The luamcallbacks package

Elie Roux
elie.roux@telecom-bretagne.eu

2009/09/18 v0.93

Abstract

This package manages the callback adding and removing, by adding `callback.add` and `callback.remove`, and overwriting `callback.register`. It also allows to create and call new callbacks. For an introduction on this package (among others), please refer to the document `luatextra-reference.pdf`.

1 Documentation

LuaTEX provides an extremely interesting feature, named callbacks. It allows to call some lua functions at some points of the TEX algorithm (a *callback*), like when TEX breaks likes, puts vertical spaces, etc. The LuaTEX core offers a function called `callback.register` that enables to register a function in a callback.

The problem with `callback.register` is that is registers only one function in a callback. For a lot of callbacks it can be common to have several packages registering their function in a callback, and thus it is impossible with them to be compatible with each other.

This package solves this problem by adding mainly one new function `callback.add` that adds a function in a callback. With this function it is possible for packages to register their function in a callback without overwriting the functions of the other packages.

The functions are called in a certain order, and when a package registers a callback it can assign a priority to its function. Conflicts can still remain even with the priority mechanism, for example in the case where two packages want to have the highest priority. In these cases the packages have to solve the conflicts themselves.

This package also privides a way to create and call new callbacks, in addition to the default LuaTEX callbacks.

This package contains only a `.lua` file, that can be called by another lua script. For example, this script is called in luatextra.

Limitations

This package only works for callbacks where it's safe to add multiple functions without changing the functions' signatures. There are callbacks, though, where

registering several functions is not possible without changing the function's signatures, like for example the readers callbacks. These callbacks take a filename and give the datas in it. One solution would be to change the functions' signature to open it when the function is the first, and to take the datas and modify them eventually if they are called after the first. But it seems rather fragile and useless, so it's not implemented. With these callbacks, in this package we simply execute the first function in the list.

Other callbacks in this case are `define_font` and `open_read_file`. There is though a solution for several packages to use these callbacks, see the implementation of `luatextra`.

2 Package code

The package contains `luamcallbacks.lua` with the new functions, and an example of the use of luamcallbacks.

First the `luamcallbacks` module is registered as a LuaTEX module, with some informations.

```
1
2 luamcallbacks          = { }
3
4 luamcallbacks.module = {
5     name          = "luamcallbacks",
6     version       = 0.93,
7     date          = "2009/09/18",
8     description   = "Module to register several functions in a callback.",
9     author        = "Hans Hagen & Elie Roux",
10    copyright     = "Hans Hagen & Elie Roux",
11    license       = "CC0",
12 }
13
14 luatextra.provides_module(luamcallbacks.module)
15
```

`callbacklist` is the main list, that contains the callbacks as keys and a table of the registered functions a values.

```
16
17 luamcallbacks.callbacklist = luamcallbacks.callbacklist or { }
18
```

A table with the default functions of the created callbacks. See `luamcallbacks.create` for further informations.

```
19
20 luamcallbacks.lua_callbacks_defaults = { }
21
22 local format = string.format
23
```

There are 4 types of callback:

- the ones taking a list of nodes and returning a boolean and eventually a new head (`list`)
- the ones taking datas and returning the modified ones (`data`)
- the ones that can't have multiple functions registered in them (`first`)
- the ones for functions that don't return anything (`simple`)

```
24
25 local list = 1
26 local data = 2
27 local first = 3
28 local simple = 4
29
```

`callbacktypes` is the list that contains the callbacks as keys and the type (list or data) as values.

```
30
31 luamcallbacks.callbacktypes = luamcallbacks.callbacktypes or {
32 buildpage_filter = simple,
33 token_filter = first,
34 pre_output_filter = list,
35 hpack_filter = list,
36 process_input_buffer = data,
37 mlist_to_hlist = list,
38 vpack_filter = list,
39 define_font = first,
40 open_read_file = first,
41 linebreak_filter = list,
42 post_linebreak_filter = list,
43 pre_linebreak_filter = list,
44 start_page_number = simple,
45 stop_page_number = simple,
46 start_run = simple,
47 show_error_hook = simple,
48 stop_run = simple,
49 hyphenate = simple,
50 ligaturing = simple,
51 kerning = data,
52 find_write_file = first,
53 find_read_file = first,
54 find_vf_file = data,
55 find_map_file = data,
56 find_format_file = data,
57 find_opentype_file = data,
58 find_output_file = data,
59 find_truetype_file = data,
60 find_type1_file = data,
61 find_data_file = data,
62 find_pk_file = data,
```

```
find_font_file = data,
find_image_file = data,
find_ocp_file = data,
find_sfd_file = data,
find_enc_file = data,
read_sfd_file = first,
read_map_file = first,
read_pk_file = first,
read_enc_file = first,
read_vf_file = first,
read_ocp_file = first,
read_opentype_file = first,
read_truetype_file = first,
read_font_file = first,
read_type1_file = first,
read_data_file = first,
}

```

In LuaTEX version 0.43, a new callback called `process_output_buffer` appeared, so we enable it.

```

if tex.luatexversion > 42 then
    luamcallbacks.callbacktypes["process_output_buffer"] = data
end

```

As we overwrite `callback.register`, we save it as `luamcallbacks.internalregister`. After that we declare some functions to write the errors or the logs.

```

luamcallbacks.internalregister = luamcallbacks.internalregister or callback.register

local callbacktypes = luamcallbacks.callbacktypes

luamcallbacks.log = luamcallbacks.log or function(...)
  luatextra.module_log('luamcallbacks', format(...))
end

luamcallbacks.info = luamcallbacks.info or function(...)
  luatextra.module_info('luamcallbacks', format(...))
end

luamcallbacks.warning = luamcallbacks.warning or function(...)
  luatextra.module_warning('luamcallbacks', format(...))
end

luamcallbacks.error = luamcallbacks.error or function(...)
  luatextra.module_error('luamcallbacks', format(...))
end

```

A simple function we'll use later to understand the arguments of the `create` function. It takes a string and returns the type corresponding to the string or nil.

```
107
108 function luamcallbacks.str_to_type(str)
109     if str == 'list' then
110         return list
111     elseif str == 'data' then
112         return data
113     elseif str == 'first' then
114         return first
115     elseif str == 'simple' then
116         return simple
117     else
118         return nil
119     end
120 end
121
```

`luamcallbacks.create`

This first function creates a new callback. The signature is `create(name, ctype, default)` where `name` is the name of the new callback to create, `ctype` is the type of callback, and `default` is the default function to call if no function is registered in this callback.

The created callback will behave the same way LuaTEX callbacks do, you can add and remove functions in it. The difference is that the callback is not automatically called, the package developer creating a new callback must also call it, see next function.

```
122
123 function luamcallbacks.create(name, ctype, default)
124     if not name then
125         luamcallbacks.error(format("unable to call callback, no proper name passed", name))
126         return nil
127     end
128     if not ctype or not default then
129         luamcallbacks.error(format("unable to create callback '%s', callbacktype or default
130         return nil
131     end
132     if callbacktypes[name] then
133         luamcallbacks.error(format("unable to create callback '%s', callback already exists'
134         return nil
135     end
136     local temp = luamcallbacks.str_to_type(ctype)
137     if not temp then
138         luamcallbacks.error(format("unable to create callback '%s', type '%s' undefined", na
139         return nil
140     end
141     ctype = temp
142     luamcallbacks.lua_callbacks_defaults[name] = default
143     callbacktypes[name] = ctype
```

```
144 end
145
```

`luamcallbacks.call` This function calls a callback. It can only call a callback created by the `create` function.

```
146
147 function luamcallbacks.call(name, ...)
148     if not name then
149         luamcallbacks.error(format("unable to call callback, no proper name passed", name))
150         return nil
151     end
152     if not luamcallbacks.lua_callbacks_defaults[name] then
153         luamcallbacks.error(format("unable to call lua callback '%s', unknown callback", nam
154         return nil
155     end
156     local l = luamcallbacks.callbacklist[name]
157     local f
158     if not l then
159         f = luamcallbacks.lua_callbacks_defaults[name]
160     else
161         if callbacktypes[name] == list then
162             f = luamcallbacks.listhandler(name)
163         elseif callbacktypes[name] == data then
164             f = luamcallbacks.datahandler(name)
165         elseif callbacktypes[name] == simple then
166             f = luamcallbacks.simplehandler(name)
167         elseif callbacktypes[name] == first then
168             f = luamcallbacks.firsthandler(name)
169         else
170             luamcallbacks.error("unknown callback type")
171         end
172     end
173     return f(...)
174 end
175
```

`luamcallbacks.add` The main function. The signature is `luamcallbacks.add (name, func, description, priority)` with `name` being the name of the callback in which the function is added; `func` is the added function; `description` is a small character string describing the function, and `priority` an optional argument describing the priority the function will have.

The functions for a callbacks are added in a list (in `luamcallbacks.callbacklist .callbackname`). If they have no priority or a high priority number, they will be added at the end of the list, and will be called after the others. If they have a low priority number, the will be added at the beginning of the list and will be called before the others.

Something that must be made clear, is that there is absolutely no solution for packages conflicts: if two packages want the top priority on a certain callback, they

will have to decide the priority they will give to their function themself. Most of the time, the priority is not needed.

```
176
177 function luamcallbacks.add (name,func,description,priority)
178     if type(func) ~= "function" then
179         luamcallbacks.error("unable to add function, no proper function passed")
180         return
181     end
182     if not name or name == "" then
183         luamcallbacks.error("unable to add function, no proper callback name passed")
184         return
185     elseif not callbacktypes[name] then
186         luamcallbacks.error(
187           format("unable to add function, '%s' is not a valid callback",
188           name))
189         return
190     end
191     if not description or description == "" then
192         luamcallbacks.error(
193           format("unable to add function to '%s', no proper description passed",
194           name))
195         return
196     end
197     if luamcallbacks.get_priority(name, description) ~= 0 then
198         luamcallbacks.warning(
199           format("function '%s' already registered in callback '%s'",
200           description, name))
201     end
202     local l = luamcallbacks.callbacklist[name]
203     if not l then
204         l = {}
205         luamcallbacks.callbacklist[name] = l
206         if not luamcallbacks.lua_callbacks_defaults[name] then
207             if callbacktypes[name] == list then
208                 luamcallbacks.internalregister(name, luamcallbacks.listhandler(name))
209             elseif callbacktypes[name] == data then
210                 luamcallbacks.internalregister(name, luamcallbacks.datahandler(name))
211             elseif callbacktypes[name] == simple then
212                 luamcallbacks.internalregister(name, luamcallbacks.simplehandler(name))
213             elseif callbacktypes[name] == first then
214                 luamcallbacks.internalregister(name, luamcallbacks.firsthandler(name))
215             else
216                 luamcallbacks.error("unknown callback type")
217             end
218         end
219     end
220     local f = {
221         func = func,
222         description = description,
```

```
223     }
224     priority = tonumber(priority)
225     if not priority or priority > #l then
226         priority = #l+1
227     elseif priority < 1 then
228         priority = 1
229     end
230     if callbacktypes[name] == first and (priority ~= 1 or #l ~= 0) then
231         luamcallbacks.warning(format("several callbacks registered in callback '%s', only th
232     end
233     table.insert(l,priority,f)
234     luamcallbacks.log(
235       format("inserting function '%s' at position %s in callback list for '%s'",
236       description,priority,name))
237 end
238
```

`luamcallbacks.get priority`

This function tells if a function has already been registered in a callback, and gives its current priority. The arguments are the name of the callback and the description of the function. If it has already been registered, it gives its priority, and if not it returns false.

```
239
240 function luamcallbacks.get_priority (name, description)
241     if not name or name == "" or not callbacktypes[name] or not description then
242         return 0
243     end
244     local l = luamcallbacks.callbacklist[name]
245     if not l then return 0 end
246     for p, f in pairs(l) do
247         if f.description == description then
248             return p
249         end
250     end
251     return 0
252 end
253
```

`luamcallbacks.remove`

The function that removes a function from a callback. The signature is `mcallbacks.remove (name, description)` with `name` being the name of callbacks, and description the description passed to `mcallbacks.add`.

```
254
255 function luamcallbacks.remove (name, description)
256     if not name or name == "" then
257         luamcallbacks.error("unable to remove function, no proper callback name passed")
258         return
259     elseif not callbacktypes[name] then
260         luamcallbacks.error(
261           format("unable to remove function, '%s' is not a valid callback",
262           name))
```

```
        return
    end
    if not description or description == "" then
        luamcallbacks.error(
          format("unable to remove function from '%s', no proper description passed",
          name))
        return
    end
    local l = luamcallbacks.callbacklist[name]
    if not l then
        luamcallbacks.error(format("no callback list for '%s'",name))
        return
    end
    for k,v in ipairs(l) do
        if v.description == description then
            table.remove(l,k)
            luamcallbacks.log(
              format("removing function '%s' from '%s'",description,name))
            if not next(l) then
              luamcallbacks.callbacklist[name] = nil
              if not luamcallbacks.lua_callbacks_defaults[name] then
                luamcallbacks.internalregister(name, nil)
              end
            end
            return
        end
    end
    luamcallbacks.warning(
      format("unable to remove function '%s' from '%s'",description,name))
end

```

`luamcallbacks.reset` This function removes all the functions registered in a callback.

```

function luamcallbacks.reset (name)
    if not name or name == "" then
        luamcallbacks.error("unable to reset, no proper callback name passed")
        return
    elseif not callbacktypes[name] then
        luamcallbacks.error(
          format("reset error, '%s' is not a valid callback",
          name))
        return
    end
    if not luamcallbacks.lua_callbacks_defaults[name] then
        luamcallbacks.internalregister(name, nil)
    end
    local l = luamcallbacks.callbacklist[name]
    if l then
        luamcallbacks.log(format("resetting callback list '%s'",name))
```

```
311         luamcallbacks.callbacklist[name] = nil
312     end
313 end
314
```

This function and the following ones are only internal. This one is the handler for the first type of callbacks: the ones that take a list head and return true, false, or a new list head.

luamcallbacks.listhandler

```
315
316 function luamcallbacks.listhandler (name)
317     return function(head,...)
318         local l = luamcallbacks.callbacklist[name]
319         if l then
320             local done = true
321             for _, f in ipairs(l) do
322                 -- the returned value can be either true or a new head plus true
323                 rtv1, rtv2 = f.func(head,...)
324                 if type(rtv1) == 'boolean' then
325                     done = rtv1
326                 elseif type (rtv1) == 'userdata' then
327                     head = rtv1
328                 end
329                 if type(rtv2) == 'boolean'  then
330                     done = rtv2
331                 elseif type(rtv2) == 'userdata' then
332                     head = rtv2
333                 end
334                 if done == false then
335                     luamcallbacks.error(format(
336                       "function \"%s\" returned false in callback '%s'",
337                       f.description, name))
338                 end
339             end
340             return head, done
341         else
342             return head, false
343         end
344     end
345 end
346
```

The handler for callbacks taking datas and returning modified ones.

luamcallbacks.datahandler

```
347
348 function luamcallbacks.datahandler (name)
349     return function(data,...)
350         local l = luamcallbacks.callbacklist[name]
```

```
        if l then
            for _, f in ipairs(l) do
                data = f.func(data,...)
            end
        end
        return data
    end
end
```

This function is for the handlers that don't support more than one functions in them. In this case we only call the first function of the list.

luamcallbacks.firsthandler

```
function luamcallbacks.firsthandler (name)
    return function(...)
        local l = luamcallbacks.callbacklist[name]
        if l then
            local f = l[1].func
            return f(...)
        else
            return nil, false
        end
    end
end
```

Handler for simple functions that don't return anything.

luamcallbacks.simplehandler

```
function luamcallbacks.simplehandler (name)
    return function(...)
        local l = luamcallbacks.callbacklist[name]
        if l then
            for _, f in ipairs(l) do
                f.func(...)
            end
        end
    end
end
```

Finally we add some functions to the `callback` module, and we overwrite `callback.register` so that it outputs an error.

```
callback.add = luamcallbacks.add
callback.remove = luamcallbacks.remove
```

```
callback.reset = luamcallbacks.reset
callback.create = luamcallbacks.create
callback.call = luamcallbacks.call
callback.get_priority = luamcallbacks.get_priority

callback.register = function (...)
luamcallbacks.error("function callback.register has been deleted by luamcallbacks, please us
end

```

3 Test file

The test file is made to run in plainTeX, but is trivial to adapt for LaTeX. First we input the package, and we typeset a small sentence to get a non-empty document.

```
\input luatextra.sty

This is just a test file.
```

Then we declare three functions that we will use.

```
\luadirect{
local function one(head,...)
    texio.write_nl("I'm number 1")
    return head, true
end

local function two(head,...)
    texio.write_nl("I'm number 2")
    return head, true
end

local function three(head,...)
    texio.write_nl("I'm number 3")
    return head, true
end
```

Then we add the three functions to the hpack_filter callback

```
callback.add("hpack_filter",one,"my example function one",1)
callback.add("hpack_filter",two,"my example function two",2)
callback.add("hpack_filter",three,"my example function three",1)
```

We remove the function three from the callback.

```
callback.remove("hpack_filter","my example function three")
```

And we remove a non-declared function to the callback, which will generate an error.

```
}

\bye
```