

lualatex.dtx  
(LuaTeX-specific support)

David Carlisle and Joseph Wright\*

2023/01/19

## Contents

<b>1</b>	<b>Overview</b>	<b>2</b>
<b>2</b>	<b>Core TeX functionality</b>	<b>2</b>
<b>3</b>	<b>Plain TeX interface</b>	<b>3</b>
<b>4</b>	<b>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</b>	<b>Implementation</b>	<b>6</b>
5.1	Minimum LuaTeX version . . . . .	6
5.2	Older L <sup>A</sup> TeX/Plain TeX setup . . . . .	7
5.3	Attributes . . . . .	9
5.4	Category code tables . . . . .	9
5.5	Named Lua functions . . . . .	11
5.6	Custom whatsits . . . . .	11
5.7	Lua bytecode registers . . . . .	11
5.8	Lua chunk registers . . . . .	12
5.9	Lua loader . . . . .	12
5.10	Lua module preliminaries . . . . .	14
5.11	Lua module utilities . . . . .	14
5.12	Accessing register numbers from Lua . . . . .	16
5.13	Attribute allocation . . . . .	17
5.14	Custom whatsit allocation . . . . .	17
5.15	Bytecode register allocation . . . . .	18
5.16	Lua chunk name allocation . . . . .	18
5.17	Lua function allocation . . . . .	18
5.18	Lua callback management . . . . .	19

---

\*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 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 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 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.

```
\newattribute \newattribute{attribute}
  Defines a named \attribute, indexed from 1 (i.e. \attribute0 is never defined).
  Attributes initially have the marker value -"7FFFFFFF ('unset') set by the engine.
\newcatcodetable \newcatcodetable{catcodetable}
  Defines a named \catcodetable, indexed from 1 (\catcodetable0 is never assigned).
  A new catcode table will be populated with exactly those values assigned by IniTeX (as described in the LuaTeX manual).
\newluafunction \newluafunction{function}
  Defines a named \luafunction, indexed from 1. (Lua indexes tables from 1 so \luafunction0 is not available).
  \newluacmd \newluacmd{function}
  Like \newluafunction, but defines the command using \luacmd instead of just assigning an integer.
\newprotectedluacmd \newluacmd{function}
  Like \newluacmd, but the defined command is not expandable.
\newwhatsit \newwhatsit{whatsit}
  Defines a custom \whatsit, indexed from 1.
\newluabytecode \newluabytecode{bytecode}
```

	Allocates a number for Lua bytecode register, indexed from 1.
<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@atletter</code>	<code>\setattribute{⟨attribute⟩}{⟨value⟩}</code>
<code>\setattribute</code>	<code>\unsetattribute{⟨attribute⟩}</code>
<code>\unsetattribute</code>	Set and unset attributes in a manner analogous to <code>\setlength</code> . 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>-"7FFFFFFF</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 `ltluatex.lua`, for example

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

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
        bad input
@MM: \mathchar"4E20
      20000
@tempdima: \dimen14
          14
@tempdimb: \dimen15
          15
strutbox: \char"B
          11
sixt@n: \char"10
         16
myattr: \attribute12
        12

```

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

### 4.3 Module utilities

`provides_module` `luatexbase.provides_module(<info>)`

This function is used by modules to identify themselves; the `info` should be a table containing information about the module. The required field `name` must contain the name of the module. It is recommended to provide a field `date` in the usual 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.

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

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

`create_callback` `luatexbase.create_callback(<name>,<type>,<default>)` Defines a user defined callback. The last argument is a default function or `false`.

`call_callback` `luatexbase.call_callback(<name>,...)` Calls a user defined callback with the supplied arguments.

`declare_callback_rule` `luatexbase.declare_callback_rule(<name>, <first>, <relation>, <second>)` Adds an ordering constraint between two callback functions for callback *<name>*.  
The kind of constraint added depends on *<relation>*:

**before** The callback function with description *<first>* will be executed before the function with description *<second>*.

**after** The callback function with description *<first>* will be executed after the function with description *<second>*.

**incompatible-warning** When both a callback function with description *<first>* and with description *<second>* is registered, then a warning is printed when the callback is executed.

**incompatible-error** When both a callback function with description *<first>* and with description *<second>* is registered, then an error is printed when the callback is executed.

**unrelated** Any previously declared callback rule between *<first>* and *<second>* gets disabled.

Every call to `declare_callback_rule` with a specific callback *<name>* and descriptions *<first>* and *<second>* overwrites all previous calls with same callback and descriptions.

The callback functions do not have to be registered yet when the functions is called. Only the constraints for which both callback descriptions refer to callbacks registered at the time the callback is called will have an effect.

## 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.dtx` have to be defined here because `ltdfn.dtx` is not loaded yet when `ltuatex.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.extraprimtives("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\@cclvi\fi}
31 \count 270=\et@xmaxregs % locally allocates \count registers
32 \count 271=\et@xmaxregs % ditto for \dimen registers
33 \count 272=\et@xmaxregs % ditto for \skip registers
34 \count 273=\et@xmaxregs % ditto for \muskip registers
35 \count 274=\et@xmaxregs % ditto for \box registers

```

```

36 \count 275=\et@xmaxregs % ditto for \toks registers
37 \count 276=\et@xmaxregs % ditto for \marks classes
    and 256 or 16 fam. (Done above due to plain/LATEX differences in lAuTex.)
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 of `luatex.sty` to leave room for inserts not really needed for `luatex` but possibly most compatible with existing use.

```

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

```

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

```

47 \chardef\e@alloc@top=65535
48 \let\e@alloc@chardef\chardef
49 \def\e@alloc#1#2#3#4#5#6{%
50   \global\advance#3\@ne
51   \e@ch@ck{#3}{#4}{#5}#1%
52   \allocationnumber#3\relax
53   \global#2#6\allocationnumber
54   \wlog{\string#6=\string#1\the\allocationnumber}}%
55 \gdef\e@ch@ck#1#2#3#4{%
56   \ifnum#1<#2\else
57     \ifnum#1=#2\relax
58       #1\@cclvi
59       \ifx\count#4\advance#1 10 \fi
60       \fi
61     \ifnum#1<#3\relax
62       \else
63         \errmessage{No room for a new \string#4}%
64       \fi
65     \fi}%

```

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

```

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

```

End of conditional setup for plain 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 LuaTeX registers we start here from 1. Notably, some code assumes that `\attribute0` is never used so this is important in this case.

```
74 \ifx\@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 LuaTeX for everything else. At the end of allocation there needs to be an initialization step. Table 0 is already taken (it's the global one for current use) so the allocation starts at 1.

```
84 \ifx\@alloc@ccodetable@count\@undefined
85   \countdef\@alloc@ccodetable@count=259
86   \@alloc@ccodetable@count=\z@
87 \fi
88 \def\newcatcodetable#1{%
89   \@alloc\catcodetable\chardef
90   \@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\setrangeatcode#1#2#3{%
                       97     \ifnum#1>#2 %
                       98       \expandafter\@gobble
                       99     \else
100       \expandafter\@firstofone
101     \fi
102     {%
103       \catcode#1=#3 %
104       \expandafter\setrangeatcode\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 \setrange\catcode{65}{90}{12}%
114 \setrange\catcode{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 }

```

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

```

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

```

## 5.6 Custom whatsits

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

```

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

```

## 5.7 Lua bytecode registers

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

```

197 \ifx\e@alloc@bytecode@count\@undefined

```

```

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

```

## 5.8 Lua chunk registers

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

```

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

```

## 5.9 Lua loader

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

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

```

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

285 <2kernel | latexrelease>\fi
286 </2kernel | tex | latexrelease>

```

## 5.10 Lua module preliminaries

287 `<*lua>`

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

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

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

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

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

295 local luatexbase_warning
296 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.

```
297 local modules = modules or { }
```

`provides_module` Local function to write to the log.

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

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

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

### 5.11.2 Module messages

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

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

```
317 local function msg_format(mod, msg_type, text)
318   local leader = ""
319   local cont
320   local first_head
321   if mod == "LaTeX" then
322     cont = string_gsub(leader, ".", " ")
323     first_head = leader .. "LaTeX: "
324   else
325     first_head = leader .. "Module " .. msg_type
326     cont = "(" .. mod .. ")"
327     .. string_gsub(first_head, ".", " ")
328     first_head = leader .. "Module " .. mod .. " " .. msg_type .. ":"
329   end
330   if msg_type == "Error" then
331     first_head = "\n" .. first_head
332   end
333   if string.sub(text,-1) ~= "\n" then
334     text = text .. " "
335   end
336   return first_head .. " "
337     .. string_gsub(
338       text
339     .. "on input line "
340       .. tex.inputlineno, "\n", "\n" .. cont .. " "
341     )
342     .. "\n"
343 end
```

`module_info` Write messages.

```
module_warning 344 local function module_info(mod, text)
module_error    345   texio_write_nl("log", msg_format(mod, "Info", text))
                346 end
                347 luatexbase.module_info = module_info
                348 local function module_warning(mod, text)
                349   texio_write_nl("term and log",msg_format(mod, "Warning", text))
                350 end
                351 luatexbase.module_warning = module_warning
                352 local function module_error(mod, text)
                353   error(msg_format(mod, "Error", text))
                354 end
                355 luatexbase.module_error = module_error
```

Dedicated versions for the rest of the code here.

```
356 function luatexbase_warning(text)
```

```

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

```

362 local luaregisterbasetable = { }
363 local registermap = {
364   attributezero = "assign_attr"   ,
365   charzero      = "char_given"    ,
366   CountZero     = "assign_int"    ,
367   dimenzero     = "assign_dimen"  ,
368   mathcharzero  = "math_given"    ,
369   muskipzero    = "assign_mu_skip",
370   skipzero      = "assign_skip"   ,
371   tokszero      = "assign_toks"   ,
372 }
373 local createtoken
374 if tex.luatexversion > 81 then
375   createtoken = token.create
376 elseif tex.luatexversion > 79 then
377   createtoken = newtoken.create
378 end
379 local hashtokens = tex.hashtokens()
380 local luatexversion = tex.luatexversion
381 for i,j in pairs (registermap) do
382   if luatexversion < 80 then
383     luaregisterbasetable[hashtokens[i][1]] =
384       hashtokens[i][2]
385   else
386     luaregisterbasetable[j] = createtoken(i).mode
387   end
388 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.

```

389 local registernumber
390 if luatexversion < 80 then
391   function registernumber(name)
392     local nt = hashtokens[name]
393     if(nt and luaregisterbasetable[nt[1]]) then
394       return nt[2] - luaregisterbasetable[nt[1]]
395     else
396       return false
397     end
398   end
399 else

```



```

400 function registernumber(name)
401   local nt = createtoken(name)
402   if(luaregisterbasetable[nt.cmdname]) then
403     return nt.mode - luaregisterbasetable[nt.cmdname]
404   else
405     return false
406   end
407 end
408 end
409 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.

```

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

```

### 5.14 Custom whatsit allocation

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

```

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

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

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

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

```

484 end
485 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.

Actually there are two tables: `realcallbacklist` directly contains the entries as described above while `callbacklist` only directly contains the already sorted entries. Other entries can be queried through `callbacklist` too which triggers a resort.

Additionally `callbackrules` describes the ordering constraints: It contains two element tables with the descriptions of the constrained callback implementations. It can additionally contain a `type` entry indicating the kind of rule. A missing value indicates a normal ordering constraint.

```

486 local realcallbacklist = {}
487 local callbackrules = {}
488 local callbacklist = setmetatable({}, {
489   __index = function(t, name)
490     local list = realcallbacklist[name]
491     local rules = callbackrules[name]
492     if list and rules then
493       local meta = {}
494       for i, entry in ipairs(list) do
495         local t = {value = entry, count = 0, pos = i}
496         meta[entry.description], list[i] = t, t
497       end
498       local count = #list
499       local pos = count
500       for i, rule in ipairs(rules) do
501         local rule = rules[i]
502         local pre, post = meta[rule[1]], meta[rule[2]]
503         if pre and post then
504           if rule.type then
505             if not rule.hidden then
506               assert(rule.type == 'incompatible-warning' and luatexbase_warning
507                 or rule.type == 'incompatible-error' and luatexbase_error)(
508                 "Incompatible functions \"" .. rule[1] .. "\" and \"" .. rule[2]
509                 .. "\" specified for callback \"" .. name .. "\".")
510               rule.hidden = true
511             end
512           else
513             local post_count = post.count
514             post.count = post_count+1

```

```

515         if post_count == 0 then
516             local post_pos = post.pos
517             if post_pos ~= pos then
518                 local new_post_pos = list[pos]
519                 new_post_pos.pos = post_pos
520                 list[post_pos] = new_post_pos
521             end
522             list[pos] = nil
523             pos = pos - 1
524         end
525         pre[#pre+1] = post
526     end
527 end
528 end
529 for i=1, count do -- The actual sort begins
530     local current = list[i]
531     if current then
532         meta[current.value.description] = nil
533         for j, cur in ipairs(current) do
534             local count = cur.count
535             if count == 1 then
536                 pos = pos + 1
537                 list[pos] = cur
538             else
539                 cur.count = count - 1
540             end
541         end
542         list[i] = current.value
543     else
544         -- Cycle occurred. TODO: Show cycle for debugging
545         -- list[i] = ...
546         local remaining = {}
547         for name, entry in next, meta do
548             local value = entry.value
549             list[#list + 1] = entry.value
550             remaining[#remaining + 1] = name
551         end
552         table.sort(remaining)
553         local first_name = remaining[1]
554         for j, name in ipairs(remaining) do
555             local entry = meta[name]
556             list[i + j - 1] = entry.value
557             for _, post_entry in ipairs(entry) do
558                 local post_name = post_entry.value.description
559                 if not remaining[post_name] then
560                     remaining[post_name] = name
561                 end
562             end
563         end
564         local cycle = {first_name}
565         local index = 1
566         local last_name = first_name
567         repeat
568             cycle[last_name] = index

```

```

569         last_name = remaining[last_name]
570         index = index + 1
571         cycle[index] = last_name
572     until cycle[last_name]
573     local length = index - cycle[last_name] + 1
574     table.move(cycle, cycle[last_name], index, 1)
575     for i=2, length//2 do
576         cycle[i], cycle[length + 1 - i] = cycle[length + 1 - i], cycle[i]
577     end
578     error('Cycle occurred at ' .. table.concat(cycle, ' -> ', 1, length))
579 end
580 end
581 end
582 realcallbacklist[name] = list
583 t[name] = list
584 return list
585 end
586 })

```

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

```

587 local list, data, exclusive, simple, reverselist = 1, 2, 3, 4, 5
588 local types = {
589     list = list,
590     data = data,
591     exclusive = exclusive,
592     simple = simple,
593     reverselist = reverselist,
594 }

```

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.)

```

595 local callbacktypes = callbacktypes or {

```

Section 8.2: file discovery callbacks.

```

596     find_read_file = exclusive,
597     find_write_file = exclusive,
598     find_font_file = data,
599     find_output_file = data,
600     find_format_file = data,
601     find_vf_file = data,
602     find_map_file = data,
603     find_enc_file = data,

```

```

604 find_pk_file      = data,
605 find_data_file    = data,
606 find_opentype_file = data,
607 find_truetype_file = data,
608 find_type1_file    = data,
609 find_image_file    = data,

610 open_read_file     = exclusive,
611 read_font_file     = exclusive,
612 read_vf_file       = exclusive,
613 read_map_file      = exclusive,
614 read_enc_file      = exclusive,
615 read_pk_file       = exclusive,
616 read_data_file     = exclusive,
617 read_truetype_file = exclusive,
618 read_type1_file    = exclusive,
619 read_opentype_file = exclusive,

```

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

```

620 find_cidmap_file  = data,
621 read_cidmap_file  = exclusive,

```

Section 8.3: data processing callbacks.

```

622 process_input_buffer = data,
623 process_output_buffer = data,
624 process_jobname      = data,

```

Section 8.4: node list processing callbacks.

```

625 contribute_filter   = simple,
626 buildpage_filter    = simple,
627 build_page_insert   = exclusive,
628 pre_linebreak_filter = list,
629 linebreak_filter     = exclusive,
630 append_to_vlist_filter = exclusive,
631 post_linebreak_filter = reverselist,
632 hpack_filter         = list,
633 vpack_filter         = list,
634 hpack_quality        = exclusive,
635 vpack_quality        = exclusive,
636 pre_output_filter    = list,
637 process_rule         = exclusive,
638 hyphenate            = simple,
639 ligaturing           = simple,
640 kerning              = simple,
641 insert_local_par     = simple,
642 % mlist_to_hlist     = exclusive,
643 new_graf             = exclusive,

```

Section 8.5: information reporting callbacks.

```

644 pre_dump            = simple,
645 start_run           = simple,
646 stop_run            = simple,
647 start_page_number   = simple,
648 stop_page_number    = simple,
649 show_error_hook     = simple,

```

```

650 show_warning_message = simple,
651 show_error_message   = simple,
652 show_lua_error_hook  = simple,
653 start_file           = simple,
654 stop_file            = simple,
655 call_edit            = simple,
656 finish_synctex       = simple,
657 wrapup_run          = simple,

```

Section 8.6: PDF-related callbacks.

```

658 finish_pdffile       = data,
659 finish_pdfpage       = data,
660 page_objnum_provider = data,
661 page_order_index     = data,
662 process_pdf_image_content = data,

```

Section 8.7: font-related callbacks.

```

663 define_font          = exclusive,
664 glyph_info           = exclusive,
665 glyph_not_found     = exclusive,
666 glyph_stream_provider = exclusive,
667 make_extensible     = exclusive,
668 font_descriptor_objnum_provider = exclusive,
669 input_level_string   = exclusive,
670 provide_charproc_data = exclusive,

```

```

671 }
672 luatexbase.callbacktypes=callbacktypes

```

Sometimes multiple callbacks correspond to a single underlying engine level callback. Then the engine level callback should be registered as long as at least one of these callbacks is in use. This is implemented though a shared table which counts how many of the involved callbacks are currently in use. The engine level callback is registered iff this count is not 0.

We add `mlist_to_hlist` directly to the list to demonstrate this, but the handler gets added later when it is actually defined.

All callbacks in this list are treated as user defined callbacks.

```

673 local shared_callbacks = {
674   mlist_to_hlist = {
675     callback = "mlist_to_hlist",
676     count = 0,
677     handler = nil,
678   },
679 }
680 shared_callbacks.pre_mlist_to_hlist_filter = shared_callbacks.mlist_to_hlist
681 shared_callbacks.post_mlist_to_hlist_filter = shared_callbacks.mlist_to_hlist

```

`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.

```

682 local callback_register = callback_register or callback.register
683 function callback.register()
684   luatexbase_error("Attempt to use callback.register() directly\n")
685 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.

```
686 local function data_handler(name)
687   return function(data, ...)
688     for _,i in ipairs(callbacklist[name]) do
689       data = i.func(data,...)
690     end
691     return data
692   end
693 end
```

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

```
694 local function data_handler_default(value)
695   return value
696 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.



```

697 local function exclusive_handler(name)
698   return function(...)
699     return callbacklist[name][1].func(...)
700   end
701 end

```

Handler for list callbacks.

```

702 local function list_handler(name)
703   return function(head, ...)
704     local ret
705     for _,i in ipairs(callbacklist[name]) do
706       ret = i.func(head, ...)
707       if ret == false then
708         luatexbase_warning(
709           "Function '" .. i.description .. "' returned false\n"
710           .. "in callback '" .. name .. "'")
711       )
712       return false
713     end
714     if ret ~= true then
715       head = ret
716     end
717   end
718   return head
719 end
720 end

```

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

```

721 local function list_handler_default(head)
722 return head
723 end

```

Handler for reverselist callbacks.

```

724 local function reverselist_handler(name)
725   return function(head, ...)
726     local ret
727     local callbacks = callbacklist[name]
728     for i = #callbacks, 1, -1 do
729       local cb = callbacks[i]
730       ret = cb.func(head, ...)
731       if ret == false then
732         luatexbase_warning(
733           "Function '" .. cb.description .. "' returned false\n"
734           .. "in callback '" .. name .. "'")
735       )
736       return false
737     end
738     if ret ~= true then
739       head = ret
740     end
741   end
742   return head
743 end
744 end

```

Handler for simple callbacks.

```

745 local function simple_handler(name)
746   return function(...)
747     for _,i in ipairs(callbacklist[name]) do
748       i.func(...)
749     end
750   end
751 end

```

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

```

752 local function simple_handler_default()
753 end

```

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

```

754 local handlers = {
755   [data]      = data_handler,
756   [exclusive] = exclusive_handler,
757   [list]      = list_handler,
758   [reverselist] = reverselist_handler,
759   [simple]     = simple_handler,
760 }
761 local defaults = {
762   [data]      = data_handler_default,
763   [exclusive] = nil,
764   [list]      = list_handler_default,
765   [reverselist] = list_handler_default,
766   [simple]     = simple_handler_default,
767 }

```

### 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.

```

768 local user_callbacks_defaults = {}

```

`create_callback` The allocator itself.

```

769 local function create_callback(name, ctype, default)
770   local ctype_id = types[ctype]
771   if not name or name == ""
772   or not ctype_id
773   then
774     luatexbase_error("Unable to create callback:\n" ..
775                       "valid callback name and type required")
776   end
777   if callbacktypes[name] then
778     luatexbase_error("Unable to create callback '" .. name ..
779                       "':\ncallback is already defined")
780   end
781   default = default or defaults[ctype_id]
782   if not default then
783     luatexbase_error("Unable to create callback '" .. name ..
784                       "':\ndefault is required for '" .. ctype ..
785                       "' callbacks")

```

```

786 elseif type (default) ~= "function" then
787     luatexbase_error("Unable to create callback '" .. name ..
788         "':\ndefault is not a function")
789 end
790 user_callbacks_defaults[name] = default
791 callbacktypes[name] = ctype_id
792 end
793 luatexbase.create_callback = create_callback

```

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

```

794 local function call_callback(name,...)
795     if not name or name == "" then
796         luatexbase_error("Unable to create callback:\n" ..
797             "valid callback name required")
798     end
799     if user_callbacks_defaults[name] == nil then
800         luatexbase_error("Unable to call callback '" .. name
801             .. "':\nunknown or empty")
802     end
803     local l = callbacklist[name]
804     local f
805     if not l then
806         f = user_callbacks_defaults[name]
807     else
808         f = handlers[callbacktypes[name]](name)
809     end
810     return f(...)
811 end
812 luatexbase.call_callback=call_callback

```

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

```

813 local function add_to_callback(name, func, description)
814     if not name or name == "" then
815         luatexbase_error("Unable to register callback:\n" ..
816             "valid callback name required")
817     end
818     if not callbacktypes[name] or
819         type(func) ~= "function" or
820         not description or
821         description == "" then
822         luatexbase_error(
823             "Unable to register callback.\n\n"
824             .. "Correct usage:\n"
825             .. "add_to_callback(<callback>, <function>, <description>)"
826         )
827     end

```

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

```

828     local l = realcallbacklist[name]
829     if l == nil then
830         l = { }
831         realcallbacklist[name] = l

```

Handle count for shared engine callbacks.

```

832 local shared = shared_callbacks[name]
833 if shared then
834     shared.count = shared.count + 1
835     if shared.count == 1 then
836         callback_register(shared.callback, shared.handler)
837     end

```

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

```

838     elseif user_callbacks_defaults[name] == nil then
839         callback_register(name, handlers[callbacktypes[name]](name))
840     end
841 end

```

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

```

842 local f = {
843     func = func,
844     description = description,
845 }
846 if callbacktypes[name] == exclusive then
847     if #l == 1 then
848         luatexbase_error(
849             "Cannot add second callback to exclusive function\n" ..
850             name .. "'")
851     end
852 end
853 table.insert(l, f)
854 callbacklist[name] = nil

```

Keep user informed.

```

855 luatexbase_log(
856     "Inserting '" .. description .. "' in '" .. name .. "'."
857 )
858 end
859 luatexbase.add_to_callback = add_to_callback

```

**declare\_callback\_rule** Add an ordering constraint between two callback implementations

```

860 local function declare_callback_rule(name, desc1, relation, desc2)
861     if not callbacktypes[name] or
862         not desc1 or not desc2 or
863         desc1 == "" or desc2 == "" then
864         luatexbase_error(
865             "Unable to create ordering constraint. "
866             .. "Correct usage:\n"
867             .. "declare_callback_rule(<callback>, <description_a>, <description_b>)"
868         )
869     end
870     if relation == 'before' then
871         relation = nil
872     elseif relation == 'after' then
873         desc2, desc1 = desc1, desc2
874         relation = nil
875     elseif relation == 'incompatible-warning' or relation == 'incompatible-error' then
876     elseif relation == 'unrelated' then
877     else

```

```

878     luatexbase_error(
879         "Unknown relation type in declare_callback_rule"
880     )
881 end
882 callbacklist[name] = nil
883 local rules = callbackrules[name]
884 if rules then
885     for i, rule in ipairs(rules) do
886         if rule[1] == desc1 and rule[2] == desc2 or rule[1] == desc2 and rule[2] == desc1 then
887             if relation == 'unrelated' then
888                 table.remove(rules, i)
889             else
890                 rule[1], rule[2], rule.type = desc1, desc2, relation
891             end
892             return
893         end
894     end
895     if relation ~= 'unrelated' then
896         rules[#rules + 1] = {desc1, desc2, type = relation}
897     end
898 elseif relation ~= 'unrelated' then
899     callbackrules[name] = {{desc1, desc2, type = relation}}
900 end
901 end
902 luatexbase.declare_callback_rule = declare_callback_rule

```

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

```

903 local function remove_from_callback(name, description)
904     if not name or name == "" then
905         luatexbase_error("Unable to remove function from callback:\n" ..
906             "valid callback name required")
907     end
908     if not callbacktypes[name] or
909         not description or
910         description == "" then
911         luatexbase_error(
912             "Unable to remove function from callback.\n\n"
913             .. "Correct usage:\n"
914             .. "remove_from_callback(<callback>, <description>)"
915         )
916     end
917     local l = realcallbacklist[name]
918     if not l then
919         luatexbase_error(
920             "No callback list for '" .. name .. "'\n")
921     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.

```

922     local index = false
923     for i,j in ipairs(l) do
924         if j.description == description then
925             index = i
926             break

```

```

927     end
928 end
929 if not index then
930     luatexbase_error(
931         "No callback '" .. description .. "' registered for '" ..
932         name .. "'\n")
933 end
934 local cb = l[index]
935 table.remove(l, index)
936 luatexbase_log(
937     "Removing '" .. description .. "' from '" .. name .. "'."
938 )
939 if #l == 0 then
940     realcallbacklist[name] = nil
941     callbacklist[name] = nil
942     local shared = shared_callbacks[name]
943     if shared then
944         shared.count = shared.count - 1
945         if shared.count == 0 then
946             callback_register(shared.callback, nil)
947         end
948     elseif user_callbacks_defaults[name] == nil then
949         callback_register(name, nil)
950     end
951 end
952 return cb.func,cb.description
953 end
954 luatexbase.remove_from_callback = remove_from_callback

```

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

```

955 local function in_callback(name, description)
956     if not name
957         or name == ""
958         or not realcallbacklist[name]
959         or not callbacktypes[name]
960         or not description then
961         return false
962     end
963     for _, i in pairs(realcallbacklist[name]) do
964         if i.description == description then
965             return true
966         end
967     end
968     return false
969 end
970 luatexbase.in_callback = in_callback

```

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

```

971 local function disable_callback(name)
972     if(realcallbacklist[name] == nil) then
973         callback_register(name, false)
974     else
975         luatexbase_error("Callback list for " .. name .. " not empty")

```

```

976 end
977 end
978 luatexbase.disable_callback = disable_callback

```

**callback\_descriptions** List the descriptions of functions registered for the given callback. This will sort the list if necessary.

```

979 local function callback_descriptions (name)
980   local d = {}
981   if not name
982     or name == ""
983     or not realcallbacklist[name]
984     or not callbacktypes[name]
985   then
986     return d
987   else
988     for k, i in pairs(callbacklist[name]) do
989       d[k]= i.description
990     end
991   end
992   return d
993 end
994 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!

```

995 local function uninstall()
996   module_info(
997     "luatexbase",
998     "Uninstalling kernel luatexbase code"
999   )
1000   callback.register = callback_register
1001   luatexbase = nil
1002 end
1003 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.

```

1004 create_callback('pre_mlist_to_hlist_filter', 'list')
1005 create_callback('mlist_to_hlist', 'exclusive', node.mlist_to_hlist)
1006 create_callback('post_mlist_to_hlist_filter', 'list')
1007 function shared_callbacks.mlist_to_hlist.handler(head, display_type, need_penalties)
1008   local current = call_callback("pre_mlist_to_hlist_filter", head, display_type, need_penalties)
1009   if current == false then
1010     flush_list(head)
1011     return nil
1012   end
1013   current = call_callback("mlist_to_hlist", current, display_type, need_penalties)
1014   local post = call_callback("post_mlist_to_hlist_filter", current, display_type, need_penalties)
1015   if post == false then
1016     flush_list(current)
1017     return nil
1018   end

```

```
1019 return post
1020 end

1021 </lua>

Reset the catcode of @.
1022 <tex>\catcode'\@=\etatcatcode\relax
```