

# lualatex.dtx

## (LuaTeX-specific support)

David Carlisle and Joseph Wright\*

2022/10/03

## Contents

<b>1 Overview</b>	<b>2</b>
<b>2 Core TeX functionality</b>	<b>2</b>
<b>3 Plain TeX interface</b>	<b>3</b>
<b>4 Lua functionality</b>	<b>3</b>
4.1 Allocators in Lua . . . . .	3
4.2 Lua access to TeX register numbers . . . . .	4
4.3 Module utilities . . . . .	5
4.4 Callback management . . . . .	5
<b>5 Implementation</b>	<b>6</b>
5.1 Minimum LuaTeX version . . . . .	6
5.2 Older L <sup>A</sup> TeX/Plain TeX setup . . . . .	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 . . . . .	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 2<sub>ε</sub> kernel level plus as a loadable file which can be used with plain TeX and L<sup>A</sup>T<sub>Ε</sub>X.

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

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

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

(`\count 256` is used for `\newmarks` allocation and `\count 257` is used for `\newXeTeXintercharclass` with XeTeX, with code defined in `ltfinal.dtx`). With any L<sup>A</sup>T<sub>Ε</sub>X 2<sub>ε</sub> kernel from 2015 onward these registers are part of the block in the extended area reserved by the kernel (prior to 2015 the L<sup>A</sup>T<sub>Ε</sub>X 2<sub>ε</sub> kernel did not provide any functionality for the extended allocation area).

## 2 Core TeX functionality

The commands defined here are defined for possible inclusion in a future L<sup>A</sup>T<sub>Ε</sub>X format, however also extracted to the file `ltluatex.tex` which may be used with older L<sup>A</sup>T<sub>Ε</sub>X formats, and with plain TeX.

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

	Allocates a number for Lua bytecode register, indexed from 1.
<code>\newluachunkname</code>	<code>newluachunkname{⟨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{ltuatex}`. This inputs `ltuatex.tex` which inputs `etex.src` (or `etex.sty` if used with L<sup>A</sup>T<sub>E</sub>X) if it is not already input, and then defines some internal commands to allow the `luatex` interface to be defined.

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

## 4 Lua functionality

### 4.1 Allocators in Lua

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

These functions all require access to a named TeX count register to manage their allocations. The standard names are those defined above for access from TeX, *e.g.* “e@alloc@attribute@count, but these can be adjusted by defining the variable `<type>_count_name` before loading `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 LuaLaTeX then the following would be produced in the log and terminal output.

```

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

```

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

### 4.3 Module utilities

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

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

back function with  $\langle description \rangle$  from the  $\langle callback \rangle$ . The removed function and its description are returned as the results of this function.

**in\_callback** `luatexbase.in_callback( $\langle callback \rangle$ ,  $\langle description \rangle$ )` Checks if the  $\langle description \rangle$  matches one of the functions added to the list for the  $\langle callback \rangle$ , returning a boolean value.

**disable\_callback** `luatexbase.disable_callback( $\langle callback \rangle$ )` Sets the  $\langle callback \rangle$  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( $\langle name \rangle$ ,  $\langle type \rangle$ ,  $\langle default \rangle$ )` Defines a user defined callback. The last argument is a default function or `false`.

**call\_callback** `luatexbase.call_callback( $\langle name \rangle$ , ...)` Calls a user defined callback with the supplied arguments.

**declare\_callback\_rule** `luatexbase.declare_callback_rule( $\langle name \rangle$ ,  $\langle first \rangle$ ,  $\langle relation \rangle$ ,  $\langle second \rangle$ )` Adds an ordering constraint between two callback functions for callback  $\langle name \rangle$ .

The kind of constraint added depends on  $\langle relation \rangle$ :

**before** The callback function with description  $\langle first \rangle$  will be executed before the function with description  $\langle second \rangle$ .

**after** The callback function with description  $\langle first \rangle$  will be executed after the function with description  $\langle second \rangle$ .

**incompatible-warning** When both a callback function with description  $\langle first \rangle$  and with description  $\langle second \rangle$  is registered, then a warning is printed when the callback is executed.

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

**unrelated** Any previously declared callback rule between  $\langle first \rangle$  and  $\langle second \rangle$  gets disabled.

Every call to `declare_callback_rule` with a specific callback  $\langle name \rangle$  and descriptions  $\langle first \rangle$  and  $\langle second \rangle$  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  $\langle *2ekernel | tex | latexrelease \rangle$ 
2  $\langle 2ekernel | latexrelease \rangle \backslash ifx \backslash directlua \backslash @undefined \backslash else$ 
```

### 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 `ltdefns.dtx` have to be defined here because `ltdefns.dtx` is not loaded yet when `ltluatex.dtx` is executed.

```

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

```

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

```

13 <*tex>

```

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

```

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

```

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

```

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

```

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

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

2015-07-13 higher range in `luatex`.

```

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

```

`luatex/xetex` also allow more math fam.

```

30 \edef \et@xmaxfam {\ifx\Umathcode\@undefined\sixt@@n\else\@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 lAuATeX.)
38 % \outer\def\newfam{\alloc@8\fam\chardef\et@xmaxfam}

    End of proposed changes to etex.src

```

### 5.2.2 luatex specific settings

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

```

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

```

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

```

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

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

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

```

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

```

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

```



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

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

### 5.3 Attributes

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

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

`\setattribute` Handy utilities.

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

### 5.4 Category code tables

`\newcatcodetable` Category code tables are allocated with a limit half of that used by LuaT<sub>E</sub>X for everything else. At the end of allocation there needs to be an initialization step. Table 0 is already taken (it's the global one for current use) so the allocation starts at 1.

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

```

104         \expandafter\setrange\catcode\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#2\fi
137         \if M#2\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\@alloc@bytecode@count\@undefined
198   \countdef\@alloc@bytecode@count=262
199   \@alloc@bytecode@count=\z@
200 \fi
201 \def\newluabytcode#1{%
202   \@alloc@luabytcode\@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   {\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  $\text{\TeX}$  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 <2kernel>\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("lualatex")}
232 \endgroup
233 <2kernel>}
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\fontencoding
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\fontencoding\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 <2ekernel | latexrelease>\fi
286 </2ekernel | tex | latexrelease>

```

## 5.10 Lua module preliminaries

```
287 <*lua>
```

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

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

```

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

```

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

```

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

296 local luatexbase_warning
297 local luatexbase_error

```

## 5.11 Lua module utilities

### 5.11.1 Module tracking

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

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

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

```

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

```

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

```

302 local function provides_module(info)
303   if not (info and info.name) then
304     luatexbase_error("Missing module name for provides_module")
305   end
306   local function spaced(text)
307     return text and (" " .. text) or ""
308   end
309   luatexbase_log(
310     "Lua module: " .. info.name
311     .. spaced(info.date)
312     .. spaced(info.version)

```

```

313         .. spaced(info.description)
314     )
315     modules[info.name] = info
316 end
317 luatexbase.provides_module = provides_module

```

### 5.11.2 Module messages

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

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

```

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

```

module\_info Write messages.

```

module_warning 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<sub>E</sub>X level into a Lua table: from version 0.80, LuaT<sub>E</sub>X makes that easy.

```

363 local luaregisterbasetable = { }
364 local registermap = {
365   attributzero = "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<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.

```

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             tex_count[luafunction_count_name] + 1)
479 if tex_count[luafunction_count_name] > 65534 then
480     luatexbase_error("No room for a new luafunction register")
481 end
482 luatexbase_log("Lua function " .. (name or "") .. " = " ..
483             tex_count[luafunction_count_name])
484 return tex_count[luafunction_count_name]
485 end
486 luatexbase.new_luafunction = new_luafunction

```

## 5.18 Lua callback management

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

### 5.18.1 Housekeeping

The main table: keys are callback names, and values are the associated lists of functions. More precisely, the entries in the list are tables holding the actual function as `func` and the identifying description as `description`. Only callbacks with a non-empty list of functions have an entry in this list.

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.

```

487 local realcallbacklist = {}
488 local callbackrules = {}
489 local callbacklist = setmetatable({}, {
490     __index = function(t, name)
491         local list = realcallbacklist[name]
492         local rules = callbackrules[name]
493         if list and rules then
494             local meta = {}
495             for i, entry in ipairs(list) do
496                 local t = {value = entry, count = 0, pos = i}
497                 meta[entry.description], list[i] = t, t
498             end
499             local count = #list
500             local pos = count
501             for i, rule in ipairs(rules) do
502                 local rule = rules[i]
503                 local pre, post = meta[rule[1]], meta[rule[2]]
504                 if pre and post then
505                     if rule.type then
506                         if not rule.hidden then
507                             assert(rule.type == 'incompatible-warning' and luatexbase_warning

```

```

508         or rule.type == 'incompatible-error' and luatexbase_error)(
509             "Incompatible functions \" .. rule[1] .. "\" and \" .. rule[2]
510             .. "\" specified for callback \" .. name .. "\".")
511         rule.hidden = true
512     end
513 else
514     local post_count = post.count
515     post.count = post_count+1
516     if post_count == 0 then
517         local post_pos = post.pos
518         if post_pos ~= pos then
519             local new_post_pos = list[pos]
520             new_post_pos.pos = post_pos
521             list[post_pos] = new_post_pos
522         end
523         list[pos] = nil
524         pos = pos - 1
525     end
526     pre[#pre+1] = post
527 end
528 end
529 end
530 for i=1, count do -- The actual sort begins
531     local current = list[i]
532     if current then
533         meta[current.value.description] = nil
534         for j, cur in ipairs(current) do
535             local count = cur.count
536             if count == 1 then
537                 pos = pos + 1
538                 list[pos] = cur
539             else
540                 cur.count = count - 1
541             end
542         end
543         list[i] = current.value
544     else
545         -- Cycle occurred. TODO: Show cycle for debugging
546         -- list[i] = ...
547         local remaining = {}
548         for name, entry in next, meta do
549             local value = entry.value
550             list[#list + 1] = entry.value
551             remaining[#remaining + 1] = name
552         end
553         table.sort(remaining)
554         local first_name = remaining[1]
555         for j, name in ipairs(remaining) do
556             local entry = meta[name]
557             list[i + j - 1] = entry.value
558             for _, post_entry in ipairs(entry) do
559                 local post_name = post_entry.value.description
560                 if not remaining[post_name] then
561                     remaining[post_name] = name

```

```

562         end
563     end
564 end
565 local cycle = {first_name}
566 local index = 1
567 local last_name = first_name
568 repeat
569     cycle[last_name] = index
570     last_name = remaining[last_name]
571     index = index + 1
572     cycle[index] = last_name
573 until cycle[last_name]
574 local length = index - cycle[last_name] + 1
575 table.move(cycle, cycle[last_name], index, 1)
576 for i=2, length//2 do
577     cycle[i], cycle[length + 1 - i] = cycle[length + 1 - i], cycle[i]
578 end
579 error('Cycle occurred at ' .. table.concat(cycle, ' -> ', 1, length))
580 end
581 end
582 end
583 realcallbacklist[name] = list
584 t[name] = list
585 return list
586 end
587 })

```

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

```

588 local list, data, exclusive, simple, reverselist = 1, 2, 3, 4, 5
589 local types = {
590     list      = list,
591     data      = data,
592     exclusive = exclusive,
593     simple    = simple,
594     reverselist = reverselist,
595 }

```

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.)

```

596 local callbacktypes = callbacktypes or {

```

Section 8.2: file discovery callbacks.

```

597 find_read_file      = exclusive,
598 find_write_file     = exclusive,
599 find_font_file      = data,
600 find_output_file    = data,
601 find_format_file    = data,
602 find_vf_file        = data,
603 find_map_file       = data,
604 find_enc_file       = data,
605 find_pk_file        = data,
606 find_data_file      = data,
607 find_opentype_file  = data,
608 find_truetype_file  = data,
609 find_type1_file     = data,
610 find_image_file     = data,

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

```

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

```

621 find_cidmap_file    = data,
622 read_cidmap_file    = exclusive,

```

Section 8.3: data processing callbacks.

```

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

```

Section 8.4: node list processing callbacks.

```

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

```

Section 8.5: information reporting callbacks.

```

645 pre_dump           = simple,
646 start_run          = simple,
647 stop_run           = simple,
648 start_page_number  = simple,
649 stop_page_number   = simple,
650 show_error_hook    = simple,
651 show_warning_message = simple,
652 show_error_message  = simple,
653 show_lua_error_hook = simple,
654 start_file          = simple,
655 stop_file           = simple,
656 call_edit           = simple,
657 finish_synctex      = simple,
658 wrapup_run          = simple,

```

Section 8.6: PDF-related callbacks.

```

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

```

Section 8.7: font-related callbacks.

```

664 define_font          = exclusive,
665 glyph_info           = exclusive,
666 glyph_not_found      = exclusive,
667 glyph_stream_provider = exclusive,
668 make_extensible      = exclusive,
669 font_descriptor_objnum_provider = exclusive,
670 input_level_string    = exclusive,
671 provide_charproc_data = exclusive,
672 }
673 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.

```

674 local shared_callbacks = {
675   mlist_to_hlist = {
676     callback = "mlist_to_hlist",
677     count = 0,
678     handler = nil,
679   },
680 }
681 shared_callbacks.pre_mlist_to_hlist_filter = shared_callbacks.mlist_to_hlist
682 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.

```
683 local callback_register = callback_register or callback.register
684 function callback.register()
685   luatexbase_error("Attempt to use callback.register() directly\n")
686 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.

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



Default for user-defined data callbacks without explicit default.

```
695 local function data_handler_default(value)
696   return value
697 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.

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

Handler for list callbacks.

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

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

```
722 local function list_handler_default(head)
723   return head
724 end
```

Handler for reverselist callbacks.

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

```

740         head = ret
741     end
742 end
743 return head
744 end
745 end

```

Handler for simple callbacks.

```

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

```

Default for user-defined simple callbacks without explicit default.

```

753 local function simple_handler_default()
754 end

```

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

```

755 local handlers = {
756     [data] = data_handler,
757     [exclusive] = exclusive_handler,
758     [list] = list_handler,
759     [reverselist] = reverselist_handler,
760     [simple] = simple_handler,
761 }
762 local defaults = {
763     [data] = data_handler_default,
764     [exclusive] = nil,
765     [list] = list_handler_default,
766     [reverselist] = list_handler_default,
767     [simple] = simple_handler_default,
768 }

```

### 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.

```

769 local user_callbacks_defaults = {}

```

`create_callback` The allocator itself.

```

770 local function create_callback(name, ctype, default)
771     local ctype_id = types[ctype]
772     if not name or name == ""
773     or not ctype_id
774     then
775         luatexbase_error("Unable to create callback:\n" ..
776             "valid callback name and type required")
777     end
778     if callbacktypes[name] then
779         luatexbase_error("Unable to create callback '" .. name ..

```

```

780             "'':\ncallback is already defined")
781     end
782     default = default or defaults[ctype_id]
783     if not default then
784         luatexbase_error("Unable to create callback '" .. name ..
785             "'':\ndefault is required for '" .. ctype ..
786             "' callbacks")
787     elseif type (default) ~= "function" then
788         luatexbase_error("Unable to create callback '" .. name ..
789             "'':\ndefault is not a function")
790     end
791     user_callbacks_defaults[name] = default
792     callbacktypes[name] = ctype_id
793 end
794 luatexbase.create_callback = create_callback

```

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

```

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

```

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

```

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

```

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

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

Handle count for shared engine callbacks.

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

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

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

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

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

Keep user informed.

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

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

```
861 local function declare_callback_rule(name, desc1, relation, desc2)
862     if not callbacktypes[name] or
863         not desc1 or not desc2 or
864         desc1 == "" or desc2 == "" then
865         luatexbase_error(
866             "Unable to create ordering constraint. "
867             .. "Correct usage:\n"
868             .. "declare_callback_rule(<callback>, <description_a>, <description_b>)"
869         )
870     end
```

```

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

```

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

```

904 local function remove_from_callback(name, description)
905     if not name or name == "" then
906         luatexbase_error("Unable to remove function from callback:\n" ..
907             "valid callback name required")
908     end
909     if not callbacktypes[name] or
910         not description or
911         description == "" then
912         luatexbase_error(
913             "Unable to remove function from callback.\n\n"
914             .. "Correct usage:\n"
915             .. "remove_from_callback(<callback>, <description>)"
916         )
917     end
918     local l = realcallbacklist[name]
919     if not l then
920         luatexbase_error(
921             "No callback list for '" .. name .. "'\n")
922     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.

```

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

```

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

```

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

```

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

```
972 local function disable_callback(name)
973   if(realcallbacklist[name] == nil) then
974     callback_register(name, false)
975   else
976     luatexbase_error("Callback list for " .. name .. " not empty")
977   end
978 end
979 luatexbase.disable_callback = disable_callback
```

`callback_descriptions` List the descriptions of functions registered for the given callback. This will sort the list if necessary.

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

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

```
996 local function uninstall()
997   module_info(
998     "luatexbase",
999     "Uninstalling kernel luatexbase code"
1000  )
1001   callback.register = callback_register
1002   luatexbase = nil
1003 end
1004 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.

```
1005 create_callback('pre_mlist_to_hlist_filter', 'list')
1006 create_callback('mlist_to_hlist', 'exclusive', node.mlist_to_hlist)
1007 create_callback('post_mlist_to_hlist_filter', 'list')
1008 function shared_callbacks.mlist_to_hlist.handler(head, display_type, need_penalties)
1009   local current = call_callback("pre_mlist_to_hlist_filter", head, display_type, need_penalties)
1010   if current == false then
1011     flush_list(head)
1012     return nil
```

```

1013 end
1014 current = call_callback("mlist_to_hlist", current, display_type, need_penalties)
1015 local post = call_callback("post_mlist_to_hlist_filter", current, display_type, need_penal
1016 if post == false then
1017     flush_list(current)
1018     return nil
1019 end
1020 return post
1021 end

1022  $\langle$ /lua $\rangle$ 

Reset the catcode of @.
1023  $\langle$ tex $\rangle$ \catcode'\@=\etatcatcode\relax

```