

# lualatex.dtx

## (LuaTeX-specific support)

David Carlisle and Joseph Wright\*

2019/10/22

## Contents

<b>1 Overview</b>	<b>2</b>
<b>2 Core TeX functionality</b>	<b>2</b>
<b>3 Plain TeX interface</b>	<b>3</b>
<b>4 Lua functionality</b>	<b>3</b>
4.1 Allocators in Lua . . . . .	3
4.2 Lua access to TeX register numbers . . . . .	4
4.3 Module utilities . . . . .	5
4.4 Callback management . . . . .	5
<b>5 Implementation</b>	<b>6</b>
5.1 Minimum LuaTeX version . . . . .	6
5.2 Older L <sup>A</sup> TeX/Plain TeX setup . . . . .	6
5.3 Attributes . . . . .	8
5.4 Category code tables . . . . .	8
5.5 Named Lua functions . . . . .	10
5.6 Custom whatsits . . . . .	10
5.7 Lua bytecode registers . . . . .	11
5.8 Lua chunk registers . . . . .	11
5.9 Lua loader . . . . .	11
5.10 Lua module preliminaries . . . . .	13
5.11 Lua module utilities . . . . .	13
5.12 Accessing register numbers from Lua . . . . .	15
5.13 Attribute allocation . . . . .	16
5.14 Custom whatsit allocation . . . . .	16
5.15 Bytecode register allocation . . . . .	17
5.16 Lua chunk name allocation . . . . .	17
5.17 Lua function allocation . . . . .	17
5.18 Lua callback management . . . . .	18

---

\*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 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 L<sup>A</sup>T<sub>Ε</sub>X 2<sub>ε</sub> kernel level plus as a loadable file which can be used with plain TeX and L<sup>A</sup>T<sub>Ε</sub>X.

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 L<sup>A</sup>T<sub>Ε</sub>X 2<sub>ε</sub> kernel from 2015 onward these registers are part of the block in the extended area reserved by the kernel (prior to 2015 the L<sup>A</sup>T<sub>Ε</sub>X 2<sub>ε</sub> 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 L<sup>A</sup>T<sub>Ε</sub>X format, however also extracted to the file `ltluatex.tex` which may be used with older L<sup>A</sup>T<sub>Ε</sub>X formats, and with plain TeX.

<code>\newattribute</code>	<code>\newattribute{&lt;attribute&gt;}</code> Defines a named <code>\attribute</code> , indexed from 1 ( <i>i.e.</i> <code>\attribute0</code> is never defined). Attributes initially have the marker value <code>-7FFFFFFF</code> ('unset') set by the engine.
<code>\newcatcodetable</code>	<code>\newcatcodetable{&lt;catcodetable&gt;}</code> Defines a named <code>\catcodetable</code> , indexed from 1 ( <code>\catcodetable0</code> is never assigned). A new catcode table will be populated with exactly those values assigned by IniTeX (as described in the LuaTeX manual).
<code>\newluafunction</code>	<code>\newluafunction{&lt;function&gt;}</code> Defines a named <code>\luafunction</code> , indexed from 1. (Lua indexes tables from 1 so <code>\luafunction0</code> is not available).
<code>\newwhatsit</code>	<code>\newwhatsit{&lt;whatsit&gt;}</code> Defines a custom <code>\whatsit</code> , indexed from 1.
<code>\newluabytecode</code>	<code>\newluabytecode{&lt;bytecode&gt;}</code> Allocates a number for Lua bytecode register, indexed from 1.
<code>\newluachunkname</code>	<code>newluachunkname{&lt;chunkname&gt;}</code> Allocates a number for Lua chunk register, indexed from 1. Also enters the name of the register (without backslash) into the <code>lua.name</code> table to be used in stack traces.

<code>\catcodetable@initex</code>	Predefined category code tables with the obvious assignments. Note that the
<code>\catcodetable@string</code>	<code>latex</code> and <code>atletter</code> tables set the full Unicode range to the codes predefined by
<code>\catcodetable@latex</code>	the kernel.
<code>\catcodetable@attribute</code>	<code>\setattribute{⟨attribute⟩}{⟨value⟩}</code>
<code>\unsetattribute</code>	<code>\unsetattribute{⟨attribute⟩}</code>

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 T<sub>E</sub>X interface

The `luatex` interface may be used with plain T<sub>E</sub>X using `\input{luatex}`. This inputs `luatex.tex` which inputs `etex.src` (or `etex.sty` if used with L<sup>A</sup>T<sub>E</sub>X) if it is not already input, and then defines some internal commands to allow the `luatex` interface to be defined.

The `luatexbase` package interface may also be used in plain T<sub>E</sub>X, as before, by inputting the package `\input luatexbase.sty`. The new version of `luatexbase` is based on this `luatex` code but implements a compatibility layer providing the interface of the original package.

## 4 Lua functionality

### 4.1 Allocators in Lua

<code>new_attribute</code>	<code>luatexbase.new_attribute(⟨attribute⟩)</code> Returns an allocation number for the <code>⟨attribute⟩</code> , indexed from 1. The attribute will be initialised with the marker value <code>-0xFFFFFFFF</code> ('unset'). The attribute allocation sequence is shared with the T <sub>E</sub> X code but this function does <i>not</i> define a token using <code>\attributedef</code> . The attribute name is recorded in the <code>attributes</code> table. A metatable is provided so that the table syntax can be used consistently for attributes declared in T <sub>E</sub> X or Lua.
<code>new_whatsit</code>	<code>luatexbase.new_whatsit(⟨whatsit⟩)</code> Returns an allocation number for the custom <code>⟨whatsit⟩</code> , indexed from 1.
<code>new_bytecode</code>	<code>luatexbase.new_bytecode(⟨bytecode⟩)</code> Returns an allocation number for a bytecode register, indexed from 1. The optional <code>⟨name⟩</code> argument is just used for logging.
<code>new_chunkname</code>	<code>luatexbase.new_chunkname(⟨chunkname⟩)</code> Returns an allocation number for a Lua chunk name for use with <code>\directlua</code> and <code>\latelua</code> , indexed from 1. The number is returned and also <code>⟨name⟩</code> argument is added to the <code>lua.name</code> array at that index.
<code>new_luafunction</code>	<code>luatexbase.new_luafunction(⟨functionname⟩)</code> Returns an allocation number for a lua function for use with <code>\luafunction</code> , <code>\lateluafunction</code> , and <code>\luadef</code> , indexed from 1. The optional <code>⟨functionname⟩</code> argument is just used for logging.

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

```

\local\attribute_count_name=\ "attributetracker"
\require("lualatex")

```

would use a T<sub>E</sub>X \count (\countdef'd token) called `attributetracker` in place of “e@alloc@attribute@count”.

## 4.2 Lua access to T<sub>E</sub>X register numbers

`registernumber` `luatexbase.registernumber(<name>)`

Sometimes (notably in the case of Lua attributes) it is necessary to access a register *by number* that has been allocated by T<sub>E</sub>X. This package provides a function to look up the relevant number using LuaT<sub>E</sub>X's internal tables. After for example `\newattribute\myattrib`, `\myattrib` would be defined by (say) `\myattrib=\attribute15`. `luatexbase.registernumber("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}

\attributedef\myattr=12
\myattr=200
\test{myattr}

```

If the demonstration code is processed with LuaL<sup>A</sup>T<sub>E</sub>X then the following would be produced in the log and terminal output.

```

undefinedrubbish:\relax
bad_input
space:\macro:->
bad_input
hbox:\hbox

```

```

uuuuuu bad_input
_u@MM:_\mathchar"4E20
uuuuuu 20000
_u@tempdima:_\dimen14
uuuuuu 14
_u@tempdimb:_\dimen15
uuuuuu 15
_ustrutbox:_\char"B
uuuuuu 11
_usixt@n:_\char"10
uuuuuu 16
_myattr:_\attribute12
uuuuuu 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X'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 callback 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.

<code>disable_callback</code>	<code>luatexbase.disable_callback(&lt;callback&gt;)</code> Sets the <code>&lt;callback&gt;</code> to <code>false</code> as described in the LuaTeX manual for the underlying <code>callback.register</code> built-in. Callbacks will only be set to <code>false</code> (and thus be skipped entirely) if there are no functions registered using the callback.
<code>callback_descriptions</code>	A list of the descriptions of functions registered to the specified callback is returned. <code>{}</code> is returned if there are no functions registered.
<code>create_callback</code>	<code>luatexbase.create_callback(&lt;name&gt;,metatype,&lt;default&gt;)</code> Defines a user defined callback. The last argument is a default function or <code>false</code> .
<code>call_callback</code>	<code>luatexbase.call_callback(&lt;name&gt;,...)</code> 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 ltuatex support *}
8   \wlog{*****}
9   \expandafter\endinput
10 \fi

```

### 5.2 Older L<sup>A</sup>T<sub>E</sub>X/Plain T<sub>E</sub>X setup

```

11 <*tex>

```

Older L<sup>A</sup>T<sub>E</sub>X formats don’t have the primitives with ‘native’ names: sort that out. If they already exist this will still be safe.

```

12 \directlua{tex.enableprimitives("",tex.extraprimitives("luatex"))}
13 \ifx\@alloc\@undefined

```

In pre-2014 L<sup>A</sup>T<sub>E</sub>X, or plain T<sub>E</sub>X, load `etex.{sty,src}`.

```

14 \ifx\documentclass\@undefined
15   \ifx\loccount\@undefined
16     \input{etex.src}%
17   \fi
18   \catcode'\@=11 %
19   \outer\expandafter\def\csname newfam\endcsname
20     {\alloc@8\fam\chardef\et@xmaxfam}
21 \else
22   \RequirePackage{etex}
23   \expandafter\def\csname newfam\endcsname
24     {\alloc@8\fam\chardef\et@xmaxfam}

```

```

25 \expandafter\let\expandafter\new@mathgroup\csname newfam\endcsname
26 \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 Lua<sub>T</sub><sub>E</sub>X-specific code, but does not define the correct range for Lua<sub>T</sub><sub>E</sub>X.

2015-07-13 higher range in luatex.

```

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

```

luatex/xetex also allow more math fam.

```

28 \edef \et@xmaxfam {\ifx\Umathchar\@undefined\sixt@@n\else\@cclvi\fi}

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

```

and 256 or 16 fam. (Done above due to plain/L<sub>A</sub>T<sub>E</sub>X differences in l<sub>T</sub>uatex.)

```

36 % \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 of `luatex.sty` to leave room for inserts not really needed for luatex but possibly most compatible with existing use.

```

37 \expandafter\let\csname newcount\expandafter\expandafter\endcsname
38 \csname globcount\endcsname
39 \expandafter\let\csname newdimen\expandafter\expandafter\endcsname
40 \csname globdimen\endcsname
41 \expandafter\let\csname newskip\expandafter\expandafter\endcsname
42 \csname globskip\endcsname
43 \expandafter\let\csname newbox\expandafter\expandafter\endcsname
44 \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.

```

45 \chardef\e@alloc@top=65535
46 \let\e@alloc\chardef\chardef

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

53 \gdef\e@ch@ck#1#2#3#4{%
54   \ifnum#1<#2\else
55     \ifnum#1=#2\relax
56     #1\@cclvi

```

```

57     \ifx\count#4\advance#1 10 \fi
58     \fi
59     \ifnum#1<#3\relax
60     \else
61     \errmessage{No room for a new \string#4}%
62     \fi
63 \fi}%

```

Two simple L<sup>A</sup>T<sub>E</sub>X macros used in `lAtex.sty`.

```

64 \long\def\@gobble#1{}
65 \long\def\@firstofone#1{#1}

```

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

```

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

```

End of conditional setup for plain T<sub>E</sub>X / old L<sup>A</sup>T<sub>E</sub>X.

```

72 \fi
73 \</tex>

```

### 5.3 Attributes

`\newattribute` As is generally the case for the LuaT<sub>E</sub>X registers we start here from 1. Notably, some code assumes that `\attribute0` is never used so this is important in this case.

```

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

```

`\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 LuaT<sub>E</sub>X for everything else. At the end of allocation there needs to be an initialisation step. Table 0 is already taken (it's the global one for current use) so the allocation starts at 1.

```

84 \ifx\@alloc@ccodetable@count\@undefined
85 \countdef\@alloc@ccodetable@count=259
86 \fi
87 \def\newcatcodetable#1{%
88 \e@alloc\catcodetable\chardef

```



```

89   \e@alloc@ccodetable@count\m@ne{"8000}#1%
90   \initcatcodetable\allocationnumber
91 }
92 \e@alloc@ccodetable@count=\z@

```

`\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
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 \fi
176 \def\newluafunction{%
177   \e@alloc@luafunction\e@alloc@chardef
178   \e@alloc@luafunction@count\m@ne\e@alloc@top
179 }
180 \e@alloc@luafunction@count=\z@

```

## 5.6 Custom whatsits

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

```

181 \ifx\e@alloc@whatsit@count\@undefined

```

```

182 \countdef\e@alloc@whatsit@count=261
183 \fi
184 \def\newwhatsit#1{%
185   \e@alloc@whatsit\e@alloc@chardef
186   \e@alloc@whatsit@count\m@ne\e@alloc@top#1%
187 }
188 \e@alloc@whatsit@count=\z@

```

## 5.7 Lua bytecode registers

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

```

189 \ifx\e@alloc@bytecode@count\@undefined
190 \countdef\e@alloc@bytecode@count=262
191 \fi
192 \def\newluabytcode#1{%
193   \e@alloc@luabytcode\e@alloc@chardef
194   \e@alloc@bytecode@count\m@ne\e@alloc@top#1%
195 }
196 \e@alloc@bytecode@count=\z@

```

## 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.

```

197 \ifx\e@alloc@luachunk@count\@undefined
198 \countdef\e@alloc@luachunk@count=263
199 \fi
200 \def\newluachunkname#1{%
201   \e@alloc@luachunk\e@alloc@chardef
202   \e@alloc@luachunk@count\m@ne\e@alloc@top#1%
203   {\escapechar\m@ne
204    \directlua{lua.name[\the\allocationnumber]="\string#1"}}%
205 }
206 \e@alloc@luachunk@count=\z@

```

## 5.9 Lua loader

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.

```

207 (2ekernel) \everyjob\expandafter{%
208 (2ekernel) \the\everyjob
209 \begingroup
210 \attributedef\attributezero=0 %
211 \chardef \charzero =0 %

```

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

```

212 \countdef \CountZero =0 %
213 \dimendef \dimenzero =0 %
214 \mathchardef \mathcharzero =0 %
215 \muskipdef \muskipzero =0 %
216 \skipdef \skipzero =0 %

```

```

217 \toksdef \tokszero =0 %
218 \directlua{require("lualatex")}
219 \endgroup
220 (2ekernel)}
221 (latexrelease)\EndIncludeInRelease

222 (latexrelease)\IncludeInRelease{0000/00/00}
223 (latexrelease) {\newluafunction}{LuaTeX}%
224 (latexrelease)\let\@alloc@attribute@count\@undefined
225 (latexrelease)\let\newattribute\@undefined
226 (latexrelease)\let\setattribute\@undefined
227 (latexrelease)\let\unsetattribute\@undefined
228 (latexrelease)\let\@alloc@ccodetable@count\@undefined
229 (latexrelease)\let\newcatcodetable\@undefined
230 (latexrelease)\let\catcodetable@initex\@undefined
231 (latexrelease)\let\catcodetable@string\@undefined
232 (latexrelease)\let\catcodetable@latex\@undefined
233 (latexrelease)\let\catcodetable@atletter\@undefined
234 (latexrelease)\let\@alloc@luafunction@count\@undefined
235 (latexrelease)\let\newluafunction\@undefined
236 (latexrelease)\let\@alloc@luafunction@count\@undefined
237 (latexrelease)\let\newwhatsit\@undefined
238 (latexrelease)\let\@alloc@whatsit@count\@undefined
239 (latexrelease)\let\newluabytecode\@undefined
240 (latexrelease)\let\@alloc@bytecode@count\@undefined
241 (latexrelease)\let\newluachunkname\@undefined
242 (latexrelease)\let\@alloc@luachunk@count\@undefined
243 (latexrelease)\directlua{luatexbase.uninstall()}
244 (latexrelease)\EndIncludeInRelease

```

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

```

245 (latexrelease)\IncludeInRelease{2017/01/01}%
246 (latexrelease) {\fontencoding}{TU in everyjob}%
247 (latexrelease)\fontencoding{TU}\let\encodingdefault\f@encoding
248 (latexrelease)\ifx\directlua\@undefined\else
249 (2ekernel)\everyjob\expandafter{%
250 (2ekernel) \the\everyjob
251 (*2ekernel, latexrelease)
252 \directlua{%
253 if xpcall(function ()%
254 require('luaotfload-main')%
255 end, texio.write_nl) then %
256 local _void = luaotfload.main ()%
257 else %
258 texio.write_nl('Error in luaotfload: reverting to OT1')%
259 tex.print('\string\def\string\encodingdefault{OT1}')%
260 end %
261 }%
262 \let\f@encoding\encodingdefault
263 \expandafter\let\csname ver@luaotfload.sty\endcsname\fmtversion
264 (/2ekernel, latexrelease)
265 (latexrelease)\fi
266 (2ekernel) }
267 (latexrelease)\EndIncludeInRelease
268 (latexrelease)\IncludeInRelease{0000/00/00}%

```

```

269 <latexrelease>                {\fontencoding}{TU in everyjob}%
270 <latexrelease>\fontencoding{OT1}\let\encodingdefault\f@encoding
271 <latexrelease>\EndIncludeInRelease

272 <2ekernel | latexrelease>\fi
273 </2ekernel | tex | latexrelease>

```

## 5.10 Lua module preliminaries

```

274 <*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.

```

275 luatexbase      = luatexbase or { }
276 local luatexbase = luatexbase

```

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

```

277 local string_gsub      = string.gsub
278 local tex_count        = tex.count
279 local tex_setattribute = tex.setattribute
280 local tex_setcount     = tex.setcount
281 local texio_write_nl   = texio.write_nl

282 local luatexbase_warning
283 local luatexbase_error

```

## 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.

```

284 local modules = modules or { }

```

**provides\_module** Local function to write to the log.

```

285 local function luatexbase_log(text)
286   texio_write_nl("log", text)
287 end

```

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

```

288 local function provides_module(info)
289   if not (info and info.name) then
290     luatexbase_error("Missing module name for provides_module")
291   end
292   local function spaced(text)
293     return text and (" " .. text) or ""
294   end
295   luatexbase_log(
296     "Lua module: " .. info.name
297     .. spaced(info.date)
298     .. spaced(info.version)
299     .. spaced(info.description)

```

```

300 )
301 modules[info.name] = info
302 end
303 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  $\text{\TeX}$ . For errors we have to make some changes. Here we give the text of the error in the  $\text{\LaTeX}$  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.

```

304 local function msg_format(mod, msg_type, text)
305   local leader = ""
306   local cont
307   local first_head
308   if mod == "LaTeX" then
309     cont = string_gsub(leader, ".", " ")
310     first_head = leader .. "LaTeX: "
311   else
312     first_head = leader .. "Module " .. msg_type
313     cont = "(" .. mod .. ")"
314     .. string_gsub(first_head, ".", " ")
315     first_head = leader .. "Module " .. mod .. " " .. msg_type .. ":"
316   end
317   if msg_type == "Error" then
318     first_head = "\n" .. first_head
319   end
320   if string.sub(text,-1) ~= "\n" then
321     text = text .. " "
322   end
323   return first_head .. " "
324   .. string_gsub(
325     text
326     .. "on input line "
327     .. tex.inputlineno, "\n", "\n" .. cont .. " "
328   )
329   .. "\n"
330 end

```

```

module_info Write messages.
module_warning 331 local function module_info(mod, text)
module_error 332   texio_write_nl("log", msg_format(mod, "Info", text))
333 end
334 luatexbase.module_info = module_info
335 local function module_warning(mod, text)
336   texio_write_nl("term and log", msg_format(mod, "Warning", text))
337 end
338 luatexbase.module_warning = module_warning
339 local function module_error(mod, text)
340   error(msg_format(mod, "Error", text))

```

```

341 end
342 luatexbase.module_error = module_error

```

Dedicated versions for the rest of the code here.

```

343 function luatexbase_warning(text)
344   module_warning("luatexbase", text)
345 end
346 function luatexbase_error(text)
347   module_error("luatexbase", text)
348 end

```

## 5.12 Accessing register numbers from Lua

Collect up the data from the T<sub>E</sub>X level into a Lua table: from version 0.80, LuaT<sub>E</sub>X makes that easy.

```

349 local luaregisterbasetable = { }
350 local registermap = {
351   attributezero = "assign_attr"    ,
352   charzero      = "char_given"    ,
353   CountZero     = "assign_int"    ,
354   dimenzero     = "assign_dimen"  ,
355   mathcharzero  = "math_given"    ,
356   muskipzero    = "assign_mu_skip",
357   skipzero      = "assign_skip"   ,
358   tokszero      = "assign_toks"   ,
359 }
360 local createtoken
361 if tex.luatexversion > 81 then
362   createtoken = token.create
363 elseif tex.luatexversion > 79 then
364   createtoken = newtoken.create
365 end
366 local hashtokens = tex.hashtokens()
367 local luatexversion = tex.luatexversion
368 for i,j in pairs (registermap) do
369   if luatexversion < 80 then
370     luaregisterbasetable[hashtokens[i][1]] =
371       hashtokens[i][2]
372   else
373     luaregisterbasetable[j] = createtoken(i).mode
374   end
375 end

```

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

```

376 local registernumber
377 if luatexversion < 80 then
378   function registernumber(name)
379     local nt = hashtokens[name]
380     if(nt and luaregisterbasetable[nt[1]]) then
381       return nt[2] - luaregisterbasetable[nt[1]]

```

```

382     else
383         return false
384     end
385 end
386 else
387     function registernumber(name)
388         local nt = createtoken(name)
389         if(luaregisterbasetable[nt.cmdname]) then
390             return nt.mode - luaregisterbasetable[nt.cmdname]
391         else
392             return false
393         end
394     end
395 end
396 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.

```

397 local attributes=setmetatable(
398 {},
399 {
400 __index = function(t,key)
401 return registernumber(key) or nil
402 end}
403 )
404 luatexbase.attributes = attributes
405 local attribute_count_name =
406         attribute_count_name or "e@alloc@attribute@count"
407 local function new_attribute(name)
408     tex_setcount("global", attribute_count_name,
409                 tex_count[attribute_count_name] + 1)
410     if tex_count[attribute_count_name] > 65534 then
411         luatexbase_error("No room for a new \\attribute")
412     end
413     attributes[name]= tex_count[attribute_count_name]
414     luatexbase_log("Lua-only attribute " .. name .. " = " ..
415                 tex_count[attribute_count_name])
416     return tex_count[attribute_count_name]
417 end
418 luatexbase.new_attribute = new_attribute

```

### 5.14 Custom whatsit allocation

**new\_whatsit** Much the same as for attribute allocation in Lua.

```

419 local whatsit_count_name = whatsit_count_name or "e@alloc@whatsit@count"
420 local function new_whatsit(name)
421     tex_setcount("global", whatsit_count_name,
422                 tex_count[whatsit_count_name] + 1)
423     if tex_count[whatsit_count_name] > 65534 then
424         luatexbase_error("No room for a new custom whatsit")
425     end

```



```

426 luatexbase_log("Custom whatsit " .. (name or "") .. " = " ..
427               tex_count[whatsit_count_name])
428 return tex_count[whatsit_count_name]
429 end
430 luatexbase.new_whatsit = new_whatsit

```

## 5.15 Bytecode register allocation

**new\_bytecode** Much the same as for attribute allocation in Lua. The optional  $\langle name \rangle$  argument is used in the log if given.

```

431 local bytecode_count_name =
432       bytecode_count_name or "e@alloc@bytecode@count"
433 local function new_bytecode(name)
434   tex_setcount("global", bytecode_count_name,
435               tex_count[bytecode_count_name] + 1)
436   if tex_count[bytecode_count_name] > 65534 then
437     luatexbase_error("No room for a new bytecode register")
438   end
439   luatexbase_log("Lua bytecode " .. (name or "") .. " = " ..
440               tex_count[bytecode_count_name])
441   return tex_count[bytecode_count_name]
442 end
443 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.

```

444 local chunkname_count_name =
445       chunkname_count_name or "e@alloc@luachunk@count"
446 local function new_chunkname(name)
447   tex_setcount("global", chunkname_count_name,
448               tex_count[chunkname_count_name] + 1)
449   local chunkname_count = tex_count[chunkname_count_name]
450   chunkname_count = chunkname_count + 1
451   if chunkname_count > 65534 then
452     luatexbase_error("No room for a new chunkname")
453   end
454   lua.name[chunkname_count]=name
455   luatexbase_log("Lua chunkname " .. (name or "") .. " = " ..
456               chunkname_count .. "\n")
457   return chunkname_count
458 end
459 luatexbase.new_chunkname = new_chunkname

```

## 5.17 Lua function allocation

**new\_luafunction** Much the same as for attribute allocation in Lua. The optional  $\langle name \rangle$  argument is used in the log if given.

```

460 local luafunction_count_name =
461       luafunction_count_name or "e@alloc@luafunction@count"
462 local function new_luafunction(name)
463   tex_setcount("global", luafunction_count_name,

```

```

464             tex_count[luafunction_count_name] + 1)
465   if tex_count[luafunction_count_name] > 65534 then
466     luatexbase_error("No room for a new luafunction register")
467   end
468   luatexbase_log("Lua function " .. (name or "") .. " = " ..
469                 tex_count[luafunction_count_name])
470   return tex_count[luafunction_count_name]
471 end
472 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.

```

473 local callbacklist = callbacklist or { }

```

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

```

474 local list, data, exclusive, simple = 1, 2, 3, 4
475 local types = {
476   list      = list,
477   data      = data,
478   exclusive = exclusive,
479   simple    = simple,
480 }

```

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

```

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

```

481 local callbacktypes = callbacktypes or {

```

Section 8.2: file discovery callbacks.

```

482   find_read_file      = exclusive,
483   find_write_file     = exclusive,
484   find_font_file      = data,
485   find_output_file    = data,

```

```

486 find_format_file = data,
487 find_vf_file      = data,
488 find_map_file      = data,
489 find_enc_file      = data,
490 find_pk_file       = data,
491 find_data_file     = data,
492 find_opentype_file = data,
493 find_truetype_file = data,
494 find_type1_file    = data,
495 find_image_file    = data,

496 open_read_file     = exclusive,
497 read_font_file     = exclusive,
498 read_vf_file       = exclusive,
499 read_map_file      = exclusive,
500 read_enc_file      = exclusive,
501 read_pk_file       = exclusive,
502 read_data_file     = exclusive,
503 read_truetype_file = exclusive,
504 read_type1_file    = exclusive,
505 read_opentype_file = exclusive,

```

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

```

506 find_cidmap_file = data,
507 read_cidmap_file = exclusive,

```

Section 8.3: data processing callbacks.

```

508 process_input_buffer = data,
509 process_output_buffer = data,
510 process_jobname      = data,

```

Section 8.4: node list processing callbacks.

```

511 contribute_filter    = simple,
512 buildpage_filter     = simple,
513 build_page_insert    = exclusive,
514 pre_linebreak_filter = list,
515 linebreak_filter     = exclusive,
516 append_to_vlist_filter = exclusive,
517 post_linebreak_filter = list,
518 hpack_filter         = list,
519 vpack_filter         = list,
520 hpack_quality        = list,
521 vpack_quality        = list,
522 pre_output_filter    = list,
523 process_rule         = exclusive,
524 hyphenate            = simple,
525 ligaturing           = simple,
526 kerning              = simple,
527 insert_local_par     = simple,
528 mlist_to_hlist       = exclusive,
529 new_graf             = simple,

```

Section 8.5: information reporting callbacks.

```

530 pre_dump             = simple,
531 start_run            = simple,

```

```

532 stop_run          = simple,
533 start_page_number = simple,
534 stop_page_number  = simple,
535 show_error_hook   = simple,
536 show_warning_message = simple,
537 show_error_message = simple,
538 show_lua_error_hook = simple,
539 start_file         = simple,
540 stop_file          = simple,
541 call_edit          = simple,
542 finish_synctex     = simple,
543 wrapup_run         = simple,

```

Section 8.6: PDF-related callbacks.

```

544 finish_pdffile     = data,
545 finish_pdfpage      = data,
546 page_objnum_provider = data,
547 process_pdf_image_content = data,

```

Section 8.7: font-related callbacks.

```

548 define_font          = exclusive,
549 glyph_not_found      = exclusive,
550 glyph_stream_provider = exclusive,
551 make_extensible       = exclusive,
552 font_descriptor_objnum_provider = exclusive,

```

```

553 }

```

```

554 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.

```

555 local callback_register = callback_register or callback.register
556 function callback.register()
557   luatexbase_error("Attempt to use callback.register() directly\n")
558 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.

**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.

```
559 local function data_handler(name)
560   return function(data, ...)
561     for _,i in ipairs(callbacklist[name]) do
562       data = i.func(data,...)
563     end
564     return data
565   end
566 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.

```
567 local function exclusive_handler(name)
568   return function(...)
569     return callbacklist[name][1].func(...)
570   end
571 end
```

Handler for *list* callbacks.

```
572 local function list_handler(name)
573   return function(head, ...)
574     local ret
575     local alltrue = true
576     for _,i in ipairs(callbacklist[name]) do
577       ret = i.func(head, ...)
578       if ret == false then
579         luatexbase_warning(
580           "Function '" .. i.description .. "' returned false\n"
581           .. "in callback '" .. name .. "'")
582       )
583       break
584     end
585     if ret ~= true then
586       alltrue = false
587       head = ret
588     end
589   end
590   return alltrue and true or head
```

```

591 end
592 end
Handler for simple callbacks.
593 local function simple_handler(name)
594   return function(...)
595     for _,i in ipairs(callbacklist[name]) do
596       i.func(...)
597     end
598   end
599 end

```

Keep a handlers table for indexed access.

```

600 local handlers = {
601   [data]      = data_handler,
602   [exclusive] = exclusive_handler,
603   [list]      = list_handler,
604   [simple]     = simple_handler,
605 }

```

### 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.

```

606 local user_callbacks_defaults = { }

```

`create_callback` The allocator itself.

```

607 local function create_callback(name, ctype, default)
608   if not name or name == ""
609   or not ctype or ctype == ""
610   then
611     luatexbase_error("Unable to create callback:\n" ..
612                       "valid callback name and type required")
613   end
614   if callbacktypes[name] then
615     luatexbase_error("Unable to create callback '" .. name ..
616                       "':\ncallback is already defined")
617   end
618   if default ~= false and type (default) ~= "function" then
619     luatexbase_error("Unable to create callback '" .. name ..
620                       ":\ndefault is not a function")
621   end
622   user_callbacks_defaults[name] = default
623   callbacktypes[name] = types[ctype]
624 end
625 luatexbase.create_callback = create_callback

```

`call_callback` Call a user defined callback. First check arguments.

```

626 local function call_callback(name,...)
627   if not name or name == "" then
628     luatexbase_error("Unable to create callback:\n" ..
629                       "valid callback name required")
630   end

```

```

631 if user_callbacks_defaults[name] == nil then
632     luatexbase_error("Unable to call callback '" .. name
633         .. "':\nunknown or empty")
634 end
635 local l = callbacklist[name]
636 local f
637 if not l then
638     f = user_callbacks_defaults[name]
639     if l == false then
640         return nil
641     end
642 else
643     f = handlers[callbacktypes[name]](name)
644 end
645 return f(...)
646 end
647 luatexbase.call_callback=call_callback

```

**add\_to\_callback** Add a function to a callback. First check arguments.

```

648 local function add_to_callback(name, func, description)
649     if not name or name == "" then
650         luatexbase_error("Unable to register callback:\n" ..
651             "valid callback name required")
652     end
653     if not callbacktypes[name] or
654         type(func) ~= "function" or
655         not description or
656         description == "" then
657         luatexbase_error(
658             "Unable to register callback.\n\n"
659             .. "Correct usage:\n"
660             .. "add_to_callback(<callback>, <function>, <description>)"
661         )
662     end

```

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

```

663     local l = callbacklist[name]
664     if l == nil then
665         l = { }
666         callbacklist[name] = l

```

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

```

667         if user_callbacks_defaults[name] == nil then
668             callback_register(name, handlers[callbacktypes[name]](name))
669         end
670     end

```

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

```

671     local f = {
672         func = func,
673         description = description,
674     }
675     local priority = #l + 1

```

```

676 if callbacktypes[name] == exclusive then
677   if #l == 1 then
678     luatexbase_error(
679       "Cannot add second callback to exclusive function\n'" ..
680       name .. "'")
681   end
682 end
683 table.insert(l, priority, f)

```

Keep user informed.

```

684 luatexbase_log(
685   "Inserting '" .. description .. "' at position "
686   .. priority .. " in '" .. name .. "'."
687 )
688 end
689 luatexbase.add_to_callback = add_to_callback

```

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

```

690 local function remove_from_callback(name, description)
691   if not name or name == "" then
692     luatexbase_error("Unable to remove function from callback:\n" ..
693       "valid callback name required")
694   end
695   if not callbacktypes[name] or
696     not description or
697     description == "" then
698     luatexbase_error(
699       "Unable to remove function from callback.\n\n"
700       .. "Correct usage:\n"
701       .. "remove_from_callback(<callback>, <description>)"
702     )
703   end
704   local l = callbacklist[name]
705   if not l then
706     luatexbase_error(
707       "No callback list for '" .. name .. "'\n")
708   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.

```

709   local index = false
710   for i,j in ipairs(l) do
711     if j.description == description then
712       index = i
713       break
714     end
715   end
716   if not index then
717     luatexbase_error(
718       "No callback '" .. description .. "' registered for '" ..
719       name .. "'\n")
720   end
721   local cb = l[index]
722   table.remove(l, index)
723   luatexbase_log(

```



```

724     "Removing '" .. description .. "' from '" .. name .. "'."
725 )
726 if #l == 0 then
727     callbacklist[name] = nil
728     callback_register(name, nil)
729 end
730 return cb.func,cb.description
731 end
732 luatexbase.remove_from_callback = remove_from_callback

```

**in\_callback** Look for a function description in a callback.

```

733 local function in_callback(name, description)
734     if not name
735         or name == ""
736         or not callbacklist[name]
737         or not callbacktypes[name]
738         or not description then
739         return false
740     end
741     for _, i in pairs(callbacklist[name]) do
742         if i.description == description then
743             return true
744         end
745     end
746     return false
747 end
748 luatexbase.in_callback = in_callback

```

**disable\_callback** As we subvert the engine interface we need to provide a way to access this functionality.

```

749 local function disable_callback(name)
750     if(callbacklist[name] == nil) then
751         callback_register(name, false)
752     else
753         luatexbase_error("Callback list for '" .. name .. "' not empty")
754     end
755 end
756 luatexbase.disable_callback = disable_callback

```

**callback\_descriptions** List the descriptions of functions registered for the given callback.

```

757 local function callback_descriptions (name)
758     local d = {}
759     if not name
760         or name == ""
761         or not callbacklist[name]
762         or not callbacktypes[name]
763     then
764         return d
765     else
766         for k, i in pairs(callbacklist[name]) do
767             d[k] = i.description
768         end
769     end
770     return d

```

```

771 end
772 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!

773 local function uninstall()
774   module_info(
775     "luatexbase",
776     "Uninstalling kernel luatexbase code"
777   )
778   callback.register = callback_register
779   luatexbase = nil
780 end
781 luatexbase.uninstall = uninstall

782  $\langle$ /lua $\rangle$ 
       Reset the catcode of @.
783  $\langle$ tex $\rangle$ \catcode'\@=\etatcatcode\relax

```