The luatextra package

Elie Roux
elie.roux@telecom-bretagne.eu

2009/04/15 v0.93

Abstract

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

Contents

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 LaTeX, 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 fist 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, LaTeX 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 LaTeX 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 LaTeX 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 higly possible that LuaTeX will replace pdfTeX as the default LaTeX engine, so it is necessary to keep backward compatibility. This lead us to the decision of renaming LuaTeX-only primitives so that they all start by `lua`, like the pdfTeX-only primitives start by `pdf`. Thus attributes become luaattributes, `\directlua` becomes `\luadirect`, etc. This also allows primitives to keep having the same name, even if their name is changed later at the engine level.

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 ε-TeXallows 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 extends the allocation scheme. luatextra loads etex, and overrides somes 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 `luaattribute`s. 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 get more control on the module system, by implementing something close to the LaTeX's `\usepackage` and `\RequirePackage` macros: the `\luaUseModule` and `\luaRequireModule` 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 agregate 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. Unfortunatly 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
```

```
    local luatextrapath = kpse.find_file("luatextra.lua")
    if luatextrapath then
        if luatextrapath:sub(1,2) == "./" then
            luatextrapath = luatextrapath:sub(3)
        end
        texio.write(' ('..luatextrapath)
    end
end

```

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

```

luatextra = {}

module("luatextra", package.seeall)

```

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.

```

luatextra.modules = {}

luatextra.modules['luatextra'] = {
    version     = 0.93,
    name        = "luatextra",
    date        = "2009/04/15",
    description = "Additional low level functions for LuaTeX",
    author      = "Elie Roux",
    copyright   = "Elie Roux, 2009",
    license     = "CC0",
}

local format = string.format

```

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

```

luatextra.internal_warning_spaces = "                    "

function luatextra.internal_warning(msg)
    if not msg then return end
    texio.write_nl(format("\nLuaTeXtra Warning: %s\n\n", msg))
end

luatextra.internal_error_spaces = "                "

function luatextra.internal_error(msg)
```

```
    if not msg then return end
    tex.sprint(format("\\immediate\\write16{}\\errmessage{LuaTeXtra error: %s^^J^^J}", msg))
end
```

2.2 Error, warning and info function for modules

Some module printing functions are provided, they have the same philosophy as the LaTeX'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.

```
function luatextra.module_error(package, msg, helpmsg)
    if not package or not msg then
        return
    end
    if helpmsg then
        tex.sprint(format("\\errhelp{%s}", helpmsg))
    end
    tex.sprint(format("\\luaModuleError{%s}{%s}", package, msg))
end

function luatextra.module_warning(modulename, msg)
    if not modulename or not msg then
        return
    end
    texio.write_nl(format("\nModule %s Warning: %s\n\n", modulename, msg))
end

function luatextra.module_log(modulename, msg)
    if not modulename or not msg then
        return
    end
    texio.write_nl('log', format("%s: %s", modulename, msg))
end

function luatextra.module_term(modulename, msg)
    if not modulename or not msg then
        return
    end
    texio.write_nl('term', format("%s: %s", modulename, msg))
end

function luatextra.module_info(modulename, msg)
    if not modulename or not msg then
        return
    end
    texio.write_nl(format("%s: %s\n", modulename, msg))
end
```

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.

```
function luatextra.find_module_file(name)
    if string.sub(name, -4) ~= '.lua' then
        name = name..'.lua'
    end
    path = kpse.find_file(name, 'tex')
    if not path then
      path = kpse.find_file(name, 'texmfscripts')
    end
    return path, name
end
```

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

```
luatextra.module = module
```

`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`.

```
function luatextra.use_module(name)
    if not name or luatextra.modules[name] then
        return
    end
    local path, filename = luatextra.find_module_file(name)
    if not path then
        luatextra.internal_error(format("unable to find lua module %s", name))
    else
        if path:sub(1,2) == "./" then
            path = path:sub(3)
        end
        texio.write(' ('..path)
        dofile(path)
        if not luatextra.modules[name] then
            luatextra.internal_warning(format("You have requested module `%s',\n%s but the f
        end
        if not package.loaded[name] then
            luatextra.module(name, package.seeall)
        end
```

```
        texio.write(')')
    end
end
```

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.

```
function luatextra.datetonumber(date)
    numbers = string.gsub(date, "(%d+)/(%d+)/(%d+)", "%1%2%3")
    return tonumber(numbers)
end

function luatextra.isdate(date)
    for _, _ in string.gmatch(date, "%d+/%d+/%d+") do
        return true
    end
    return false
end

local date, number = 1, 2

function luatextra.versiontonumber(version)
    if luatextra.isdate(version) then
        return {type = date, version = luatextra.datetonumber(version), orig = version}
    else
        return {type = number, version = tonumber(version), orig = version}
    end
end

luatextra.requiredversions = {}
```

`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).

```
function luatextra.require_module(name, version)
    if not name then
        return
    end
    if not version then
        return luatextra.use_module(name)
    end
    luaversion = luatextra.versiontonumber(version)
    if luatextra.modules[name] then
        if luaversion.type == date then
            if luatextra.datetonumber(luatextra.modules[name].date) < luaversion.version the
```

```
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 LaTeX 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.datetonumber
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.ve
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 luatextra.use_module('luaextra')
199
```

`luatextra.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 luatextra.kpse_module_loader(mod)
202   local file = luatextra.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[luatextra.kpse_module_loader] Loading error:\n\t"
210            .. error
211   end
212   return "\n\t[luatextra.kpse_module_loader] Search failed"
213 end
214
215 table.insert(package.loaders, luatextra.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 `luatextra.attributedef_from_tex`. This function registers the number in the table `tex.attributenumber`. For example to get the number of the attribute `myattribute` registered with `\newluaattribute\myattribute`, you can simply call `tex.attributenumber[myattribute]`.

```
217
218 luatextra.attributes = {}
219
220 tex.attributenumber = luatextra.attributes
221
222 function luatextra.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 overriden.
- `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)
```

```
    path = env.path
    if not path then
        return
    end
    local f = env.file
    if not f then
        f = io.open(path)
        env.file = f
    end
    if not f then
        return
    end
    env.reader = luatextra.reader
    env.close = luatextra.close
    return env
end

```

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

```


function luatextra.reader(env)
    local line = (env.file):read()
    line = luamcallbacks.call('file_reader', env, line)
    return line
end

function luatextra.close(env)
    (env.file):close()
    luamcallbacks.call('file_close', env)
end

```

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.

```


function luatextra.default_reader(env, line)
    return line
end

function luatextra.default_close(env)
    return
end

function luatextra.default_pre_read(env)
    return env
end

```

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 `\ltxtra@RegisterFontCallback`) to know how to load this mechanism anyway.

```
289
290 do
291   if tex.luatexversion < 36 then
292       fontloader = fontforge
293   end
294 end
295
296 function luatextra.find_font(name)
297     local types = {'ofm', 'ovf', 'opentype fonts', 'truetype fonts'}
298     local path = kpse.find_file(name)
299     if path then return path end
300     for _,t in pairs(types) do
301         path = kpse.find_file(name, t)
302         if path then return path end
303     end
304     return nil
305 end
306
307 function luatextra.font_load_error(error)
308     luatextra.module_warning('luatextra', string.format('%s\nloading lmr10 instead...', erro
309 end
310
```

```
311 function luatextra.load_default_font(size)
312     return font.read_tfm("lmr10", size)
313 end
314
315 function luatextra.define_font(name, size)
316     if (size < 0) then size = (- 655.36) * size end
317     local fontinfos = {
318         asked_name = name,
319         name = name,
320         size = size
321         }
322     callback.call('font_syntax', fontinfos)
323     name = fontinfos.name
324     local path = fontinfos.path
325     if not path then
326         path = luatextra.find_font(name)
327         fontinfos.path = luatextra.find_font(name)
328     end
329     if not path then
330         luatextra.font_load_error("unable to find font "..name)
331         return luatextra.load_default_font(size)
332     end
333     if not fontinfos.filename then
334         fontinfos.filename = fpath.basename(path)
335     end
336     local ext = fpath.suffix(path)
337     local f
338     if ext == 'tfm' or ext == 'ofm' then
339         f =  font.read_tfm(name, size)
340     elseif ext == 'vf' or ext == 'ovf' then
341         f =  font.read_vf(name, size)
342     elseif ext == 'ttf' or ext == 'otf' or ext == 'ttc' then
343         f = callback.call('open_otf_font', fontinfos)
344     else
345         luatextra.font_load_error("unable to determine the type of font "..name)
346         f = luatextra.load_default_font(size)
347     end
348     if not f then
349         luatextra.font_load_error("unable to load font "..name)
350         f = luatextra.load_default_font(size)
351     end
352     callback.call('post_font_opening', f, fontinfos)
353     return f
354 end
355
356 function luatextra.default_font_syntax(fontinfos)
357     return
358 end
359
360 function luatextra.default_open_otf(fontinfos)
```

```
    return nil
end

function luatextra.default_post_font(f, fontinfos)
    return true
end

function luatextra.register_font_callback()
    callback.add('define_font', luatextra.define_font, 'luatextra.define_font')
end

do
    luatextra.use_module('luamcallbacks')
    callback.create('pre_read_file', 'simple', luatextra.default_pre_read)
    callback.create('file_reader', 'data', luatextra.default_reader)
    callback.create('file_close', 'simple', luatextra.default_close)
    callback.add('open_read_file', luatextra.open_read_file, 'luatextra.open_read_file')
    callback.create('font_syntax', 'simple', luatextra.default_font_syntax)
    callback.create('open_otf_font', 'first', luatextra.default_open_otf)
    callback.create('post_font_opening', 'simple', luatextra.default_post_font)

    if luatextrapath then
        texio.write(')')
    end
end
```

3 luatextra.sty

3.1 Initializations

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

```
\csname ifluatextraloaded\endcsname
\let\ifluatextraloaded\endinput

```

Then we load ifluatex.

```

\expandafter\ifx\csname ProvidesPackage\endcsname\relax
  \expandafter\ifx\csname ifluatex\endcsname\relax
    \input ifluatex.sty
  \fi
\else
  \RequirePackage{ifluatex}
\fi

\expandafter\ifx\csname 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.

```
\ifluatex\else
  \immediate\write16{}
  \errmessage{Package luatextra Error: This package must be used with LuaTeX}
\fi
\def\luaRequireModule#1#2{\luadirect{luatextra.require_module([[#1]], [[#2]])}}
\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 package) and one optional (the version or the date). We also define the environment `luacode`.

```
\ifluatex\else
  \PackageError{luatextra}{This package must be used with LuaTeX.}
\fi
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luatextra}
  [2009/04/15 v0.93 LuaTeX extra low-level macros]
\RequirePackage{environ}
\NewEnviron{luacode}{\luadirect{\BODY}}
\newcommand\luaRequireModule[2][0]{\luadirect{luatextra.require_module([[#2]], [[#1]])}}
```

We also require the package etex to be loaded. The `\input` is a hack that modifies some values in the register attribution scheme of ε-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.

```
  \RequirePackage{etex}[1998/03/26]%
  \input luatextra-latex.tex
\fi

```

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).

```

\def\LuaTeX{Lua\TeX }
\def\LuaLaTeX{Lua\LaTeX }

```

3.2 Primitives renaming

Here we differenciate two very different cases: LuaTeX 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.

```

\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.

```
  \def\directlua{\pdfprimitive\directlua0}
  \def\latelua{\pdfprimitive\latelua0}
  \def\luadirect{\pdfprimitive\directlua0}
  \def\lualate{\pdfprimitive\latelua0}
  \def\luaattribute{\attribute}
  \def\luaattributedef{\attributedef}
  \def\luaclearmarks{\pdfprimitive\luaclearmarks}
  \def\luaformatname{\pdfprimitive\formatname}
  \def\luascantexttokens{\pdfprimitive\scantexttokens}
  \def\luacatcodetable{\catcodetable}
\else
```

For newer versions, we first enable all primitives with their default name. Then we add a prefix to some. Note that with this method, `\luaattribute` will always have the good meaning (except, of course, if it was defined before, but I think we can call it perversion...).

```
  \directlua{tex.enableprimitives('lua', {'attribute'})}
  \directlua{tex.enableprimitives('lua', {'attributedef'})}
  \directlua{tex.enableprimitives('lua', {'clearmarks'})}
  \directlua{tex.enableprimitives('lua', {'formatname'})}
  \directlua{tex.enableprimitives('lua', {'scantexttokens'})}
  \directlua{tex.enableprimitives('lua', {'catcodetable'})}
  \def\luadirect{\pdfprimitive\directlua}
  \def\lualate{\pdfprimitive\latelua}
\fi

```

Finally we add some common definitions that have lua inside their name, or that must remove their suffix.

```
\def\initluacatcodetable{\pdfprimitive\initcatcodetable}
\def\saveluacatcodetable{\pdfprimitive\savecatcodetable}
\def\luaclose{\pdfprimitive\closelua}

```

We load the `lua` file.

```

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

```

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

```

\def\ltxtra@RegisterFontCallback{
```

```
  \luadirect{luatextra.register_font_callback()}
}

```

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 `\luaModuleError` 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 `\luaUseModule` that simply calls `luatextra.use_module`. Remember that `\luaRequireModule` is defined at the beginning of this file.

```


\def\luaModuleError#1#2{%
  \errorcontextlines=0\relax
  \immediate\write16{}%
  \errmessage{Module #1 error: #2^^J^^J%
See the module #1 documentation for explanation.^^J ...^^J}%
}

\def\luaUseModule#1{\luadirect{luatextra.use_module([[#1]])}}

```

3.5 Attributes handling

The most important macro here is `\newluaattribute` 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.

```


\newcount\luaattdefcounter
\luaattdefcounter = 1

\def\newluaattribute#1{%
  \ifnum\luaattdefcounter<65535\relax %
    \global\advance\luaattdefcounter by 1\relax %
    \allocationnumber\luaattdefcounter %
    \ifluatex %
      \global\luaattributedef#1=\allocationnumber %
    \fi %
    \wlog{\string#1=\string\attribute\the\allocationnumber}%
```

```
    \luadirect{%
      luatextra.attributedef_from_tex([[\noexpand#1]], '\number\allocationnumber')}%
  \else %
    \errmessage{No room for a new \string\attribute}%
  \fi %
}

```

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

```

\def\setluaattribute#1#2{%
  #1=\numexpr#2\relax %
}

\def\unsetluaattribute#1{%
  \ifnum\luatexversion<37\relax %
    #1=-1\relax %
  \else %
    #1=-"7FFFFFFF\relax %
  \fi %
}

```

3.6 Catcodetables handling

Here we allocate catcodetables the same way we handle attributes.

```

\newcount\luacatcodetabledefcounter

\luacatcodetabledefcounter = 1

\def\newluacatcodetable#1{%
  \ifnum\luacatcodetabledefcounter<1114110\relax % 0x10FFFF is maximal \chardef
    \global\advance\luacatcodetabledefcounter by 1\relax %
    \allocationnumber=\luacatcodetabledefcounter %
    \global\chardef#1=\allocationnumber %
    \luadirect{%
      luatextra.catcodetabledef_from_tex([[\noexpand#1]], '\number\allocationnumber')}%
    \wlog{\string#1=\string\catcodetable\the\allocationnumber}%
  \else %
    \errmessage{No room for a new \string\catcodetable}%
  \fi %
}

```

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

```
\expandafter\edef\csname ltxtra@AtEnd\endcsname{%
  \catcode64 \the\catcode64\relax
}

\catcode 64=11\relax

\expandafter\ifx\csname @tempcnta\endcsname\relax
  \csname newcount\endcsname\@tempcnta
\fi
\expandafter\ifx\csname @tempcntb\endcsname\relax
  \csname newcount\endcsname\@tempcntb
\fi
```

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.

```
\def\luasetcatcoderange#1#2#3{%
  \edef\luaSCR@temp{%
    \noexpand\@tempcnta=\the\@tempcnta
    \noexpand\@tempcntb=\the\@tempcntb
    \noexpand\count@=\the\count@
    \relax
  }%
  \@tempcnta=#1\relax
  \@tempcntb=#2\relax
  \count@=#3\relax
  \loop\unless\ifnum\@tempcnta>\@tempcntb
    \catcode\@tempcnta=\count@
    \advance\@tempcnta by 1\relax
  \repeat
  \luaSCR@temp
}
```

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

- `\CatcodeTableIniTeX`: the base TEX catcodes
- `\CatcodeTableString`: almost all characters have catcode 12
- `\CatcodeTableOther`: all characters have catcode 12 (even space)
- `\CatcodeTableLaTeX`: the LATEX classical catcodes

```
\newluacatcodetable\CatcodeTableIniTeX
```

```
\newluacatcodetable\CatcodeTableString
\newluacatcodetable\CatcodeTableOther
\newluacatcodetable\CatcodeTableLaTeX
\initluacatcodetable\CatcodeTableIniTeX

\expandafter\ifx\csname @firstofone\endcsname\relax
  \long\def\@firstofone#1{#1}%
\fi

\begingroup
  \def\@makeother#1{\catcode#1=12\relax}%
  \@firstofone{%
    \luacatcodetable\CatcodeTableIniTeX
    \begingroup
      \luasetcatcoderange{0}{8}{15}%
      \catcode9=10  % tab
      \catcode11=15 %
      \catcode12=13 % form feed
      \luasetcatcoderange{14}{31}{15}%
      \catcode35=6 % hash
      \catcode36=3 % dollar
      \catcode38=4 % ampersand
      \catcode94=7 % circumflex
      \catcode95=8 % underscore
      \catcode123=1 % brace left
      \catcode125=2 % brace right
      \catcode126=13 % tilde
      \catcode127=15 %
      \saveluacatcodetable\CatcodeTableLaTeX
    \endgroup
    \@makeother{0}% nul
    \@makeother{13}% carriage return
    \@makeother{37}% percent
    \@makeother{92}% backslash
    \@makeother{127}%
    \luasetcatcoderange{65}{90}{12}% A-Z
    \luasetcatcoderange{97}{122}{12}% a-z
    \saveluacatcodetable\CatcodeTableString
    \@makeother{32}% space
    \saveluacatcodetable\CatcodeTableOther
  \endgroup
}
```

We load luaotfload.

```
\expandafter\ifx\csname ProvidesPackage\endcsname\relax
  \input luaotfload.sty
\else
  \RequirePackage{luaotfload}
```

```
\fi

\ltxtra@AtEnd
```

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`.

```
\def\ltxtra@temp#1{%
\ifnum\count27#1=32768 %
  \count27#1=65536 %
\fi
}%
\ltxtra@temp0%
\ltxtra@temp1%
\ltxtra@temp2%
\ltxtra@temp3%
\ltxtra@temp4%
\ltxtra@temp5%
\ltxtra@temp6%
\let\newcount\globcount
\let\newdimen\globdimen
\let\newskip\globskip
\let\newbox\globbox
```