

The `luatextra` package

Elie Roux

`elie.roux@telecom-bretagne.eu`

2009/09/22 v0.94

Abstract

`luatextra` provides low-level addition to the formats Plain and \LaTeX to be used with the engine \LuaTeX .

Contents

1	Documentation	2
1.1	Preamble	2
1.2	History of formats and engines	2
1.3	choices made in this package	2
1.4	registers allocation scheme	3
1.5	attributes	3
1.6	Module system	3
1.7	Multiple callbacks handling	3
2	<code>luatextra.lua</code>	3
2.1	Initialization and internal functions	3
2.2	Error, warning and info function for modules	5
2.3	module loading and providing functions	6
2.4	Attributes handling	9
2.5	Catcodetables handling	10
2.6	Multiple callbacks on the <code>open_read_file</code> callback	10
2.7	Multiple callbacks on the <code>define_font</code> callback	12
3	<code>luatextra.sty</code>	14
3.1	Initializations	14
3.2	Primitives renaming	15
3.3	Module handling	17
3.4	Module handling	17
3.5	Attributes handling	17
3.6	Catcodetables handling	18
4	<code>luatextra-latex.tex</code>	21

1 Documentation

1.1 Preamble

This document is made for people wanting to understand how the package was made. For an introduction to the use of LuaTeX with the formats Plain and L^ATeX, please read the document `luatextra-reference.pdf` that you can find in your TeX distribution (TeXLive from version 2009) or on the CTAN.

1.2 History of formats and engines

To understand this package, one must first understand some historical choices in the TeX world.

A TeX engine is a binary executable that provides some very low-level primitives, for example `\count` to set a counter to a certain value. A TeX format is a macro package that provides higher-level macros for the user and the package developer, for example `\newcount` that allocates a new counter and gives it a name. Examples of engines are the old TeX 82, ϵ -TeX, pdfTeX, Omega/Aleph, LuaTeX and XeTeX. Examples of formats are Plain, L^ATeX and ConTeXt.

This distinction is hard to make as only one command is invoked, for example when you call the command `tex`, you often have no idea that the engine TeX 82 is invoked with the format Plain.

Evolution is also something confusing in the TeX world: engines often evolve, and new engines have always appeared, when most formats are frozen: the Plain and L^ATeX formats do not accept any new code to cope with the new engines. In theory, this package shouldn't exist, or at least shouldn't be a package, but its code should be integrated into a format. But as Plain and L^ATeX are frozen, people wanting to take advantage of the new engines have to use a package.

This package is really necessary to take advantage of LuaTeX as it provides things users are expecting a macro package to provide, for example `\newluaattribute` that acts like `\newcount` for lua attributes. It also enables all LuaTeX primitives, that are disabled by default.

1.3 choices made in this package

In the very long term, it is highly possible that LuaTeX will replace pdfTeX as the default L^ATeX engine, so it is necessary to keep backward compatibility. This led us to the decision of renaming LuaTeX-only primitives so that they all start by `luatex`, like the pdfTeX-only primitives start by `pdf`. Thus attributes become `luatexattributes`, etc. This also allows primitives to keep having the same name, even if their name is changed later at the engine level. Also some new functions like `newluatexattribute` are provided with the `lua` prefix, to shorten the already too long name.

1.4 registers allocation scheme

The default register allocation scheme of \LaTeX is old and limited (like the one of \TeX82) to 256 values. The engine $\varepsilon\text{-TeX}$ allows more different registers (up to 32768), and LuaTeX allows even 65536 ones. These new limits were not acknowledged by \LaTeX . A package `etex` was created for \LaTeX to extend the allocation scheme. `luatextra` loads `etex`, and overrides some values to extend the allocation max number to 65536.

1.5 attributes

Attributes are a new concept in LuaTeX (see the LuaTeX documentation for details). As the macro `\attribute` is certainly very common in the user's documents, they are renamed `luaattributes`. This package provides a simple way to allocate new attributes, with the macro `\newluaattribute`. For more informations about attribute handling in lua, please read section 2.4.

1.6 Module system

Lua has some embedded module management, with the functions `module` and `require`. With this package we try to get more control on the module system, by implementing something close to the \LaTeX 's `\usepackage` and `\RequirePackage` macros: the `\luatexUseModule` and `\luatexRequireModule` that act like them, but for lua files. The functions `module` and `require` should not be used, in profit of the lua functions `luatextra.provides_module` and `luatextra.use_module` or `luatextra.require_module`.

1.7 Multiple callbacks handling

LuaTeX has no way to register multiple functions in a callback. This package loads `luamcallbacks` that provides a safe way to do so. But the `luamcallbacks` package can't register several functions in some callbacks, like `open_read_file` and `define_font`. This package takes advantage of the callback creation possibilities of `luamcallbacks` to split these callbacks into several ones that can aggregate several functions. Thus it allows several packages to safely use the callbacks. See section 2.6 for more details.

2 luatextra.lua

2.1 Initialization and internal functions

\TeX always prints the names of the files that are input. Unfortunately it can't do so with lua files called with `dofile`. We will fix it with the `luatextra.use_module` function, but in the meantime we print this information for the `luatextra.lua` file.

A change compared to usual filename printings is the fact that LuaTeX does not print the `./` for files in the current directory. We keep this convention for lua filename printings.

```

1 do
2   local luatextrapath = kpse.find_file("luatextra.lua")
3   if luatextrapath then
4     if luatextrapath:sub(1,2) == "./" then
5       luatextrapath = luatextrapath:sub(3)
6     end
7     texio.write(' '..luatextrapath)
8   end
9 end
10

```

We create the `luatextra` table that will contain all the functions and variables, and we register it as a normal lua module.

```

11
12 luatextra = {}
13
14 module("luatextra", package.seeall)
15

```

We initiate the `modules` table that will contain informations about the loaded modules. And we register the `luatextra` module. The informations contained in the table describing the module are always the same, it can be taken as a template. See `luatextra.provides_module` for more details.

```

16
17 luatextra.modules = {}
18
19 luatextra.modules['luatextra'] = {
20   version    = 0.94,
21   name       = "luatextra",
22   date       = "2009/09/22",
23   description = "Additional low level functions for LuaTeX",
24   author     = "Elie Roux",
25   copyright  = "Elie Roux, 2009",
26   license    = "CC0",
27 }
28
29 local format = string.format
30

```

Here we define the warning and error functions specific to `luatextra`.

```

31
32 luatextra.internal_warning_spaces = " "
33
34 function luatextra.internal_warning(msg)
35   if not msg then return end
36   texio.write_nl(format("\nLuaTeXtra Warning: %s\n\n", msg))

```

```

37 end
38
39 luatextra.internal_error_spaces = "
40
41 function luatextra.internal_error(msg)
42     if not msg then return end
43     tex.sprint(format("\immediate\write16{\errmessage{LuaTeXtra error: %s^^J^^J}", msg))
44 end
45

```

2.2 Error, warning and info function for modules

Some module printing functions are provided, they have the same philosophy as the L^AT_EX's `\PackageError` and `\PackageWarning` macros: their first argument is the name of the module, and the second is the message. These functions are meant to be used by lua module writers.

```

46
47 function luatextra.module_error(package, msg, helpmsg)
48     if not package or not msg then
49         return
50     end
51     if helpmsg then
52         tex.sprint(format("\errhelp{%s}", helpmsg))
53     end
54     tex.sprint(format("\luatexModuleError{%s}{%s}", package, msg))
55 end
56
57 function luatextra.module_warning(modulename, msg)
58     if not modulename or not msg then
59         return
60     end
61     texio.write_nl(format("\nModule %s Warning: %s\n\n", modulename, msg))
62 end
63
64 function luatextra.module_log(modulename, msg)
65     if not modulename or not msg then
66         return
67     end
68     texio.write_nl('log', format("%s: %s", modulename, msg))
69 end
70
71 function luatextra.module_term(modulename, msg)
72     if not modulename or not msg then
73         return
74     end
75     texio.write_nl('term', format("%s: %s", modulename, msg))
76 end
77
78 function luatextra.module_info(modulename, msg)

```

```

79     if not modulename or not msg then
80         return
81     end
82     texio.write_nl(format("%s: %s\n", modulename, msg))
83 end
84

```

2.3 module loading and providing functions

A small function to find a lua module file according to its name, with or without the .lua at the end of the filename.

```

85
86 function luatextra.find_module_file(name)
87     if string.sub(name, -4) ~= '.lua' then
88         name = name..'.lua'
89     end
90     path = kpse.find_file(name, 'tex')
91     if not path then
92         path = kpse.find_file(name, 'texmfscripts')
93     end
94     return path, name
95 end
96

```

A small patch, for the module function to work in this file. I can't understand why it doesn't otherwise.

```

97
98 luatextra.module = module
99

```

luatextra.use module This macro is the one used to simply load a lua module file. It does not reload it if it's already loaded, and prints the filename in the terminal and the log. A lua module must call the macro `luatextra.provides_module`.

```

100
101
102 function luatextra.use_module(name)
103     if not name or luatextra.modules[name] then
104         return
105     end
106     local path, filename = luatextra.find_module_file(name)
107     if not path then
108         luatextra.internal_error(format("unable to find lua module %s", name))
109     else
110         if path:sub(1,2) == "./" then
111             path = path:sub(3)
112         end
113         texio.write(' ('..path)
114         dofile(path)
115         if not luatextra.modules[name] then

```

```

116         luatextra.internal_warning(format("You have requested module '%s',\n%s but the f
117     end
118     if not package.loaded[name] then
119         luatextra.module(name, package.seeall)
120     end
121     texio.write('')
122 end
123 end
124

```

Some internal functions to convert a date into a number, and to determine if a string is a date. It is useful for `luatextra.require_package` to understand if a user asks a version with a date or a version number.

```

125
126 function luatextra.datetonenumber(date)
127     numbers = string.gsub(date,("(%d+)/(%d+)/(%d+)", "%1%2%3")
128     return tonumber(numbers)
129 end
130
131 function luatextra.isdate(date)
132     for _, _ in string.gmatch(date, "%d+/%d+/%d+") do
133         return true
134     end
135     return false
136 end
137
138 local date, number = 1, 2
139
140 function luatextra.versiontonumber(version)
141     if luatextra.isdate(version) then
142         return {type = date, version = luatextra.datetonenumber(version), orig = version}
143     else
144         return {type = number, version = tonumber(version), orig = version}
145     end
146 end
147
148 luatextra.requiredversions = {}
149

```

`luatextra.require module` This function is like the `luatextra.use_module` function, but can accept a second argument that checks for the version of the module. The version can be a number or a date (format `yyyy/mm/dd`).

```

150
151 function luatextra.require_module(name, version)
152     if not name then
153         return
154     end
155     if not version then
156         return luatextra.use_module(name)

```

```

157     end
158     luaversion = luatextra.versiontonumber(version)
159     if luatextra.modules[name] then
160         if luaversion.type == date then
161             if luatextra.datetonenumber(luatextra.modules[name].date) < luaversion.version then
162                 luatextra.internal_error(format("found module '%s' loaded in version %s, but",
163             end
164         else
165             if luatextra.modules[name].version < luaversion.version then
166                 luatextra.internal_error(format("found module '%s' loaded in version %.02f,
167             end
168         end
169     else
170         luatextra.requiredversions[name] = luaversion
171         luatextra.use_module(name)
172     end
173 end
174

```

`luatextra.provides module` This macro is the one that must be called in the module files. It takes a table as argument. You can put any information you want in this table, but the mandatory ones are `name` (a string), `version` (a number), `date` (a string) and `description` (a string). Other fields are usually `copyright`, `author` and `license`.

This function logs informations about the module the same way L^AT_EX does for informations about packages.

```

175
176 function luatextra.provides_module(mod)
177     if not mod then
178         luatextra.internal_error('cannot provide nil module')
179         return
180     end
181     if not mod.version or not mod.name or not mod.date or not mod.description then
182         luatextra.internal_error('invalid module registered, fields name, version, date and
183     return
184 end
185 requiredversion = luatextra.requiredversions[mod.name]
186 if requiredversion then
187     if requiredversion.type == date and requiredversion.version > luatextra.datetonenumber
188         luatextra.internal_error(format("loading module %s in version %s, but version %s
189     elseif requiredversion.type == number and requiredversion.version > mod.version then
190         luatextra.internal_error(format("loading module %s in version %.02f, but version
191     end
192 end
193 luatextra.modules[mod.name] = module
194 texio.write_nl('log', format("Lua module: %s %s v%.02f %s\n", mod.name, mod.date, mod.v
195 end
196

```


Here we load the `luaextra` module, that contains a bunch of very useful functions. See the documentation of `luaextra` for more details.

```

197
198 luaextra.use_module('luaextra')
199
    luaextra.kpse_module_loader finds a module with the kpse library. This
    function is then registered in the table of the functions used by the lua function
    require to look for modules.

200
201 function luaextra.kpse_module_loader(mod)
202   local file = luaextra.find_module_file(mod)
203   if file then
204     local loader, error = loadfile(file)
205     if loader then
206       texio.write_nl("(" .. file .. ")")
207       return loader
208     end
209     return "\n\t[luaextra.kpse_module_loader] Loading error:\n\t"
210       .. error
211   end
212   return "\n\t[luaextra.kpse_module_loader] Search failed"
213 end
214
215 table.insert(package.loaders, luaextra.kpse_module_loader)
216

```

2.4 Attributes handling

Attribute allocation is done mainly in the `sty` file, but there is also a lua addition for attribute handling: `LuaTeX` is by default unable to tell the attribute number corresponding to an attribute name. This attribute number is necessary for functions such as `node.has_attribute`, which is used very often. The solution until now was to give a chosen attribute number to each attribute, and pray that someone else didn't use it before. With this method it was easy to know the number of an attribute, as it was chosen. Now with the `\newluaattribute` macro, it's impossible to know the number of an attribute. To fix it, when `\newluaattribute` is called, it calls `luaextra.attributedef_from_tex`. This function registers the number in the table `tex.attributenum`. For example to get the number of the attribute `myattribute` registered with `\newluaattribute\myattribute`, you can simply call `tex.attributenum[myattribute]`.

```

217
218 luaextra.attributes = {}
219
220 tex.attributenum = luaextra.attributes
221
222 function luaextra.attributedef_from_tex(name, number)

```

```

223     truename = name:gsub('[\\ ]', '')
224     luatextra.attributes[truename] = tonumber(number)
225 end
226

```

2.5 Catcodetables handling

In the same way, the table `tex.catcodetablenumber` contains the numbers of the catcodetables registered with `\newluacatcodetable`.

```

227
228 luatextra.catcodetables = {}
229
230 tex.catcodetablenumber = luatextra.catcodetables
231
232 function luatextra.catcodetabledef_from_tex(name, number)
233     truename = name:gsub('[\\ ]', '')
234     luatextra.catcodetables[truename] = tonumber(number)
235 end
236

```

2.6 Multiple callbacks on the `open_read_file` callback

The `luamcallbacks` (see documentation for details) cannot really provide a simple and reliable way of registering multiple functions in some callbacks. To be able to do so, the solution we implemented is to register one function in these callbacks, and to create "sub-callbacks" that can accept several functions. That's what we do here for the callback `open_read_file`.

`luatextra.open_read_file` This function is the one that will be registered in the callback. It calls new callbacks, that will be created later. These callbacks are:

- `pre_read_file` in which you can register a function with the signature `pre_read_file(env)`, with `env` being a table containing the fields `filename` which is the argument of the callback `open_read_file`, and `path` which is the result of `kpse.find_file`. You can put any field you want in the `env` table, you can even override the existing fields. This function is called at the very beginning of the callback, it allows for instance to register functions in the other callbacks. It is useless to add a field `reader` or `close`, as they will be overridden.
- `file_reader` is automatically registered in the `reader` callback for every file, it has the same signature.
- `file_close` is registered in the `close` callback for every file, and has the same signature.

```

237
238 function luatextra.open_read_file(filename)

```

```

239     local path = kpse.find_file(filename)
240     local env = {
241         ['filename'] = filename,
242         ['path'] = path,
243     }
244     luamcallbacks.call('pre_read_file', env)
245     path = env.path
246     if not path then
247         return
248     end
249     local f = env.file
250     if not f then
251         f = io.open(path)
252         env.file = f
253     end
254     if not f then
255         return
256     end
257     env.reader = luatextra.reader
258     env.close = luatextra.close
259     return env
260 end
261

```

The two next functions are the one called in the `open_read_file` callback.

```

262
263 function luatextra.reader(env)
264     local line = (env.file):read()
265     line = luamcallbacks.call('file_reader', env, line)
266     return line
267 end
268
269 function luatextra.close(env)
270     (env.file):close()
271     luamcallbacks.call('file_close', env)
272 end
273

```

In the callback creation process we need to have default behaviours. Here they are. These are called only when no function is registered in the created callback. See the documentation of `luamcallbacks` for more details.

```

274
275 function luatextra.default_reader(env, line)
276     return line
277 end
278
279 function luatextra.default_close(env)
280     return
281 end

```

```

282
283 function luatextra.default_pre_read(env)
284     return env
285 end
286

```

2.7 Multiple callbacks on the `define_font` callback

The same principle is applied to the `define_font` callback. The main difference is that this mechanism is not applied by default. The reason is that the callback most people will register in the `define_font` callback is the one from ConTeXt allowing the use of OT fonts. When the code will be more adapted (not so soon certainly), this mechanism will certainly be used, as it allows more flexibility in the font syntax, the OT font load mechanism, etc.

The callbacks we register here are the following ones:

- `font_syntax` that takes a table with the fields `asked_name`, `name` and `size`, and modifies this table to add more information. It must add at least a `path` field. The structure of the final table is not precisely defined, as it can vary from one syntax to another.
- `open_otf_font` takes the previous table, and must return a valid font structure as described in the LuaTeX manual.
- `post_font_opening` takes the final font table and can modify it, before this table is returned to the `define_font` callback.

But first, we acknowledge the fact that `fontforge` has been renamed to `fontloader`. This check allows older versions of LuaTeX to use `fontloader`.

As this mechanism is not loaded by default and certainly won't be until version 1.0 of LuaTeX, we don't document it further. See the documentation of `luatextra.sty` (macro `\ltextra@RegisterFontCallback`) to know how to load this mechanism anyway.

```

287
288 do
289     if tex.luatexversion < 36 then
290         fontloader = fontforge
291     end
292 end
293
294 function luatextra.find_font(name)
295     local types = {'ofm', 'ovf', 'opentype fonts', 'truetype fonts'}
296     local path = kpse.find_file(name)
297     if path then return path end
298     for _,t in pairs(types) do
299         path = kpse.find_file(name, t)
300         if path then return path end
301     end

```

```

302     return nil
303 end
304
305 function luatextra.font_load_error(error)
306     luatextra.module_warning('luatextra', string.format('%s\nloading lmr10 instead...', error))
307 end
308
309 function luatextra.load_default_font(size)
310     return font.read_tfm("lmr10", size)
311 end
312
313 function luatextra.define_font(name, size)
314     if (size < 0) then size = (- 655.36) * size end
315     local fontinfos = {
316         asked_name = name,
317         name = name,
318         size = size
319     }
320     callback.call('font_syntax', fontinfos)
321     name = fontinfos.name
322     local path = fontinfos.path
323     if not path then
324         path = luatextra.find_font(name)
325         fontinfos.path = luatextra.find_font(name)
326     end
327     if not path then
328         luatextra.font_load_error("unable to find font "..name)
329         return luatextra.load_default_font(size)
330     end
331     if not fontinfos.filename then
332         fontinfos.filename = fpath.basename(path)
333     end
334     local ext = fpath.suffix(path)
335     local f
336     if ext == 'tfm' or ext == 'ofm' then
337         f = font.read_tfm(name, size)
338     elseif ext == 'vf' or ext == 'ovf' then
339         f = font.read_vf(name, size)
340     elseif ext == 'ttf' or ext == 'otf' or ext == 'ttc' then
341         f = callback.call('open_otf_font', fontinfos)
342     else
343         luatextra.font_load_error("unable to determine the type of font "..name)
344         f = luatextra.load_default_font(size)
345     end
346     if not f then
347         luatextra.font_load_error("unable to load font "..name)
348         f = luatextra.load_default_font(size)
349     end
350     callback.call('post_font_opening', f, fontinfos)
351     return f

```

```

352 end
353
354 function luatextra.default_font_syntax(fontinfos)
355     return
356 end
357
358 function luatextra.default_open_otf(fontinfos)
359     return nil
360 end
361
362 function luatextra.default_post_font(f, fontinfos)
363     return true
364 end
365
366 function luatextra.register_font_callback()
367     callback.add('define_font', luatextra.define_font, 'luatextra.define_font')
368 end
369
370 do
371     luatextra.use_module('luamcallbacks')
372     callback.create('pre_read_file', 'simple', luatextra.default_pre_read)
373     callback.create('file_reader', 'data', luatextra.default_reader)
374     callback.create('file_close', 'simple', luatextra.default_close)
375     callback.add('open_read_file', luatextra.open_read_file, 'luatextra.open_read_file')
376     callback.create('font_syntax', 'simple', luatextra.default_font_syntax)
377     callback.create('open_otf_font', 'first', luatextra.default_open_otf)
378     callback.create('post_font_opening', 'simple', luatextra.default_post_font)
379
380     if luatextrapath then
381         texio.write('')
382     end
383 end

```

3 luatextra.sty

3.1 Initializations

First we prevent multiple loads of the file (useful for plain-TEX).

```

384 \csname ifluatextraloaded\endcsname
385 \let\ifluatextraloaded\endinput
386

```

Then we load ifluatex.

```

387
388 \expandafter\ifx\csname ProvidesPackage\endcsname\relax
389     \expandafter\ifx\csname ifluatex\endcsname\relax
390         \input ifluatex.sty
391     \fi
392 \else

```

```

393 \RequirePackage{ifluatex}
394 \fi
395
396 \expandafter\ifx\cscname ProvidesPackage\endcsname\relax
    If the package is loaded with Plain, we raise an error if the package is called with
    a non-LuaTeX engine, and we define \luaRequireModule with two mandatory
    arguments.
397 \ifluatex\else
398 \immediate\write16{}
399 \errmessage{Package luatextra Error: This package must be used with LuaTeX}
400 \fi
401 \def\luatexRequireModule#1#2{\luadirect{luatextra.require_module([[#1]], [[#2]])}}
402 \else
    If the package is loaded with LATEX, we also print the error message, and we
    define \luaRequireModule with one mandatory argument (the name of the pack-
    age) and one optional (the version or the date). We also define the environment
    luacode.
403 \ifluatex\else
404 \PackageError{luatextra}{This package must be used with LuaTeX.}
405 \fi
406 \NeedsTeXFormat{LaTeX2e}
407 \ProvidesPackage{luatextra}
408 [2009/04/15 v0.93 LuaTeX extra low-level macros]
409 \RequirePackage{environ}
410 \NewEnviron{luacode}{\luadirect{\BODY}}
411 \newcommand\luatexRequireModule[2][0]{\luadirect{luatextra.require_module([[#2]], [[#1]])}}
    We also require the package etex to be loaded. The \input is a hack that modi-
    fies some values in the register attribution scheme of  $\varepsilon$ -TEX and remaps \newcount
    to etex's \globcount. We have to do such a remapping in a separate file that Plain
    doesn't see, otherwise it outputs an error if we try to change \newcount (because
    it is an \outer macro). See section 4 for the file content.
412 \RequirePackage{etex}[1998/03/26]%
413 \input luatextra-latex.tex
414 \fi
415
    The two macros \LuaTeX and \LuaLaTeX are defined to LuaTEX and LuaLATEX,
    because that's the way it's written in the LuaTEX's manual (not in small capitals).
416
417 \def\LuaTeX{Lua\TeX }
418 \def\LuaLaTeX{Lua\LaTeX }
419

```

3.2 Primitives renaming

Here we differentiate two very different cases: LuaT_EX version ≥ 0.36 has no `tex.enableprimitives` function, and has support for multiple lua states, and for

versions ≥ 0.35 , the `tex.enableprimitives` is provided, and the old `\directlua` syntax prints a warning.

420

421 `\ifnum\luatexversion<36`

For old versions, we simply rename the primitives. You can note that `\attribute` (and also others) have no `\primitive` before them, because it would make users unable to call `\global\luaattribute`, which is a strong restriction. With this method, we can call it, but if `\attribute` was defined before, this means that `\luaattribute` will get its meaning, which is dangerous. Note also that you cannot use multiple states.

```
422 \def\directlua{\pdfprimitive\directlua0}
423 \def\latelua{\pdfprimitive\latelua0}
424 \def\luadirect{\pdfprimitive\directlua0}
425 \def\lualate{\pdfprimitive\latelua0}
426 \def\luatexattribute{\attribute}
427 \def\luatexattributedef{\attributedef}
428 \def\luatexclearmarks{\pdfprimitive\luaclearmarks}
429 \def\luatexformatname{\pdfprimitive\formatname}
430 \def\luatexscantexttokens{\pdfprimitive\scantexttokens}
431 \def\luatexcatcodetable{\catcodetable}
432 \def\initluatexcatcodetable{\pdfprimitive\initcatcodetable}
433 \def\saveluatexcatcodetable{\pdfprimitive\savecatcodetable}
434 \def\luaclose{\pdfprimitive\closelua}
435 \else
```

From TeXLive 2009, all primitives should be provided with the `luatex` prefix. For TeXLive 2008, we provide some primitives with this prefix too, to keep backward compatibility.

```
436 \directlua{tex.enableprimitives('luatex', {'attribute'})}
437 \directlua{tex.enableprimitives('luatex', {'attributedef'})}
438 \directlua{tex.enableprimitives('luatex', {'clearmarks'})}
439 \directlua{tex.enableprimitives('luatex', {'formatname'})}
440 \directlua{tex.enableprimitives('luatex', {'scantexttokens'})}
441 \directlua{tex.enableprimitives('luatex', {'catcodetable'})}
442 \directlua{tex.enableprimitives('luatex', {'latelua'})}
443 \directlua{tex.enableprimitives('luatex', {'initcatcodetable'})}
444 \directlua{tex.enableprimitives('luatex', {'savecatcodetable'})}
445 \directlua{tex.enableprimitives('luatex', {'closelua'})}
446 \let\luadirect\directlua
447 \let\lualate\luatexlatelua
448 \let\initluatexcatcodetable\luatexinitcatcodetable
449 \let\saveluatexcatcodetable\luatexsavecatcodetable
450 \let\luaclose\luatexcloselua
451 \fi
452
453
```

We load the lua file.


```

454
455 \luadirect{dofile(kpse.find_file("luatextra.lua"))}
456

```

A small macro to register the `define_font` callback from `luatextra`. See section 2.7 for more details.

```

457
458 \def\ltxtra@RegisterFontCallback{
459   \luadirect{luatextra.register_font_callback()}
460 }
461

```

3.3 Module handling

The `\luaModuleError` macro is called by the lua function `luatextra.module_error`. It is necessary because we can't call directly `\errmessage` in lua.

3.4 Module handling

The `\luatexModuleError` macro is called by the lua function `luatextra.module_error`. It is necessary because we can't call directly `\errmessage` in lua. Then we define `\luatexUseModule` that simply calls `luatextra.use_module`. Remember that `\luatexRequireModule` is defined at the beginning of this file.

```

462
463 \def\luatexModuleError#1#2{%
464   \errorcontextlines=0\relax
465   \immediate\write16{}%
466   \errmessage{Module #1 error: #2^^J^^J%
467 See the module #1 documentation for explanation.^^J ...^^J}%
468 }
469
470 \def\luatexUseModule#1{\luadirect{luatextra.use_module([[#1]])}}
471

```

3.5 Attributes handling

The most important macro here is `\newluatexattribute` that allocates a new attribute, and adds it in the `tex.attributename` table (see `luatextra.attributedef_from_tex` for more details. It works just like the other `\new*` macros, we can allocate up to 65536 different attributes.

```

472
473
474 \newcount\luatexattdefcounter
475 \luatexattdefcounter = 1
476
477 \def\newluatexattribute#1{%
478   \ifnum\luatexattdefcounter<65535\relax %
479     \global\advance\luatexattdefcounter by 1\relax %

```

```

480 \allocationnumber\luatexattdefcounter %
481 \ifluatex %
482   \global\luatexattributedef#1=\allocationnumber %
483 \fi %
484 \wlog{\string#1=\string\luatexattribute\the\allocationnumber}%
485 \luadirect{%
486   luatextra.attributedef_from_tex([[noexpand#1]], '\number\allocationnumber'))%
487 \else %
488   \errmessage{No room for a new \string\attribute}%
489 \fi %
490 }
491

```

Two convenient macros, one to set an attribute (basically just a wrapper), and another one to unset it. Unsetting attributes with this function is important, as the `unset` value may change, as it already has in the 0.37 version.

```

492
493 \def\setluatexattribute#1#2{%
494   #1=\numexpr#2\relax %
495 }
496
497 \def\unsetluatexattribute#1{%
498   \ifnum\luatexversion<37\relax %
499     #1=-1\relax %
500   \else %
501     #1=-"7FFFFFFF\relax %
502   \fi %
503 }
504

```

3.6 Catcodetables handling

Here we allocate catcodetables the same way we handle attributes.

```

505
506 \newcount\luatexcatcodetabledefcounter
507
508 \luatexcatcodetabledefcounter = 1
509
510 \def\newluatexcatcodetable#1{%
511   \ifnum\luatexcatcodetabledefcounter<1114110\relax % 0x10FFFF is maximal \chardef
512     \global\advance\luatexcatcodetabledefcounter by 1\relax %
513     \allocationnumber=\luatexcatcodetabledefcounter %
514     \global\chardef#1=\allocationnumber %
515     \luadirect{%
516       luatextra.catcodetabledef_from_tex([[noexpand#1]], '\number\allocationnumber'))%
517     \wlog{\string#1=\string\catcodetable\the\allocationnumber}%
518   \else %
519     \errmessage{No room for a new \string\catcodetable}%
520   \fi %

```

521 }
 522

A small patch to manage the catcode of `\in` in Plain, and to get two new counters in Plain too.

523
 524 \expandafter\edef\csname ltxtra@AtEnd\endcsname{%
 525 \catcode64 \the\catcode64\relax
 526 }
 527
 528 \catcode 64=11\relax
 529
 530 \expandafter\ifx\csname @tempcnta\endcsname\relax
 531 \csname newcount\endcsname\@tempcnta
 532 \fi
 533 \expandafter\ifx\csname @tempcntb\endcsname\relax
 534 \csname newcount\endcsname\@tempcntb
 535 \fi
 536

A macro that sets the catcode of a range of characters. The first parameter is the character number of the first character of the range, the second parameter is one for the last character, and the third parameter is the catcode we want them to have.

537
 538 \def\luatexsetcatcoderange#1#2#3{%
 539 \edef\luaSCR@temp{%
 540 \noexpand\@tempcnta=\the\@tempcnta
 541 \noexpand\@tempcntb=\the\@tempcntb
 542 \noexpand\count@=\the\count@
 543 \relax
 544 }%
 545 \@tempcnta=#1\relax
 546 \@tempcntb=#2\relax
 547 \count@=#3\relax
 548 \loop\unless\ifnum\@tempcnta>\@tempcntb
 549 \catcode\@tempcnta=\count@
 550 \advance\@tempcnta by 1\relax
 551 \repeat
 552 \luaSCR@temp
 553 }
 554

Finally we create several catcodetables that may be used by the user. These are:

- \CatcodeTableIniTeX: the base T_EX catcodes
- \CatcodeTableString: almost all characters have catcode 12
- \CatcodeTableOther: all characters have catcode 12 (even space)

- \CatcodeTableLaTeX: the L^AT_EX classical catcodes

```

555
556 \newluatexcatcodetable\CatcodeTableIniTeX
557 \newluatexcatcodetable\CatcodeTableString
558 \newluatexcatcodetable\CatcodeTableOther
559 \newluatexcatcodetable\CatcodeTableLaTeX
560 \initluatexcatcodetable\CatcodeTableIniTeX
561
562 \expandafter\ifx\csname @firstofone\endcsname\relax
563   \long\def\@firstofone#1{#1}%
564 \fi
565
566 \begingroup
567   \def\@makeother#1{\catcode#1=12\relax}%
568   \@firstofone{%
569     \luatexcatcodetable\CatcodeTableIniTeX
570     \begingroup
571       \luatexsetcatcoderange{0}{8}{15}%
572       \catcode9=10 % tab
573       \catcode11=15 %
574       \catcode12=13 % form feed
575       \luatexsetcatcoderange{14}{31}{15}%
576       \catcode35=6 % hash
577       \catcode36=3 % dollar
578       \catcode38=4 % ampersand
579       \catcode94=7 % circumflex
580       \catcode95=8 % underscore
581       \catcode123=1 % brace left
582       \catcode125=2 % brace right
583       \catcode126=13 % tilde
584       \catcode127=15 %
585       \saveluatexcatcodetable\CatcodeTableLaTeX
586     \endgroup
587     \@makeother{0}% nul
588     \@makeother{13}% carriage return
589     \@makeother{37}% percent
590     \@makeother{92}% backslash
591     \@makeother{127}%
592     \luatexsetcatcoderange{65}{90}{12}% A-Z
593     \luatexsetcatcoderange{97}{122}{12}% a-z
594     \saveluatexcatcodetable\CatcodeTableString
595     \@makeother{32}% space
596     \saveluatexcatcodetable\CatcodeTableOther
597   \endgroup
598 }
599
600 \ltxtra@AtEnd
601

```

We provide some functions for backward compatibility with old versions of luatextra.

```

602
603 \let\newluaattribute\newluatexattribute
604 \let\luaattribute\luatexattribute
605 \let\unsetluaattribute\unsetluatexattribute
606 \let\initluacatcodetable\initluatexcatcodetable
607 \let\luasetcatcoderange\luatexsetcatcoderange
608 \let\newluacatcodetable\newluatexcatcodetable
609 \let\setluaattribute\setluatexattribute
610 \let\luaModuleError\luatexModuleError
611 \let\luaRequireModule\luatexRequireModule
612 \let\luaUseModule\luatexUseModule
613

```

Finally, we load luaotfload.

```

614
615 \expandafter\ifx\csname ProvidesPackage\endcsname\relax
616   \input luaotfload.sty
617 \else
618   \RequirePackage{luaotfload}
619 \fi
620

```

4 luatextra-latex.tex

This file is very small, it just changes the maximum values of allowed registers from 32768 to 65536, and remaps `\newcount` (and friends) to `etex's \globcount`.

```

621 \def\ltxtra@temp#1{%
622 \ifnum\count27#1=32768 %
623   \count27#1=65536 %
624 \fi
625 }%
626 \ltxtra@temp0%
627 \ltxtra@temp1%
628 \ltxtra@temp2%
629 \ltxtra@temp3%
630 \ltxtra@temp4%
631 \ltxtra@temp5%
632 \ltxtra@temp6%
633 \let\newcount\globcount
634 \let\newdimen\globdimen
635 \let\newskip\globskip
636 \let\newbox\globbox

```