The luamcallbacks package

Elie Roux
elie.roux@telecom-bretagne.eu

2009/03/19 v0.92

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       = 0.92,
7     date          = "2009/03/19",
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,
}

```

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.

```

function luamcallbacks.str_to_type(str)
    if str == 'list' then
        return list
    elseif str == 'data' then
```

```
107         return data
108     elseif str == 'first' then
109         return first
110     elseif str == 'simple' then
111         return simple
112     else
113         return nil
114     end
115 end
116
```

`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.

```
117
118 function luamcallbacks.create(name, ctype, default)
119     if not name then
120         luamcallbacks.error(format("unable to call callback, no proper name passed", name))
121         return nil
122     end
123     if not ctype or not default then
124         luamcallbacks.error(format("unable to create callback '%s', callbacktype or default
125         return nil
126     end
127     if callbacktypes[name] then
128         luamcallbacks.error(format("unable to create callback '%s', callback already exists'
129         return nil
130     end
131     local temp = luamcallbacks.str_to_type(ctype)
132     if not temp then
133         luamcallbacks.error(format("unable to create callback '%s', type '%s' undefined", na
134         return nil
135     end
136     ctype = temp
137     luamcallbacks.lua_callbacks_defaults[name] = default
138     callbacktypes[name] = ctype
139 end
140
```

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

```
141
142 function luamcallbacks.call(name, ...)
143     if not name then
```

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

`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.

```
171
172 function luamcallbacks.add (name,func,description,priority)
173     if type(func) ~= "function" then
174         luamcallbacks.error("unable to add function, no proper function passed")
175         return
176     end
```

```
177     if not name or name == "" then
178         luamcallbacks.error("unable to add function, no proper callback name passed")
179         return
180     elseif not callbacktypes[name] then
181         luamcallbacks.error(
182           format("unable to add function, '%s' is not a valid callback",
183           name))
184         return
185     end
186     if not description or description == "" then
187         luamcallbacks.error(
188           format("unable to add function to '%s', no proper description passed",
189           name))
190         return
191     end
192     if luamcallbacks.get_priority(name, description) ~= 0 then
193         luamcallbacks.warning(
194           format("function '%s' already registered in callback '%s'",
195           description, name))
196     end
197     local l = luamcallbacks.callbacklist[name]
198     if not l then
199         l = {}
200         luamcallbacks.callbacklist[name] = l
201         if not luamcallbacks.lua_callbacks_defaults[name] then
202             if callbacktypes[name] == list then
203                 luamcallbacks.internalregister(name, luamcallbacks.listhandler(name))
204             elseif callbacktypes[name] == data then
205                 luamcallbacks.internalregister(name, luamcallbacks.datahandler(name))
206             elseif callbacktypes[name] == simple then
207                 luamcallbacks.internalregister(name, luamcallbacks.simplehandler(name))
208             elseif callbacktypes[name] == first then
209                 luamcallbacks.internalregister(name, luamcallbacks.firsthandler(name))
210             else
211                 luamcallbacks.error("unknown callback type")
212             end
213         end
214     end
215     local f = {
216         func = func,
217         description = description,
218     }
219     priority = tonumber(priority)
220     if not priority or priority > #l then
221         priority = #l+1
222     elseif priority < 1 then
223         priority = 1
224     end
225     if callbacktypes[name] == first and (priority ~= 1 or #l ~= 0) then
226         luamcallbacks.warning(format("several callbacks registered in callback '%s', only th
```

```
    end
    table.insert(l,priority,f)
    luamcallbacks.log(
      format("inserting function '%s' at position %s in callback list for '%s'",
      description,priority,name))
end
```

`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.

```
function luamcallbacks.get_priority (name, description)
    if not name or name == "" or not callbacktypes[name] or not description then
        return 0
    end
    local l = luamcallbacks.callbacklist[name]
    if not l then return 0 end
    for p, f in pairs(l) do
        if f.description == description then
            return p
        end
    end
    return 0
end
```

`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`.

```
function luamcallbacks.remove (name, description)
    if not name or name == "" then
        luamcallbacks.error("unable to remove function, no proper callback name passed")
        return
    elseif not callbacktypes[name] then
        luamcallbacks.error(
          format("unable to remove function, '%s' is not a valid callback",
          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 l == {} and not luamcallbacks.lua_callbacks_defaults[name] then
                luamcallbacks.internalregister(name, nil)
            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))
        luamcallbacks.callbacklist[name] = nil
    end
end

```

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`

```

```

```
function luamcallbacks.listhandler (name)
    return function(head,...)
        local l = luamcallbacks.callbacklist[name]
        if l then
            local done = true
            for _, f in ipairs(l) do
                -- the returned value can be either true or a new head plus true
                rtv1, rtv2 = f.func(head,...)
                if type(rtv1) == 'boolean' then
                    done = rtv1
                elseif type (rtv1) == 'userdata' then
                    head = rtv1
                end
                if type(rtv2) == 'boolean'  then
                    done = rtv2
                elseif type(rtv2) == 'userdata' then
                    head = rtv2
                end
                if done == false then
                    luamcallbacks.error(format(
                      "function \"%s\" returned false in callback '%s'",
                      f.description, name))
                end
            end
            return head, done
        else
            return head, false
        end
    end
end

```

The handler for callbacks taking datas and returning modified ones.

`luamcallbacks.datahandler`

```

function luamcallbacks.datahandler (name)
    return function(data,...)
        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

```
352
353 function luamcallbacks.firsthandler (name)
354     return function(...)
355         local l = luamcallbacks.callbacklist[name]
356         if l then
357             local f = l[1].func
358             return f(...)
359         else
360             return nil, false
361         end
362     end
363 end
364
```

Handler for simple functions that don't return anything.

luamcallbacks.simplehandler

```
365
366 function luamcallbacks.simplehandler (name)
367     return function(...)
368         local l = luamcallbacks.callbacklist[name]
369         if l then
370             for _, f in ipairs(l) do
371                 f.func(...)
372             end
373         end
374     end
375 end
376
```

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

```
377
378 callback.add = luamcallbacks.add
379 callback.remove = luamcallbacks.remove
380 callback.reset = luamcallbacks.reset
381 callback.create = luamcallbacks.create
382 callback.call = luamcallbacks.call
383 callback.get_priority = luamcallbacks.get_priority
384
385 callback.register = function (...)
386 luamcallbacks.error("function callback.register has been deleted by luamcallbacks, please us
387 end
388
```

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
```