ltluatex.dtx
(LuaTeX-specific support)

David Carlisle and Joseph Wright*

2021/12/27

Contents

---

*Significant portions of the code here are adapted/simplified from the packages luatex and luatexbase written by Heiko Oberdiek, Élie Roux, Manuel Pégourié-Gonnar and Philipp Gesang.

1

1 Overview

LuaTeX adds a number of engine-specific functions to TeX. Several of these require set up that is best done in the kernel or need related support functions. This file provides *basic* support for LuaTeX at the LaTeX 2_ε kernel level plus as a loadable file which can be used with plain TeX and LaTeX.

This file contains code for both TeX (to be stored as part of the format) and Lua (to be loaded at the start of each job). In the Lua code, the kernel uses the namespace `luatexbase`.

The following `\count` registers are used here for register allocation:

`\e@alloc@attribute@count`	Attributes (default 258)
`\e@alloc@ccodetable@count`	Category code tables (default 259)
`\e@alloc@luafunction@count`	Lua functions (default 260)
`\e@alloc@whatsit@count`	User whatsits (default 261)
`\e@alloc@bytecode@count`	Lua bytecodes (default 262)
`\e@alloc@luachunk@count`	Lua chunks (default 263)

(`\count` 256 is used for `\newmarks` allocation and `\count` 257 is used for `\newXeTeXintercharclass` with XeTeX, with code defined in `ltfinal.dtx`). With any LaTeX 2_ε kernel from 2015 onward these registers are part of the block in the extended area reserved by the kernel (prior to 2015 the LaTeX 2_ε kernel did not provide any functionality for the extended allocation area).

2 Core TeX functionality

The commands defined here are defined for possible inclusion in a future LaTeX format, however also extracted to the file `ltluatex.tex` which may be used with older LaTeX formats, and with plain TeX.

`\newattribute` `\newattribute{`⟨*attribute*⟩`}`
Defines a named `\attribute`, indexed from 1 (*i.e.* `\attribute0` is never defined). Attributes initially have the marker value `-"7FFFFFFF` ('unset') set by the engine.

`\newcatcodetable` `\newcatcodetable{`⟨*catcodetable*⟩`}`
Defines a named `\catcodetable`, indexed from 1 (`\catcodetable0` is never assigned). A new catcode table will be populated with exactly those values assigned by IniTeX (as described in the LuaTeX manual).

`\newluafunction` `\newluafunction{`⟨*function*⟩`}`
Defines a named `\luafunction`, indexed from 1. (Lua indexes tables from 1 so `\luafunction0` is not available).

`\newluacmd` `\newluadef{`⟨*function*⟩`}`
Like `\newluafunction`, but defines the command using `\luadef` instead of just assigning an integer.

`\newprotectedluacmd` `\newluadef{`⟨*function*⟩`}`
Like `\newluacmd`, but the defined command is not expandable.

`\newwhatsit` `\newwhatsit{`⟨*whatsit*⟩`}`
Defines a custom `\whatsit`, indexed from 1.

`\newluabytecode` `\newluabytecode{`⟨*bytecode*⟩`}`

Allocates a number for Lua bytecode register, indexed from 1.

`\newluachunkname`   `newluachunkname{⟨chunkname⟩}`
Allocates a number for Lua chunk register, indexed from 1. Also enters the name of the register (without backslash) into the `lua.name` table to be used in stack traces.

`\catcodetable@initex`   Predefined category code tables with the obvious assignments. Note that the
`\catcodetable@string`   `latex` and `atletter` tables set the full Unicode range to the codes predefined by
`\catcodetable@latex`   the kernel.
`\catcodetable@atletter`   `\setattribute{⟨attribute⟩}{⟨value⟩}`
`\setattribute`   `\unsetattribute{⟨attribute⟩}`
`\unsetattribute`   Set and unset attributes in a manner analogous to `\setlength`. Note that attributes take a marker value when unset so this operation is distinct from setting the value to zero.

3 Plain TEX interface

The ltluatex interface may be used with plain TEX using `\input{ltluatex}`. This inputs `ltluatex.tex` which inputs `etex.src` (or `etex.sty` if used with LATEX) if it is not already input, and then defines some internal commands to allow the ltluatex interface to be defined.

The luatexbase package interface may also be used in plain TEX, as before, by inputting the package `\input luatexbase.sty`. The new version of luatexbase is based on this ltluatex code but implements a compatibility layer providing the interface of the original package.

4 Lua functionality

4.1 Allocators in Lua

`new_attribute`   `luatexbase.new_attribute(⟨attribute⟩)`
Returns an allocation number for the ⟨attribute⟩, indexed from 1. The attribute will be initialised with the marker value -"7FFFFFFF ('unset'). The attribute allocation sequence is shared with the TEX code but this function does *not* define a token using `\attributedef`. The attribute name is recorded in the `attributes` table. A metatable is provided so that the table syntax can be used consistently for attributes declared in TEX or Lua.

`new_whatsit`   `luatexbase.new_whatsit(⟨whatsit⟩)`
Returns an allocation number for the custom ⟨whatsit⟩, indexed from 1.

`new_bytecode`   `luatexbase.new_bytecode(⟨bytecode⟩)`
Returns an allocation number for a bytecode register, indexed from 1. The optional ⟨name⟩ argument is just used for logging.

`new_chunkname`   `luatexbase.new_chunkname(⟨chunkname⟩)`
Returns an allocation number for a Lua chunk name for use with `\directlua` and `\latelua`, indexed from 1. The number is returned and also ⟨name⟩ argument is added to the `lua.name` array at that index.

`new_luafunction`   `luatexbase.new_luafunction(⟨functionname⟩)`
Returns an allocation number for a lua function for use with `\luafunction`, `\lateluafunction`, and `\luadef`, indexed from 1. The optional ⟨functionname⟩ argument is just used for logging.

These functions all require access to a named TeX count register to manage their allocations. The standard names are those defined above for access from TeX, *e.g.* "e@alloc@attribute@count, but these can be adjusted by defining the variable ⟨*type*⟩_count_name before loading ltluatex.lua, for example

```
local attribute_count_name = "attributetracker"
require("ltluatex")
```

would use a TeX \count (\countdef'd token) called attributetracker in place of "e@alloc@attribute@count.

4.2   Lua access to TeX register numbers

registernumber luatexbase.registernumer(⟨*name*⟩)

Sometimes (notably in the case of Lua attributes) it is necessary to access a register *by number* that has been allocated by TeX. This package provides a function to look up the relevant number using LuaTeX's internal tables. After for example \newattribute\myattrib, \myattrib would be defined by (say) \myattrib=\attribute15. luatexbase.registernumer("myattrib") would then return the register number, 15 in this case. If the string passed as argument does not correspond to a token defined by \attributedef, \countdef or similar commands, the Lua value false is returned.

As an example, consider the input:

```
\newcommand\test[1]{%
\typeout{#1: \expandafter\meaning\csname#1\endcsname^^J
\space\space\space\space
\directlua{tex.write(luatexbase.registernumber("#1") or "bad input")}%
}}

\test{undefinedrubbish}

\test{space}

\test{hbox}

\test{@MM}

\test{@tempdima}
\test{@tempdimb}

\test{strutbox}

\test{sixt@@n}

\attrbutedef\myattr=12
\myattr=200
\test{myattr}
```

If the demonstration code is processed with LuaLaTeX then the following would be produced in the log and terminal output.

```
undefinedrubbish: \relax
     bad input
space: macro:->
     bad input
hbox: \hbox
     bad input
@MM: \mathchar"4E20
     20000
@tempdima: \dimen14
     14
@tempdimb: \dimen15
     15
strutbox: \char"B
     11
sixt@@n: \char"10
     16
myattr: \attribute12
     12
```

Notice how undefined commands, or commands unrelated to registers do not produce an error, just return `false` and so print `bad input` here. Note also that commands defined by `\newbox` work and return the number of the box register even though the actual command holding this number is a `\chardef` defined token (there is no `\boxdef`).

4.3   Module utilities

provides_module luatexbase.provides_module(⟨*info*⟩)

This function is used by modules to identify themselves; the `info` should be a table containing information about the module. The required field `name` must contain the name of the module. It is recommended to provide a field `date` in the usual LaTeX format yyyy/mm/dd. Optional fields `version` (a string) and `description` may be used if present. This information will be recorded in the log. Other fields are ignored.

module_info luatexbase.module_info(⟨*module*⟩, ⟨*text*⟩)

module_warning luatexbase.module_warning(⟨*module*⟩, ⟨*text*⟩)

module_error luatexbase.module_error(⟨*module*⟩, ⟨*text*⟩)

These functions are similar to LaTeX's `\PackageError`, `\PackageWarning` and `\PackageInfo` in the way they format the output. No automatic line breaking is done, you may still use `\n` as usual for that, and the name of the package will be prepended to each output line.

Note that `luatexbase.module_error` raises an actual Lua error with `error()`, which currently means a call stack will be dumped. While this may not look pretty, at least it provides useful information for tracking the error down.

4.4   Callback management

add_to_callback luatexbase.add_to_callback(⟨*callback*⟩, ⟨*function*⟩, ⟨*description*⟩) Registers the ⟨*function*⟩ into the ⟨*callback*⟩ with a textual ⟨*description*⟩ of the function. Functions are inserted into the callback in the order loaded.

remove_from_callback luatexbase.remove_from_callback(⟨*callback*⟩, ⟨*description*⟩) Removes the call-

back function with ⟨*description*⟩ from the ⟨*callback*⟩. The removed function and its description are returned as the results of this function.

in_callback `luatexbase.in_callback(`⟨*callback*⟩`, `⟨*description*⟩`)` Checks if the ⟨*description*⟩ matches one of the functions added to the list for the ⟨*callback*⟩, returning a boolean value.

disable_callback `luatexbase.disable_callback(`⟨*callback*⟩`)` Sets the ⟨*callback*⟩ to `false` as described in the LuaTeX manual for the underlying `callback.register` built-in. Callbacks will only be set to false (and thus be skipped entirely) if there are no functions registered using the callback.

callback_descriptions A list of the descriptions of functions registered to the specified callback is returned. `{}` is returned if there are no functions registered.

create_callback `luatexbase.create_callback(`⟨*name*⟩`,metatype,`⟨*default*⟩`)` Defines a user defined callback. The last argument is a default function or `false`.

call_callback `luatexbase.call_callback(`⟨*name*⟩`,...)` Calls a user defined callback with the supplied arguments.

5 Implementation

```
1 ⟨∗2ekernel | tex | latexrelease⟩
2 ⟨2ekernel | latexrelease⟩\ifx\directlua\@undefined\else
```

5.1 Minimum LuaTeX version

LuaTeX has changed a lot over time. In the kernel support for ancient versions is not provided: trying to build a format with a very old binary therefore gives some information in the log and loading stops. The cut-off selected here relates to the tree-searching behaviour of `require()`: from version 0.60, LuaTeX will correctly find Lua files in the `texmf` tree without 'help'.

```
 3 ⟨latexrelease⟩\IncludeInRelease{2015/10/01}
 4 ⟨latexrelease⟩                    {\newluafunction}{LuaTeX}%
 5 \ifnum\luatexversion<60 %
 6   \wlog{************************************************}
 7   \wlog{* LuaTeX version too old for ltluatex support *}
 8   \wlog{************************************************}
 9   \expandafter\endinput
10 \fi
```

Two simple LaTeX macros from `ltdefns.dtx` have to be defined here because ltdefns.dtx is not loaded yet when ltluatex.dtx is executed.

```
11 \long\def\@gobble#1{}
12 \long\def\@firstofone#1{#1}
```

5.2 Older LaTeX/Plain TeX setup

```
13 ⟨∗tex⟩
```

Older LaTeX formats don't have the primitives with 'native' names: sort that out. If they already exist this will still be safe.

```
14 \directlua{tex.enableprimitives("",tex.extraprimitives("luatex"))}

15 \ifx\e@alloc\@undefined
```

In pre-2014 LaTeX, or plain TeX, load `etex.{sty,src}`.

```
16   \ifx\documentclass\@undefined
```

```
17    \ifx\loccount\@undefined
18      \input{etex.src}%
19    \fi
20    \catcode'\@=11 %
21    \outer\expandafter\def\csname newfam\endcsname
22                     {\alloc@8\fam\chardef\et@xmaxfam}
23  \else
24    \RequirePackage{etex}
25    \expandafter\def\csname newfam\endcsname
26                   {\alloc@8\fam\chardef\et@xmaxfam}
27    \expandafter\let\expandafter\new@mathgroup\csname newfam\endcsname
28  \fi
```

5.2.1 Fixes to `etex.src`/`etex.sty`

These could and probably should be made directly in an update to `etex.src` which already has some LuaTeX-specific code, but does not define the correct range for LuaTeX.

2015-07-13 higher range in luatex.

```
29 \edef \et@xmaxregs {\ifx\directlua\@undefined 32768\else 65536\fi}
```

luatex/xetex also allow more math fam.

```
30 \edef \et@xmaxfam {\ifx\Umathcode\@undefined\sixt@@n\else\@cclvi\fi}
```

```
31 \count 270=\et@xmaxregs % locally allocates \count registers
32 \count 271=\et@xmaxregs % ditto for \dimen registers
33 \count 272=\et@xmaxregs % ditto for \skip registers
34 \count 273=\et@xmaxregs % ditto for \muskip registers
35 \count 274=\et@xmaxregs % ditto for \box registers
36 \count 275=\et@xmaxregs % ditto for \toks registers
37 \count 276=\et@xmaxregs % ditto for \marks classes
```

and 256 or 16 fam. (Done above due to plain/LaTeX differences in ltluatex.)

```
38 % \outer\def\newfam{\alloc@8\fam\chardef\et@xmaxfam}
```

End of proposed changes to `etex.src`

5.2.2 luatex specific settings

Switch to global cf `luatex.sty` to leave room for inserts not really needed for luatex but possibly most compatible with existing use.

```
39 \expandafter\let\csname newcount\expandafter\expandafter\endcsname
40              \csname globcount\endcsname
41 \expandafter\let\csname newdimen\expandafter\expandafter\endcsname
42              \csname globdimen\endcsname
43 \expandafter\let\csname newskip\expandafter\expandafter\endcsname
44              \csname globskip\endcsname
45 \expandafter\let\csname newbox\expandafter\expandafter\endcsname
46              \csname globbox\endcsname
```

Define `\e@alloc` as in latex (the existing macros in `etex.src` hard to extend to further register types as they assume specific 26x and 27x count range. For compatibility the existing register allocation is not changed.

```
47 \chardef\e@alloc@top=65535
48 \let\e@alloc@chardef\chardef
```

```
49 \def\e@alloc#1#2#3#4#5#6{%
50   \global\advance#3\@ne
51   \e@ch@ck{#3}{#4}{#5}#1%
52   \allocationnumber#3\relax
53   \global#2#6\allocationnumber
54   \wlog{\string#6=\string#1\the\allocationnumber}}%

55 \gdef\e@ch@ck#1#2#3#4{%
56   \ifnum#1<#2\else
57     \ifnum#1=#2\relax
58       #1\@cclvi
59       \ifx\count#4\advance#1 10 \fi
60     \fi
61     \ifnum#1<#3\relax
62     \else
63       \errmessage{No room for a new \string#4}%
64     \fi
65   \fi}%
```

Fix up allocations not to clash with `etex.src`.

```
66 \expandafter\csname newcount\endcsname\e@alloc@attribute@count
67 \expandafter\csname newcount\endcsname\e@alloc@ccodetable@count
68 \expandafter\csname newcount\endcsname\e@alloc@luafunction@count
69 \expandafter\csname newcount\endcsname\e@alloc@whatsit@count
70 \expandafter\csname newcount\endcsname\e@alloc@bytecode@count
71 \expandafter\csname newcount\endcsname\e@alloc@luachunk@count
```

End of conditional setup for plain TeX / old LaTeX.

```
72 \fi
73 ⟨/tex⟩
```

5.3   Attributes

\newattribute As is generally the case for the LuaTeX registers we start here from 1. Notably, some code assumes that `\attribute0` is never used so this is important in this case.

```
74 \ifx\e@alloc@attribute@count\@undefined
75   \countdef\e@alloc@attribute@count=258
76   \e@alloc@attribute@count=\z@
77 \fi
78 \def\newattribute#1{%
79   \e@alloc\attribute\attributedef
80     \e@alloc@attribute@count\m@ne\e@alloc@top#1%
81 }
```

\setattribute   Handy utilities.
\unsetattribute
```
82 \def\setattribute#1#2{#1=\numexpr#2\relax}
83 \def\unsetattribute#1{#1=-"7FFFFFFF\relax}
```

5.4   Category code tables

\newcatcodetable Category code tables are allocated with a limit half of that used by LuaTeX for everything else. At the end of allocation there needs to be an initialization step.

Table 0 is already taken (it's the global one for current use) so the allocation starts at 1.

```
84 \ifx\e@alloc@ccodetable@count\@undefined
85   \countdef\e@alloc@ccodetable@count=259
86   \e@alloc@ccodetable@count=\z@
87 \fi
88 \def\newcatcodetable#1{%
89   \e@alloc\catcodetable\chardef
90     \e@alloc@ccodetable@count\m@ne{"8000}#1%
91   \initcatcodetable\allocationnumber
92 }
```

\catcodetable@initex   Save a small set of standard tables. The Unicode data is read here in using a parser
\catcodetable@string   simplified from that in `load-unicode-data`: only the nature of letters needs to
\catcodetable@latex   be detected.
\catcodetable@atletter

```
 93 \newcatcodetable\catcodetable@initex
 94 \newcatcodetable\catcodetable@string
 95 \begingroup
 96   \def\setrangecatcode#1#2#3{%
 97     \ifnum#1>#2 %
 98       \expandafter\@gobble
 99     \else
100       \expandafter\@firstofone
101     \fi
102     {%
103       \catcode#1=#3 %
104       \expandafter\setrangecatcode\expandafter
105         {\number\numexpr#1 + 1\relax}{#2}{#3}
106     }%
107   }
108   \@firstofone{%
109     \catcodetable\catcodetable@initex
110       \catcode0=12 %
111       \catcode13=12 %
112       \catcode37=12 %
113       \setrangecatcode{65}{90}{12}%
114       \setrangecatcode{97}{122}{12}%
115       \catcode92=12 %
116       \catcode127=12 %
117       \savecatcodetable\catcodetable@string
118     \endgroup
119   }%
120 \newcatcodetable\catcodetable@latex
121 \newcatcodetable\catcodetable@atletter
122 \begingroup
123   \def\parseunicodedataI#1;#2;#3;#4\relax{%
124     \parseunicodedataII#1;#3;#2 First>\relax
125   }%
126   \def\parseunicodedataII#1;#2;#3 First>#4\relax{%
127     \ifx\relax#4\relax
128       \expandafter\parseunicodedataIII
129     \else
130       \expandafter\parseunicodedataIV
131     \fi
```

9

```
132        {#1}#2\relax%
133      }%
134      \def\parseunicodedataIII#1#2#3\relax{%
135        \ifnum 0%
136          \if L#21\fi
137          \if M#21\fi
138          >0 %
139          \catcode"#1=11 %
140        \fi
141      }%
142      \def\parseunicodedataIV#1#2#3\relax{%
143        \read\unicoderead to \unicodedataline
144        \if L#2%
145          \count0="#1 %
146          \expandafter\parseunicodedataV\unicodedataline\relax
147        \fi
148      }%
149      \def\parseunicodedataV#1;#2\relax{%
150        \loop
151          \unless\ifnum\count0>"#1 %
152            \catcode\count0=11 %
153            \advance\count0 by 1 %
154        \repeat
155      }%
156      \def\storedpar{\par}%
157      \chardef\unicoderead=\numexpr\count16 + 1\relax
158      \openin\unicoderead=UnicodeData.txt %
159      \loop\unless\ifeof\unicoderead %
160        \read\unicoderead to \unicodedataline
161        \unless\ifx\unicodedataline\storedpar
162          \expandafter\parseunicodedataI\unicodedataline\relax
163        \fi
164      \repeat
165      \closein\unicoderead
166      \@firstofone{%
167        \catcode64=12 %
168        \savecatcodetable\catcodetable@latex
169        \catcode64=11 %
170        \savecatcodetable\catcodetable@atletter
171      }
172 \endgroup
```

5.5   Named Lua functions

\newluafunction   Much the same story for allocating LuaTeX functions except here they are just numbers so they are allocated in the same way as boxes. Lua indexes from 1 so once again slot 0 is skipped.

```
173 \ifx\e@alloc@luafunction@count\@undefined
174   \countdef\e@alloc@luafunction@count=260
175   \e@alloc@luafunction@count=\z@
176 \fi
177 \def\newluafunction{%
178   \e@alloc\luafunction\e@alloc@chardef
```

```
179        \e@alloc@luafunction@count\m@ne\e@alloc@top
180 }
```

\newluacmd Additionally two variants are provided to make the passed control sequence call
\newprotectedluacmd the function directly.

```
181 \def\newluacmd{%
182   \e@alloc\luafunction\luadef
183     \e@alloc@luafunction@count\m@ne\e@alloc@top
184 }
185 \def\newprotectedluacmd{%
186   \e@alloc\luafunction{\protected\luadef}
187     \e@alloc@luafunction@count\m@ne\e@alloc@top
188 }
```

5.6   Custom whatsits

\newwhatsit These are only settable from Lua but for consistency are definable here.

```
189 \ifx\e@alloc@whatsit@count\@undefined
190   \countdef\e@alloc@whatsit@count=261
191   \e@alloc@whatsit@count=\z@
192 \fi
193 \def\newwhatsit#1{%
194   \e@alloc\whatsit\e@alloc@chardef
195     \e@alloc@whatsit@count\m@ne\e@alloc@top#1%
196 }
```

5.7   Lua bytecode registers

\newluabytecode These are only settable from Lua but for consistency are definable here.

```
197 \ifx\e@alloc@bytecode@count\@undefined
198   \countdef\e@alloc@bytecode@count=262
199   \e@alloc@bytecode@count=\z@
200 \fi
201 \def\newluabytecode#1{%
202   \e@alloc\luabytecode\e@alloc@chardef
203     \e@alloc@bytecode@count\m@ne\e@alloc@top#1%
204 }
```

5.8   Lua chunk registers

\newluachunkname As for bytecode registers, but in addition we need to add a string to the lua.name
table to use in stack tracing. We use the name of the command passed to the
allocator, with no backslash.

```
205 \ifx\e@alloc@luachunk@count\@undefined
206   \countdef\e@alloc@luachunk@count=263
207   \e@alloc@luachunk@count=\z@
208 \fi
209 \def\newluachunkname#1{%
210   \e@alloc\luachunk\e@alloc@chardef
211     \e@alloc@luachunk@count\m@ne\e@alloc@top#1%
212     {\escapechar\m@ne
213     \directlua{lua.name[\the\allocationnumber]="\string#1"}}%
214 }
```

11

5.9 Lua loader

Lua code loaded in the format often has to be loaded again at the beginning of every job, so we define a helper which allows us to avoid duplicated code:

```
215 \def\now@and@everyjob#1{%
216   \everyjob\expandafter{\the\everyjob
217     #1%
218   }%
219   #1%
220 }
```

Load the Lua code at the start of every job. For the conversion of TeX into numbers at the Lua side we need some known registers: for convenience we use a set of systematic names, which means using a group around the Lua loader.

```
221 ⟨2ekernel⟩\now@and@everyjob{%
222   \begingroup
223     \attributedef\attributezero=0 %
224     \chardef     \charzero     =0 %
```

Note name change required on older luatex, for hash table access.

```
225     \countdef    \CountZero    =0 %
226     \dimendef    \dimenzero    =0 %
227     \mathchardef \mathcharzero =0 %
228     \muskipdef   \muskipzero   =0 %
229     \skipdef     \skipzero     =0 %
230     \toksdef     \tokszero     =0 %
231     \directlua{require("ltluatex")}
232   \endgroup
233 ⟨2ekernel⟩}
234 ⟨latexrelease⟩\EndIncludeInRelease

235 ⟨latexrelease⟩\IncludeInRelease{0000/00/00}
236 ⟨latexrelease⟩                    {\newluafunction}{LuaTeX}%
237 ⟨latexrelease⟩\let\e@alloc@attribute@count\@undefined
238 ⟨latexrelease⟩\let\newattribute\@undefined
239 ⟨latexrelease⟩\let\setattribute\@undefined
240 ⟨latexrelease⟩\let\unsetattribute\@undefined
241 ⟨latexrelease⟩\let\e@alloc@ccodetable@count\@undefined
242 ⟨latexrelease⟩\let\newcatcodetable\@undefined
243 ⟨latexrelease⟩\let\catcodetable@initex\@undefined
244 ⟨latexrelease⟩\let\catcodetable@string\@undefined
245 ⟨latexrelease⟩\let\catcodetable@latex\@undefined
246 ⟨latexrelease⟩\let\catcodetable@atletter\@undefined
247 ⟨latexrelease⟩\let\e@alloc@luafunction@count\@undefined
248 ⟨latexrelease⟩\let\newluafunction\@undefined
249 ⟨latexrelease⟩\let\e@alloc@luafunction@count\@undefined
250 ⟨latexrelease⟩\let\newwhatsit\@undefined
251 ⟨latexrelease⟩\let\e@alloc@whatsit@count\@undefined
252 ⟨latexrelease⟩\let\newluabytecode\@undefined
253 ⟨latexrelease⟩\let\e@alloc@bytecode@count\@undefined
254 ⟨latexrelease⟩\let\newluachunkname\@undefined
255 ⟨latexrelease⟩\let\e@alloc@luachunk@count\@undefined
256 ⟨latexrelease⟩\directlua{luatexbase.uninstall()}
257 ⟨latexrelease⟩\EndIncludeInRelease
```

In \everyjob, if luaotfload is available, load it and switch to TU.

```
258 ⟨latexrelease⟩\IncludeInRelease{2017/01/01}%
259 ⟨latexrelease⟩                      {\fontencoding}{TU in everyjob}%
260 ⟨latexrelease⟩\fontencoding{TU}\let\encodingdefault\f@encoding
261 ⟨latexrelease⟩\ifx\directlua\@undefined\else
262 ⟨2ekernel⟩\everyjob\expandafter{%
263 ⟨2ekernel⟩  \the\everyjob
264 ⟨*2ekernel, latexrelease⟩
265   \directlua{%
266   if xpcall(function ()%
267               require('luaotfload-main')%
268             end,texio.write_nl) then %
269   local _void = luaotfload.main ()%
270   else %
271   texio.write_nl('Error in luaotfload: reverting to OT1')%
272   tex.print('\string\\def\string\\encodingdefault{OT1}')%
273   end %
274   }%
275   \let\f@encoding\encodingdefault
276   \expandafter\let\csname ver@luaotfload.sty\endcsname\fmtversion
277 ⟨/2ekernel, latexrelease⟩
278 ⟨latexrelease⟩\fi
279 ⟨2ekernel⟩  }
280 ⟨latexrelease⟩\EndIncludeInRelease
281 ⟨latexrelease⟩\IncludeInRelease{0000/00/00}%
282 ⟨latexrelease⟩                      {\fontencoding}{TU in everyjob}%
283 ⟨latexrelease⟩\fontencoding{OT1}\let\encodingdefault\f@encoding
284 ⟨latexrelease⟩\EndIncludeInRelease

285 ⟨2ekernel | latexrelease⟩\fi
286 ⟨/2ekernel | tex | latexrelease⟩
```

5.10 Lua module preliminaries

```
287 ⟨*lua⟩
```

Some set up for the Lua module which is needed for all of the Lua functionality added here.

luatexbase Set up the table for the returned functions. This is used to expose all of the public functions.

```
288 luatexbase        = luatexbase or { }
289 local luatexbase = luatexbase
```

Some Lua best practice: use local versions of functions where possible.

```
290 local string_gsub      = string.gsub
291 local tex_count        = tex.count
292 local tex_setattribute = tex.setattribute
293 local tex_setcount     = tex.setcount
294 local texio_write_nl   = texio.write_nl
295 local flush_list       = node.flush_list

296 local luatexbase_warning
297 local luatexbase_error
```

13

5.11   Lua module utilities

5.11.1   Module tracking

modules To allow tracking of module usage, a structure is provided to store information and to return it.

```
298 local modules = modules or { }
```

provides_module Local function to write to the log.

```
299 local function luatexbase_log(text)
300   texio_write_nl("log", text)
301 end
```

Modelled on `\ProvidesPackage`, we store much the same information but with a little more structure.

```
302 local function provides_module(info)
303   if not (info and info.name) then
304     luatexbase_error("Missing module name for provides_module")
305   end
306   local function spaced(text)
307     return text and (" " .. text) or ""
308   end
309   luatexbase_log(
310     "Lua module: " .. info.name
311       .. spaced(info.date)
312       .. spaced(info.version)
313       .. spaced(info.description)
314   )
315   modules[info.name] = info
316 end
317 luatexbase.provides_module = provides_module
```

5.11.2   Module messages

There are various warnings and errors that need to be given. For warnings we can get exactly the same formatting as from TeX. For errors we have to make some changes. Here we give the text of the error in the LaTeX format then force an error from Lua to halt the run. Splitting the message text is done using `\n` which takes the place of `\MessageBreak`.

First an auxiliary for the formatting: this measures up the message leader so we always get the correct indent.

```
318 local function msg_format(mod, msg_type, text)
319   local leader = ""
320   local cont
321   local first_head
322   if mod == "LaTeX" then
323     cont = string_gsub(leader, ".", " ")
324     first_head = leader .. "LaTeX: "
325   else
326     first_head = leader .. "Module "  .. msg_type
327     cont = "(" .. mod .. ")"
328       .. string_gsub(first_head, ".", " ")
329     first_head =  leader .. "Module "  .. mod .. " " .. msg_type  .. ":"
330   end
```

```
331  if msg_type == "Error" then
332    first_head = "\n" .. first_head
333  end
334  if string.sub(text,-1) ~= "\n" then
335    text = text .. " "
336  end
337  return first_head .. " "
338    .. string_gsub(
339        text
340  .. "on input line "
341      .. tex.inputlineno, "\n", "\n" .. cont .. " "
342    )
343    .. "\n"
344 end
```

module_info   Write messages.
module_warning
module_error
```
345 local function module_info(mod, text)
346   texio_write_nl("log", msg_format(mod, "Info", text))
347 end
348 luatexbase.module_info = module_info
349 local function module_warning(mod, text)
350   texio_write_nl("term and log",msg_format(mod, "Warning", text))
351 end
352 luatexbase.module_warning = module_warning
353 local function module_error(mod, text)
354   error(msg_format(mod, "Error", text))
355 end
356 luatexbase.module_error = module_error
```

Dedicated versions for the rest of the code here.

```
357 function luatexbase_warning(text)
358   module_warning("luatexbase", text)
359 end
360 function luatexbase_error(text)
361   module_error("luatexbase", text)
362 end
```

5.12 Accessing register numbers from Lua

Collect up the data from the TeX level into a Lua table: from version 0.80, LuaTeX makes that easy.

```
363 local luaregisterbasetable = { }
364 local registermap = {
365   attributezero = "assign_attr"    ,
366   charzero      = "char_given"     ,
367   CountZero     = "assign_int"     ,
368   dimenzero     = "assign_dimen"   ,
369   mathcharzero  = "math_given"     ,
370   muskipzero    = "assign_mu_skip" ,
371   skipzero      = "assign_skip"    ,
372   tokszero      = "assign_toks"    ,
373 }
374 local createtoken
```

```
375 if tex.luatexversion > 81 then
376   createtoken = token.create
377 elseif tex.luatexversion > 79 then
378   createtoken = newtoken.create
379 end
380 local hashtokens    = tex.hashtokens()
381 local luatexversion = tex.luatexversion
382 for i,j in pairs (registermap) do
383   if luatexversion < 80 then
384     luaregisterbasetable[hashtokens[i][1]] =
385       hashtokens[i][2]
386   else
387     luaregisterbasetable[j] = createtoken(i).mode
388   end
389 end
```

registernumber Working out the correct return value can be done in two ways. For older LuaTeX
releases it has to be extracted from the `hashtokens`. On the other hand, newer
LuaTeX's have `newtoken`, and whilst `.mode` isn't currently documented, Hans
Hagen pointed to this approach so we should be OK.

```
390 local registernumber
391 if luatexversion < 80 then
392   function registernumber(name)
393     local nt = hashtokens[name]
394     if(nt and luaregisterbasetable[nt[1]]) then
395       return nt[2] - luaregisterbasetable[nt[1]]
396     else
397       return false
398     end
399   end
400 else
401   function registernumber(name)
402     local nt = createtoken(name)
403     if(luaregisterbasetable[nt.cmdname]) then
404       return nt.mode - luaregisterbasetable[nt.cmdname]
405     else
406       return false
407     end
408   end
409 end
410 luatexbase.registernumber = registernumber
```

5.13   Attribute allocation

new_attribute As attributes are used for Lua manipulations its useful to be able to assign from
this end.

```
411 local attributes=setmetatable(
412 {},
413 {
414 __index = function(t,key)
415 return registernumber(key) or nil
416 end}
417 )
```

```
418 luatexbase.attributes = attributes
419 local attribute_count_name =
420                     attribute_count_name or "e@alloc@attribute@count"
421 local function new_attribute(name)
422   tex_setcount("global", attribute_count_name,
423                     tex_count[attribute_count_name] + 1)
424   if tex_count[attribute_count_name] > 65534 then
425     luatexbase_error("No room for a new \\attribute")
426   end
427   attributes[name]= tex_count[attribute_count_name]
428   luatexbase_log("Lua-only attribute " .. name .. " = " ..
429                 tex_count[attribute_count_name])
430   return tex_count[attribute_count_name]
431 end
432 luatexbase.new_attribute = new_attribute
```

5.14   Custom whatsit allocation

new_whatsit Much the same as for attribute allocation in Lua.

```
433 local whatsit_count_name = whatsit_count_name or "e@alloc@whatsit@count"
434 local function new_whatsit(name)
435   tex_setcount("global", whatsit_count_name,
436                     tex_count[whatsit_count_name] + 1)
437   if tex_count[whatsit_count_name] > 65534 then
438     luatexbase_error("No room for a new custom whatsit")
439   end
440   luatexbase_log("Custom whatsit " .. (name or "") .. " = " ..
441                 tex_count[whatsit_count_name])
442   return tex_count[whatsit_count_name]
443 end
444 luatexbase.new_whatsit = new_whatsit
```

5.15   Bytecode register allocation

new_bytecode Much the same as for attribute allocation in Lua. The optional ⟨name⟩ argument is used in the log if given.

```
445 local bytecode_count_name =
446                     bytecode_count_name or "e@alloc@bytecode@count"
447 local function new_bytecode(name)
448   tex_setcount("global", bytecode_count_name,
449                     tex_count[bytecode_count_name] + 1)
450   if tex_count[bytecode_count_name] > 65534 then
451     luatexbase_error("No room for a new bytecode register")
452   end
453   luatexbase_log("Lua bytecode " .. (name or "") .. " = " ..
454                 tex_count[bytecode_count_name])
455   return tex_count[bytecode_count_name]
456 end
457 luatexbase.new_bytecode = new_bytecode
```

5.16 Lua chunk name allocation

new_chunkname — As for bytecode registers but also store the name in the `lua.name` table.

```
458 local chunkname_count_name =
459                        chunkname_count_name or "e@alloc@luachunk@count"
460 local function new_chunkname(name)
461   tex_setcount("global", chunkname_count_name,
462                        tex_count[chunkname_count_name] + 1)
463   local chunkname_count = tex_count[chunkname_count_name]
464   chunkname_count = chunkname_count + 1
465   if chunkname_count > 65534 then
466     luatexbase_error("No room for a new chunkname")
467   end
468   lua.name[chunkname_count]=name
469   luatexbase_log("Lua chunkname " .. (name or "") .. " = " ..
470                  chunkname_count .. "\n")
471   return chunkname_count
472 end
473 luatexbase.new_chunkname = new_chunkname
```

5.17 Lua function allocation

new_luafunction — Much the same as for attribute allocation in Lua. The optional ⟨*name*⟩ argument is used in the log if given.

```
474 local luafunction_count_name =
475                        luafunction_count_name or "e@alloc@luafunction@count"
476 local function new_luafunction(name)
477   tex_setcount("global", luafunction_count_name,
478                        tex_count[luafunction_count_name] + 1)
479   if tex_count[luafunction_count_name] > 65534 then
480     luatexbase_error("No room for a new luafunction register")
481   end
482   luatexbase_log("Lua function " .. (name or "") .. " = " ..
483                  tex_count[luafunction_count_name])
484   return tex_count[luafunction_count_name]
485 end
486 luatexbase.new_luafunction = new_luafunction
```

5.18 Lua callback management

The native mechanism for callbacks in LuaTeX allows only one per function. That is extremely restrictive and so a mechanism is needed to add and remove callbacks from the appropriate hooks.

5.18.1 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.

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

18

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

```
488 local list, data, exclusive, simple, reverselist = 1, 2, 3, 4, 5
489 local types    = {
490   list         = list,
491   data         = data,
492   exclusive    = exclusive,
493   simple       = simple,
494   reverselist = reverselist,
495 }
```

Now, list all predefined callbacks with their current type, based on the LuaTeX manual version 1.01. A full list of the currently-available callbacks can be obtained using

```
\directlua{
  for i,_ in pairs(callback.list()) do
    texio.write_nl("- " .. i)
  end
}
\bye
```

in plain LuaTeX. (Some undocumented callbacks are omitted as they are to be removed.)

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

Section 8.2: file discovery callbacks.

```
497   find_read_file     = exclusive,
498   find_write_file    = exclusive,
499   find_font_file     = data,
500   find_output_file   = data,
501   find_format_file   = data,
502   find_vf_file       = data,
503   find_map_file      = data,
504   find_enc_file      = data,
505   find_pk_file       = data,
506   find_data_file     = data,
507   find_opentype_file = data,
508   find_truetype_file = data,
509   find_type1_file    = data,
510   find_image_file    = data,

511   open_read_file     = exclusive,
512   read_font_file     = exclusive,
513   read_vf_file       = exclusive,
514   read_map_file      = exclusive,
515   read_enc_file      = exclusive,
516   read_pk_file       = exclusive,
517   read_data_file     = exclusive,
518   read_truetype_file = exclusive,
519   read_type1_file    = exclusive,
520   read_opentype_file = exclusive,
```

Not currently used by luatex but included for completeness. may be used by a font handler.

```
521  find_cidmap_file   = data,
522  read_cidmap_file   = exclusive,
```

Section 8.3: data processing callbacks.

```
523  process_input_buffer  = data,
524  process_output_buffer = data,
525  process_jobname       = data,
```

Section 8.4: node list processing callbacks.

```
526  contribute_filter     = simple,
527  buildpage_filter      = simple,
528  build_page_insert     = exclusive,
529  pre_linebreak_filter  = list,
530  linebreak_filter      = exclusive,
531  append_to_vlist_filter = exclusive,
532  post_linebreak_filter = reverselist,
533  hpack_filter          = list,
534  vpack_filter          = list,
535  hpack_quality         = exclusive,
536  vpack_quality         = exclusive,
537  pre_output_filter     = list,
538  process_rule          = exclusive,
539  hyphenate             = simple,
540  ligaturing            = simple,
541  kerning               = simple,
542  insert_local_par      = simple,
543  pre_mlist_to_hlist_filter = list,
544  mlist_to_hlist        = exclusive,
545  post_mlist_to_hlist_filter = reverselist,
546  new_graf              = exclusive,
```

Section 8.5: information reporting callbacks.

```
547  pre_dump            = simple,
548  start_run           = simple,
549  stop_run            = simple,
550  start_page_number   = simple,
551  stop_page_number    = simple,
552  show_error_hook     = simple,
553  show_warning_message = simple,
554  show_error_message  = simple,
555  show_lua_error_hook = simple,
556  start_file          = simple,
557  stop_file           = simple,
558  call_edit           = simple,
559  finish_synctex      = simple,
560  wrapup_run          = simple,
```

Section 8.6: PDF-related callbacks.

```
561  finish_pdffile         = data,
562  finish_pdfpage         = data,
563  page_objnum_provider   = data,
564  page_order_index       = data,
565  process_pdf_image_content = data,
```

Section 8.7: font-related callbacks.

```
566  define_font                = exclusive,
```

```
567  glyph_info                          = exclusive,
568  glyph_not_found                     = exclusive,
569  glyph_stream_provider               = exclusive,
570  make_extensible                     = exclusive,
571  font_descriptor_objnum_provider = exclusive,
572  input_level_string                  = exclusive,
573  provide_charproc_data               = exclusive,

574 }
575 luatexbase.callbacktypes=callbacktypes
```

callback.register Save the original function for registering callbacks and prevent the original being used. The original is saved in a place that remains available so other more sophisticated code can override the approach taken by the kernel if desired.

```
576 local callback_register = callback_register or callback.register
577 function callback.register()
578   luatexbase_error("Attempt to use callback.register() directly\n")
579 end
```

5.18.2 Handlers

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, the 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.

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.

reverselist is a specialized variant of *list* which executes functions in inverse order.

exclusive is for functions with more complex signatures; functions in this type of callback are *not* combined: An error is raised if a second callback is registered.

Handler for `data` callbacks.

```
580 local function data_handler(name)
581   return function(data, ...)
582     for _,i in ipairs(callbacklist[name]) do
583       data = i.func(data,...)
584     end
585     return data
586   end
587 end
```

Default for user-defined `data` callbacks without explicit default.

```
588 local function data_handler_default(value)
589   return value
590 end
```

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

```
591 local function exclusive_handler(name)
592   return function(...)
593     return callbacklist[name][1].func(...)
594   end
595 end
```

Handler for `list` callbacks.

```
596 local function list_handler(name)
597   return function(head, ...)
598     local ret
599     for _,i in ipairs(callbacklist[name]) do
600       ret = i.func(head, ...)
601       if ret == false then
602         luatexbase_warning(
603           "Function '" .. i.description .. "' returned false\n"
604             .. "in callback '" .. name .."'"
605          )
606         return false
607       end
608       if ret ~= true then
609         head = ret
610       end
611     end
612     return head
613   end
614 end
```

Default for user-defined `list` and `reverselist` callbacks without explicit default.

```
615 local function list_handler_default(head)
616 return head
617 end
```

Handler for `reverselist` callbacks.

```
618 local function reverselist_handler(name)
619   return function(head, ...)
```

```
620    local ret
621    local callbacks = callbacklist[name]
622    for i = #callbacks, 1, -1 do
623      local cb = callbacks[i]
624      ret = cb.func(head, ...)
625      if ret == false then
626        luatexbase_warning(
627          "Function '" .. cb.description .. "' returned false\n"
628            .. "in callback '" .. name .."'"
629        )
630        return false
631      end
632      if ret ~= true then
633        head = ret
634      end
635    end
636    return head
637  end
638 end
```

Handler for `simple` callbacks.

```
639 local function simple_handler(name)
640   return function(...)
641     for _,i in ipairs(callbacklist[name]) do
642       i.func(...)
643     end
644   end
645 end
```

Default for user-defined `simple` callbacks without explicit default.

```
646 local function simple_handler_default()
647 end
```

Keep a handlers table for indexed access and a table with the corresponding default functions.

```
648 local handlers  = {
649   [data]       = data_handler,
650   [exclusive]  = exclusive_handler,
651   [list]       = list_handler,
652   [reverselist] = reverselist_handler,
653   [simple]     = simple_handler,
654 }
655 local defaults = {
656   [data]       = data_handler_default,
657   [exclusive]  = nil,
658   [list]       = list_handler_default,
659   [reverselist] = list_handler_default,
660   [simple]     = simple_handler_default,
661 }
```

5.18.3 Public functions for callback management

Defining user callbacks perhaps should be in package code, but impacts on `add_to_callback`. If a default function is not required, it may be declared as `false`. First we need a list of user callbacks.

```
662 local user_callbacks_defaults = {
663   pre_mlist_to_hlist_filter = list_handler_default,
664   mlist_to_hlist = node.mlist_to_hlist,
665   post_mlist_to_hlist_filter = list_handler_default,
666 }
```

create_callback The allocator itself.

```
667 local function create_callback(name, ctype, default)
668   local ctype_id = types[ctype]
669   if not name  or name  == ""
670   or not ctype_id
671   then
672     luatexbase_error("Unable to create callback:\n" ..
673                      "valid callback name and type required")
674   end
675   if callbacktypes[name] then
676     luatexbase_error("Unable to create callback '" .. name ..
677                        "':\ncallback is already defined")
678   end
679   default = default or defaults[ctype_id]
680   if not default then
681     luatexbase_error("Unable to create callback '" .. name ..
682                        "':\ndefault is required for '" .. ctype ..
683                        "' callbacks")
684   elseif type (default) ~= "function" then
685     luatexbase_error("Unable to create callback '" .. name ..
686                        "':\ndefault is not a function")
687   end
688   user_callbacks_defaults[name] = default
689   callbacktypes[name] = ctype_id
690 end
691 luatexbase.create_callback = create_callback
```

call_callback Call a user defined callback. First check arguments.

```
692 local function call_callback(name,...)
693   if not name or name == "" then
694     luatexbase_error("Unable to create callback:\n" ..
695                      "valid callback name required")
696   end
697   if user_callbacks_defaults[name] == nil then
698     luatexbase_error("Unable to call callback '" .. name
699                        .. "':\nunknown or empty")
700    end
701   local l = callbacklist[name]
702   local f
703   if not l then
704     f = user_callbacks_defaults[name]
705   else
706     f = handlers[callbacktypes[name]](name)
707   end
708   return f(...)
709 end
710 luatexbase.call_callback=call_callback
```

add_to_callback Add a function to a callback. First check arguments.

```
711 local function add_to_callback(name, func, description)
712   if not name or name == "" then
713     luatexbase_error("Unable to register callback:\n" ..
714                      "valid callback name required")
715   end
716   if not callbacktypes[name] or
717     type(func) ~= "function" or
718     not description or
719     description == "" then
720     luatexbase_error(
721       "Unable to register callback.\n\n"
722         .. "Correct usage:\n"
723         .. "add_to_callback(<callback>, <function>, <description>)"
724     )
725   end
```

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

```
726   local l = callbacklist[name]
727   if l == nil then
728     l = { }
729     callbacklist[name] = l
```

If it is not a user defined callback use the primitive callback register.

```
730     if user_callbacks_defaults[name] == nil then
731       callback_register(name, handlers[callbacktypes[name]](name))
732     end
733   end
```

Actually register the function and give an error if more than one `exclusive` one is registered.

```
734   local f = {
735     func        = func,
736     description = description,
737   }
738   local priority = #l + 1
739   if callbacktypes[name] == exclusive then
740     if #l == 1 then
741       luatexbase_error(
742         "Cannot add second callback to exclusive function\n'" ..
743         name .. "'")
744     end
745   end
746   table.insert(l, priority, f)
```

Keep user informed.

```
747   luatexbase_log(
748     "Inserting '" .. description .. "' at position "
749       .. priority .. " in '" .. name .. "'."
750   )
751 end
752 luatexbase.add_to_callback = add_to_callback
```

remove_from_callback Remove a function from a callback. First check arguments.

```
753 local function remove_from_callback(name, description)
754   if not name or name == "" then
755     luatexbase_error("Unable to remove function from callback:\n" ..
756                      "valid callback name required")
757   end
758   if not callbacktypes[name] or
759      not description or
760      description == "" then
761     luatexbase_error(
762       "Unable to remove function from callback.\n\n"
763         .. "Correct usage:\n"
764         .. "remove_from_callback(<callback>, <description>)"
765     )
766   end
767   local l = callbacklist[name]
768   if not l then
769     luatexbase_error(
770       "No callback list for '" .. name .. "'\n")
771   end
```

Loop over the callback's function list until we find a matching entry. Remove it
and check if the list is empty: if so, unregister the callback handler.

```
772   local index = false
773   for i,j in ipairs(l) do
774     if j.description == description then
775       index = i
776       break
777     end
778   end
779   if not index then
780     luatexbase_error(
781       "No callback '" .. description .. "' registered for '" ..
782       name .. "'\n")
783   end
784   local cb = l[index]
785   table.remove(l, index)
786   luatexbase_log(
787     "Removing  '" .. description .. "' from '" .. name .. "'."
788   )
789   if #l == 0 then
790     callbacklist[name] = nil
791     if user_callbacks_defaults[name] == nil then
792       callback_register(name, nil)
793     end
794   end
795   return cb.func,cb.description
796 end
797 luatexbase.remove_from_callback = remove_from_callback
```

in_callback Look for a function description in a callback.

```
798 local function in_callback(name, description)
799   if not name
800     or name == ""
801     or not callbacklist[name]
```

```
802      or not callbacktypes[name]
803      or not description then
804         return false
805    end
806    for _, i in pairs(callbacklist[name]) do
807      if i.description == description then
808         return true
809      end
810    end
811    return false
812 end
813 luatexbase.in_callback = in_callback
```

disable_callback As we subvert the engine interface we need to provide a way to access this functionality.

```
814 local function disable_callback(name)
815    if(callbacklist[name] == nil) then
816      callback_register(name, false)
817    else
818      luatexbase_error("Callback list for " .. name .. " not empty")
819    end
820 end
821 luatexbase.disable_callback = disable_callback
```

callback_descriptions List the descriptions of functions registered for the given callback.

```
822 local function callback_descriptions (name)
823    local d = {}
824    if not name
825      or name == ""
826      or not callbacklist[name]
827      or not callbacktypes[name]
828      then
829      return d
830    else
831    for k, i in pairs(callbacklist[name]) do
832      d[k]= i.description
833      end
834    end
835    return d
836 end
837 luatexbase.callback_descriptions =callback_descriptions
```

uninstall Unlike at the TeX level, we have to provide a back-out mechanism here at the same time as the rest of the code. This is not meant for use by anything other than latexrelease: as such this is *deliberately* not documented for users!

```
838 local function uninstall()
839    module_info(
840      "luatexbase",
841      "Uninstalling kernel luatexbase code"
842    )
843    callback.register = callback_register
844    luatexbase = nil
845 end
846 luatexbase.uninstall = uninstall
```

27

mlist_to_hlist To emulate these callbacks, the "real" `mlist_to_hlist` is replaced by a wrapper calling the wrappers before and after.

```
847 callback_register("mlist_to_hlist", function(head, display_type, need_penalties)
848   local current = call_callback("pre_mlist_to_hlist_filter", head, display_type, need_penalti
849   if current == false then
850     flush_list(head)
851     return nil
852   end
853   current = call_callback("mlist_to_hlist", current, display_type, need_penalties)
854   local post = call_callback("post_mlist_to_hlist_filter", current, display_type, need_penalt
855   if post == false then
856     flush_list(current)
857     return nil
858   end
859   return post
860 end)
```

861 ⟨/lua⟩

Reset the catcode of @.

862 ⟨tex⟩\catcode'\@=\etatcatcode\relax