

# lualatex.dtx

## (LuaTeX-specific support)

David Carlisle and Joseph Wright\*

2021/10/15

## 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 . . . . .	17
5.15 Bytecode register allocation . . . . .	17
5.16 Lua chunk name allocation . . . . .	17
5.17 Lua function allocation . . . . .	18
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{⟨attribute⟩}</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{⟨catcodetable⟩}</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{⟨function⟩}</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{⟨whatsit⟩}</code> Defines a custom <code>\whatsit</code> , indexed from 1.
<code>\newluabytecode</code>	<code>\newluabytecode{⟨bytecode⟩}</code> Allocates a number for Lua bytecode register, indexed from 1.
<code>\newluachunkname</code>	<code>newluachunkname{⟨chunkname⟩}</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>\luaodef</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("ltnlua")

```

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.registernumber(<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 Lua $\TeX$ '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 Lua $\TeX$  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
sist@@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 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

```

Two simple L<sup>A</sup>T<sub>E</sub>X macros from `ltdfn.s.dtx` have to be defined here because `ltdfn.s.dtx` is not loaded yet when `ltluatex.dtx` is executed.

```

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

```

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

```

13 <*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.

```

14 \directlua{tex.enableprimitives("",tex.extraprimitives("luatex"))}
15 \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}`.

```

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\ccclvi\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 ltuatex.)

```

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\@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\@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 TEX / old LATEX.

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   \@alloc@attribute@count=\z@
77 \fi
78 \def\newattribute#1{%
79   \@alloc\attribute\attributedef
80   \@alloc@attribute@count\m@ne\@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 LuaT<sub>E</sub>X 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\@alloc@ccodetable@count\@undefined
85   \countdef\@alloc@ccodetable@count=259
86   \@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
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 }

```

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

```

## 5.7 Lua bytecode registers

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

```

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

```

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

```

## 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:

```

207 \def\now@and@everyjob#1{%
208   \everyjob\expandafter{\the\everyjob
209     #1%
210   }%
211   #1%
212 }

```

Load the Lua code at the start of every job. For the conversion of T<sub>E</sub>X 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.

```

213 \<2kernel>\now@and@everyjob{%
214   \begingroup

```

```

215 \attributedef\attributezero=0 %
216 \chardef \charzero =0 %

```

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

```

217 \countdef \CountZero =0 %
218 \dimendef \dimenzero =0 %
219 \mathchardef \mathcharzero =0 %
220 \muskipdef \muskipzero =0 %
221 \skipdef \skipzero =0 %
222 \toksdef \tokszero =0 %
223 \directlua{require("lualatex")}
224 \endgroup
225 (2ekernel)}
226 (latexrelease)\EndIncludeInRelease

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

```

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

```

250 (latexrelease)\IncludeInRelease{2017/01/01}%
251 (latexrelease) {\fontencoding}{TU in everyjob}%
252 (latexrelease)\fontencoding{TU}\let\encodingdefault\f@encoding
253 (latexrelease)\ifx\directlua\@undefined\else
254 (2ekernel)\everyjob\expandafter{%
255 (2ekernel) \the\everyjob
256 (*2ekernel, latexrelease)
257 \directlua{%
258 if xpcall(function ()%
259 require('luaotfload-main')%
260 end, texio.write_nl) then %
261 local _void = luaotfload.main ()%
262 else %
263 texio.write_nl('Error in luaotfload: reverting to OT1')%
264 tex.print('\string\def\string\encodingdefault{OT1}')%

```

```

265 end %
266 }%
267 \let\f@encoding\encodingdefault
268 \expandafter\let\csname ver@luaotfload.sty\endcsname\fmtversion
269 \<2ekernel, latexrelease>
270 \<latexrelease>\fi
271 \<2ekernel> }
272 \<latexrelease>\EndIncludeInRelease
273 \<latexrelease>\IncludeInRelease{0000/00/00}%
274 \<latexrelease> {\fontencoding}{TU in everyjob}%
275 \<latexrelease>\fontencoding{OT1}\let\encodingdefault\f@encoding
276 \<latexrelease>\EndIncludeInRelease
277 \<2ekernel | latexrelease>\fi
278 \</2ekernel | tex | latexrelease>

```

## 5.10 Lua module preliminaries

```

279 \*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.

```

280 luatexbase = luatexbase or { }
281 local luatexbase = luatexbase

```

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

```

282 local string_gsub = string.gsub
283 local tex_count = tex.count
284 local tex_setattribute = tex.setattribute
285 local tex_setcount = tex.setcount
286 local texio_write_nl = texio.write_nl
287 local flush_list = node.flush_list
288 local luatexbase_warning
289 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.

```

290 local modules = modules or { }

```

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

```

291 local function luatexbase_log(text)
292   texio_write_nl("log", text)
293 end

```

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

```

294 local function provides_module(info)
295   if not (info and info.name) then

```

```

296     luatexbase_error("Missing module name for provides_module")
297 end
298 local function spaced(text)
299     return text and (" " .. text) or ""
300 end
301 luatexbase_log(
302     "Lua module: " .. info.name
303     .. spaced(info.date)
304     .. spaced(info.version)
305     .. spaced(info.description)
306 )
307 modules[info.name] = info
308 end
309 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.

```

310 local function msg_format(mod, msg_type, text)
311     local leader = ""
312     local cont
313     local first_head
314     if mod == "LaTeX" then
315         cont = string_gsub(leader, ".", " ")
316         first_head = leader .. "LaTeX: "
317     else
318         first_head = leader .. "Module " .. msg_type
319         cont = "(" .. mod .. ")"
320         .. string_gsub(first_head, ".", " ")
321         first_head = leader .. "Module " .. mod .. " " .. msg_type .. ":"
322     end
323     if msg_type == "Error" then
324         first_head = "\n" .. first_head
325     end
326     if string.sub(text,-1) ~= "\n" then
327         text = text .. " "
328     end
329     return first_head .. " "
330     .. string_gsub(
331         text
332         .. "on input line "
333         .. tex.inputlineno, "\n", "\n" .. cont .. " "
334     )
335     .. "\n"
336 end

```

```

module_info Write messages.
module_warning
module_error

```

```

337 local function module_info(mod, text)
338   texio_write_nl("log", msg_format(mod, "Info", text))
339 end
340 luatexbase.module_info = module_info
341 local function module_warning(mod, text)
342   texio_write_nl("term and log", msg_format(mod, "Warning", text))
343 end
344 luatexbase.module_warning = module_warning
345 local function module_error(mod, text)
346   error(msg_format(mod, "Error", text))
347 end
348 luatexbase.module_error = module_error

```

Dedicated versions for the rest of the code here.

```

349 function luatexbase_warning(text)
350   module_warning("luatexbase", text)
351 end
352 function luatexbase_error(text)
353   module_error("luatexbase", text)
354 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.

```

355 local luaregisterbasetable = { }
356 local registermap = {
357   attributezero = "assign_attr"    ,
358   charzero      = "char_given"     ,
359   CountZero     = "assign_int"     ,
360   dimenzero     = "assign_dimen"   ,
361   mathcharzero  = "math_given"     ,
362   muskipzero    = "assign_mu_skip" ,
363   skipzero      = "assign_skip"    ,
364   tokszero      = "assign_toks"    ,
365 }
366 local createtoken
367 if tex.luatexversion > 81 then
368   createtoken = token.create
369 elseif tex.luatexversion > 79 then
370   createtoken = newtoken.create
371 end
372 local hashtokens = tex.hashtokens()
373 local luatexversion = tex.luatexversion
374 for i,j in pairs (registermap) do
375   if luatexversion < 80 then
376     luaregisterbasetable[hashtokens[i][1]] =
377       hashtokens[i][2]
378   else
379     luaregisterbasetable[j] = createtoken(i).mode
380   end
381 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.

```

382 local registernumber
383 if luatexversion < 80 then
384   function registernumber(name)
385     local nt = hashtokens[name]
386     if(nt and luaregisterbasetable[nt[1]]) then
387       return nt[2] - luaregisterbasetable[nt[1]]
388     else
389       return false
390     end
391   end
392 else
393   function registernumber(name)
394     local nt = createtoken(name)
395     if(luaregisterbasetable[nt.cmdname]) then
396       return nt.mode - luaregisterbasetable[nt.cmdname]
397     else
398       return false
399     end
400   end
401 end
402 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.

```

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

```



## 5.14 Custom whatsit allocation

`new_whatsit` Much the same as for attribute allocation in Lua.

```
425 local whatsit_count_name = whatsit_count_name or "e@alloc@whatsit@count"
426 local function new_whatsit(name)
427   tex_setcount("global", whatsit_count_name,
428               tex_count[whatsit_count_name] + 1)
429   if tex_count[whatsit_count_name] > 65534 then
430     luatexbase_error("No room for a new custom whatsit")
431   end
432   luatexbase_log("Custom whatsit " .. (name or "") .. " = " ..
433               tex_count[whatsit_count_name])
434   return tex_count[whatsit_count_name]
435 end
436 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.

```
437 local bytecode_count_name =
438     bytecode_count_name or "e@alloc@bytecode@count"
439 local function new_bytecode(name)
440   tex_setcount("global", bytecode_count_name,
441               tex_count[bytecode_count_name] + 1)
442   if tex_count[bytecode_count_name] > 65534 then
443     luatexbase_error("No room for a new bytecode register")
444   end
445   luatexbase_log("Lua bytecode " .. (name or "") .. " = " ..
446               tex_count[bytecode_count_name])
447   return tex_count[bytecode_count_name]
448 end
449 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.

```
450 local chunkname_count_name =
451     chunkname_count_name or "e@alloc@luachunk@count"
452 local function new_chunkname(name)
453   tex_setcount("global", chunkname_count_name,
454               tex_count[chunkname_count_name] + 1)
455   local chunkname_count = tex_count[chunkname_count_name]
456   chunkname_count = chunkname_count + 1
457   if chunkname_count > 65534 then
458     luatexbase_error("No room for a new chunkname")
459   end
460   lua.name[chunkname_count]=name
461   luatexbase_log("Lua chunkname " .. (name or "") .. " = " ..
462               chunkname_count .. "\n")
463   return chunkname_count
464 end
465 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.

```
466 local luafunction_count_name =
467     luafunction_count_name or "e@alloc@luafunction@count"
468 local function new_luafunction(name)
469     tex_setcount("global", luafunction_count_name,
470         tex_count[luafunction_count_name] + 1)
471     if tex_count[luafunction_count_name] > 65534 then
472         luatexbase_error("No room for a new luafunction register")
473     end
474     luatexbase_log("Lua function " .. (name or "") .. " = " ..
475         tex_count[luafunction_count_name])
476     return tex_count[luafunction_count_name]
477 end
478 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.

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

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

```
480 local list, data, exclusive, simple, reverselist = 1, 2, 3, 4, 5
481 local types = {
482     list      = list,
483     data      = data,
484     exclusive = exclusive,
485     simple    = simple,
486     reverselist = reverselist,
487 }
```

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.)

488 local callbacktypes = callbacktypes or {

Section 8.2: file discovery callbacks.

```

489 find_read_file      = exclusive,
490 find_write_file     = exclusive,
491 find_font_file      = data,
492 find_output_file    = data,
493 find_format_file    = data,
494 find_vf_file        = data,
495 find_map_file       = data,
496 find_enc_file       = data,
497 find_pk_file        = data,
498 find_data_file      = data,
499 find_opentype_file  = data,
500 find_truetype_file  = data,
501 find_type1_file     = data,
502 find_image_file     = data,

503 open_read_file      = exclusive,
504 read_font_file      = exclusive,
505 read_vf_file        = exclusive,
506 read_map_file       = exclusive,
507 read_enc_file       = exclusive,
508 read_pk_file        = exclusive,
509 read_data_file      = exclusive,
510 read_truetype_file  = exclusive,
511 read_type1_file     = exclusive,
512 read_opentype_file  = exclusive,

```

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

```

513 find_cidmap_file    = data,
514 read_cidmap_file    = exclusive,

```

Section 8.3: data processing callbacks.

```

515 process_input_buffer = data,
516 process_output_buffer = data,
517 process_jobname      = data,

```

Section 8.4: node list processing callbacks.

```

518 contribute_filter    = simple,
519 buildpage_filter     = simple,
520 build_page_insert    = exclusive,
521 pre_linebreak_filter = list,
522 linebreak_filter      = exclusive,
523 append_to_vlist_filter = exclusive,
524 post_linebreak_filter = reverselist,
525 hpack_filter          = list,
526 vpack_filter          = list,
527 hpack_quality         = list,
528 vpack_quality         = list,
529 pre_output_filter     = list,
530 process_rule          = exclusive,
531 hyphenate             = simple,

```

```

532 ligaturing          = simple,
533 kerning             = simple,
534 insert_local_par    = simple,
535 pre_mlist_to_hlist_filter = list,
536 mlist_to_hlist       = exclusive,
537 post_mlist_to_hlist_filter = reverselist,
538 new_graf             = exclusive,

```

Section 8.5: information reporting callbacks.

```

539 pre_dump            = simple,
540 start_run            = simple,
541 stop_run             = simple,
542 start_page_number    = simple,
543 stop_page_number     = simple,
544 show_error_hook      = simple,
545 show_warning_message = simple,
546 show_error_message   = simple,
547 show_lua_error_hook  = simple,
548 start_file           = simple,
549 stop_file            = simple,
550 call_edit            = simple,
551 finish_synctex       = simple,
552 wrapup_run           = simple,

```

Section 8.6: PDF-related callbacks.

```

553 finish_pdffile       = data,
554 finish_pdfpage       = data,
555 page_objnum_provider = data,
556 page_order_index     = data,
557 process_pdf_image_content = data,

```

Section 8.7: font-related callbacks.

```

558 define_font          = exclusive,
559 glyph_info           = exclusive,
560 glyph_not_found      = exclusive,
561 glyph_stream_provider = exclusive,
562 make_extensible      = exclusive,
563 font_descriptor_objnum_provider = exclusive,
564 input_level_string    = exclusive,
565 provide_charproc_data = exclusive,
566 }
567 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.

```

568 local callback_register = callback_register or callback.register
569 function callback.register()
570   luatexbase_error("Attempt to use callback.register() directly\n")
571 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.

```
572 local function data_handler(name)
573   return function(data, ...)
574     for _,i in ipairs(callbacklist[name]) do
575       data = i.func(data,...)
576     end
577     return data
578   end
579 end
```

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

```
580 local function data_handler_default(value)
581   return value
582 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.

```
583 local function exclusive_handler(name)
584   return function(...)
585     return callbacklist[name][1].func(...)
586   end
587 end
```

Handler for list callbacks.

```
588 local function list_handler(name)
589   return function(head, ...)
590     local ret
591     local alltrue = true
592     for _,i in ipairs(callbacklist[name]) do
593       ret = i.func(head, ...)
594       if ret == false then
595         luatexbase_warning(
596           "Function '" .. i.description .. "' returned false\n"
597           .. "in callback '" .. name .. "'")
598       )
599       return false
600     end
601     if ret ~= true then
602       alltrue = false
603       head = ret
604     end
605   end
606   return alltrue and true or head
607 end
608 end
```

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

```
609 local function list_handler_default()
610   return true
611 end
```

Handler for reverselist callbacks.

```
612 local function reverselist_handler(name)
613   return function(head, ...)
614     local ret
615     local alltrue = true
616     local callbacks = callbacklist[name]
617     for i = #callbacks, 1, -1 do
618       local cb = callbacks[i]
619       ret = cb.func(head, ...)
620       if ret == false then
621         luatexbase_warning(
622           "Function '" .. cb.description .. "' returned false\n"
623           .. "in callback '" .. name .. "'")
624       )
625       return false
626     end
627     if ret ~= true then
628       alltrue = false
629       head = ret
630     end
631   end
632   return alltrue and true or head
633 end
634 end
```

Handler for simple callbacks.

```
635 local function simple_handler(name)
```

```

636 return function(...)
637   for _,i in ipairs(callbacklist[name]) do
638     i.func(...)
639   end
640 end
641 end

```

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

```

642 local function simple_handler_default()
643 end

```

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

```

644 local handlers = {
645   [data]      = data_handler,
646   [exclusive] = exclusive_handler,
647   [list]      = list_handler,
648   [reverselist] = reverselist_handler,
649   [simple]     = simple_handler,
650 }
651 local defaults = {
652   [data]      = data_handler_default,
653   [exclusive] = nil,
654   [list]      = list_handler_default,
655   [reverselist] = list_handler_default,
656   [simple]     = simple_handler_default,
657 }

```

### 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.

```

658 local user_callbacks_defaults = {
659   pre_mlist_to_hlist_filter = list_handler_default,
660   mlist_to_hlist = node.mlist_to_hlist,
661   post_mlist_to_hlist_filter = list_handler_default,
662 }

```

`create_callback` The allocator itself.

```

663 local function create_callback(name, ctype, default)
664   local ctype_id = types[ctype]
665   if not name or name == ""
666   or not ctype_id
667   then
668     luatexbase_error("Unable to create callback:\n" ..
669                       "valid callback name and type required")
670   end
671   if callbacktypes[name] then
672     luatexbase_error("Unable to create callback '" .. name ..
673                       "':\ncallback is already defined")
674   end
675   default = default or defaults[ctype_id]
676   if not default then

```

```

677     luatexbase_error("Unable to create callback '" .. name ..
678                       "':\ndefault is required for '" .. ctype ..
679                       "' callbacks")
680 elseif type (default) ~= "function" then
681     luatexbase_error("Unable to create callback '" .. name ..
682                       "':\ndefault is not a function")
683 end
684 user_callbacks_defaults[name] = default
685 callbacktypes[name] = ctype_id
686 end
687 luatexbase.create_callback = create_callback

```

**call\_callback** Call a user defined callback. First check arguments.

```

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

```

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

```

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

```

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

```

722     local l = callbacklist[name]
723     if l == nil then

```



```

724     l = { }
725     callbacklist[name] = l

```

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

```

726     if user_callbacks_defaults[name] == nil then
727         callback_register(name, handlers[callbacktypes[name]](name))
728     end
729 end

```

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

```

730 local f = {
731     func      = func,
732     description = description,
733 }
734 local priority = #l + 1
735 if callbacktypes[name] == exclusive then
736     if #l == 1 then
737         luatexbase_error(
738             "Cannot add second callback to exclusive function\n" ..
739             name .. "'")
740     end
741 end
742 table.insert(l, priority, f)

```

Keep user informed.

```

743 luatexbase_log(
744     "Inserting '" .. description .. "' at position "
745     .. priority .. " in '" .. name .. "'")
746 )
747 end
748 luatexbase.add_to_callback = add_to_callback

```

**remove\_from\_callback** Remove a function from a callback. First check arguments.

```

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

```

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

```

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

```

794 local function in_callback(name, description)
795     if not name
796         or name == ""
797         or not callbacklist[name]
798         or not callbacktypes[name]
799         or not description then
800         return false
801     end
802     for _, i in pairs(callbacklist[name]) do
803         if i.description == description then
804             return true
805         end
806     end
807     return false
808 end
809 luatexbase.in_callback = in_callback

```

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

```

810 local function disable_callback(name)
811     if(callbacklist[name] == nil) then
812         callback_register(name, false)
813     else
814         luatexbase_error("Callback list for '" .. name .. "' not empty")
815     end
816 end

```

```
817 luatexbase.disable_callback = disable_callback
```

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

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

**uninstall** Unlike at the T<sub>E</sub>X 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!

```
834 local function uninstall()
835   module_info(
836     "luatexbase",
837     "Uninstalling kernel luatexbase code"
838   )
839   callback.register = callback_register
840   luatexbase = nil
841 end
842 luatexbase.uninstall = uninstall
```

**mlist\_to\_hlist** To emulate these callbacks, the “real” mlist\_to\_hlist is replaced by a wrapper calling the wrappers before and after.

```
843 callback_register("mlist_to_hlist", function(head, display_type, need_penalties)
844   local current = call_callback("pre_mlist_to_hlist_filter", head, display_type, need_penalties)
845   if current == false then
846     flush_list(head)
847     return nil
848   elseif current == true then
849     current = head
850   end
851   current = call_callback("mlist_to_hlist", current, display_type, need_penalties)
852   local post = call_callback("post_mlist_to_hlist_filter", current, display_type, need_penalties)
853   if post == true then
854     return current
855   elseif 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`