	65, 72
<code>\meaningasis</code>	65
<code>\meaningfull</code>	65, 72
<code>\meaningless</code>	65, 72
<code>\medmuskip</code>	116
<code>\middle</code>	204
<code>\muskip</code>	44, 116, 198
<code>\muskipdef</code>	44
<code>\mutable</code>	72
<code>\newlinechar</code>	30
<code>\noalign</code>	53, 72
<code>\noaligned</code>	72
<code>\noboundary</code>	64, 95, 99, 143
<code>\noexpand</code>	55
<code>\nohrule</code>	61, 62
<code>\normalizelinemode</code>	68
<code>\nospaces</code>	86, 87
<code>\novrule</code>	61, 62
<code>\number</code>	43, 74, 207
<code>\numericsscale</code>	70
<code>\numexpr</code>	74, 75
<code>\numexpression</code>	74
<code>\orelse</code>	60
<code>\orphanpenalties</code>	131
<code>\orphanpenalty</code>	131



<code>\orunless</code>	60, 61	<code>\setbox</code>	45
<code>\output</code>	179	<code>\setfontid</code>	84
<code>\outputbox</code>	61	<code>\setlanguage</code>	90, 95, 99
<code>\over</code>	107, 109, 110, 204	<code>\setmathspacing</code>	116
<code>\overline</code>	108	<code>\sfcode</code>	33, 44, 200
<code>\overloaded</code>	73	<code>\shapingpenaltiesmode</code>	131
<code>\overloadmode</code>	72, 73, 195	<code>\shapingpenaltymode</code>	131
<code>\overwithdelims</code>	107	<code>\skewchar</code>	77, 122
<code>\pageboundary</code>	133	<code>\skip</code>	44, 198, 199
<code>\par</code>	46, 55, 175	<code>\skipdef</code>	44, 198, 199
<code>\parattribute</code>	48	<code>\snapshotpar</code>	131
<code>\parfillskip</code>	176, 216	<code>\spaceskip</code>	86
<code>\parindent</code>	195	<code>\splitbotmark</code>	132
<code>\patterns</code>	96, 98, 99	<code>\splitfirstmark</code>	132
<code>\penalty</code>	141	<code>\string</code>	52
<code>\permanent</code>	72	<code>\tabsize</code>	53, 54
<code>\postexhyphenchar</code>	98	<code>\tabskip</code>	54
<code>\posthyphenchar</code>	98	<code>\textdirection</code>	16, 68, 143, 205
<code>\predisplayskip</code>	127	<code>\textfont</code>	111, 129
<code>\preexhyphenchar</code>	98	<code>\textstyle</code>	106
<code>\prehyphenchar</code>	98	<code>\the</code>	43, 46, 74, 195, 197, 199, 205
<code>\protrudechars</code>	31, 32, 79	<code>\thickmuskip</code>	116
<code>\protrusionboundary</code>	64, 87, 143	<code>\thinmuskip</code>	116
<code>\pxdimen</code>	32	<code>\tinymuskip</code>	116
<code>\quitvmode</code>	31	<code>\todimension</code>	74
<code>\radical</code>	104	<code>\tointeger</code>	74
<code>\relax</code>	72, 75, 96, 206, 210, 222	<code>\tokenized</code>	52
<code>\retokenized</code>	52	<code>\toks</code>	44, 197, 198, 199, 205
<code>\right</code>	121	<code>\toksapp</code>	52
<code>\righthyphenmin</code>	64, 90	<code>\toksdef</code>	44, 198, 199
<code>\rightmarginkern</code>	31	<code>\tokspre</code>	52
<code>\romannumeral</code>	106, 207	<code>\tolerant</code>	70, 72
<code>\rpcode</code>	31, 44, 78	<code>\topmark</code>	132
<code>\SuperscriptBaselineDropMax</code>	110	<code>\toscaled</code>	74
<code>\SuperscriptShiftUp</code>	110	<code>\tpack</code>	64
<code>\SuperscriptShiftUpCramped</code>	110	<code>\tracingadjusts</code>	133
<code>\savecatcodetable</code>	52	<code>\tracingalignments</code>	53
<code>\savingshyphcodes</code>	90, 91, 96, 102	<code>\tracingassigns</code>	31, 33
<code>\scaledfontdimen</code>	85	<code>\tracingcommands</code>	65, 95, 195
<code>\scantextokens</code>	52	<code>\tracingexpressions</code>	75
<code>\scantokens</code>	49, 52	<code>\tracingfonts</code>	32, 66
<code>\scriptfont</code>	111	<code>\tracinghyphenation</code>	65, 95
<code>\scriptscriptfont</code>	111	<code>\tracinginserts</code>	132
<code>\scriptscriptstyle</code>	122	<code>\tracinglevels</code>	65, 183
<code>\scriptspace</code>	115	<code>\tracingmarks</code>	53, 132
<code>\scriptstyle</code>	108	<code>\tracingmath</code>	65



<code>\tracingnesting</code>	209	<code>\Umathlimitbelowbgap</code>	110
<code>\tracingonline</code>	64	<code>\Umathlimitbelowkern</code>	110, 114
<code>\tracingrestores</code>	31, 33	<code>\Umathlimitbelowvgap</code>	110
<code>\Uabove</code>	125	<code>\Umathnolimitsubfactor</code>	110, 118
<code>\Uabovewithdelims</code>	125	<code>\Umathnolimitsupfactor</code>	110, 118
<code>\Uatop</code>	125	<code>\Umathoperatorsize</code>	105, 109, 115
<code>\Uatopwithdelims</code>	125	<code>\Umathoverbarkern</code>	109
<code>\Uchar</code>	44	<code>\Umathoverbarrule</code>	109
<code>\Udelcode</code>	104, 201	<code>\Umathoverbarvgap</code>	109
<code>\Udelcodenum</code>	104	<code>\Umathoverdelimiterbgap</code>	110, 124
<code>\Udelimiter</code>	104	<code>\Umathoverdelimitervgap</code>	110, 124
<code>\Udelimiterover</code>	105, 123	<code>\Umathprimebaselinedropmax</code>	110
<code>\Udelimiterunder</code>	105, 123	<code>\Umathprimeraisepercent</code>	110
<code>\Uhexensible</code>	123, 124	<code>\Umathprimeshiftup</code>	110
<code>\Uleft</code>	105	<code>\Umathprimeshiftupcramped</code>	110
<code>\UmathAccentBaseHeight</code>	111	<code>\Umathprimespaceafter</code>	110
<code>\Umathaccent</code>	104, 105, 122	<code>\Umathprimewidthpercent</code>	110
<code>\Umathaccentbasedepth</code>	111	<code>\Umathquad</code>	109, 114
<code>\Umathaccentbottomshiftup</code>	110	<code>\Umathradicaldegreeafter</code>	109, 115, 122
<code>\Umathaccentflattenedbasedepth</code>	111	<code>\Umathradicaldegreebefore</code>	109, 115, 122
<code>\Umathaccenttopshiftup</code>	110	<code>\Umathradicaldegreeraise</code>	109, 115, 122
<code>\Umathaxis</code>	109	<code>\Umathradicalkern</code>	109
<code>\Umathchar</code>	104, 129	<code>\Umathradicalrule</code>	109, 114
<code>\Umathcharclass</code>	127	<code>\Umathradicalvgap</code>	109, 115
<code>\Umathchardef</code>	104, 129	<code>\UmathSpaceAfterScript</code>	110
<code>\Umathcharfam</code>	127	<code>\Umathskewedfractionhgap</code>	124
<code>\Umathcharnum</code>	104	<code>\Umathskewedfractionvgap</code>	124
<code>\Umathcharnumdef</code>	104	<code>\Umathspaceafterscript</code>	110, 115
<code>\Umathcharslot</code>	127	<code>\Umathspacebeforescript</code>	110
<code>\Umathclass</code>	130	<code>\Umathstackdenomdown</code>	109
<code>\Umathcode</code>	104, 127	<code>\Umathstacknumup</code>	109
<code>\Umathcodenum</code>	104	<code>\Umathstackvgap</code>	109
<code>\Umathconnectoroverlapmin</code>	110, 115	<code>\Umathsubshiftup</code>	110, 119
<code>\UmathFlattenedAccentBaseHeight</code>	111	<code>\Umathsubshiftupdrop</code>	110
<code>\Umathflattenedaccentbottomshiftup</code>	111	<code>\Umathsubsupshiftup</code>	110, 119
<code>\Umathflattenedaccenttopshiftup</code>	111	<code>\Umathsubsupvgap</code>	110
<code>\Umathfractiondelsize</code>	110	<code>\Umathsubtopmax</code>	110
<code>\Umathfractiondenomdown</code>	110	<code>\Umathsubtopmin</code>	110
<code>\Umathfractiondenomvgap</code>	110	<code>\Umathsupshiftup</code>	110, 119
<code>\Umathfractionnumup</code>	110	<code>\Umathsupsubbottommax</code>	110
<code>\Umathfractionnumvgap</code>	110	<code>\Umathunderbarkern</code>	109
<code>\Umathfractionrule</code>	109	<code>\Umathunderbarrule</code>	109
<code>\Umathlimitabovebgap</code>	110	<code>\Umathunderbarvgap</code>	109
<code>\Umathlimitabovekern</code>	110, 114	<code>\Umathunderdelimiterbgap</code>	110, 124
<code>\Umathlimitabovevgap</code>	110	<code>\Umathunderdelimitervgap</code>	110, 124



<code>\Umathxscale</code>	130	<code>\wd</code>	45, 131
<code>\Umathyscale</code>	130	<code>\widowpenalties</code>	216
<code>\Umiddle</code>	105, 126	<code>\wordboundary</code>	64, 91, 143
<code>\Unosubscript</code>	128	<code>\wrapuppar</code>	131
<code>\Unosuperscript</code>	128	<code>\xdefcsname</code>	54
<code>\Uoperator</code>	105	<code>\xtoksapp</code>	52
<code>\Uover</code>	105, 125	<code>\xtokspre</code>	52
<code>\Uoverdelimater</code>	105, 123, 124	<code>\-</code>	139
<code>\Uoverwithdelims</code>	125		
<code>\Uradical</code>	104, 122		
<code>\Uright</code>	105, 126		
<code>\Uroot</code>	105, 122, 147		
<code>\Uskewed</code>	124		
<code>\Uskewedwithdelims</code>	124		
<code>\Ustack</code>	107, 108		
<code>\Ustartdisplaymath</code>	128		
<code>\Ustartmath</code>	128		
<code>\Ustopdisplaymath</code>	128		
<code>\Ustopmath</code>	128		
<code>\Ustyle</code>	125		
<code>\Usubscript</code>	128		
<code>\Usuperprescript</code>	128		
<code>\Usuperscript</code>	128		
<code>\UUskewed</code>	125		
<code>\UUskewedwithdelims</code>	125		
<code>\Uunderdelimater</code>	105, 123, 124		
<code>\UVextensible</code>	105		
<code>\uccode</code>	33, 44, 200		
<code>\uchyph</code>	90, 93, 142		
<code>\unexpanded</code>	221		
<code>\unhbox</code>	45		
<code>\unhcopy</code>	45		
<code>\unless</code>	61		
<code>\untraced</code>	73		
<code>\unvbox</code>	45		
<code>\unvcopy</code>	45		
<code>\uppercase</code>	54, 97		
<code>\vadjust</code>	132, 138, 175		
<code>\valign</code>	53, 176		
<code>\vbox</code>	16, 45, 64, 176, 201, 216		
<code>\vcenter</code>	48, 64, 176		
<code>\vpack</code>	64		
<code>\vrule</code>	16, 61, 137		
<code>\vskip</code>	16, 140		
<code>\vsplit</code>	45, 62, 176, 202		
<code>\vtop</code>	16, 64, 176, 201		





# Callbacks

## **b**

buildpage\_filter 175  
build\_page\_insert 175

## **c**

contribute\_filter 174

## **d**

define\_font 182

## **f**

find\_format\_file 174  
find\_log\_file 174

## **g**

glyph\_run 177

## **h**

hpack\_filter 176, 178  
hyphenate 179

## **i**

intercept\_lua\_error 181  
intercept\_tex\_error 181

## **k**

kerning 180

## **l**

ligaturing 179, 180  
linebreak\_filter 177, 216

## **m**

mlisttohlist 162  
mlist\_to\_hlist 120, 180

## **o**

open\_data\_file 174

## **p**

post\_linebreak\_filter 177  
pre\_dump 181  
pre\_linebreak\_filter 176, 216

process\_jobname 174  
process\_rule 179

## **s**

show\_error\_message 181  
show\_warning\_message 181  
show\_whatsit 183  
start\_file 182  
start\_run 181  
stop\_file 182  
stop\_run 181

## **v**

vpack\_filter 176, 178

## **w**

wrapup\_run 182





# Nodes

This register contains the nodes that are known to LuaTeX. The primary nodes are in bold, whatsits that are determined by their subtype are normal.

<b>a</b>	<b>math_char</b> 145
<b>accent</b> 146	<b>math_text_char</b> 145
<b>adjust</b> 92, 138	
<b>attribute</b> 157	<b>n</b>
	<b>noad</b> 146
<b>b</b>	
<b>boundary</b> 64, 92, 143	<b>p</b>
	<b>par</b> 143, 216
<b>c</b>	<b>parameter</b> 146
<b>choice</b> 147	<b>penalty</b> 92, 141
<b>d</b>	<b>r</b>
<b>delimiter</b> 145	<b>radical</b> 147
<b>delta</b> 207	<b>rule</b> 16, 92, 137
<b>dir</b> 16, 92, 143	<b>rules</b> 155
<b>disc</b> 16, 45, 139, 155	
	<b>s</b>
<b>f</b>	<b>style</b> 146
<b>fence</b> 148	<b>sub_box</b> 145
<b>fraction</b> 122, 147	<b>sub_mlist</b> 145
<b>g</b>	<b>t</b>
<b>glue</b> 16, 45, 92, 140, 155	<b>temp</b> 136
<b>glue-spec</b> 199	
<b>glue_spec</b> 140, 195, 197, 199	<b>v</b>
<b>glyph</b> 16, 45, 89, 90, 94, 142, 154	<b>vlist</b> 16, 45, 92, 136, 154, 155
<b>glyphs</b> 155	
	<b>w</b>
<b>h</b>	<b>whatsit</b> 92
<b>hlist</b> 15, 45, 47, 92, 136, 154, 155	
<b>i</b>	
<b>insert</b> 92, 138	
<b>k</b>	
<b>kern</b> 16, 45, 92, 141	
<b>m</b>	
<b>mark</b> 138	
<b>math</b> 139	





**288 Nodes**

# Libraries

This register contains the functions available in libraries. Not all functions are documented, for instance because they can be experimental or obsolete.

## **callback**

- find 173
- known 173
- list 173
- register 173

## **lang**

- clean 101
- clearhyphenation 101
- clearpatterns 101
- gethjcode 102
- hyphenate 102
- hyphenation 100
- hyphenationmin 101
- id 100
- new 100
- patterns 101
- postexhyphenchar 101
- posthyphenchar 101
- preexhyphenchar 101
- prehyphenchar 101
- sethjcode 102

## **lua**

- bytecode 185
- callbytecode 185
- getbytecode 185
- getcurrenttime 186
- getprecisesecounds 186
- getpreciseticks 186
- getruntime 186
- getstacktop 186
- getstartupfile 185
- getversion 185
- newindex 185
- newtable 185
- setbytecode 185

## **mplib**

- execute 229
- finish 230
- getcallbackstate 231
- gethashentries 230

- gethashentry 230

- getstates 230

- getstatus 230

- gettolerance 230

- new 227

- settolerance 230

- showcontext 230

- statistics 229

- version 227

## **node**

- checkdiscretionaries 160

- checkdiscretionary 160

- copy 151, 164

- copylist 151, 164

- count 152

- currentattr 157

- currentattributes 164

- dimensions 161

- endofmath 162

- fields 135, 150

- findattribute 158

- findnode 156

- firstglyph 159

- flattendiscretionaries 160

- flushlist 151, 164

- flushnode 151, 165

- free 151, 165

- getattribute 158, 165

- getfield 165

- getglue 156

- getpropertystable 165, 169

- getproperty 165

- gettotal 166

- hasattribute 158, 166

- hasfield 150, 166

- hasglyph 159

- hpack 160

- id 150

- insertafter 153, 166

- insertbefore 152, 166



- ischar 159
- isglyph 159
- isnode 150, 166
- iszeroglue 157
- kerning 159
- lastnode 153
- length 152
- ligaturing 159
- mlisttohlist 162
- new 150, 166
- protectglyph 160
- protectglyphs 160
- protrusionskipable 160
- rangedimensions 161
- remove 152, 167
- setAttribute 158, 167
- setfield 167
- setglue 156, 167
- setpropertiesmode 169
- setProperty 167
- slide 152
- subtypes 135
- tail 152, 168
- todirect 162
- tonode 162
- tostring 162, 168
- traverse 153, 168
- traversechar 154, 168
- traversecontent 155, 168
- traverseglyph 154, 168
- traverseid 154, 168
- traverselist 154, 168
- type 150, 168
- types 149
- unprotectglyph 159
- unprotectglyphs 159
- unsetattribute 158, 168
- values 135
- vpack 161
- write 151, 168

#### **node.direct**

- checkdiscretionaries 164
- checkdiscretionary 164
- copy 164
- copylist 164
- count 164

- currentattributes 164
- dimensions 164
- effectiveglue 164
- endofmath 164
- findattribute 164
- findattributerange 164
- findnode 164
- firstglyph 164
- flattendiscretionaries 164
- flushlist 164
- flushnode 165
- free 165
- getAttribute 165
- getAttributeList 165
- getattributes 165
- getboth 165
- getbox 165
- getchar 165
- getdata 165
- getdepth 165
- getdirection 165
- getdisc 165
- getexpansion 165
- getfam 165
- getfield 165
- getfont 165
- getglue 165
- getglyphdata 165
- getglyphdimensions 165
- getglyphscript 165
- getglyphstate 165
- getheight 165
- getid 165
- getindex 165
- getkern 165
- getkerndimension 165
- getlanguage 165
- getleader 165
- getlist 165
- getnext 165
- getnormalizedline 165
- getnucleus 165
- getoffsets 165
- getoptions 165
- getorientation 165
- getparstate 165



getpenalty 165  
getpost 165  
getpre 165  
getprev 165  
getpropertytable 165  
getproperty 165  
getreplace 165  
getscales 165  
getscript 165  
getshift 166  
getstate 166  
getsub 166  
getsubpre 166  
getsubtype 166  
getsup 166  
getsuppre 166  
getsynctexfields 165  
gettotal 166  
getwhd 166  
getwidth 166  
getxscale 166  
getxyscale 166  
getyscale 166  
hasattribute 166  
hasdimensions 166  
hasfield 166  
hasglyph 166  
hasglyphoption 166  
hpack 166  
hyphenating 166  
ignoremathskip 166  
insertafter 166  
insertbefore 166  
ischar 166  
isdirect 166  
isglyph 166  
isnextchar 166  
isnextglyph 166  
isnode 166  
isprevchar 166  
isprevglyph 166  
isvalid 166  
iszeroglue 166  
kerning 166  
lastnode 166  
length 166

ligaturing 166  
makeextensible 166  
migrate 166  
mlisttohlist 166  
naturalwidth 166  
new 166  
protectglyph 166  
protectglyphs 166  
protrusionskippable 166  
rangedimensions 166  
remove 167  
setattribute 167  
setattributelist 167  
setattributes 167  
setboth 167  
setbox 167  
setchar 167  
setdata 167  
setdepth 167  
setdirection 167  
setdisc 167  
setexpansion 167  
setfam 167  
setfield 167  
setfont 167  
setglue 167  
setglyphdata 167  
setglyphscript 167  
setglyphstate 167  
setheight 167  
setindex 167  
setkern 167  
setlanguage 167  
setleader 167  
setlink 167  
setlist 167  
setnext 167  
setnucleus 167  
setoffsets 167  
setoptions 167  
setorientation 167  
setpenalty 167  
setpost 167  
setpre 167  
setprev 167  
setproperty 167



setreplace 167  
setscales 167  
setscript 167  
setshift 167  
setsplit 167  
setstate 167  
setsub 167  
setsubpre 167  
setsubtype 167  
setup 168  
setsuppre 168  
setsynctexfields 167  
setwhd 168  
setwidth 168  
slide 168  
startofpar 168  
tail 168  
todirect 168  
tonode 168  
total 168  
tovaliddirect 168  
traverse 168  
traversechar 168  
traversecontent 168  
traverseglyph 168  
traverseid 168  
traverselist 168  
unprotectglyph 168  
unprotectglyphs 168  
unsetattribute 168  
unsetattributes 168  
usedlist 168  
usesfont 168  
verticalbreak 168  
vpack 168  
write 168

## **os**

env 252  
gettimeofday 252  
name 252  
selfarg 252  
selfdir 252  
setenv 252  
type 252  
uname 252

## **pdfc**

arraytotable 242  
close 239  
closestream 241  
dictionarytotable 242  
getarray 241  
getboolean 240  
getbox 240  
getcatalag 240  
getdictionary 241  
getfromarray 241, 242  
getfromdictionary 241, 242  
getfromreference 242  
getfromstream 241  
getinfo 240  
getinteger 240  
getname 240  
getnofobjects 240  
getnofpages 240  
getnumber 240  
getpage 240  
getsize 240  
getstatus 239  
getstream 241  
getstring 240  
gettrailer 240  
getversion 240  
new 239, 243  
open 239  
openstream 241  
readfromstream 241  
readfromwholestream 241  
unencrypt 239

## **sio**

getposition 245  
readbytes 245  
readbytetable 245  
readcardinaltable 245  
readcardinal1 245  
readcardinal2 245  
readcardinal3 245  
readcardinal4 245  
readfixed2 245  
readfixed4 245  
readintegertable 245  
readinteger1 245  
readinteger2 245





readinteger3 245  
 readinteger4 245  
 read2dot14 245  
 setposition 245  
 skipposition 245  
**status**  
 list 186  
 resetmessages 186  
 setexitcode 186  
**string**  
 bytepairs 251  
 bytes 251  
 characterpairs 251  
 characters 251  
 explode 251  
 utfcharacter 252  
 utfcharacters 251  
 utflength 252  
 utfvalue 252  
 utfvalues 251  
**tex**  
 attribute 198  
 badness 215  
 box 198, 201  
 catcode 200  
 count 198  
 cprint 206  
 definefont 210  
 delcode 200  
 dimen 198  
 enableprimitives 210  
 error 208  
 extraprimitives 211  
 fontidentifier 207  
 fontname 207  
 forcehmode 210  
 forcesynctexline 217  
 forcesynctextag 217  
 get 195  
 getattribute 198  
 getbox 198, 201  
 getcatcode 200  
 getcount 198  
 getdelcode 200  
 getdelcodes 200  
 getdimen 198  
 getfamilyoffont 208  
 getglue 198  
 gethelptext 208  
 getinteraction 209  
 getlccode 200  
 getlinenumber 208  
 getlist 203  
 getlocallevel 217  
 getmark 198  
 getmath 202  
 getmathcode 200  
 getmathcodes 200  
 getmuglue 198  
 getmuskip 198  
 getnest 204  
 getpagestate 217  
 getsfcode 200  
 getskip 198  
 getsynctexline 217  
 getsynctexmode 217  
 getsynctextag 217  
 gettoks 198  
 getuccode 200  
 glue 198  
 hashtokens 210  
 isattribute 198  
 isbox 198  
 iscount 198  
 isdimen 198  
 isglue 198  
 ismuglue 198  
 ismuskip 198  
 isskip 198  
 istoks 198  
 lccode 200  
 linebreak 215  
 lists 203  
 mathcode 200  
 muglue 198  
 muskip 198  
 nest 204  
 number 207  
 primitives 215  
 print 205  
 ptr 204  
 resetparagraph 215



- romannumeral 207
- round 207
- scale 207
- scantoks 198
- set 195
- setattribute 198
- setbox 198, 201
- setcatcode 200
- setcount 198
- setdelcode 200
- setdelcodes 200
- setdimen 198
- setglue 198
- setinteraction 209
- setlccode 200
- setlinenumber 208
- setlist 203
- setmath 202
- setmathcode 200
- setmathcodes 200
- setmuglue 198
- setmuskip 198
- setsfcode 200
- setskip 198
- setsynctexline 217
- setsynctexmode 217
- setsynctexnofiles 217
- setsynctextag 217
- settoks 198
- setuccode 200
- sfcode 200
- shipout 217
- show\_context 208
- skip 198
- sp 208
- splitbox 202
- sprint 205
- toks 198
- tprint 206
- triggerbuildpage 202
- uccode 200
- write 207

#### **texio**

- closeinput 219
- setescape 219
- write 218

- writenl 219
- writeselector 218
- writeselectornl 219

#### **token**

- biggest\_char 222
- commands 222
- command\_id 222
- create 222
- expand 222
- getactive 222
- getcmdname 222
- getcommand 222
- getcsname 222
- getexpandable 222
- getfrozen 222
- getfunctionstable 223
- getid 222
- getindex 222
- getmacro 223
- getmeaning 223
- getmode 222
- getprotected 222
- gettok 222
- getuser 222
- is\_defined 222
- is\_token 222
- new 222
- peeknext 222
- peeknextexpanded 222
- popmacro 223
- pushmacro 223
- putnext 224
- scanargument 219
- scancode 219
- scancsname 219
- scandimen 219
- scanfloat 219
- scanglue 219
- scanint 219
- scankeyword 219
- scankeywordcs 219
- scanlist 219
- scannext 222, 224
- scannextexpanded 222
- scanreal 219
- scanstring 219



scantoken 222  
scantoks 219  
scanword 219  
setchar 223  
setlua 223  
setmacro 223  
skipnext 222  
skipnextexpanded 222





# Statistics

The following fonts are used in this document:

used	filesize	version	filename
2	988.684	5.000	cambmath.ttf
1	927.280	5.020	cambria.ttf
1	163.452	1.802	LucidaBrightMathOT-Demi.otf
1	348.296	1.802	LucidaBrightMathOT.otf
1	73.284	1.801	LucidaBrightOT.otf
2	733.500	1.958	latinmodern-math.otf
1	64.684	2.004	lmmono10-regular.otf
2	64.160	2.004	lmmonoltcond10-regular.otf
1	111.536	2.004	lmroman10-regular.otf
3	525.008	1.106	texgyredejavu-math.otf
2	601.220	1.632	texgyrepagella-math.otf
1	218.100	2.501	texgyrepagella-regular.otf
1	693.876	2.340	DejaVuSans-Bold.ttf
1	741.536	2.340	DejaVuSans.ttf
1	318.392	2.340	DejaVuSansMono-Bold.ttf
1	245.948	2.340	DejaVuSansMono-Oblique.ttf
1	335.068	2.340	DejaVuSansMono.ttf
2	345.364	2.340	DejaVuSerif-Bold.ttf
1	336.884	2.340	DejaVuSerif-BoldItalic.ttf
1	343.388	2.340	DejaVuSerif-Italic.ttf
1	367.260	2.340	DejaVuSerif.ttf
<b>28</b>	<b>8.546.920</b>		<b>21 files loaded</b>





# Some remarks

Here I collect remarks that I'd like to make but that don't fit into the manual. Consider in a notebook.

**remark:** LuaMetaTeX development is mostly done by Hans Hagen and in adapting the macros to the new features Wolfgang Schuster, who knows the code inside-out is a instrumental. In the initial phase Alan Braslau, who love playing with the three languages did extensive testing and compiled for several platforms. Later Mojca Miklavc make sure all compiles well on the buildbot infrastructure. After the first release more users got involved in testing. Many thanks for their patience! The development also triggered upgrading of the wiki support infrastructure where Taco Hoekwater and Paul Mazaitis have teamed up. So, progress all around.

**remark:** When there are non-intrusive features that also make sense in LuaTeX, these will be applied in the experimental branch first, so that there is no interference with the stable release. However, given that in the meantime the code bases differs a lot, it is unlikely that much will trickle back. This is no real problem as there's not much demand for that anyway.

**remark:** Most ConTeXt users seem always willing to keep up with the latest versions which means that LMTX is tested well. We can therefore safely claim that end of 2019 the code has become quite stable, although after that in some areas there were substantial additions. There are no complaints about performance (on my 2013 laptop this manual compiles at 24.5 pps with LMTX versus 20.7 pps for the LuaTeX manual with MkIV). After updating some of the ConTeXt code to use recently added features by the end of 2020 I could do more than 25.5 pps and in 2021 at some point to measured some 29.1 pps (probably also due to some performance improvements in the MetaFun code) but don't expect spectacular bumps in performance (I need a new machine for that to happen). Probably no one notices it, but memory consumption stepwise got reduced too. And ... the binary is still below 3 MegaBytes on all platforms.

**remark:** I tried to only add features that are sort of generic and much relates to controlling and opening up the engine. That also means that there are extensions that (at least not now) are used in ConTeXt, simply because there are already mechanisms in place that work well. So, it's also about trying to be complete in order not have to add more later, which makes it possible to shift to larger interval between updates. That way local experiments are also better isolated from stable versions.

In that perspective arguments like “This got added because ConTeXt needs it.” or “That got done because features creep.” as well as “Because of such features ConTeXt performs better.” are merely distractions from the fact that we are dealing with a project that just wants to upgrade the machinery while making that effort fun to do. There has not been much community drive and demand for substantial extensions over the last decades, so it has to be the fun factor, right? And the ConTeXt community being willing to join the experiment makes it even more fun. Just keep that in mind.

**remark:** It's is kind of strange to run into arguments for not using LuaTeX or for what it is worth LuaMetaTeX. No one forces anyone to use TeX in the first place, also because often word processors or web based editing provides plenty of benefits. And no one forces a TeX users to use a specific engine. I bet that for most users pdfTeX suits well, especially when you only need TeX for relative simple publications and reports in English, using default styles that put constraints



on the user. Often the math is what matters there. Also, using  $\text{X}\text{\_}\text{TeX}$  is quite okay because it ships with built in font handling (of course that also has disadvantages, just consider the fact that it changed over time). When you want scripting  $\text{Lua}\text{\_}\text{TeX}$  is fine. When you need specific cjk support there are specialized engines for that. The same is true for  $\text{Con}\text{\_}\text{TeX}$ . You don't have to dislike it: just ignore it and don't waste time on barking against a tree. But when you use  $\text{Con}\text{\_}\text{TeX}$  the Lua enhanced engines are what you use.

**remark:** Yes there are bugs but I always consider the 'many' in "There are many bugs." to be an indication of frustration. Given the number of extensions and experiment one can expect bugs. But if someone can only mention a few, of which some fit into the category of engine limitations, it's probably more about ego. Abusing a mechanism for what it's not meant to, stretching it to the limits, running into a border case, those are not really bugs, more missing features. A crash is a bug indeed but we can count those in a few digits. The same is true for something missing in the manual: maybe it has a simple reason and explanation.

We have a fast cycle of resolving issues on the  $\text{Con}\text{\_}\text{TeX}$  list where user also test new functionality so that it can get improved. Complaints are also kind of puzzling because when we talk new features we're also talking of something that could not be done before. No one forces anyone to use experimental features. Yes, trying out something that is not perfect is no fun, but I clearly remember working around many limitations which is not always fun but can also be interesting. Just choose a better program if you don't like it, and definitely stick to the robust older engines!

As a warning: the tone in an email of a complaint or remark nowadays determines how high it ends up on the to-be-dealt-with list: pretty low. There are always more interesting things on top.

**remark:** Some extensions involve the way macro arguments are dealt with. Combined with the possibility to parse the input stream using Lua one can come up with solution that are hard (or maybe even impossible) otherwise. For me it meant throwing away nice (but often complex) solutions that evolved over decades. That can hurt, especially when you consider the time spent on it. But all this doesn't change the concept of  $\text{TeX}$  the macro language. When pondering some criticism, just wonder first why  $\text{TeX}$  attracts users, some of which like to write code.

I'm always puzzled by folks who complain about  $\text{TeX}$  as a language (the other part being the typesetter). Why use it if you don't like it? A macro language has its own characteristics so live with it. After years of writing  $\text{TeX}$  code it's this language that intrigues me. It's also a reason why MetaPost and Lua are embedded: they are different languages and depending on the task they might suit better. When Alan, Aditya, I and others are playing with MetaPost extensions using the new scanners and interfaces resulting from that we do just that. We could invent a new language, with lots of fruitless debate, with limitations, but in the end there's nothing wrong with MetaPost (coming from MetaFont).

