

The luatexbase-mcb package

Heiko Oberdiek (primary author of luatex)
Élie Roux, Manuel Pégourié-Gonnard, Philipp Gesang*

<https://github.com/lualatex/luatexbase>
luatex-dev@tug.org

2013/04/13 v0.5

Abstract

The primary feature of this package is to allow many functions to be registered in the same callback. Depending of the type of the callback, the functions will be combined in some way when the callback is called. Functions are provided for addition and removal of individual functions from a callback’s list, with a priority system.

Additionally, you can create new callbacks that will be handled the same way as predefined callbacks, except that they must be called explicitly.

Contents

1	Documentation	2
1.1	Managing functions in callbacks	2
1.2	Creating new callbacks	3
1.2.1	Limitations	3
1.3	Compatibility	4
2	Implementation	4
2.1	T _E X package	4
2.1.1	Preliminaries	4
2.1.2	Load supporting Lua module	5
2.2	Lua module	5
2.2.1	Module identification	5
2.2.2	Housekeeping	6
2.2.3	Handlers	8
2.2.4	Public functions for functions management	9
2.2.5	Public functions for user-defined callbacks	12
3	Test files	13

*See “History” in [luatexbase.pdf](#) for details.

1 Documentation

Before we start, let me mention that test files are provided (they should be in the same directory as this PDF file). You can have a look at them, compile them and have a look at the log, if you want examples of how this module works.

1.1 Managing functions in callbacks

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

The problem with `callback.register` is that it registers only one function in a callback. This package solves the problem by disabling `callback.register` and providing a new interface allowing many functions to be registered in a single callback.

The way the functions are combined together depends on the type of the callback. There are currently 4 types of callback, depending on the calling convention of the functions the callback can hold:

simple is for functions that don't return anything: they are called in order, all with the same argument;

data is for functions receiving a piece of data of any type except node list head (and possibly other arguments) and returning it (possibly modified): the functions are called in order, and each is passed the return value of the previous (and the other arguments untouched, if any). The return value is that of the last function;

list is a specialized variant of *data* for functions filtering node lists. Such functions may return either the head of a modified node list, or the boolean values `true` or `false`. The functions are chained the same way as for *data* except that for the following. If one function returns `false`, then `false` is immediately returned and the following functions are *not* called. If one function returns `true`, then the same head is passed to the next function. If all functions return `true`, then `true` is returned, otherwise the return value of the last function not returning `true` is used.

first is for functions with more complex signatures; functions in this type of callback are *not* combined: only the first one (according to priorities) is executed.

To add a function to a callback, use:

```
luatexbase.add_to_callback(name, func, description, priority)
```

The first argument is the name of the callback, the second is a function, the third one is a string used to identify the function later, and the optional priority is a positive integer, representing the rank of the function in the list of functions to be executing for this callback. So, 1 is the highest priority. If no priority is specified, the function is appended to the list, that is, its priority is the one of the last function plus one.

The priority system is intended to help resolving conflicts between packages competing on the same callback, but it cannot solve every possible issue. If two packages request priority 1 on the same callback, then the last one loaded will win.

To remove a function from a callback, use:

```
luatexbase.remove_from_callback(name, description)
```

The first argument must be the name of the callback, and the second one the description used when adding the function to this callback. You can also remove all functions from a callback at once using

```
luatexbase.reset_callback(name, make_false)
```

The `make_false` argument is optional. If it is `true` (repeat: `true`, not `false`) then the value `false` is registered in the callback, which has a special meaning for some callback.

Note that `reset_callback` is very invasive since it removes all functions possibly installed by other packages in this callback. So, use it with care if there is any chance that another package wants to share this callback with you.

When new functions are added at the beginning of the list, other functions are shifted down the list. To get the current rank of a function in a callback's list, use:

```
priority = luatexbase.priority_in_callback(name, description)
```

Again, the description is the string used when adding the function. If the function identified by this string is not in this callback's list, the priority returned is the boolean value `false`.

1.2 Creating new callbacks

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

```
luatexbase.create_callback(name, type, default)
```

The first argument is the callback's name, it must be unique. Then, the type goes as explained above, it is given as a string. Finally all user-defined callbacks have a default function which must¹ be provided as the third argument. It will be used when no other function is registered for this callback.

Functions are added to and removed from user-defined callbacks just the same way as predefined callback, so the previous section still applies. There is one difference, however: user-defined callbacks must be called explicitly at some point in your code, while predefined callbacks are called automatically by LuaTeX. To do so, use:

```
luatexbase.call_callback(name, arguments...)
```

The functions registered for this callback (or the default function) will be called with `arguments...` as arguments.

1.2.1 Limitations

For callbacks of type `first`, our new management system isn't actually better than good old `callback.register`. For some of them, it may be possible to split them into many callbacks, so that these callbacks can accept multiple functions. However, it seems risky and limited in use and is therefore not implemented.

At some point, `luatextra` used to split `open_read_file` that way, but support for this was removed. It may be added back (as well as support for other split callbacks) if it appears there is an actual need for it.

¹You can obviously provide a dummy function. If you're doing so often, please tell me, I may want to make this argument optional.

1.3 Compatibility

Some callbacks have a calling convention that varies depending on the version of LuaTeX used. This package *does not* try to track the type of the callbacks in every possible version of LuaTeX. The types are based on the last stable beta version (0.60.2 at the time this doc is written).

However, for callbacks that have the same calling convention for every version of LuaTeX, this package should work with the same range of LuaTeX version as other packages in the `luatexbase` bundle (currently, 0.25.4 to 0.60.2).

2 Implementation

2.1 T_EX package

```
1 (*texpackage)
```

2.1.1 Preliminaries

Catcode defenses and reload protection.

```
2 \begingroup\catcode61\catcode48\catcode32=10\relax% = and space
3 \catcode123 1 % {
4 \catcode125 2 % }
5 \catcode 35 6 % #
6 \toks0\expandafter{\expandafter\endlinechar\the\endlinechar}%
7 \edef\x{\endlinechar13}%
8 \def\y#1 #2 {%
9   \toks0\expandafter{\the\toks0 \catcode#1 \the\catcode#1}%
10  \edef\x{\x \catcode#1 #2}}%
11 \y 13 5 % carriage return
12 \y 61 12 % =
13 \y 32 10 % space
14 \y 123 1 % {
15 \y 125 2 % }
16 \y 35 6 % #
17 \y 64 11 % @ (letter)
18 \y 10 12 % new line ^^J
19 \y 39 12 % '
20 \y 40 12 % (
21 \y 41 12 % )
22 \y 45 12 % -
23 \y 46 12 % .
24 \y 47 12 % /
25 \y 58 12 % :
26 \y 91 12 % [
27 \y 93 12 % ]
28 \y 94 7 % ^
29 \y 96 12 % '
30 \toks0\expandafter{\the\toks0 \relax\noexpand\endinput}%
31 \edef\y#1{\noexpand\expandafter\endgroup%
32   \noexpand\ifx#1\relax \edef#1{\the\toks0}\x\relax%
33   \noexpand\else \noexpand\expandafter\noexpand\endinput%
34   \noexpand\fi}%
35 \expandafter\y\csname luatexbase@mcb@sty@endinput\endcsname%
```

Package declaration.

```
36 \begingroup
37 \expandafter\ifx\csname ProvidesPackage\endcsname\relax
38   \def\x#1[#2]{\immediate\write16{Package: #1 #2}}
39 \else
40   \let\x\ProvidesPackage
41 \fi
42 \expandafter\endgroup
43 \x{luatexbase-mcb}[2013/04/13 v0.5 Callback management for LuaTeX]
```

Make sure LuaTeX is used.

```
44 \begingroup\expandafter\expandafter\expandafter\endgroup
45 \expandafter\ifx\csname RequirePackage\endcsname\relax
46   \input ifluatex.sty
47 \else
48   \RequirePackage{ifluatex}
49 \fi
50 \ifluatex\else
51   \begingroup
52     \expandafter\ifx\csname PackageError\endcsname\relax
53       \def\x#1#2#3{\begingroup \newlinechar10
54         \errhelp{#3}\errmessage{Package #1 error: #2}\endgroup}
55     \else
56       \let\x\PackageError
57     \fi
58   \expandafter\endgroup
59   \x{luatexbase-mcb}{LuaTeX is required for this package. Aborting.}{%
60     This package can only be used with the LuaTeX engine^^J%
61     (command 'lualatex' or 'luatex').^^J%
62     Package loading has been stopped to prevent additional errors.}
63   \expandafter\luatexbase@mcb@sty@endinput%
64 \fi
```

2.1.2 Load supporting Lua module

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

```
65 \begingroup\expandafter\expandafter\expandafter\endgroup
66 \expandafter\ifx\csname RequirePackage\endcsname\relax
67   \input luatexbase-modutils.sty
68 \else
69   \RequirePackage{luatexbase-modutils}
70 \fi
71 \luatexbase@directlua{require('luatexbase.mcb')}
```

That's all folks!

```
72 \luatexbase@mcb@sty@endinput%
73 </texpackage>
```

2.2 Lua module

```
74 <lua>
```

2.2.1 Module identification

```

75 luatexbase      = luatexbase or { }
76 local luatexbase = luatexbase
77 local err, warning, info, log = luatexbase.provides_module({
78   name      = "luatexbase-mcb",
79   version   = 0.5,
80   date      = "2013/04/13",
81   description = "register several functions in a callback",
82   author     = "Hans Hagen, Elie Roux, Manuel Pegourie-Gonnard and Philipp Gesang",
83   copyright  = "Hans Hagen, Elie Roux, Manuel Pegourie-Gonnard and Philipp Gesang",
84   license    = "CC0",
85 })

```

First we declare the function references for the entire scope.

```

86 local add_to_callback
87 local call_callback
88 local create_callback
89 local priority_in_callback
90 local remove_from_callback
91 local reset_callback

```

2.2.2 Housekeeping

The main table: keys are callback names, and values are the associated lists of functions. More precisely, the entries in the list are tables holding the actual function as **func** and the identifying description as **description**. Only callbacks with a non-empty list of functions have an entry in this list.

```

92 local callbacklist = callbacklist or { }

```

Numerical codes for callback types, and name to value association (the table keys are strings, the values are numbers).

```

93 local list, data, first, simple = 1, 2, 3, 4
94 local types = {
95   list   = list,
96   data   = data,
97   first  = first,
98   simple = simple,
99 }

```

Now, list all predefined callbacks with their current type, based on the LuaTeX manual version 0.60.2.

```

100 local callbacktypes = callbacktypes or {

```

Section 4.1.1: file discovery callbacks.

```

101   find_read_file    = first,
102   find_write_file   = first,
103   find_font_file    = data,
104   find_output_file  = data,
105   find_format_file  = data,
106   find_vf_file      = data,
107   find_otp_file     = data,
108   find_map_file     = data,
109   find_enc_file     = data,
110   find_sfd_file     = data,
111   find_pk_file      = data,

```

```

112     find_data_file      = data,
113     find_opentype_file  = data,
114     find_truetype_file  = data,
115     find_type1_file     = data,
116     find_image_file     = data,

```

Section 4.1.2: file reading callbacks.

```

117     open_read_file      = first,
118     read_font_file      = first,
119     read_vf_file        = first,
120     read_otp_file       = first,
121     read_map_file       = first,
122     read_enc_file       = first,
123     read_sfd_file       = first,
124     read_pk_file        = first,
125     read_data_file      = first,
126     read_truetype_file  = first,
127     read_type1_file     = first,
128     read_opentype_file  = first,

```

Section 4.1.3: data processing callbacks.

```

129     process_input_buffer = data,
130     process_output_buffer = data,
131     token_filter         = first,

```

Section 4.1.4: node list processing callbacks.

```

132     buildpage_filter     = simple,
133     pre_linebreak_filter = list,
134     linebreak_filter     = list,
135     post_linebreak_filter = list,
136     hpack_filter         = list,
137     vpack_filter         = list,
138     pre_output_filter    = list,
139     hyphenate            = simple,
140     ligaturing           = simple,
141     kerning              = simple,
142     mlist_to_hlist       = list,

```

Section 4.1.5: information reporting callbacks.

```

143     start_run            = simple,
144     stop_run             = simple,
145     start_page_number    = simple,
146     stop_page_number     = simple,
147     show_error_hook      = simple,

```

Section 4.1.6: font-related callbacks.

```

148     define_font = first,
149 }

```

All user-defined callbacks have a default function. The following table's keys are the names of the user-defined callback, the associated value is the default function for this callback. This table is also used to identify the user-defined callbacks.

```

150 local lua_callbacks_defaults = { }

```

Overwrite `callback.register`, but save it first. Also define a wrapper that automatically raise an error when something goes wrong.

```

151 local original_register = original_register or callback.register
152 callback.register = function ()
153   err("function callback.register has been trapped,\n"
154   .."please use luatexbase.add_to_callback instead.")
155 end
156 local function register_callback(...)
157   return assert(original_register(...))
158 end

```

2.2.3 Handlers

Normal (as opposed to user-defined) callbacks have handlers depending on their type. The handler function is registered into the callback when the first function is added to this callback's list. Then, when the callback is called, then handler takes care of running all functions in the list. When the last function is removed from the callback's list, the handler is unregistered.

More precisely, the functions below are used to generate a specialized function (closure) for a given callback, which is the actual handler.

Handler for list callbacks.

```

159 local function listhandler (name)
160   return function(head,...)
161     local ret
162     local alltrue = true
163     for _, f in ipairs(callbacklist[name]) do
164       ret = f.func(head, ...)
165       if ret == false then
166         warning("function '%s' returned false\nin callback '%s'",
167           f.description, name)
168         break
169       end
170       if ret ~= true then
171         alltrue = false
172         head = ret
173       end
174     end
175     return alltrue and true or head
176   end
177 end

```

Handler for data callbacks.

```

178 local function datahandler (name)
179   return function(data, ...)
180     for _, f in ipairs(callbacklist[name]) do
181       data = f.func(data, ...)
182     end
183     return data
184   end
185 end

```

Handler for first callbacks. We can assume `callbacklist[name]` is not empty: otherwise, the function wouldn't be registered in the callback any more.


```

186 local function firsthandler (name)
187     return function(...)
188         return callbacklist[name][1].func(...)
189     end
190 end

```

Handler for simple callbacks.

```

191 local function simplehandler (name)
192     return function(...)
193         for _, f in ipairs(callbacklist[name]) do
194             f.func(...)
195         end
196     end
197 end

```

Finally, keep a handlers table for indexed access.

```

198 local handlers = {
199     [list]    = listhandler,
200     [data]    = datahandler,
201     [first]   = firsthandler,
202     [simple]   = simplehandler,
203 }

```

2.2.4 Public functions for functions management

Add a function to a callback. First check arguments.

```

204 function add_to_callback (name,func,description,priority)
205     if type(func) ~= "function" then
206         return err("unable to add function:\nno proper function passed")
207     end
208     if not name or name == "" then
209         err("unable to add function:\nno proper callback name passed")
210         return
211     elseif not callbacktypes[name] then
212         err("unable to add function:\n'%s' is not a valid callback", name)
213         return
214     end
215     if not description or description == "" then
216         err("unable to add function to '%s':\nno proper description passed",
217             name)
218         return
219     end
220     if priority_in_callback(name, description) then
221         err("function '%s' already registered\nin callback '%s'",
222             description, name)
223         return
224     end

```

Then test if this callback is already in use. If not, initialise its list and register the proper handler.

```

225     local l = callbacklist[name]
226     if not l then
227         l = {}

```

```

228     callbacklist[name] = 1
229     if not lua_callbacks_defaults[name] then
230         register_callback(name, handlers[callbacktypes[name]](name))
231     end
232 end

```

Actually register the function.

```

233     local f = {
234         func = func,
235         description = description,
236     }
237     priority = tonumber(priority)
238     if not priority or priority > #l then
239         priority = #l+1
240     elseif priority < 1 then
241         priority = 1
242     end
243     table.insert(l,priority,f)

```

Keep user informed.

```

244     if callbacktypes[name] == first and #l ~= 1 then
245         warning("several functions in '%s',\n"
246             .."only one will be active.", name)
247     end
248     log("inserting '%s'\nat position %s in '%s'",
249         description, priority, name)
250 end
251 luatexbase.add_to_callback = add_to_callback

```

Remove a function from a callback. First check arguments.

```

252 function remove_from_callback (name, description)
253     if not name or name == "" then
254         err("unable to remove function:\nno proper callback name passed")
255         return
256     elseif not callbacktypes[name] then
257         err("unable to remove function:\n'%s' is not a valid callback", name)
258         return
259     end
260     if not description or description == "" then
261         err(
262             "unable to remove function from '%s':\nno proper description passed",
263             name)
264         return
265     end
266     local l = callbacklist[name]
267     if not l then
268         err("no callback list for '%s'",name)
269         return
270     end

```

Then loop over the callback's function list until we find a matching entry. Remove it and check if the list gets empty: if so, unregister the callback handler unless the callback is user-defined.

```

271     local index = false
272     for k,v in ipairs(l) do

```

```

273         if v.description == description then
274             index = k
275             break
276         end
277     end
278     if not index then
279         err("unable to remove '%s'\nfrom '%s'", description, name)
280         return
281     end
282     table.remove(l, index)
283     log("removing '%s'\nfrom '%s'", description, name)
284     if #l == 0 then
285         callbacklist[name] = nil
286         if not lua_callbacks_defaults[name] then
287             register_callback(name, nil)
288         end
289     end
290     return
291 end
292 luatexbase.remove_from_callback = remove_from_callback

```

Remove all the functions registered in a callback. Unregisters the callback handler unless the callback is user-defined.

```

293 function reset_callback (name, make_false)
294     if not name or name == "" then
295         err("unable to reset:\nno proper callback name passed")
296         return
297     elseif not callbacktypes[name] then
298         err("unable to reset '%s':\nis not a valid callback", name)
299         return
300     end
301     log("resetting callback '%s'", name)
302     callbacklist[name] = nil
303     if not lua_callbacks_defaults[name] then
304         if make_false == true then
305             log("setting '%s' to false", name)
306             register_callback(name, false)
307         else
308             register_callback(name, nil)
309         end
310     end
311 end
312 luatexbase.reset_callback = reset_callback

```

Get a function's priority in a callback list, or false if the function is not in the list.

```

313 function priority_in_callback (name, description)
314     if not name or name == ""
315         or not callbacktypes[name]
316         or not description then
317         return false
318     end
319     local l = callbacklist[name]
320     if not l then return false end
321     for p, f in pairs(l) do

```

```

322         if f.description == description then
323             return p
324         end
325     end
326     return false
327 end
328 luatexbase.priority_in_callback = priority_in_callback

```

2.2.5 Public functions for user-defined callbacks

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.

```

329 function create_callback(name, ctype, default)
330     if not name then
331         err("unable to call callback:\nno proper name passed", name)
332         return nil
333     end
334     if not ctype or not default then
335         err("unable to create callback '%s':\n"
336             .."callbacktype or default function not specified", name)
337         return nil
338     end
339     if callbacktypes[name] then
340         err("unable to create callback '%s':\ncallback already exists", name)
341         return nil
342     end
343     ctype = types[ctype]
344     if not ctype then
345         err("unable to create callback '%s':\nctype '%s' undefined", name, ctype)
346         return nil
347     end
348     log("creating '%s' type %s", name, ctype)
349     lua_callbacks_defaults[name] = default
350     callbacktypes[name] = ctype
351 end
352 luatexbase.create_callback = create_callback

```

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

```

353 function call_callback(name, ...)
354     if not name then
355         err("unable to call callback:\nno proper name passed", name)
356         return nil
357     end
358     if not lua_callbacks_defaults[name] then
359         err("unable to call lua callback '%s':\nunknown callback", name)
360         return nil
361     end
362     local l = callbacklist[name]
363     local f

```

```

364     if not l then
365         f = lua_callbacks_defaults[name]
366     else
367         f = handlers[callbacktypes[name]](name)
368         if not f then
369             err("unknown callback type")
370             return
371         end
372     end
373     return f(...)
374 end
375 luatexbase.call_callback = call_callback

```

That's all folks!

```

376 </lua>

```

3 Test files

A few basic tests for Plain and LaTeX. Use a separate Lua file for convenience, since this package works on the Lua side of the force.

```

377 <*testlua>
378 local msg = texio.write_nl

    Test the management functions with a predefined callback.

379 local function sample(head,...)
380     return head, true
381 end
382 local prio = luatexbase.priority_in_callback
383 msg("\n*****\n* Testing management functions\n*****")
384 luatexbase.add_to_callback("hpack_filter", sample, "sample one", 1)
385 luatexbase.add_to_callback("hpack_filter", sample, "sample two", 2)
386 luatexbase.add_to_callback("hpack_filter", sample, "sample three", 1)
387 assert(prio("hpack_filter", "sample three"))
388 luatexbase.remove_from_callback("hpack_filter", "sample three")
389 assert(not prio("hpack_filter", "sample three"))
390 luatexbase.reset_callback("hpack_filter")
391 assert(not prio("hpack_filter", "sample one"))

```

Create a callback, and check that the managment functions work with this callback too.

```

392 local function data_one(s)
393     texio.write_nl("I'm data 1 whith argument: "..s)
394     return s
395 end
396 local function data_two(s)
397     texio.write_nl("I'm data 2 whith argument: "..s)
398     return s
399 end
400 local function data_three(s)
401     texio.write_nl("I'm data 3 whith argument: "..s)
402     return s
403 end
404 msg("\n*****\n* Testing user-defined callbacks\n*****")

```

```

405 msg("* create one")
406 luatexbase.create_callback("fooback", "data", data_one)
407 msg("* call it")
408 luatexbase.call_callback("fooback", "default")
409 msg("* add two functions")
410 luatexbase.add_to_callback("fooback", data_two, "function two", 2)
411 luatexbase.add_to_callback("fooback", data_three, "function three", 1)
412 msg("* call")
413 luatexbase.call_callback("fooback", "all")
414 msg("* rm one function")
415 luatexbase.remove_from_callback("fooback", "function three")
416 msg("* call")
417 luatexbase.call_callback("fooback", "all but three")
418 msg("* reset")
419 luatexbase.reset_callback("fooback")
420 msg("* call")
421 luatexbase.call_callback("fooback", "default")

```

Now, we want to make each handler run at least once. So, define dummy functions and register them in various callbacks. We will make sure the callbacks are executed on the \TeX end. Also, we want to check that everything works when we unload the functions either one by one, or using reset.

A list callback.

```

422 function add_hpack_filter()
423     luatexbase.add_to_callback('hpack_filter', function(head, ...)
424         texio.write_nl("I'm a dummy hpack_filter")
425         return head
426     end,
427     'dummy hpack filter')
428     luatexbase.add_to_callback('hpack_filter', function(head, ...)
429         texio.write_nl("I'm an optimized dummy hpack_filter")
430         return true
431     end,
432     'optimized dummy hpack filter')
433 end
434 function rm_one_hpack_filter()
435     luatexbase.remove_from_callback('hpack_filter', 'dummy hpack filter')
436 end
437 function rm_two_hpack_filter()
438     luatexbase.remove_from_callback('hpack_filter',
439         'optimized dummy hpack filter')
440 end

```

A simple callback.

```

441 function add_hyphenate()
442     luatexbase.add_to_callback('hyphenate', function(head, tail)
443         texio.write_nl("I'm a dummy hyphenate")
444     end,
445     'dummy hyphenate')
446     luatexbase.add_to_callback('hyphenate', function(head, tail)
447         texio.write_nl("I'm an other dummy hyphenate")
448     end,
449     'other dummy hyphenate')

```

```

450 end
451 function rm_one_hyphenate()
452     luatexbase.remove_from_callback('hyphenate', 'dummy hyphenate')
453 end
454 function rm_two_hyphenate()
455     luatexbase.remove_from_callback('hyphenate', 'other dummy hyphenate')
456 end

```

A first callback.

```

457 function add_find_write_file()
458     luatexbase.add_to_callback('find_write_file', function(id, name)
459         texio.write_nl("I'm a dummy find_write_file")
460         return "dummy-"..name
461     end,
462     'dummy find_write_file')
463     luatexbase.add_to_callback('find_write_file', function(id, name)
464         texio.write_nl("I'm an other dummy find_write_file")
465         return "dummy-other-"..name
466     end,
467     'other dummy find_write_file')
468 end
469 function rm_one_find_write_file()
470     luatexbase.remove_from_callback('find_write_file',
471     'dummy find_write_file')
472 end
473 function rm_two_find_write_file()
474     luatexbase.remove_from_callback('find_write_file',
475     'other dummy find_write_file')
476 end

```

A data callback.

```

477 function add_process_input_buffer()
478     luatexbase.add_to_callback('process_input_buffer', function(buffer)
479         return buffer.."\\msg{dummy}"
480     end,
481     'dummy process_input_buffer')
482     luatexbase.add_to_callback('process_input_buffer', function(buffer)
483         return buffer.."\\msg{otherdummy}"
484     end,
485     'other dummy process_input_buffer')
486 end
487 function rm_one_process_input_buffer()
488     luatexbase.remove_from_callback('process_input_buffer',
489     'dummy process_input_buffer')
490 end
491 function rm_two_process_input_buffer()
492     luatexbase.remove_from_callback('process_input_buffer',
493     'other dummy process_input_buffer')
494 end
495 </testlua>
496 <testplain>\input luatexbase-mcb.sty
497 <testlatex>\RequirePackage{luatexbase-mcb}
498 <*testplain,testlatex>

```

```

499 \catcode 64 11
500 \def\msg{\immediate\write16}
501 \msg{===== BEGIN =====}

```

Loading the lua files tests that the management functions can be called without raising errors.

```

502 \luatexbase@directlua{dofile('test-mcb.lua')}

```

We now want to load and unload stuff from the various callbacks have them called to test the handlers. Here is a helper macro for that.

```

503 \def\test#1#2{%
504   \msg{^^J*****^^J* Testing #1 (type #2)^^J*****}
505   \msg{* Add two functions}
506   \luatexbase@directlua{add_#1()}
507   \csname test_#1\endcsname
508   \msg{* Remove one}
509   \luatexbase@directlua{rm_one_#1()}
510   \csname test_#1\endcsname
511   \msg{* Remove the second}
512   \luatexbase@directlua{rm_two_#1()}
513   \csname test_#1\endcsname
514   \msg{* Add two functions again}
515   \luatexbase@directlua{add_#1()}
516   \csname test_#1\endcsname
517   \msg{* Remove all functions}
518   \luatexbase@directlua{luatexbase.reset_callback("#1")}
519   \csname test_#1\endcsname
520 }

```

For each callback, we need a specific macro that triggers it. For the hyphenate test, we need to untrap \everypar first, in the L^AT_EX case.

```

521 \catcode'\_ 11
522 \testlatex\everypar{}
523 \def\test_hpack_filter{\setbox0=\hbox{bla}}
524 \def\test_hyphenate{\showhyphens{hyphenation}}
525 \def\test_find_write_file{\immediate\openout15 test-mcb-out.log}
526 \def\test_process_input_buffer{\input test-mcb-aux.tex}

```

Now actually test them

```

527 \test{hpack_filter}{list}
528 \test{hyphenate}{simple}
529 \test{find_write_file}{first}
530 \test{process_input_buffer}{data}

```

Done.

```

531 \msg{===== END =====}
532 \testplain,testlatex
533 \testplain\bye
534 \testlatex\stop

```