The **luamcallbacks** package

Elie Roux
`elie.roux@telecom-bretagne.eu`

2007/12/12 v0.1

Abstract

This package manages the callback adding and removing, by adding `callback.add` and `callback.remove`, and overwriting `callback.register`.

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.

2 Package files

The package contains `luamcallbacks.lua` with the new functions, and two wrappers: `luamcallbacks.tex` for plainTEX, and `luamcallbacks.sty` for LATEX.

2.1 `luamcallbacks.lua`

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

```

if not modules then modules = { } end modules ['mcallbacks'] = {
    version   = 0.99,
    comment   = "simple module to have several functions in a callback",
    author    = "Hans Hagen & Elie Roux",
    copyright = "Hans Hagen & Elie Roux",
    license   = "public domain",
}

if mcallbacks and mcallbacks.version then
else

mcallbacks          = mcallbacks or { }
mcallbacks.version  = 0.99
```

If `mcallbacks.showlog` is set to `true`, each time a function is called it will log its actions.

```
mcallbacks.showlog  = mcallbacks.showlog or false

```

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

```

mcallbacks.callbacklist = mcallbacks.callbacklist or { }

local list = 1
local data = 2
local reader = 3
local simple = 4
local file = 5

local format = string.format

```

There are n types of callback:

- the ones for file reading (`reader`)
- the ones taking datas and returning the modified ones (`data`)
- the ones for reading and finding text files, like aux files (`file`)
- the ones for functions that don't return anything (`simple`)
- the ones taking a list of nodes and returning a boolean and a new head (`list`)

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

```

mcallbacks.callbacktypes = mcallbacks.callbacktypes or {
```

```
buildpage_filter = simple, -- ???
token_filter = data, -- ???
pre_output_filter = data, -- data but special2
hpack_filter = data, -- data but special2
process_input_buffer = data,
mlist_to_hlist = data, -- data but special
vpack_filter = data, -- data but special2
define_font = data,  -- alone
open_read_file = data, -- special
linebreak_filter = data, -- data but special
post_linebreak_filter = data, -- data or true
pre_linebreak_filter = data, -- data or true
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 = file,
find_read_file = file,
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 = reader,
read_map_file = reader,
read_pk_file = reader,
read_enc_file = reader,
read_vf_file = reader,
read_ocp_file = reader,
read_opentype_file = reader,
read_truetype_file = reader,
read_font_file = reader,
read_type1_file = reader,
read_data_file = reader,
}


```

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

```

mcallbacks.internalregister = mcallbacks.internalregister or callback.register

local callbacktypes = mcallbacks.callbacktypes

mcallbacks.report = mcallbacks.report or function(...)
  texio.write_nl(format("(mcallbacks: %s)",string.format(...)))
end

mcallbacks.log = mcallbacks.log or function(...)
  if mcallbacks.showlog then
    mcallbacks.report(...)
  end
end

mcallbacks.error = mcallbacks.error or function(...)
  mcallbacks.report(...)
end

```

`mcallbacks.add`

The main function. The signature is `mcallbacks.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 `mcallbacks.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.

```

mcallbacks.add = mcallbacks.add or function (name,func,description,priority)
    if type(func) ~= "function" then
        mcallbacks.error("unable to add function, no proper function passed")
        return
    end
    if not name or name == "" then
        mcallbacks.error("unable to add function, no proper callback name passed")
        return
    elseif not callbacktypes[name] then
        mcallbacks.error(
          format("unable to add function, ’%s’ is not a valid callback",
          name))
        return
    end

    if not description or description == "" then
        mcallbacks.error(
          format("unable to add function to '%s', no proper description passed",
          name))
        return
    end
    local l = mcallbacks.callbacklist[name]
    if not l then
        l = { }
        mcallbacks.callbacklist[name] = l
        if callbacktypes[name] == list then
            mcallbacks.internalregister(name, mcallbacks.listhandler(name))
        elseif callbacktypes[name] == data then
            mcallbacks.internalregister(name, mcallbacks.listhandler(name))
        else
            mcallbacks.error("unknown callback type")
        end
        mcallbacks.log(format("creating callback list for '%s'",name))
    end
    local f = {
        func = func,
        description = description,
    }
    priority = tonumber(priority)
    if not priority or priority > #l then
        priority = #l+1
    elseif priority < 1 then
        priority = 1
    end
    table.insert(l,priority,f)
    mcallbacks.log(
      format("inserting function '%s' at position %s in callback list for '%s'",
      description,priority,name))
end

```

`mcallbacks.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`.

```

mcallbacks.remove = mcallbacks.remove or function (name, description)
    if not name or name == "" then
        mcallbacks.error("unable to remove function, no proper callback name passed")
        return
    elseif not callbacktypes[name] then
```

```
        mcallbacks.error(
          format("unable to remove function, ’%s’ is not a valid callback",
          name))
        return
    end
    if not description or description == "" then
        mcallbacks.error(
          format("unable to remove function from ’%s’, no proper description passed",
          name))
        return
    end
    local l = mcallbacks.callbacklist[name]
    if not l then
        mcallbacks.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)
            mcallbacks.log(
              format("removing function ’%s’ from ’%s’",description,name))
            return
        end
    end
    if l == {} then
        mcallbacks.internalregister(name, nil)
    end
    mcallbacks.error(
      format("unable to remove function ’%s’ from ’%s’",description,name))
end

```

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

```

mcallbacks.reset = mcallbacks.reset or function (name)
    if not name or name == "" then
        mcallbacks.error("unable to reset, no proper callback name passed")
        return
    elseif not callbacktypes[name] then
        mcallbacks.error(
          format("reset error, ’%s’ is not a valid callback",
          name))
        return
    end
    mcallbacks.internalregister(name, nil)
    local l = mcallbacks.callbacklist[name]
    if not l then
        mcallbacks.error(format("no function registered for callback ’%s’",name))
    else
        mcallbacks.log(format("resetting callback list ’%s’",name))
        mcallbacks.callbacklist[name] = { }
    end
end

```

This function and the following ones are only internal. This one is the handler for

`mcallbacks.listhandler`

```

mcallbacks.listhandler = mcallbacks.listhandler or function (name)
    return function(head,...)
        local l = mcallbacks.callbacklist[name]
        if l then
            local done = false
            for _, f in ipairs(l) do
                head, ok = f.func(head,...)
                if ok then done = true end
            end
            return head, done
        else
            return head, false
        end
    end
end

```

The handler for callbacks modifying datas.

`mcallbacks.datahandler`

```

mcallbacks.datahandler = mcallbacks.datahandler or function (name)
    return function(data,...)
        local l = mcallbacks.callbacklist[name]
        if l then
            for _, f in ipairs(l) do
                data = f.func(data,...)
            end
        end
        return data
    end
end

```

This function is the handler for the reader_xxx callbacks.

`mcallbacks.readerhandler`

```

mcallbacks.readerhandler = mcallbacks.readerhandler or function (name)
    return function(filename)
```

```
        local l = mcallbacks.callbacklist[name]
        if l then
            printTable(l)
            f = l[1].func
            return f(filename)
        else
            return false, nil, 0
        end
    end
end

```

Handler for simple functions that don't return anything.

`mcallbacks.simplehandler`

```

mcallbacks.simplehandler = mcallbacks.simplehandler or function (name)
    return function(...)
        local l = mcallbacks.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 = mcallbacks.add
callback.remove = mcallbacks.remove
callback.reset = mcallbacks.reset

callback.register = function (...)
mcallbacks.error("function callback.register is considered too dangerous to use and has bee
end

end
```

2.2 `luamcallbacks.sty`

The LaTeXpackage is just a small wrapper for the lua file.

```
\directlua0{dofile(kpse.find_file("luamcallbacks.lua"))}
```

2.3 `luamcallbacks.tex`

The plainTeXpackage is almost the same.

```
\directlua0{dofile(kpse.find_file("luamcallbacks.lua"))}
```

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 luamcallbacks.tex

This is just a test file.
```

Then we declare three functions that we will use.

```
\directlua0{
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
```

Finally we try a few calls to the functions.First we try to add a callback to an invalid callback, it will generate an error:

```
callback.add("hpacsfsdffilter",one,"my example function one",1)
```

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 try to register a callback with the lua base call, it generates an error.

```
callback.register("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.

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

```
}

\bye
```