

lualatex.dtx

(LuaTeX-specific support)

David Carlisle and Joseph Wright*

2026-10-28

Contents

1 Overview	2
2 Core TeX functionality	2
3 Plain TeX interface	3
4 Lua functionality	3
4.1 Allocators in Lua	3
4.2 Lua access to TeX register numbers	4
4.3 Module utilities	5
4.4 Callback management	6
5 Implementation	7
5.1 Minimum LuaTeX version	7
5.2 Older LaTeX/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	12
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	18
5.15 Bytecode register allocation	18
5.16 Lua chunk name allocation	18
5.17 Lua function allocation	19
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^AT_ΕX 2_ε kernel level plus as a loadable file which can be used with plain TeX and L^AT_ΕX.

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^AT_ΕX 2_ε kernel from 2015 onward these registers are part of the block in the extended area reserved by the kernel (prior to 2015 the L^AT_ΕX 2_ε kernel did not provide any functionality for the extended allocation area).

2 Core TeX functionality

The commands defined here are defined for possible inclusion in a future L^AT_ΕX format, however also extracted to the file `ltluatex.tex` which may be used with older L^AT_ΕX 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 \luadef instead of just assigning an integer.
\newprotectedluacmd \newluadef{<function>}
  Like \newluacmd, but the defined command is not expandable.
\newwhatsit \newwhatsit{<whatsit>}
  Defines a custom \whatsit, indexed from 1.
\newluabytecode \newluabytecode{<bytecode>}
```

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

The `luatex` interface may be used with plain T_EX using `\input{luatex}`. This inputs `luatex.tex` which inputs `etex.src` (or `etex.sty` if used with L^AT_EX) 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_EX, 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 initialized with the marker value <code>-0xFFFFFFFF</code> ('unset'). The attribute allocation sequence is shared with the T _E 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 _E 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>\l_uatexlua</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>\l_uatexluafunction</code> , and <code>\lua_udef</code> , indexed from 1. The optional <code>⟨functionname⟩</code> argument is just used for logging.

`new_luacommand` `luatexbase.new_luacommand(<cname>, <fn>, <...>)`

Convenience wrapper combining `new_luafunction` with `token.set_lua`. Allocates a new lua function slot for `<cname>` (as `new_luafunction` does), stores the Lua function `<fn>` in it, then calls `token.set_lua(<cname>, <fnalloc>, <...>)` to bind `<cname>` to that slot. Any further `<...>` arguments are passed on unchanged to `token.set_lua`.

These functions all require access to a named TeX count register to manage their allocations. The standard names are those defined above for access from TeX, e.g. “e@alloc@attribute@count, but these can be adjusted by defining the variable `<type>.count_name` before loading `ltxlua.tex`, for example

```
local attribute_count_name = "attributetracker"
require("ltxlua.tex")
```

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 LuaTeX'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 $\text{\LaTeX}$ 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 $\text{\newbox}$ work and return the number of the box register even though the actual command holding this number is a $\text{\chardef}$ defined token (there is no $\text{\boxdef}$).

4.3 Module utilities

provides_module `luatexbase.provides_module( $\langle info \rangle$ )`

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 $\text{\LaTeX}$ format `yyyy/mm/dd`. Optional fields **version** (a string) and **description** may be used if present. This information will be recorded in the log. Other fields are ignored. If the **version** begins with a digit, a **v** will be added at the start in the log.

module_info `luatexbase.module_info( $\langle module \rangle$ ,  $\langle text \rangle$ )`
module_warning `luatexbase.module_warning( $\langle module \rangle$ ,  $\langle text \rangle$ )`
module_error `luatexbase.module_error( $\langle module \rangle$ ,  $\langle text \rangle$ )`

These functions are similar to $\text{\LaTeX}$'s $\text{\PackageError}$, $\text{\PackageWarning}$ and $\text{\PackageInfo}$ in the way they format the output. No automatic line breaking is done, you may still use $\text{\n}$ 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>, <identifier>)` Registers the *<function>* into the *<callback>* with a textual *<identifier>* of the function. Functions are inserted into the callback in the order loaded. The *<identifier>* should be unique for each callback.
- remove_from_callback** `luatexbase.remove_from_callback(<callback>, <identifier>)` Removes the callback function with *<identifier>* from the *<callback>*. The removed function and its identifier are returned as the results of this function.
- in_callback** `luatexbase.in_callback(<callback>, <identifier>)` Checks if the *<identifier>* 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_identifiers** A list of the identifiers of functions registered to the specified callback is returned. `{}` is returned if there are no functions registered.
- callback_descriptions** This is a historical alias for **callback_identifiers**.
- 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 identifier *<first>* will be executed before the function with identifier *<second>*.
 - after** The callback function with identifier *<first>* will be executed after the function with identifier *<second>*.
 - incompatible-warning** When both a callback function with identifier *<first>* and with identifier *<second>* is registered, then a warning is printed when the callback is executed.
 - incompatible-error** When both a callback function with identifier *<first>* and with identifier *<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 identifier *<first>* and *<second>* overwrites all previous calls with same callback and identifiers.

The callback functions do not have to be registered yet when the functions is called. Only the constraints for which both callback identifiers 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^AT_EX macros from `ltdfns.dtx` have to be defined here because `ltdfns.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^AT_EX/Plain T_EX setup

```

13 <*tex>

```

Older L^AT_EX 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^AT_EX, or plain T_EX, 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/L^AT_EX differences in l^Auatex.)

```
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 L^AT_EX (the existing macros in `etex.src` are 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\@alloc@attribute@count
67 \expandafter\csname newcount\endcsname\@alloc@ccodetable@count
68 \expandafter\csname newcount\endcsname\@alloc@luafunction@count
69 \expandafter\csname newcount\endcsname\@alloc@whatsit@count
70 \expandafter\csname newcount\endcsname\@alloc@bytecode@count
71 \expandafter\csname newcount\endcsname\@alloc@luachunk@count
```

End of conditional setup for plain T_EX / old L^AT_EX.

```
72 \fi
73 \</tex>
```

5.3 Attributes

`\newattribute` As is generally the case for the LuaT_EX 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_EX 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\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 }

```

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

```

5.7 Lua bytecode registers

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

```

197 \ifx\@alloc@bytecode@count\@undefined
198 \countdef\@alloc@bytecode@count=262
199 \@alloc@bytecode@count=\z@
200 \fi
201 \def\newluabytocode#1{%
202 \@alloc@luabytocode\@alloc@chardef
203 \@alloc@bytecode@count\m@ne\@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\@alloc@luachunk@count\@undefined
206 \countdef\@alloc@luachunk@count=263
207 \@alloc@luachunk@count=\z@
208 \fi
209 \def\newluachunkname#1{%
210 \@alloc@luachunk\@alloc@chardef
211 \@alloc@luachunk@count\m@ne\@alloc@top#1%
212 \directlua{lua.name[\the\allocationnumber]="\csstring#1"}%
213 }

```

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:

```

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

```

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.

```

220 <2ekernel>\now@and@everyjob{%
221 \begingroup
222 \attributedef\attributezero=0 %
223 \chardef \charzero =0 %

```

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

```

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

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

```

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

```

257 <latexrelease>\IncludeInRelease{2017/01/01}%
258 <latexrelease> {\fontencoding}{TU in everyjob}%
259 <latexrelease>\fontencoding{TU}\let\encodingdefault\f@encoding
260 <latexrelease>\ifx\directlua\@undefined\else
261 <2ekernel>\everyjob\expandafter{%
262 <2ekernel> \the\everyjob
263 <*2ekernel, latexrelease>
264 \directlua{%
265 if xpcall(function ()%
266 require('luaotfload-main')%
267 end, texio.write_nl) then %
268 local _void = luaotfload.main ()%
269 else %
270 texio.write_nl('Error in luaotfload: reverting to OT1')%
271 tex.print('\string\def\string\encodingdefault{OT1}')%
272 end %
273 }%
274 \let\f@encoding\encodingdefault

```

```

275 \expandafter\let\csname ver@luaotfload.sty\endcsname\fmtversion
276 </2ekernel, latexrelease>
277 <latexrelease>\fi
278 <2ekernel> }
279 <latexrelease>\EndIncludeInRelease
280 <latexrelease>\IncludeInRelease{0000/00/00}%
281 <latexrelease> {\fontencoding}{TU in everyjob}%
282 <latexrelease>\fontencoding{OT1}\let\encodingdefault\f@encoding
283 <latexrelease>\EndIncludeInRelease

284 <2ekernel | latexrelease>\fi
285 </2ekernel | tex | latexrelease>

```

5.10 Lua module preliminaries

```
286 <*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.

```

287 luatexbase      = luatexbase or { }
288 local luatexbase = luatexbase

```

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

```

289 local string_format    = string.format
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 token_set_lua     = token.set_lua

296 local luatexbase_warning
297 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.

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

provides_module Local function to write to the log.

```

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

```

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

```

302 local function provides_module(info)
303   if not (info and info.name) then
304     luatexbase_error("Missing module name for provides_module")
305   end

```

```

306 local function spaced(text)
307     return text and (" " .. text) or ""
308 end
309 luatexbase_log(
310     "Lua module: " .. info.name
311     .. spaced(info.date)
312     .. spaced(info.version and string_gsub(info.version or "", "%d"), "v%1")
313     .. spaced(info.description)
314 )
315 modules[info.name] = info
316 end
317 luatexbase.provides_module = provides_module

```

5.11.2 Module messages

There are various warnings and errors that need to be given. For warnings we can get exactly the same formatting as from $\text{\TeX}$. 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.

```

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

```

module_info Write messages.

```

module_warning 345 local function module_info(mod, text)
module_error   346     texio_write_nl("log", msg_format(mod, "Info", text))

```

```

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

```

Dedicated versions for the rest of the code here.

```

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

```

5.12 Accessing register numbers from Lua

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

```

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

```

registernumber Working out the correct return value can be done in two ways. For older LuaT_EX releases it has to be extracted from the `hashtokens`. On the other hand, newer

LuaTeX's have `newtoken`, and whilst `.mode` isn't currently documented, Hans Hagen pointed to this approach so we should be OK.

```

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

```

5.13 Attribute allocation

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

```

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

```

5.14 Custom whatsit allocation

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

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

5.15 Bytecode register allocation

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

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

5.16 Lua chunk name allocation

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

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

5.17 Lua function allocation

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

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

new_luacommand A convenience wrapper around **new_luafunction** and **token.set_lua**: it allocates the lua function slot, stores $\langle fn \rangle$ there, and binds $\langle csname \rangle$ to it in one step.

```
491 local function new_luacommand(csname, fn, ...)
492     local fntable = lua.get_functions_table()
493     local fnalloc = new_luafunction(csname)
494     fntable[fnalloc] = fn
495     token_set_lua(csname, fnalloc, ...)
496 end
497 luatexbase.new_luacommand = new_luacommand
```

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 identifier **identifier**. 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 identifiers 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.

```

498 local realcallbacklist = {}
499 local callbackrules = {}
500 local callbacklist = setmetatable({}, {
501   __index = function(t, name)
502     local list = realcallbacklist[name]
503     local rules = callbackrules[name]
504     if list and rules then
505       local meta = {}
506       for i, entry in ipairs(list) do
507         local t = {value = entry, count = 0, pos = i}
508         meta[entry.identifier], list[i] = t, t
509       end
510       local count = #list
511       local pos = count
512       for i, rule in ipairs(rules) do
513         local rule = rules[i]
514         local pre, post = meta[rule[1]], meta[rule[2]]
515         if pre and post then
516           if rule.type then
517             if not rule.hidden then
518               assert(rule.type == 'incompatible-warning' and luatexbase_warning
519                 or rule.type == 'incompatible-error' and luatexbase_error)(
520                 "Incompatible functions \"" .. rule[1] .. "\" and \"" .. rule[2]
521                 .. "\" specified for callback \"" .. name .. "\".")
522               rule.hidden = true
523             end
524           else
525             local post_count = post.count
526             post.count = post_count+1
527             if post_count == 0 then
528               local post_pos = post.pos
529               if post_pos ~= pos then
530                 local new_post_pos = list[pos]
531                 new_post_pos.pos = post_pos
532                 list[post_pos] = new_post_pos
533               end
534               list[pos] = nil
535               pos = pos - 1
536             end
537             pre[#pre+1] = post
538           end
539         end
540       end
541       for i=1, count do -- The actual sort begins
542         local current = list[i]
543         if current then
544           meta[current.value.identifier] = nil
545           for j, cur in ipairs(current) do
546             local count = cur.count
547             if count == 1 then
548               pos = pos + 1
549               list[pos] = cur
550             else
551               cur.count = count - 1

```

```

552         end
553     end
554     list[i] = current.value
555 else
556     -- Cycle occurred. TODO: Show cycle for debugging
557     -- list[i] = ...
558     local remaining = {}
559     for name, entry in next, meta do
560         local value = entry.value
561         list[#list + 1] = entry.value
562         remaining[#remaining + 1] = name
563     end
564     table.sort(remaining)
565     local first_name = remaining[1]
566     for j, name in ipairs(remaining) do
567         local entry = meta[name]
568         list[i + j - 1] = entry.value
569         for _, post_entry in ipairs(entry) do
570             local post_name = post_entry.value.identifier
571             if not remaining[post_name] then
572                 remaining[post_name] = name
573             end
574         end
575     end
576     local cycle = {first_name}
577     local index = 1
578     local last_name = first_name
579     repeat
580         cycle[last_name] = index
581         last_name = remaining[last_name]
582         index = index + 1
583         cycle[index] = last_name
584     until cycle[last_name]
585     local length = index - cycle[last_name] + 1
586     table.move(cycle, cycle[last_name], index, 1)
587     for i=2, length//2 do
588         cycle[i], cycle[length + 1 - i] = cycle[length + 1 - i], cycle[i]
589     end
590     error('Cycle occurred at ' .. table.concat(cycle, ' -> ', 1, length))
591 end
592 end
593 end
594 realcallbacklist[name] = list
595 t[name] = list
596 return list
597 end
598 })

```

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

```

599 local list, data, exclusive, simple, reverselist = 1, 2, 3, 4, 5
600 local types = {
601     list      = list,
602     data      = data,

```

```

603 exclusive = exclusive,
604 simple    = simple,
605 reverselist = reverselist,
606 }

```

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.)

```

607 local callbacktypes = callbacktypes or {
Section 8.2: file discovery callbacks.

```

```

608 find_read_file      = exclusive,
609 find_write_file     = exclusive,
610 find_font_file      = data,
611 find_output_file    = data,
612 find_format_file    = data,
613 find_vf_file        = data,
614 find_map_file       = data,
615 find_enc_file       = data,
616 find_pk_file        = data,
617 find_data_file      = data,
618 find_opentype_file  = data,
619 find_truetype_file  = data,
620 find_type1_file     = data,
621 find_image_file     = data,

622 open_read_file      = exclusive,
623 read_font_file      = exclusive,
624 read_vf_file        = exclusive,
625 read_map_file       = exclusive,
626 read_enc_file       = exclusive,
627 read_pk_file        = exclusive,
628 read_data_file      = exclusive,
629 read_truetype_file  = exclusive,
630 read_type1_file     = exclusive,
631 read_opentype_file  = exclusive,

```

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

```

632 find_cidmap_file    = data,
633 read_cidmap_file    = exclusive,

```

Section 8.3: data processing callbacks.

```

634 process_input_buffer = data,
635 process_output_buffer = data,
636 process_jobname      = data,

```

Section 8.4: node list processing callbacks.

```
637 contribute_filter      = simple,
638 buildpage_filter       = simple,
639 build_page_insert      = exclusive,
640 pre_linebreak_filter    = list,
641 linebreak_filter        = exclusive,
642 append_to_vlist_filter  = exclusive,
643 post_linebreak_filter   = reverselist,
644 hpack_filter            = list,
645 vpack_filter            = list,
646 hpack_quality           = exclusive,
647 vpack_quality           = exclusive,
648 pre_output_filter       = list,
649 process_rule            = exclusive,
650 hyphenate               = simple,
651 ligaturing              = simple,
652 kerning                 = simple,
653 insert_local_par        = simple,
654 % mlist_to_hlist        = exclusive,
655 new_graf                = exclusive,
```

Section 8.5: information reporting callbacks.

```
656 pre_dump               = simple,
657 start_run               = simple,
658 stop_run                = simple,
659 start_page_number       = simple,
660 stop_page_number        = simple,
661 show_error_hook         = simple,
662 show_warning_message    = simple,
663 show_error_message      = simple,
664 show_lua_error_hook     = simple,
665 start_file              = simple,
666 stop_file               = simple,
667 call_edit               = simple,
668 finish_synctex          = simple,
669 wrapup_run              = simple,
```

Section 8.6: PDF-related callbacks.

```
670 finish_pdffile         = data,
671 finish_pdfpage          = data,
672 page_objnum_provider    = data,
673 page_order_index        = data,
674 process_pdf_image_content = data,
```

Section 8.7: font-related callbacks.

```
675 define_font            = exclusive,
676 glyph_info              = exclusive,
677 glyph_not_found         = exclusive,
678 glyph_stream_provider   = exclusive,
679 make_extensible         = exclusive,
680 font_descriptor_objnum_provider = exclusive,
681 input_level_string       = exclusive,
682 provide_charproc_data   = exclusive,
683 }
684 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 through 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.

```

685 local shared_callbacks = {
686   mlist_to_hlist = {
687     callback = "mlist_to_hlist",
688     count = 0,
689     handler = nil,
690   },
691 }
692 shared_callbacks.pre_mlist_to_hlist_filter = shared_callbacks.mlist_to_hlist
693 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.

```

694 local callback_register = callback_register or callback.register
695 function callback.register()
696   luatexbase_error("Attempt to use callback.register() directly\n")
697 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 are called with a node list head as the first argument and 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 for the following cases. If a function returns `false`, then `false` is immediately returned and

the following functions are *not* called. If a function returns `true`, then the same head is passed to the next function. If all functions return `true`, then the original head 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.

```

698 local function data_handler(name)
699   return function(data, ...)
700     for _,i in ipairs(callbacklist[name]) do
701       data = i.func(data,...)
702     end
703     return data
704   end
705 end

```

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

```

706 local function data_handler_default(value)
707   return value
708 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.

```

709 local function exclusive_handler(name)
710   return function(...)
711     return callbacklist[name][1].func(...)
712   end
713 end

```

Handler for **list** callbacks.

```

714 local function list_handler(name)
715   return function(head, ...)
716     local ret
717     for _,i in ipairs(callbacklist[name]) do
718       ret = i.func(head, ...)
719       if ret == false then
720         luatexbase_warning(
721           "Function '" .. i.identifier .. "' returned false\n"
722           .. "in callback '" .. name .. "'")
723       )
724       return false
725     end
726     if ret ~= true then
727       head = ret
728     end
729   end
730   return head
731 end
732 end

```

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

```
733 local function list_handler_default(head)
734 return head
735 end
```

Handler for `reverselist` callbacks.

```
736 local function reverselist_handler(name)
737   return function(head, ...)
738     local ret
739     local callbacks = callbacklist[name]
740     for i = #callbacks, 1, -1 do
741       local cb = callbacks[i]
742       ret = cb.func(head, ...)
743       if ret == false then
744         luatexbase_warning(
745           "Function '" .. cb.identifier .. "' returned false\n"
746           .. "in callback '" .. name .. "'")
747       )
748       return false
749     end
750     if ret ~= true then
751       head = ret
752     end
753   end
754   return head
755 end
756 end
```

Handler for `simple` callbacks.

```
757 local function simple_handler(name)
758   return function(...)
759     for _,i in ipairs(callbacklist[name]) do
760       i.func(...)
761     end
762   end
763 end
```

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

```
764 local function simple_handler_default()
765 end
```

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

```
766 local handlers = {
767   [data] = data_handler,
768   [exclusive] = exclusive_handler,
769   [list] = list_handler,
770   [reverselist] = reverselist_handler,
771   [simple] = simple_handler,
772 }
773 local defaults = {
774   [data] = data_handler_default,
775   [exclusive] = nil,
776   [list] = list_handler_default,
777   [reverselist] = list_handler_default,
```

```

778 [simple]      = simple_handler_default,
779 }

```

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.

```

780 local user_callbacks_defaults = {}

```

`create_callback` The allocator itself.

```

781 local function create_callback(name, ctype, default)
782   local ctype_id = types[ctype]
783   if not name or name == ""
784   or not ctype_id
785   then
786     luatexbase_error("Unable to create callback:\n" ..
787                       "valid callback name and type required")
788   end
789   if callbacktypes[name] then
790     luatexbase_error("Unable to create callback '" .. name ..
791                       "':\ncallback is already defined")
792   end
793   default = default or defaults[ctype_id]
794   if not default then
795     luatexbase_error("Unable to create callback '" .. name ..
796                       "':\ndefault is required for '" .. ctype ..
797                       "' callbacks")
798   elseif type (default) ~= "function" then
799     luatexbase_error("Unable to create callback '" .. name ..
800                       "':\ndefault is not a function")
801   end
802   user_callbacks_defaults[name] = default
803   callbacktypes[name] = ctype_id
804 end
805 luatexbase.create_callback = create_callback

```

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

```

806 local function call_callback(name,...)
807   if not name or name == "" then
808     luatexbase_error("Unable to create callback:\n" ..
809                       "valid callback name required")
810   end
811   if user_callbacks_defaults[name] == nil then
812     luatexbase_error("Unable to call callback '" .. name
813                       .. "':\nunknown or empty")
814   end
815   local l = callbacklist[name]
816   local f
817   if not l then
818     f = user_callbacks_defaults[name]
819   else
820     f = handlers[callbacktypes[name]](name)

```

```

821 end
822 return f(...)
823 end
824 luatexbase.call_callback=call_callback

```

`in_callback` Look for a function identifier in a callback.

```

825 local function in_callback(name, identifier)
826   if not name
827     or name == ""
828     or not realcallbacklist[name]
829     or not callbacktypes[name]
830     or not identifier then
831     return false
832   end
833   for _, i in pairs(realcallbacklist[name]) do
834     if i.identifier == identifier then
835       return true
836     end
837   end
838   return false
839 end
840 luatexbase.in_callback = in_callback

```

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

```

841 local function add_to_callback(name, func, identifier)
842   if not name or name == "" then
843     luatexbase_error("Unable to register callback:\n" ..
844                       "valid callback name required")
845   end
846   if not callbacktypes[name] or
847     type(func) ~= "function" or
848     not identifier or
849     identifier == "" then
850     luatexbase_error(
851       "Unable to register callback.\n\n"
852       .. "Correct usage:\n"
853       .. "add_to_callback(<callback>, <function>, <identifier>)"
854     )
855   end

```

Check if the specified identifier is already registered. This used to be allowed but will become an error in a future release.

```

856   if in_callback(name, identifier) then
857     luatexbase_warning(string_format(
858       "The identifier %s is already in use in callback %s.\n\z
859       Please change the identifier of the newly registered callback \z
860       to avoid errors in the next LaTeX release", identifier, name))
861   end

```

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

```

862   local l = realcallbacklist[name]
863   if l == nil then
864     l = { }
865     realcallbacklist[name] = l

```

Handle count for shared engine callbacks.

```
866     local shared = shared_callbacks[name]
867     if shared then
868         shared.count = shared.count + 1
869         if shared.count == 1 then
870             callback_register(shared.callback, shared.handler)
871         end
872     end
```

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

```
872     elseif user_callbacks_defaults[name] == nil then
873         callback_register(name, handlers[callbacktypes[name]](name))
874     end
875 end
```

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

```
876 local f = {
877     func      = func,
878     identifier = identifier,
879 }
880 if callbacktypes[name] == exclusive then
881     if #l == 1 then
882         luatexbase_error(
883             "Cannot add second callback to exclusive function\n" ..
884             name .. "'")
885     end
886 end
887 table.insert(l, f)
888 callbacklist[name] = nil
```

Keep user informed.

```
889 luatexbase_log(
890     "Inserting '" .. identifier .. "' in '" .. name .. "'")
891 )
892 end
893 luatexbase.add_to_callback = add_to_callback
```

declare_callback_rule Add an ordering constraint between two callback implementations

```
894 local function declare_callback_rule(name, desc1, relation, desc2)
895     if not callbacktypes[name] or
896         not desc1 or not desc2 or
897         desc1 == "" or desc2 == "" then
898         luatexbase_error(
899             "Unable to create ordering constraint. "
900             .. "Correct usage:\n"
901             .. "declare_callback_rule(<callback>, <identifier_a>, <identifier_b>)"
902         )
903     end
904     if relation == 'before' then
905         relation = nil
906     elseif relation == 'after' then
907         desc2, desc1 = desc1, desc2
908         relation = nil
909     elseif relation == 'incompatible-warning' or relation == 'incompatible-error' then
910     elseif relation == 'unrelated' then
```

```

911 else
912     luatexbase_error(
913         "Unknown relation type in declare_callback_rule"
914     )
915 end
916 callbacklist[name] = nil
917 local rules = callbackrules[name]
918 if rules then
919     for i, rule in ipairs(rules) do
920         if rule[1] == desc1 and rule[2] == desc2 or rule[1] == desc2 and rule[2] == desc1 then
921             if relation == 'unrelated' then
922                 table.remove(rules, i)
923             else
924                 rule[1], rule[2], rule.type = desc1, desc2, relation
925             end
926             return
927         end
928     end
929     if relation ~= 'unrelated' then
930         rules[#rules + 1] = {desc1, desc2, type = relation}
931     end
932 elseif relation ~= 'unrelated' then
933     callbackrules[name] = {{desc1, desc2, type = relation}}
934 end
935 end
936 luatexbase.declare_callback_rule = declare_callback_rule

```

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

```

937 local function remove_from_callback(name, identifier)
938     if not name or name == "" then
939         luatexbase_error("Unable to remove function from callback:\n" ..
940             "valid callback name required")
941     end
942     if not callbacktypes[name] or
943         not identifier or
944         identifier == "" then
945         luatexbase_error(
946             "Unable to remove function from callback.\n\n"
947             .. "Correct usage:\n"
948             .. "remove_from_callback(<callback>, <identifier>)"
949         )
950     end
951     local l = realcallbacklist[name]
952     if not l then
953         luatexbase_error(
954             "No callback list for '" .. name .. "'\n")
955     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.

```

956     local index = false
957     for i,j in ipairs(l) do
958         if j.identifier == identifier then
959             index = i

```

```

960     break
961   end
962 end
963 if not index then
964   luatexbase_error(
965     "No callback '" .. identifier .. "' registered for '" ..
966     name .. "'\n")
967 end
968 local cb = l[index]
969 table.remove(l, index)
970 luatexbase_log(
971   "Removing '" .. identifier .. "' from '" .. name .. "'."
972 )
973 if #l == 0 then
974   realcallbacklist[name] = nil
975   callbacklist[name] = nil
976   local shared = shared_callbacks[name]
977   if shared then
978     shared.count = shared.count - 1
979     if shared.count == 0 then
980       callback_register(shared.callback, nil)
981     end
982   elseif user_callbacks_defaults[name] == nil then
983     callback_register(name, nil)
984   end
985 end
986 return cb.func, cb.identifier
987 end
988 luatexbase.remove_from_callback = remove_from_callback

```

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

```

989 local function disable_callback(name)
990   if(realcallbacklist[name] == nil) then
991     callback_register(name, false)
992   else
993     luatexbase_error("Callback list for " .. name .. " not empty")
994   end
995 end
996 luatexbase.disable_callback = disable_callback

```

callback_identifiers List the identifiers of functions registered for the given callback. This will sort the list if necessary.

```

997 local function callback_identifiers (name)
998   local d = {}
999   if not name
1000     or name == ""
1001     or not realcallbacklist[name]
1002     or not callbacktypes[name]
1003   then
1004     return d
1005   else
1006     for k, i in pairs(callbacklist[name]) do
1007       d[k] = i.identifier

```

```

1008     end
1009 end
1010 return d
1011 end
1012 luatexbase.callback_descriptions =callback_identifiers
1013 luatexbase.callback_identifiers = callback_identifiers

```

uninstall Unlike at the T_EX 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!

```

1014 local function uninstall()
1015   module_info(
1016     "luatexbase",
1017     "Uninstalling kernel luatexbase code"
1018   )
1019   callback.register = callback_register
1020   luatexbase = nil
1021 end
1022 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.

```

1023 create_callback('pre_mlist_to_hlist_filter', 'list')
1024 create_callback('mlist_to_hlist', 'exclusive', node.mlist_to_hlist)
1025 create_callback('post_mlist_to_hlist_filter', 'reverselist')
1026 function shared_callbacks.mlist_to_hlist.handler(head, display_type, need_penalties)
1027   local current = call_callback("pre_mlist_to_hlist_filter", head, display_type, need_penalties)
1028   if current == false then
1029     flush_list(head)
1030     return nil
1031   end
1032   current = call_callback("mlist_to_hlist", current, display_type, need_penalties)
1033   local post = call_callback("post_mlist_to_hlist_filter", current, display_type, need_penalties)
1034   if post == false then
1035     flush_list(current)
1036     return nil
1037   end
1038   return post
1039 end

```

```

1040 </lua>

```

Reset the catcode of @.

```

1041 <tex>\catcode'\@=\etacatcode\relax

```