

# The luatexbase-mcb package

Manuel Pégourié-Gonnard & Élie Roux  
Support: [lualatex-dev@tug.org](mailto:lualatex-dev@tug.org)

2010/10/10 v0.3

## 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

<b>1</b>	<b>Documentation</b>	<b>1</b>
1.1	Managing functions in callbacks . . . . .	2
1.2	Creating new callbacks . . . . .	3
1.2.1	Limitations . . . . .	3
1.3	Compatibility . . . . .	4
<b>2</b>	<b>Implementation</b>	<b>4</b>
2.1	TeX package . . . . .	4
2.1.1	Preliminaries . . . . .	4
2.1.2	Load supporting Lua module . . . . .	5
2.2	Lua module . . . . .	6
2.2.1	Module identification . . . . .	6
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
<b>3</b>	<b>Test files</b>	<b>13</b>

## 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 $\TeX$  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 lines, puts vertical spaces, etc. The Lua $\TeX$  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 must<sup>1</sup> 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.

---

<sup>1</sup>You 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 TeX package

```
1 (*texpackage)
```

#### 2.1.1 Preliminaries

Reload protection, especially for Plain TeX.

```
2             \csname lltxb@mcbl@loaded\endcsname
3 \expandafter\let\csname lltxb@mcbl@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
17  \y 34 12 % "
18  \y 36 3 % $ $
19  \y 39 12 % '
20  \y 40 12 % (
21  \y 41 12 % )
22  \y 42 12 % *
23  \y 43 12 % +
24  \y 44 12 % ,
25  \y 45 12 % -
26  \y 46 12 % .
27  \y 47 12 % /
28  \y 60 12 % <
29  \y 61 12 % =
30  \y 64 11 % @ (letter)
31  \y 62 12 % >
32  \y 95 12 % _ (other)
33  \y 96 12 % `
34  \edef\y#1{\endgroup\edef#1{\the\toks0\relax}\x}%
```

```

35 \expandafter\y\csname lltxb@mcb@AtEnd\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}[2010/10/10 v0.3 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-attr}{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   \lltxb@mcb@AtEnd
64 \expandafter\endinput
65 \fi

```

### 2.1.2 Load supporting Lua module

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

```

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

    That's all folks!
73 \lltxb@mcb@AtEnd
74 </texpackage>

```

## 2.2 Lua module

75 `(*lua)`

### 2.2.1 Module identification

```
76 module('luatexbase', package.seeall)
77 local err, warning, info = luatexbase.provides_module({
78   name      = "luatexbase-mcb",
79   version   = 0.2,
80   date      = "2010/05/12",
81   description = "register several functions in a callback",
82   author    = "Hans Hagen, Elie Roux and Manuel Pegourie-Gonnard",
83   copyright  = "Hans Hagen, Elie Roux and Manuel Pegourie-Gonnard",
84   license   = "CC0",
85 })
```

### 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.

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

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

```
87 local list, data, first, simple = 1, 2, 3, 4
88 local types = {
89   list   = list,
90   data   = data,
91   first  = first,
92   simple = simple,
93 }
```

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

```
94 local callbacktypes = callbacktypes or {
```

Section 4.1.1: file discovery callbacks.

```
95   find_read_file   = first,
96   find_write_file  = first,
97   find_font_file   = data,
98   find_output_file = data,
99   find_format_file = data,
100  find_vf_file      = data,
101  find_ocr_file     = data,
102  find_map_file     = data,
103  find_enc_file     = data,
104  find_sfd_file     = data,
105  find_pk_file      = data,
106  find_data_file    = data,
107  find_opentype_file = data,
108  find_truetype_file = data,
109  find_type1_file   = data,
110  find_image_file   = data,
```

Section 4.1.2: file reading callbacks.

```
111   open_read_file      = first,
112   read_font_file     = first,
113   read_vf_file       = first,
114   read_ocr_file      = first,
115   read_map_file      = first,
116   read_enc_file      = first,
117   read_sfd_file      = first,
118   read_pk_file       = first,
119   read_data_file     = first,
120   read_truetype_file = first,
121   read_type1_file    = first,
122   read_opentype_file = first,
```

Section 4.1.3: data processing callbacks.

```
123   process_input_buffer = data,
124   process_output_buffer = data,
125   token_filter         = first,
```

Section 4.1.4: node list processing callbacks.

```
126   buildpage_filter    = simple,
127   pre_linebreak_filter = list,
128   linebreak_filter    = list,
129   post_linebreak_filter = list,
130   hpack_filter        = list,
131   vpack_filter        = list,
132   pre_output_filter   = list,
133   hyphenate           = simple,
134   ligaturing          = simple,
135   kerning             = simple,
136   mlist_to_hlist     = list,
```

Section 4.1.5: information reporting callbacks.

```
137   start_run          = simple,
138   stop_run           = simple,
139   start_page_number = simple,
140   stop_page_number  = simple,
141   show_error_hook   = simple,
```

Section 4.1.6: font-related callbacks.

```
142   define_font = first,
143 }
```

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.

```
144 local lua_callbacks_defaults = { }
```

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

```
145 local original_register = original_register or callback.register
146 callback.register = function ()
147   err("function callback.register has been trapped,\n"
```

```

148 .."please use luatexbase.add_to_callback instead.")
149 end
150 local function register_callback(...)
151     return assert(original_register(...))
152 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.

```

153 local function listhandler (name)
154     return function(head,...)
155         local ret
156         local alltrue = true
157         for _, f in ipairs(callbacklist[name]) do
158             ret = f.func(head, ...)
159             if ret == false then
160                 warn("function '%s' returned false\nin callback '%s'",
161                     f.description, name)
162                 break
163             end
164             if ret ~= true then
165                 alltrue = false
166                 head = ret
167             end
168         end
169         return alltrue and true or head
170     end
171 end

```

Handler for data callbacks.

```

172 local function datahandler (name)
173     return function(data, ...)
174         for _, f in ipairs(callbacklist[name]) do
175             data = f.func(data, ...)
176         end
177         return data
178     end
179 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.

```

180 local function firsthandler (name)
181     return function(...)
182         return callbacklist[name][1].func(...)
183     end
184 end

```



Handler for simple callbacks.

```
185 local function simplehandler (name)
186     return function(...)
187         for _, f in ipairs(callbacklist[name]) do
188             f.func(...)
189         end
190     end
191 end
```

Finally, keep a handlers table for indexed access.

```
192 local handlers = {
193     [list] = listhandler,
194     [data] = datahandler,
195     [first] = firsthandler,
196     [simple] = simplehandler,
197 }
```

## 2.2.4 Public functions for functions management

Add a function to a callback. First check arguments.

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

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

```
219     local l = callbacklist[name]
220     if not l then
221         l = {}
222         callbacklist[name] = l
223         if not lua_callbacks_defaults[name] then
224             register_callback(name, handlers[callbacktypes[name]](name))
225         end
226     end
```

Actually register the function.

```
227     local f = {
228         func = func,
229         description = description,
230     }
231     priority = tonumber(priority)
232     if not priority or priority > #l then
233         priority = #l+1
234     elseif priority < 1 then
235         priority = 1
236     end
237     table.insert(l,priority,f)
```

Keep user informed.

```
238     if callbacktypes[name] == first and #l ~= 1 then
239         warning("several functions in '%s',\n"
240             .."only one will be active.", name)
241     end
242     info("inserting '%s'\nat position %s in '%s'",
243         description, priority, name)
244 end
```

Remove a function from a callback. First check arguments.

```
245 function remove_from_callback (name, description)
246     if not name or name == "" then
247         err("unable to remove function:\nno proper callback name passed")
248         return
249     elseif not callbacktypes[name] then
250         err("unable to remove function:\n'%s' is not a valid callback", name)
251         return
252     end
253     if not description or description == "" then
254         err(
255             "unable to remove function from '%s':\nno proper description passed",
256             name)
257         return
258     end
259     local l = callbacklist[name]
260     if not l then
261         err("no callback list for '%s'",name)
262         return
263     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.

```
264     local index = false
265     for k,v in ipairs(l) do
266         if v.description == description then
267             index = k
268             break
269         end
270     end
271     if not index then
```

```

272     err("unable to remove '%s'\nfrom '%s'", description, name)
273     return
274 end
275 table.remove(l, index)
276 info("removing '%s'\nfrom '%s'", description, name)
277 if table.maxn(l) == 0 then
278     callbacklist[name] = nil
279     if not lua_callbacks_defaults[name] then
280         register_callback(name, nil)
281     end
282 end
283 return
284 end

```

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

```

285 function reset_callback (name, make_false)
286     if not name or name == "" then
287         err("unable to reset:\nno proper callback name passed")
288         return
289     elseif not callbacktypes[name] then
290         err("unable to reset '%s':\nis not a valid callback", name)
291         return
292     end
293     info("resetting callback '%s'", name)
294     callbacklist[name] = nil
295     if not lua_callbacks_defaults[name] then
296         if make_false == true then
297             info("setting '%s' to false", name)
298             register_callback(name, false)
299         else
300             register_callback(name, nil)
301         end
302     end
303 end

```

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

```

304 function priority_in_callback (name, description)
305     if not name or name == ""
306         or not callbacktypes[name]
307         or not description then
308         return false
309     end
310     local l = callbacklist[name]
311     if not l then return false end
312     for p, f in pairs(l) do
313         if f.description == description then
314             return p
315         end
316     end
317     return false
318 end

```

### 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 Lua<sub>TEX</sub> 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.

```
319 function create_callback(name, ctype, default)
320   if not name then
321     err("unable to call callback:\nno proper name passed", name)
322     return nil
323   end
324   if not ctype or not default then
325     err("unable to create callback '%s':\n"
326       .."callbacktype or default function not specified", name)
327     return nil
328   end
329   if callbacktypes[name] then
330     err("unable to create callback '%s':\ncallback already exists", name)
331     return nil
332   end
333   ctype = types[ctype]
334   if not ctype then
335     err("unable to create callback '%s':\ntype '%s' undefined", name, ctype)
336     return nil
337   end
338   info("creating '%s' type %s", name, ctype)
339   lua_callbacks_defaults[name] = default
340   callbacktypes[name] = ctype
341 end
```

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

```
342 function call_callback(name, ...)
343   if not name then
344     err("unable to call callback:\nno proper name passed", name)
345     return nil
346   end
347   if not lua_callbacks_defaults[name] then
348     err("unable to call lua callback '%s':\nunknown callback", name)
349     return nil
350   end
351   local l = callbacklist[name]
352   local f
353   if not l then
354     f = lua_callbacks_defaults[name]
355   else
356     f = handlers[callbacktypes[name]](name)
357     if not f then
358       err("unknown callback type")
359       return
360     end
361   end
```

```

362     return f(...)
363 end

```

That's all folks!

```

364 </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.

```

365 (*testlua)
366 local msg = texio.write_nl

    Test the management functions with a predefined callback.

367 local function sample(head,...)
368     return head, true
369 end
370 local prio = luatexbase.priority_in_callback
371 msg("\n*****\n* Testing management functions\n*****")
372 luatexbase.add_to_callback("hpack_filter", sample, "sample one", 1)
373 luatexbase.add_to_callback("hpack_filter", sample, "sample two", 2)
374 luatexbase.add_to_callback("hpack_filter", sample, "sample three", 1)
375 assert(prio("hpack_filter", "sample three"))
376 luatexbase.remove_from_callback("hpack_filter", "sample three")
377 assert(not prio("hpack_filter", "sample three"))
378 luatexbase.reset_callback("hpack_filter")
379 assert(not prio("hpack_filter", "sample one"))

```

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

```

380 local function data_one(s)
381     texio.write_nl("I'm data 1 whith argument: "..s)
382     return s
383 end
384 local function data_two(s)
385     texio.write_nl("I'm data 2 whith argument: "..s)
386     return s
387 end
388 local function data_three(s)
389     texio.write_nl("I'm data 3 whith argument: "..s)
390     return s
391 end
392 msg("\n*****\n* Testing user-defined callbacks\n*****")
393 msg("* create one")
394 luatexbase.create_callback("fooback", "data", data_one)
395 msg("* call it")
396 luatexbase.call_callback("fooback", "default")
397 msg("* add two functions")
398 luatexbase.add_to_callback("fooback", data_two, "function two", 2)
399 luatexbase.add_to_callback("fooback", data_three, "function three", 1)
400 msg("* call")
401 luatexbase.call_callback("fooback", "all")
402 msg("* rm one function")

```

```

403 luatexbase.remove_from_callback("fooback", "function three")
404 msg("* call")
405 luatexbase.call_callback("fooback", "all but three")
406 msg("* reset")
407 luatexbase.reset_callback("fooback")
408 msg("* call")
409 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  $\text{\TeX}$  end. Also, we want to check that everything works when we unload the functions either one by one, or using reset.

A list callback.

```

410 function add_hpack_filter()
411     luatexbase.add_to_callback('hpack_filter', function(head, ...)
412         texio.write_nl("I'm a dummy hpack_filter")
413         return head
414     end,
415     'dummy hpack filter')
416 luatexbase.add_to_callback('hpack_filter', function(head, ...)
417     texio.write_nl("I'm an optimized dummy hpack_filter")
418     return true
419 end,
420 'optimized dummy hpack filter')
421 end
422 function rm_one_hpack_filter()
423     luatexbase.remove_from_callback('hpack_filter', 'dummy hpack filter')
424 end
425 function rm_two_hpack_filter()
426     luatexbase.remove_from_callback('hpack_filter',
427     'optimized dummy hpack filter')
428 end

```

A simple callback.

```

429 function add_hyphenate()
430     luatexbase.add_to_callback('hyphenate', function(head, tail)
431         texio.write_nl("I'm a dummy hyphenate")
432     end,
433     'dummy hyphenate')
434 luatexbase.add_to_callback('hyphenate', function(head, tail)
435     texio.write_nl("I'm an other dummy hyphenate")
436     end,
437     'other dummy hyphenate')
438 end
439 function rm_one_hyphenate()
440     luatexbase.remove_from_callback('hyphenate', 'dummy hyphenate')
441 end
442 function rm_two_hyphenate()
443     luatexbase.remove_from_callback('hyphenate', 'other dummy hyphenate')
444 end

```

A first callback.

```

445 function add_find_write_file()

```

```

446   luatexbase.add_to_callback('find_write_file', function(id, name)
447       texio.write_nl("I'm a dummy find_write_file")
448       return "dummy-"..name
449   end,
450   'dummy find_write_file')
451   luatexbase.add_to_callback('find_write_file', function(id, name)
452       texio.write_nl("I'm an other dummy find_write_file")
453       return "dummy-other-"..name
454   end,
455   'other dummy find_write_file')
456 end
457 function rm_one_find_write_file()
458   luatexbase.remove_from_callback('find_write_file',
459   'dummy find_write_file')
460 end
461 function rm_two_find_write_file()
462   luatexbase.remove_from_callback('find_write_file',
463   'other dummy find_write_file')
464 end

```

A data callback.

```

465 function add_process_input_buffer()
466   luatexbase.add_to_callback('process_input_buffer', function(buffer)
467       return buffer.."\\msg{dummy}"
468   end,
469   'dummy process_input_buffer')
470   luatexbase.add_to_callback('process_input_buffer', function(buffer)
471       return buffer.."\\msg{otherdummy}"
472   end,
473   'other dummy process_input_buffer')
474 end
475 function rm_one_process_input_buffer()
476   luatexbase.remove_from_callback('process_input_buffer',
477   'dummy process_input_buffer')
478 end
479 function rm_two_process_input_buffer()
480   luatexbase.remove_from_callback('process_input_buffer',
481   'other dummy process_input_buffer')
482 end
483 </testlua>
484 <testplain>\input luatexbase-mcb.sty
485 <testlatex>\RequirePackage{luatexbase-mcb}
486 <*testplain,testlatex>
487 \catcode 64 11
488 \def\msg{\immediate\write16}
489 \msg{===== BEGIN =====}

```

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

```

490 \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.

```

491 \def\test#1#2{%

```

```

492 \msg{^^J*****^^J* Testing #1 (type #2)^^J*****}
493 \msg{* Add two functions}
494 \luatexbase@directlua{add_#1()}
495 \csname test_#1\endcsname
496 \msg{* Remove one}
497 \luatexbase@directlua{rm_one_#1()}
498 \csname test_#1\endcsname
499 \msg{* Remove the second}
500 \luatexbase@directlua{rm_two_#1()}
501 \csname test_#1\endcsname
502 \msg{* Add two functions again}
503 \luatexbase@directlua{add_#1()}
504 \csname test_#1\endcsname
505 \msg{* Remove all functions}
506 \luatexbase@directlua{luatexbase.reset_callback("#1")}
507 \csname test_#1\endcsname
508 }

```

For each callback, we need a specific macro that triggers it. For the hyphenate test, we need to untrap `\everypar` first, in the  $\text{\LaTeX}$  case.

```

509 \catcode'\_ 11
510 \testlatex\everypar{}
511 \def\test_hpack_filter{\setbox0=\hbox{bla}}
512 \def\test_hyphenate{\showhyphens{hyphenation}}
513 \def\test_find_write_file{\immediate\openout15 test-mcb-out.log}
514 \def\test_process_input_buffer{\input test-mcb-aux.tex}

```

Now actually test them

```

515 \test{hpack_filter}{list}
516 \test{hyphenate}{simple}
517 \test{find_write_file}{first}
518 \test{process_input_buffer}{data}

```

Done.

```

519 \msg{===== END =====}
520 \testplain,testlatex)
521 \testplain)\bye
522 \testlatex)\stop

```