The luatexbase-mcb package

Manuel Pégourié-Gonnard
mpg@elzevir.fr

Élie Roux
elie.roux@telecom-bretagne.eu

2010/05/12 v0.2

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`.

Warning. Currently assumes that luatexbase-modutils has been previously loaded. (This is a temporary limitation.)

Contents

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.

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 Implementation

2.1 TEX package

1 ⟨*texpackage⟩

2.1.1 Preliminaries

Reload protection, especially for Plain TEX.

```
2                \csname lltxb@mcb@loaded\endcsname
3 \expandafter\let\csname lltxb@mcb@loaded\endcsname\endinput
```

Catcode defenses.

```
4 \begingroup
5   \catcode123 1 % {
6   \catcode125 2 % }
7   \catcode 35 6 % #
8   \toks0{}%
9   \def\x{}%
10  \def\y#1 #2 {%
11    \toks0\expandafter{\the\toks0 \catcode#1 \the\catcode#1}%
12    \edef\x{\x \catcode#1 #2}}%
13  \y 123 1  % {
14  \y 125 2  % }
15  \y  35 6  % #
16  \y  10 12 % ^^J
```

```
  \y  34 12 % "
  \y  36 3  % $ $
  \y  39 12 % ’
  \y  40 12 % (
  \y  41 12 % )
  \y  42 12 % *
  \y  43 12 % +
  \y  44 12 % ,
  \y  45 12 % -
  \y  46 12 % .
  \y  47 12 % /
  \y  60 12 % <
  \y  61 12 % =
  \y  64 11 % @ (letter)
  \y  62 12 % >
  \y  95 12 % _ (other)
  \y  96 12 % ‘
  \edef\y#1{\endgroup\edef#1{\the\toks0\relax}\x}%
\expandafter\y\csname lltxb@mcb@AtEnd\endcsname
```

Package declaration.

```
\begingroup
  \expandafter\ifx\csname ProvidesPackage\endcsname\relax
    \def\x#1[#2]{\immediate\write16{Package: #1 #2}}
  \else
    \let\x\ProvidesPackage
  \fi
\expandafter\endgroup
\x{luatexbase-mcb}[2010/05/12 v0.2 Callback management for LuaTeX]
```

Make sure LuaTEX is 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-mcb}{LuaTeX is required for this package. Aborting.}
  \lltxb@mcb@AtEnd
  \expandafter\endinput
\fi
```

2.1.2 Load supporting Lua module

First load luatexbase-loader (hence luatexbase-compat), then the supporting Lua module.

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname RequirePackage\endcsname\relax
  \input luatexbase-modutils.sty
\else
  \RequirePackage{luatexbase-modutils}
\fi
\luatexbase@directlua{require('luatexbase.mcb')}
```

That's all folks!

```
\lltxb@mcb@AtEnd
```
⟨/texpackage⟩

2.2 Lua module

⟨*lua⟩

2.2.1 Module identification

```
module('luatexbase', package.seeall)
luatexbase.provides_module({
    name          = "luamcallbacks",
    version       = 0.2,
    date          = "2010/05/12",
    description   = "register several functions in a callback",
    author        = "Hans Hagen, Elie Roux and Manuel PÃľgourie-Gonnard",
    copyright     = "Hans Hagen, Elie Roux and Manuel PÃľgourie-Gonnard",
    license       = "CC0",
})
```

Shortcuts for error functions.

```
local log = log or function(...)
  luatexbase.module_log('luamcallbacks', string.format(...))
end
local info = info or function(...)
  luatexbase.module_info('luamcallbacks', string.format(...))
end
local warning = warning or function(...)
  luatexbase.module_warning('luamcallbacks', string.format(...))
end
local err = err or function(...)
  luatexbase.module_error('luamcallbacks', string.format(...))
end
```

2.2.2 Initialisations

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

```
local callbacklist = callbacklist or { }
```

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

```
local lua_callbacks_defaults = { }
```

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`)

```
local list = 1
local data = 2
local first = 3
local simple = 4
```

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

```
local callbacktypes = callbacktypes or {
  buildpage_filter = simple,
  token_filter = first,
  pre_output_filter = list,
  hpack_filter = list,
  process_input_buffer = data,
  mlist_to_hlist = list,
  vpack_filter = list,
  define_font = first,
  open_read_file = first,
  linebreak_filter = list,
  post_linebreak_filter = list,
  pre_linebreak_filter = list,
  start_page_number = simple,
  stop_page_number = simple,
  start_run = simple,
  show_error_hook = simple,
  stop_run = simple,
  hyphenate = simple,
  ligaturing = simple,
  kerning = data,
  find_write_file = first,
  find_read_file = first,
  find_vf_file = data,
  find_map_file = data,
  find_format_file = data,
  find_opentype_file = data,
  find_output_file = data,
  find_truetype_file = data,
  find_type1_file = data,
  find_data_file = data,
  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. Test the version using the compat package for, well, compatibility.

```
if luatexbase.luatexversion > 42 then
    callbacktypes["process_output_buffer"] = data
end
```

As we overwrite `callback.register`, we save it as `internalregister`.

```
local internalregister = internalregister or callback.register
```

2.2.3 Unsorted stuff

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.

```
local function str_to_type(str)
    if str == 'list' then
        return list
    elseif str == 'data' then
        return data
    elseif str == 'first' then
        return first
    elseif str == 'simple' then
        return simple
    else
        return 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.

```
-- local
function listhandler (name)
    return function(head,...)
```

```
        local l = callbacklist[name]
        if l then
            local done = true
            for _, f in ipairs(l) do
                -- the returned value is 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
                    err("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.

```
local function datahandler (name)
    return function(data,...)
        local l = 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.

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

```
215             return nil, false
216         end
217     end
218 end
```

Handler for simple functions that don't return anything.

```
219 local function simplehandler (name)
220     return function(...)
221         local l = callbacklist[name]
222         if l then
223             for _, f in ipairs(l) do
224                 f.func(...)
225             end
226         end
227     end
228 end
```

2.2.4 Public functions

The main function. The signature is `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 `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 themselves. Most of the time, the priority is not needed.

```
229 function add_to_callback (name,func,description,priority)
230     if type(func) ~= "function" then
231         err("unable to add function, no proper function passed")
232         return
233     end
234     if not name or name == "" then
235         err("unable to add function, no proper callback name passed")
236         return
237     elseif not callbacktypes[name] then
238         err("unable to add function, '%s' is not a valid callback", name)
239         return
240     end
241     if not description or description == "" then
242         err("unable to add function to '%s', no proper description passed",
243           name)
244         return
245     end
246     if priority_in_callback(name, description) ~= 0 then
247         warning("function '%s' already registered in callback '%s'",
```

```
248           description, name)
249     end
250     local l = callbacklist[name]
251     if not l then
252         l = {}
253         callbacklist[name] = l
254         if not lua_callbacks_defaults[name] then
255             if callbacktypes[name] == list then
256                 internalregister(name, listhandler(name))
257             elseif callbacktypes[name] == data then
258                 internalregister(name, datahandler(name))
259             elseif callbacktypes[name] == simple then
260                 internalregister(name, simplehandler(name))
261             elseif callbacktypes[name] == first then
262                 internalregister(name, firsthandler(name))
263             else
264                 err("unknown callback type")
265             end
266         end
267     end
268     local f = {
269         func = func,
270         description = description,
271     }
272     priority = tonumber(priority)
273     if not priority or priority > #l then
274         priority = #l+1
275     elseif priority < 1 then
276         priority = 1
277     end
278     if callbacktypes[name] == first and (priority ~= 1 or #l ~= 0) then
279         warning("several callbacks registered in callback '%s', "
280         .."only the first function will be active.", name)
281     end
282     table.insert(l,priority,f)
283     log("inserting function '%s' at position %s in callback list for '%s'",
284       description, priority, name)
285 end
```

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

```
286 function remove_from_callback (name, description)
287     if not name or name == "" then
288         err("unable to remove function, no proper callback name passed")
289         return
290     elseif not callbacktypes[name] then
291         err("unable to remove function, '%s' is not a valid callback", name)
292         return
293     end
```

```
294     if not description or description == "" then
295         err(
296           "unable to remove function from '%s', no proper description passed",
297           name)
298         return
299     end
300     local l = callbacklist[name]
301     if not l then
302         err("no callback list for '%s'",name)
303         return
304     end
305     for k,v in ipairs(l) do
306         if v.description == description then
307             table.remove(l,k)
308             log("removing function '%s' from '%s'",description,name)
309             if not next(l) then
310               callbacklist[name] = nil
311               if not lua_callbacks_defaults[name] then
312                 internalregister(name, nil)
313               end
314             end
315             return
316         end
317     end
318     warning("unable to remove function '%s' from '%s'",description,name)
319 end
```

This function removes all the functions registered in a callback.

```
320 function reset_callback (name)
321     if not name or name == "" then
322         err("unable to reset, no proper callback name passed")
323         return
324     elseif not callbacktypes[name] then
325         err("reset error, '%s' is not a valid callback", name)
326         return
327     end
328     if not lua_callbacks_defaults[name] then
329         internalregister(name, nil)
330     end
331     local l = callbacklist[name]
332     if l then
333         log("resetting callback list '%s'",name)
334         callbacklist[name] = nil
335     end
336 end
```

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.

```
function create_callback(name, ctype, default)
    if not name then
        err("unable to call callback, no proper name passed", name)
        return nil
    end
    if not ctype or not default then
        err("unable to create callback '%s': "
        .."callbacktype or default function not specified", name)
        return nil
    end
    if callbacktypes[name] then
        err("unable to create callback '%s', callback already exists", name)
        return nil
    end
    local temp = str_to_type(ctype)
    if not temp then
        err("unable to create callback '%s', type '%s' undefined", name, ctype)
        return nil
    end
    ctype = temp
    lua_callbacks_defaults[name] = default
    callbacktypes[name] = ctype
end
```

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

```
function call_callback(name, ...)
    if not name then
        err("unable to call callback, no proper name passed", name)
        return nil
    end
    if not lua_callbacks_defaults[name] then
        err("unable to call lua callback '%s', unknown callback", name)
        return nil
    end
    local l = callbacklist[name]
    local f
    if not l then
        f = lua_callbacks_defaults[name]
    else
        if callbacktypes[name] == list then
            f = listhandler(name)
        elseif callbacktypes[name] == data then
            f = datahandler(name)
        elseif callbacktypes[name] == simple then
            f = simplehandler(name)
        elseif callbacktypes[name] == first then
            f = firsthandler(name)
        else
            err("unknown callback type")
```

```
        end
    end
    return f(...)
end
```

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 priority_in_callback (name, description)
    if not name or name == ""
            or not callbacktypes[name]
            or not description then
        return 0
    end
    local l = 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
```

Finally we overwrite `callback.register` so that it outputs an error.

```
callback.register = function ()
err("function callback.register has been deleted by luamcallbacks, "
.."please use callback.add instead.")
end
```

That's all folks!

⟨/lua⟩

3 Test files

A few basic tests for Plain and LaTeX.

```
⟨testplain⟩\input luatexbase-mcb.sty
⟨testlatex⟩\RequirePackage{luatexbase-mcb}
⟨*testplain, testlatex⟩
\catcode 64 11
\luatexbase@directlua{
  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

  luatexbase.add_to_callback("hpack_filter",one,"my sample function one",1)
  luatexbase.add_to_callback("hpack_filter",two,"my sample function two",2)
  luatexbase.add_to_callback("hpack_filter",three,"my sample function three",1)

  luatexbase.remove_from_callback("hpack_filter","my sample function three")
}
⟨/testplain, testlatex⟩
⟨testplain⟩\bye
⟨testlatex⟩\stop
```