The luatextra package

Elie Roux
elie.roux@telecom-bretagne.eu

2010/05/10 v0.97

Abstract

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.

Warning. This package is undergoing major changes, so the documentation may sometimes be inconsistent. Also, be aware that everything may change.

Contents

1 `luatextra.lua`

1.1 Initialization and internal functions

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

```
1 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.

```
luatexbase.provides_module {
    version     = 0.97,
    name        = "luatextra",
    date        = "2010/05/10",
    description = "Additional low level functions for LuaTeX",
    author      = "Elie Roux and Manuel Pegourie-Gonnard",
    copyright   = "Elie Roux, 2009 and Manuel Pegourie-Gonnard, 2010",
    license     = "CC0",
}
local format = string.format
```

1.2 Multiple callbacks on the `open_read_file` callback

`luatexbase` (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.

```
function luatextra.open_read_file(filename)
    local path = kpse.find_file(filename)
    local env = {
      ['filename'] = filename,
      ['path'] = path,
    }
    luatexbase.call_callback('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 = luatexbase.call_callback('file_reader', env, line)
    return line
end
function luatextra.close(env)
    (env.file):close()
    luatexbase.call_callback('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 `luatexbase` 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
```

1.3 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.

```
do
  if tex.luatexversion < 36 then
     fontloader = fontforge
  end
end
function luatextra.find_font(name)
    local types = {'ofm', 'ovf', 'opentype fonts', 'truetype fonts'}
    local path = kpse.find_file(name)
    if path then return path end
    for _,t in pairs(types) do
        path = kpse.find_file(name, t)
        if path then return path end
    end
    return nil
end
function luatextra.font_load_error(error)
    luatextra.module_warning('luatextra', string.format('%s\nloading lmr10 instead...', erro
end
function luatextra.load_default_font(size)
    return font.read_tfm("lmr10", size)
end
function luatextra.define_font(name, size)
    if (size < 0) then size = (- 655.36) * size end
    local fontinfos = {
        asked_name = name,
        name = name,
        size = size
        }
    callback.call('font_syntax', fontinfos)
    name = fontinfos.name
    local path = fontinfos.path
    if not path then
        path = luatextra.find_font(name)
        fontinfos.path = luatextra.find_font(name)
```

```
87      end
88      if not path then
89          luatextra.font_load_error("unable to find font "..name)
90          return luatextra.load_default_font(size)
91      end
92      if not fontinfos.filename then
93          fontinfos.filename = file.basename(path)
94      end
95      local ext = file.suffix(path)
96      local f
97      if ext == 'tfm' or ext == 'ofm' then
98          f =  font.read_tfm(name, size)
99      elseif ext == 'vf' or ext == 'ovf' then
100         f =  font.read_vf(name, size)
101     elseif ext == 'ttf' or ext == 'otf' or ext == 'ttc' then
102         f = luatexbase.call_callback('open_otf_font', fontinfos)
103     else
104         luatextra.font_load_error("unable to determine the type of font "..name)
105         f = luatextra.load_default_font(size)
106     end
107     if not f then
108         luatextra.font_load_error("unable to load font "..name)
109         f = luatextra.load_default_font(size)
110     end
111     luatexbase.call_callback('post_font_opening', f, fontinfos)
112     return f
113 end
114 function luatextra.default_font_syntax(fontinfos)
115     return
116 end
117 function luatextra.default_open_otf(fontinfos)
118     return nil
119 end
120 function luatextra.default_post_font(f, fontinfos)
121     return true
122 end
123 function luatextra.register_font_callback()
124     luatexbase.add_to_callback('define_font', luatextra.define_font, 'luatextra.define_font'
125 end
```

Initialise a few callbacks.

```
126     luatexbase.create_callback('pre_read_file', 'simple', luatextra.default_pre_read)
127     luatexbase.create_callback('file_reader', 'data', luatextra.default_reader)
128     luatexbase.create_callback('file_close', 'simple', luatextra.default_close)
129     luatexbase.add_to_callback('open_read_file', luatextra.open_read_file, 'luatextra.open_r
130     luatexbase.create_callback('font_syntax', 'simple', luatextra.default_font_syntax)
131     luatexbase.create_callback('open_otf_font', 'first', luatextra.default_open_otf)
132     luatexbase.create_callback('post_font_opening', 'simple', luatextra.default_post_font)
```

2 luatextra.sty

2.1 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}
  \NeedsTeXFormat{LaTeX2e}
  \ProvidesPackage{luatextra}
    [2010/05/10 v0.97 LuaTeX extra low-level macros]
\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).

These two macros are the only two loaded if we are under a non-LuaTEX engine.

```

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

```

Make sure LuaTEX is being used.

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname RequirePackage\endcsname\relax
  \input ifluatex.sty
\else
  \RequirePackage{ifluatex}
\fi
\ifluatex\else
  \begingroup
    \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
      \def\x#1#2{\begingroup\newlinechar10
        \immediate\write16{Package #1 warning: #2}\endgroup}
    \else
      \let\x\PackageWarningNoLine
    \fi
  \expandafter\endgroup
  \x{luatexbase-modutils}{LuaTeX is required for this package.^^J
```

```
    Aborting package loading.}
  \expandafter\endinput
\fi
```

Load other usefull packages.

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname RequirePackage\endcsname\relax
  \input luatexbase-modutils.sty
  \input luatexbase-attr.sty
  \input luatexbase-cctb.sty
  \input luatexbase-regs.sty
  \input luatexbase-mcb.sty
\else
  \RequirePackage{luatexbase-modutils}
  \RequirePackage{luatexbase-attr}
  \RequirePackage{luatexbase-cctb}
  \RequirePackage{luatexbase-regs}
  \RequirePackage{luatexbase-mcb}
\fi
\luatexUseModule{lualibs}
```

If the package is loaded with LaTeX, we also define the environment `luacode`.

```
\expandafter\ifx\csname RequirePackage\endcsname\relax \else
  \RequirePackage{environ}
  \NewEnviron{luacode}{\directlua{\BODY}}
\fi
```

2.2 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\luatexattribute{\attribute}
  \def\luatexattributedef{\attributedef}
  \def\luatexclearmarks{\pdfprimitive\luaclearmarks}
  \def\luatexformatname{\pdfprimitive\formatname}
```

```
  \def\luatexscantexttokens{\pdfprimitive\scantexttokens}
  \def\luatexcatcodetable{\catcodetable}
  \def\initluatexcatcodetable{\pdfprimitive\initcatcodetable}
  \def\saveluatexcatcodetable{\pdfprimitive\savecatcodetable}
  \def\luaclose{\pdfprimitive\closelua}
\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.

```
  \directlua{tex.enableprimitives('luatex', {'attribute'})}
  \directlua{tex.enableprimitives('luatex', {'attributedef'})}
  \directlua{tex.enableprimitives('luatex', {'clearmarks'})}
  \directlua{tex.enableprimitives('luatex', {'formatname'})}
  \directlua{tex.enableprimitives('luatex', {'scantexttokens'})}
  \directlua{tex.enableprimitives('luatex', {'catcodetable'})}
  \directlua{tex.enableprimitives('luatex', {'latelua'})}
  \directlua{tex.enableprimitives('luatex', {'initcatcodetable'})}
  \directlua{tex.enableprimitives('luatex', {'savecatcodetable'})}
  \directlua{tex.enableprimitives('luatex', {'closelua'})}
  \let\luadirect\directlua
  \let\lualate\luatexlatelua
  \let\initluatexcatcodetable\luatexinitcatcodetable
  \let\saveluatexcatcodetable\luatexsavecatcodetable
  \let\luaclose\luatexcloselua
\fi
```

We load the `lua` file.

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

A small macro to register the `define_font` callback from luatextra. See section 1.3 for more details.

```
\def\ltxtra@RegisterFontCallback{
  \directlua{luatextra.register_font_callback()}
}
```

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

```
\let\newluaattribute\newluatexattribute
\let\luaattribute\luatexattribute
\let\unsetluaattribute\unsetluatexattribute
\let\initluacatcodetable\initluatexcatcodetable
\let\luasetcatcoderange\luatexsetcatcoderange
\let\newluacatcodetable\newluatexcatcodetable
\let\setluaattribute\setluatexattribute
\let\luaModuleError\luatexModuleError
\let\luaRequireModule\luatexRequireModule
\let\luaUseModule\luatexUseModule
```

Finally, we load luaotfload.

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