The luatextra package

Manuel Pégourié-Gonnard mpg@elzevir.fr
Élie Roux elie.roux@telecom-bretagne.eu

2010/11/08 v0.99

Abstract

The luatextra package loads essential and convenient packages for LuaTEX, and provides few additional user-level goodies. It is meant as a convenience package for users. Developers should load directly the underlying packages they specifically need.

Warning. This package is undergoing major changes, see the "Planned changes" section on page 2.

Contents

1 Documentation

The luatextra package loads the following essential packages:

- luatexbase: low-level functions for resources handling and compatibility. Divided in sub-packages, like luatexbase-mcb (formerly luamcallbacks) that allows to register several functions in a callback;
- lualibs: additional lua functions (previously luaextra) ;

- luaotfload: implements extended syntax *à la* XeTeX for the `\font` primitive.

When running LaTeX luatextra loads the following additional packages:

- metalogo: provides commands for typesetting the LuaTeX and LuaLaTeX logos amongst others;
- luacode: provides tools for easy integration of Lua code in LaTeX;
- fixltx2e: fixes bugs or suboptimal things in the LaTeX kernel.

In case you are not yet familiar with the available LuaLaTeX packages, you might want to check the document `lualatex-doc.pdf` from the eponymous package.

1.1 Planned changes

If, for some reason, you disagree with one of the upcoming changes described below, please discuss it on the development list: lualatex-dev@tug.org.

The most major change planned is to make this package compatible with LaTeX only. Rationale: it is (or will soon be) only a convenience package loading other packages. Users of Plain TeX are assumed not to care that much about such convenience. Then, fontspec would be loaded rather than luaotfload.

Note that luatextra currently expands on luatexbase-mcb by proposing solutions for the `define_font` and `open_read_file` callbacks. See the implementation section below for details. In the future, this code (only half of which was activated) will be removed, since it seems very unlikely that different packages want to (or are able to usefully) share these callbacks. The `luatextra` Lua namespace (aka table) will then disappear.

All name aliases (for primitives and macro from luatexbase and previous versions) will be dropped, so as to maintain a the name space clean. Please check section 2.2.2 for the complete list of what will be dropped.

2 Implementation

2.1 Lua module

2.1.1 Initialization and internal functions

1 ⟨*lua⟩

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

```
2 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     = 0.99,
    name        = "luatextra",
    date        = "2010/10/08",
    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
```

2.1.2 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
```

```
26         f = io.open(path)
27         env.file = f
28     end
29     if not f then
30         return
31     end
32     env.reader = luatextra.reader
33     env.close = luatextra.close
34     return env
35 end
```

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

```
36 function luatextra.reader(env)
37     local line = (env.file):read()
38     line = luatexbase.call_callback('file_reader', env, line)
39     return line
40 end
41 function luatextra.close(env)
42     (env.file):close()
43     luatexbase.call_callback('file_close', env)
44 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.

```
45 function luatextra.default_reader(env, line)
46     return line
47 end
48 function luatextra.default_close(env)
49     return
50 end
51 function luatextra.default_pre_read(env)
52     return env
53 end
```

2.1.3 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...', error))
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)
    end
    if not path then
        luatextra.font_load_error("unable to find font "..name)
```

```
        return luatextra.load_default_font(size)
    end
    if not fontinfos.filename then
        fontinfos.filename = file.basename(path)
    end
    local ext = file.suffix(path)
    local f
    if ext == 'tfm' or ext == 'ofm' then
        f =  font.read_tfm(name, size)
    elseif ext == 'vf' or ext == 'ovf' then
        f =  font.read_vf(name, size)
    elseif ext == 'ttf' or ext == 'otf' or ext == 'ttc' then
        f = luatexbase.call_callback('open_otf_font', fontinfos)
    else
        luatextra.font_load_error("unable to determine the type of font "..name)
        f = luatextra.load_default_font(size)
    end
    if not f then
        luatextra.font_load_error("unable to load font "..name)
        f = luatextra.load_default_font(size)
    end
    luatexbase.call_callback('post_font_opening', f, fontinfos)
    return f
end
function luatextra.default_font_syntax(fontinfos)
    return
end
function luatextra.default_open_otf(fontinfos)
    return nil
end
function luatextra.default_post_font(f, fontinfos)
    return true
end
function luatextra.register_font_callback()
    luatexbase.add_to_callback('define_font', luatextra.define_font, 'luatextra.define_font')
end
```

Initialise a few callbacks.

```
    luatexbase.create_callback('pre_read_file', 'simple', luatextra.default_pre_read)
    luatexbase.create_callback('file_reader', 'data', luatextra.default_reader)
    luatexbase.create_callback('file_close', 'simple', luatextra.default_close)
    luatexbase.add_to_callback('open_read_file', luatextra.open_read_file, 'luatextra.open_read
    luatexbase.create_callback('font_syntax', 'simple', luatextra.default_font_syntax)
    luatexbase.create_callback('open_otf_font', 'first', luatextra.default_open_otf)
    luatexbase.create_callback('post_font_opening', 'simple', luatextra.default_post_font)
```

⟨/lua⟩

2.2 TEX package

```
⟨*package⟩
```

2.2.1 Initializations

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

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

```

Then we load ifluatex and identify.

```

\bgroup\expandafter\expandafter\expandafter\egroup
\expandafter\ifx\csname ProvidesPackage\endcsname\relax
  \expandafter\ifx\csname ifluatex\endcsname\relax
    \input ifluatex.sty
  \fi
\else
  \RequirePackage{ifluatex}
  \NeedsTeXFormat{LaTeX2e}
  \ProvidesPackage{luatextra}
    [2010/10/08 v0.98 LuaTeX extra low-level macros]
\fi

```

Make sure LuaTEX is being used.

```
\ifluatex\else
  \begingroup
    \expandafter\ifx\csname PackageError\endcsname\relax
      \def\x#1#2#3{\begingroup \newlinechar10
        \errhelp{#3}\errmessage{Package #1 error: #2}\endgroup}
    \else
      \let\x\PackageError
    \fi
  \expandafter\endgroup
  \x{luatextra}{LuaTeX is required for this package. Aborting.}{%
    This package can only be used with the LuaTeX engine^^J%
    (command 'lualatex' or 'luatex').^^J%
    Package loading has been stopped to prevent additional errors.}
  \expandafter\endinput
\fi
```

Load packages.

```
\bgroup\expandafter\expandafter\expandafter\egroup
\expandafter\ifx\csname ProvidesPackage\endcsname\relax
  \input luatexbase.sty
  \input luatexbase-modutils.sty
  \input luatexbase-mcb.sty
  \luatexUseModule{lualibs}
  \input luaotfload.sty
\else
```

```
  \RequirePackage{luatexbase}
  \RequirePackage{luatexbase-modutils}
  \RequirePackage{luatexbase-mcb}
  \luatexUseModule{lualibs}
  \RequirePackage{luaotfload}
  %
  \RequirePackage{metalogo}
  \RequirePackage{luacode}
  \RequirePackage{fixltx2e}
\fi
```

2.2.2 Primitives and macros renaming

Warning. This entire section will be dropped in a future release. Check section 1.1 for what to do if you deperately need it.

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\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\lualate\luatexlatelua
  \let\initluatexcatcodetable\luatexinitcatcodetable
  \let\saveluatexcatcodetable\luatexsavecatcodetable
  \let\luaclose\luatexcloselua
\fi
```

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
```

2.3 Callbacks

We load the `lua` file.

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

A small macro to register the `define_font` callback from luatextra.

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

⟨/package⟩

3 Test file

Very minimal, just check that the package correctly loads.

```
⟨*test⟩
\RequirePackage{luatextra}
\stop
⟨/test⟩
```