The luatexbase-mcb package

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

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

2013/05/11 v0.6

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 TEX 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 . . . 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

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

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 lines, 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. 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[1] 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, is may be possible to split them into many callbacks, so that these callbacks can accept multiple functions. However, its seems risky and limited in use and is therefore nor 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.

[1] 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

Catcode defenses and reload protection.

```
\begingroup\catcode61\catcode48\catcode32=10\relax% = and space
  \catcode123 1 % {
  \catcode125 2 % }
  \catcode 35 6 % #
  \toks0\expandafter{\expandafter\endlinechar\the\endlinechar}%
  \edef\x{\endlinechar13}%
  \def\y#1 #2 {%
    \toks0\expandafter{\the\toks0 \catcode#1 \the\catcode#1}%
    \edef\x{\x \catcode#1 #2}}%
  \y  13  5 % carriage return
  \y  61 12 % =
  \y  32 10 % space
  \y 123  1 % {
  \y 125  2 % }
  \y  35  6 % #
  \y  64 11 % @ (letter)
  \y  10 12 % new line ^^J
  \y  39 12 % '
  \y  40 12 % (
  \y  41 12 % )
  \y  45 12 % -
  \y  46 12 % .
  \y  47 12 % /
  \y  58 12 % :
  \y  91 12 % [
  \y  93 12 % ]
  \y  94  7 % ^
  \y  96 12 % `
  \toks0\expandafter{\the\toks0 \relax\noexpand\endinput}%
  \edef\y#1{\noexpand\expandafter\endgroup%
    \noexpand\ifx#1\relax \edef#1{\the\toks0}\x\relax%
    \noexpand\else \noexpand\expandafter\noexpand\endinput%
    \noexpand\fi}%
\expandafter\y\csname luatexbase@mcb@sty@endinput\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}[2013/05/11 v0.6 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 PackageError\endcsname\relax
      \def\x#1#2#3{\begingroup \newlinechar10
        \errhelp{#3}\errmessage{Package #1 error: #2}\endgroup}
    \else
      \let\x\PackageError
    \fi
  \expandafter\endgroup
  \x{luatexbase-mcb}{LuaTeX is required for this package. Aborting.}{%
    This package can only be used with the LuaTeX engine^^J%
    (command 'lualatex' or 'luatex').^^J%
    Package loading has been stopped to prevent additional errors.}
  \expandafter\luatexbase@mcb@sty@endinput%
\fi
```

2.1.2 Load supporting Lua module

First load luatexbase-modutils (hence luatexbase-loader and 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!

```
\luatexbase@mcb@sty@endinput%
```

⟨/texpackage⟩

2.2 Lua module

⟨*lua⟩

2.2.1 Module identification

```
luatexbase          = luatexbase or { }
local luatexbase    = luatexbase
local err, warning, info, log = luatexbase.provides_module({
    name          = "luatexbase-mcb",
    version       = 0.6,
    date          = "2013/05/11",
    description   = "register several functions in a callback",
    author        = "Hans Hagen, Elie Roux, Manuel Pegourie-Gonnard and Philipp Gesang",
    copyright     = "Hans Hagen, Elie Roux, Manuel Pegourie-Gonnard and Philipp Gesang",
    license       = "CC0",
})
```

First we declare the function references for the entire scope.

```
local add_to_callback
local call_callback
local create_callback
local priority_in_callback
local remove_from_callback
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.

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

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

```
local list, data, first, simple = 1, 2, 3, 4
local types = {
    list   = list,
    data   = data,
    first  = first,
    simple = simple,
}
```

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

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

Section 4.1.1: file discovery callbacks.

```
    find_read_file     = first,
    find_write_file    = first,
    find_font_file     = data,
    find_output_file   = data,
    find_format_file   = data,
    find_vf_file       = data,
    find_ocp_file      = data,
    find_map_file      = data,
    find_enc_file      = data,
    find_sfd_file      = data,
```

```
    find_pk_file       = data,
    find_data_file     = data,
    find_opentype_file = data,
    find_truetype_file = data,
    find_type1_file    = data,
    find_image_file    = data,
```

Section 4.1.2: file reading callbacks.

```
    open_read_file     = first,
    read_font_file     = first,
    read_vf_file       = first,
    read_ocp_file      = first,
    read_map_file      = first,
    read_enc_file      = first,
    read_sfd_file      = first,
    read_pk_file       = first,
    read_data_file     = first,
    read_truetype_file = first,
    read_type1_file    = first,
    read_opentype_file = first,
```

Section 4.1.3: data processing callbacks.

```
    process_input_buffer  = data,
    process_output_buffer = data,
    token_filter          = first,
```

Section 4.1.4: node list processiong callbacks.

```
    buildpage_filter      = simple,
    pre_linebreak_filter  = list,
    linebreak_filter      = list,
    post_linebreak_filter = list,
    hpack_filter          = list,
    vpack_filter          = list,
    pre_output_filter     = list,
    hyphenate             = simple,
    ligaturing            = simple,
    kerning               = simple,
    mlist_to_hlist        = list,
```

Section 4.1.5: information reporting callbacks.

```
    start_run         = simple,
    stop_run          = simple,
    start_page_number = simple,
    stop_page_number  = simple,
    show_error_hook   = simple,
```

Section 4.1.6: font-related callbacks.

```
    define_font = first,
}
```

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 functon for this callback. This table is also used to identify the user-defined callbacks.

```
local lua_callbacks_defaults = { }
```

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

```
local original_register = original_register or callback.register
callback.register = function ()
  err("function callback.register has been trapped,\n"
  .."please use luatexbase.add_to_callback instead.")
end
local function register_callback(...)
    return assert(original_register(...))
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.

```
local function listhandler (name)
    return function(head,...)
        local ret
        local alltrue = true
        for _, f in ipairs(callbacklist[name]) do
            ret = f.func(head, ...)
            if ret == false then
                warning("function '%s' returned false\nin callback '%s'",
                    f.description, name)
                break
            end
            if ret ~= true then
                alltrue = false
                head = ret
            end
        end
        return alltrue and true or head
    end
end
```

Handler for `data` callbacks.

```
local function datahandler (name)
    return function(data, ...)
        for _, f in ipairs(callbacklist[name]) do
            data = f.func(data, ...)
        end
        return data
    end
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.

```
local function firsthandler (name)
    return function(...)
        return callbacklist[name][1].func(...)
    end
end
```

Handler for `simple` callbacks.

```
local function simplehandler (name)
    return function(...)
        for _, f in ipairs(callbacklist[name]) do
            f.func(...)
        end
    end
end
```

Finally, keep a handlers table for indexed access.

```
local handlers = {
  [list]   = listhandler,
  [data]   = datahandler,
  [first]  = firsthandler,
  [simple] = simplehandler,
}
```

2.2.4 Public functions for functions management

Add a function to a callback. First check arguments.

```
function add_to_callback (name,func,description,priority)
    if type(func) ~= "function" then
        return err("unable to add function:\nno proper function passed")
    end
    if not name or name == "" then
        err("unable to add function:\nno proper callback name passed")
        return
    elseif not callbacktypes[name] then
        err("unable to add function:\n'%s' is not a valid callback", name)
        return
    end
    if not description or description == "" then
        err("unable to add function to '%s':\nno proper description passed",
          name)
        return
    end
    if priority_in_callback(name, description) then
        err("function '%s' already registered\nin callback '%s'",
          description, name)
        return
    end
```

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

```
    local l = callbacklist[name]
    if not l then
        l = {}
```

```
        callbacklist[name] = l
        if not lua_callbacks_defaults[name] then
            register_callback(name, handlers[callbacktypes[name]](name))
        end
    end
```

Actually register the function.

```
    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)
```

Keep user informed.

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

Remove a function from a callback. First check arguments.

```
function remove_from_callback (name, description)
    if not name or name == "" then
        err("unable to remove function:\nno proper callback name passed")
        return
    elseif not callbacktypes[name] then
        err("unable to remove function:\n'%s' is not a valid callback", name)
        return
    end
    if not description or description == "" then
        err(
          "unable to remove function from '%s':\nno proper description passed",
          name)
        return
    end
    local l = callbacklist[name]
    if not l then
        err("no callback list for '%s'",name)
        return
    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.

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

```
        if v.description == description then
            index = k
            break
        end
    end
    if not index then
        err("unable to remove '%s'\nfrom '%s'", description, name)
        return
    end
    table.remove(l, index)
    log("removing '%s'\nfrom '%s'", description, name)
    if #l == 0 then
        callbacklist[name] = nil
        if not lua_callbacks_defaults[name] then
            register_callback(name, nil)
        end
    end
    return
end
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.

```
function reset_callback (name, make_false)
    if not name or name == "" then
        err("unable to reset:\nno proper callback name passed")
        return
    elseif not callbacktypes[name] then
        err("unable to reset '%s':\nis not a valid callback", name)
        return
    end
    log("resetting callback '%s'", name)
    callbacklist[name] = nil
    if not lua_callbacks_defaults[name] then
        if make_false == true then
            log("setting '%s' to false", name)
            register_callback(name, false)
        else
            register_callback(name, nil)
        end
    end
end
luatexbase.reset_callback = reset_callback
```

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

```
function priority_in_callback (name, description)
    if not name or name == ""
            or not callbacktypes[name]
            or not description then
        return false
    end
    local l = callbacklist[name]
    if not l then return false end
    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':\ntype '%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
```

```
    if not l then
        f = lua_callbacks_defaults[name]
    else
        f = handlers[callbacktypes[name]](name)
        if not f then
            err("unknown callback type")
            return
        end
    end
    return f(...)
end
luatexbase.call_callback = call_callback
```

That's all folks!

⟨/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.

⟨*testlua⟩

```
local msg = texio.write_nl
```

Test the management functions with a predefined callback.

```
local function sample(head,...)
    return head, true
end
local prio = luatexbase.priority_in_callback
msg("\n*********\n* Testing management functions\n*********")
luatexbase.add_to_callback("hpack_filter", sample, "sample one", 1)
luatexbase.add_to_callback("hpack_filter", sample, "sample two", 2)
luatexbase.add_to_callback("hpack_filter", sample, "sample three", 1)
assert(prio("hpack_filter", "sample three"))
luatexbase.remove_from_callback("hpack_filter", "sample three")
assert(not prio("hpack_filter", "sample three"))
luatexbase.reset_callback("hpack_filter")
assert(not prio("hpack_filter", "sample one"))
```

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

```
local function data_one(s)
  texio.write_nl("I'm data 1 whith argument: "..s)
  return s
end
local function data_two(s)
  texio.write_nl("I'm data 2 whith argument: "..s)
  return s
end
local function data_three(s)
  texio.write_nl("I'm data 3 whith argument: "..s)
  return s
end
msg("\n*********\n* Testing user-defined callbacks\n*********")
```

```
msg("* create one")
luatexbase.create_callback("fooback", "data", data_one)
msg("* call it")
luatexbase.call_callback("fooback", "default")
msg("* add two functions")
luatexbase.add_to_callback("fooback", data_two, "function two", 2)
luatexbase.add_to_callback("fooback", data_three, "function three", 1)
msg("* call")
luatexbase.call_callback("fooback", "all")
msg("* rm one function")
luatexbase.remove_from_callback("fooback", "function three")
msg("* call")
luatexbase.call_callback("fooback", "all but three")
msg("* reset")
luatexbase.reset_callback("fooback")
msg("* call")
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.

```
function add_hpack_filter()
    luatexbase.add_to_callback('hpack_filter', function(head, ...)
            texio.write_nl("I'm a dummy hpack_filter")
            return head
        end,
        'dummy hpack filter')
    luatexbase.add_to_callback('hpack_filter', function(head, ...)
            texio.write_nl("I'm an optimized dummy hpack_filter")
            return true
        end,
        'optimized dummy hpack filter')
end
function rm_one_hpack_filter()
    luatexbase.remove_from_callback('hpack_filter', 'dummy hpack filter')
end
function rm_two_hpack_filter()
    luatexbase.remove_from_callback('hpack_filter',
        'optimized dummy hpack filter')
end
```

A `simple` callback.

```
function add_hyphenate()
    luatexbase.add_to_callback('hyphenate', function(head, tail)
            texio.write_nl("I'm a dummy hyphenate")
        end,
        'dummy hyphenate')
    luatexbase.add_to_callback('hyphenate', function(head, tail)
            texio.write_nl("I'm an other dummy hyphenate")
        end,
        'other dummy hyphenate')
```

```
end
function rm_one_hyphenate()
    luatexbase.remove_from_callback('hyphenate', 'dummy hyphenate')
end
function rm_two_hyphenate()
    luatexbase.remove_from_callback('hyphenate', 'other dummy hyphenate')
end
```

A `first` callback.

```
function add_find_write_file()
    luatexbase.add_to_callback('find_write_file', function(id, name)
            texio.write_nl("I'm a dummy find_write_file")
            return "dummy-"..name
        end,
        'dummy find_write_file')
    luatexbase.add_to_callback('find_write_file', function(id, name)
            texio.write_nl("I'm an other dummy find_write_file")
            return "dummy-other-"..name
        end,
        'other dummy find_write_file')
end
function rm_one_find_write_file()
    luatexbase.remove_from_callback('find_write_file',
        'dummy find_write_file')
end
function rm_two_find_write_file()
    luatexbase.remove_from_callback('find_write_file',
        'other dummy find_write_file')
end
```

A `data` callback.

```
function add_process_input_buffer()
    luatexbase.add_to_callback('process_input_buffer', function(buffer)
            return buffer.."\\msg{dummy}"
        end,
        'dummy process_input_buffer')
    luatexbase.add_to_callback('process_input_buffer', function(buffer)
            return buffer.."\\msg{otherdummy}"
        end,
        'other dummy process_input_buffer')
end
function rm_one_process_input_buffer()
    luatexbase.remove_from_callback('process_input_buffer',
        'dummy process_input_buffer')
end
function rm_two_process_input_buffer()
    luatexbase.remove_from_callback('process_input_buffer',
        'other dummy process_input_buffer')
end
```

⟨/testlua⟩

⟨testplain⟩`\input luatexbase-mcb.sty`
⟨testlatex⟩`\RequirePackage{luatexbase-mcb}`
⟨∗testplain, testlatex⟩

```
\catcode 64 11
\def\msg{\immediate\write16}
\msg{===== BEGIN =====}
```

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

```
\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.

```
\def\test#1#2{%
  \msg{^^J*********^^J* Testing #1 (type #2)^^J*********}
  \msg{* Add two functions}
  \luatexbase@directlua{add_#1()}
  \csname test_#1\endcsname
  \msg{* Remove one}
  \luatexbase@directlua{rm_one_#1()}
  \csname test_#1\endcsname
  \msg{* Remove the second}
  \luatexbase@directlua{rm_two_#1()}
  \csname test_#1\endcsname
  \msg{* Add two functions again}
  \luatexbase@directlua{add_#1()}
  \csname test_#1\endcsname
  \msg{* Remove all functions}
  \luatexbase@directlua{luatexbase.reset_callback("#1")}
  \csname test_#1\endcsname
}
```

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

```
\catcode`\_ 11
⟨testlatex⟩\everypar{}
\def\test_hpack_filter{\setbox0=\hbox{bla}}
\def\test_hyphenate{\showhyphens{hyphenation}}
\def\test_find_write_file{\immediate\openout15 test-mcb-out.log}
\def\test_process_input_buffer{\input test-mcb-aux.tex}
```

Now actually test them

```
\test{hpack_filter}{list}
\test{hyphenate}{simple}
\test{find_write_file}{first}
\test{process_input_buffer}{data}
```

Done.

```
\msg{=====  END =====}
⟨/testplain, testlatex⟩
⟨testplain⟩\bye
⟨testlatex⟩\stop
```