

# lualuatex.dtx

## (LuaTeX-specific support)

David Carlisle and Joseph Wright\*

2025/12/23

## 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><br>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><br>Returns an allocation number for the <code>⟨attribute⟩</code> , indexed from 1. The attribute will be initialised with the marker value <code>-"7FFFFFFF</code> ('unset'). The attribute allocation sequence is shared with the T <sub>E</sub> X code but this function does <i>not</i> define a token using <code>\attributedef</code> . The attribute name is recorded in the <code>attributes</code> table. A metatable is provided so that the table syntax can be used consistently for attributes declared in T <sub>E</sub> X or Lua. |
| <code>new_whatsit</code>     | <code>luatexbase.new_whatsit(⟨whatsit⟩)</code><br>Returns an allocation number for the custom <code>⟨whatsit⟩</code> , indexed from 1.   |
| <code>new_bytecode</code>    | <code>luatexbase.new_bytecode(⟨bytecode⟩)</code><br>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><br>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><br>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 T<sub>E</sub>X count register to manage their allocations. The standard names are those defined above for access from T<sub>E</sub>X, *e.g.* “e@alloc@attribute@count, but these can be adjusted by defining the variable `<type>_count_name` before loading `ltluatex.lua`, for example

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

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

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

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

Sometimes (notably in the case of Lua attributes) it is necessary to access a register *by number* that has been allocated by T<sub>E</sub>X. This package provides a function to look up the relevant number using LuaT<sub>E</sub>X’s internal tables. After for example `\newattribute\myattrib`, `\myattrib` would be defined by (say) `\myattrib=\attribute15`. `luatexbase.registernumber("myattrib")` would then return the register number, 15 in this case. If the string passed as argument does not correspond to a token defined by `\attributedef`, `\countdef` or similar commands, the Lua value `false` is returned.

As an example, consider the input:

```
\newcommand\test[1]{%
\typeout{#1: \expandafter\meaning\csname#1\endcsname^^J
\space\space\space\space
\directlua{tex.write(luatexbase.registernumber("#1") or "bad input")}}%
}

\test{undefinedrubbish}

\test{space}

\test{hbox}

\test{@MM}

\test{@tempdima}
\test{@tempdimb}

\test{strutbox}

\test{sixt@@n}

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

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

```
undefinedrubbish: \relax
```

```

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

```

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

### 4.3 Module utilities

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

This function is used by modules to identify themselves; the `info` should be a table containing information about the module. The required field `name` must contain the name of the module. It is recommended to provide a field `date` in the usual L<sup>A</sup>T<sub>E</sub>X format `yyyy/mm/dd`. Optional fields `version` (a string) and `description` may be used if present. This information will be recorded in the log. Other fields are ignored. If the `version` begins with a digit, a `v` will be added at the start in the log.

```

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 `ltdfn.dtx` have to be defined here because `ltdfn.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.extraprimatives("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 luatex.)
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<sup>A</sup>T<sub>E</sub>X (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\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

`\newluaopcode` 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\newluaopcode#1{%
202   \@alloc@luaopcode\@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\@alloc@attribute@count\@undefined
237 <latexrelease>\let\newattribute\@undefined
238 <latexrelease>\let\setattribute\@undefined
239 <latexrelease>\let\unsetattribute\@undefined
240 <latexrelease>\let\@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\@alloc@luafunction@count\@undefined
247 <latexrelease>\let\newluafunction\@undefined
248 <latexrelease>\let\@alloc@luafunction@count\@undefined
249 <latexrelease>\let\newwhatsit\@undefined
250 <latexrelease>\let\@alloc@whatsit@count\@undefined
251 <latexrelease>\let\newluabytecode\@undefined
252 <latexrelease>\let\@alloc@bytecode@count\@undefined
253 <latexrelease>\let\newluachunkname\@undefined
254 <latexrelease>\let\@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_gsub      = string.gsub
290 local tex_count        = tex.count
291 local tex_setcount      = tex.setcount
292 local texio_write_nl    = texio.write_nl
293 local flush_list        = node.flush_list

294 local luatexbase_warning
295 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.

```
296 local modules = modules or { }
```

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

```

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

```

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

```

300 local function provides_module(info)
301   if not (info and info.name) then
302     luatexbase_error("Missing module name for provides_module")
303   end
304   local function spaced(text)
305     return text and (" " .. text) or ""
306   end
307   luatexbase_log(
308     "Lua module: " .. info.name
309     .. spaced(info.date)
310     .. spaced(info.version and string_gsub(info.version or "", "%d"), "v%1")
311     .. spaced(info.description)
312   )

```

```

313 modules[info.name] = info
314 end
315 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.

```

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

```

module\_info Write messages.

```

module_warning 343 local function module_info(mod, text)
module_error    344   texio_write_nl("log", msg_format(mod, "Info", text))
                 345 end
                 346 luatexbase.module_info = module_info
                 347 local function module_warning(mod, text)
                 348   texio_write_nl("term and log",msg_format(mod, "Warning", text))
                 349 end
                 350 luatexbase.module_warning = module_warning
                 351 local function module_error(mod, text)
                 352   error(msg_format(mod, "Error", text))
                 353 end

```

```
354 luatexbase.module_error = module_error
```

Dedicated versions for the rest of the code here.

```
355 function luatexbase_warning(text)
356   module_warning("luatexbase", text)
357 end
358 function luatexbase_error(text)
359   module_error("luatexbase", text)
360 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.

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

```
388 local registernumber
389 if luatexversion < 80 then
390   function registernumber(name)
391     local nt = hashtokens[name]
392     if (nt and luaregisterbasetable[nt[1]]) then
393       return nt[2] - luaregisterbasetable[nt[1]]
394     else
```



```

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

```

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

```

### 5.14 Custom whatsit allocation

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

```

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

```

```

439             tex_count[whatsit_count_name])
440 return tex_count[whatsit_count_name]
441 end
442 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.

```

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

```

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

```

472 local luafunction_count_name =
473     luafunction_count_name or "e@alloc@luafunction@count"
474 local function new_luafunction(name)
475     tex_setcount("global", luafunction_count_name,
476         math.max(

```

```

477             #(lua.get_functions_table()),
478             tex_count[luafunction_count_name])
479         + 1)
480 lua.get_functions_table()[tex_count[luafunction_count_name]] = false
481 if tex_count[luafunction_count_name] > 65534 then
482     luatexbase_error("No room for a new luafunction register")
483 end
484 luatexbase_log("Lua function " .. (name or "") .. " = " ..
485               tex_count[luafunction_count_name])
486 return tex_count[luafunction_count_name]
487 end
488 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.

```

489 local realcallbacklist = {}
490 local callbackrules = {}
491 local callbacklist = setmetatable({}, {
492   __index = function(t, name)
493     local list = realcallbacklist[name]
494     local rules = callbackrules[name]
495     if list and rules then
496       local meta = {}
497       for i, entry in ipairs(list) do
498         local t = {value = entry, count = 0, pos = i}
499         meta[entry.description], list[i] = t, t
500       end
501       local count = #list
502       local pos = count
503       for i, rule in ipairs(rules) do
504         local rule = rules[i]
505         local pre, post = meta[rule[1]], meta[rule[2]]
506         if pre and post then
507           if rule.type then

```

```

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

```

```

562         if not remaining[post_name] then
563             remaining[post_name] = name
564         end
565     end
566 end
567 local cycle = {first_name}
568 local index = 1
569 local last_name = first_name
570 repeat
571     cycle[last_name] = index
572     last_name = remaining[last_name]
573     index = index + 1
574     cycle[index] = last_name
575 until cycle[last_name]
576 local length = index - cycle[last_name] + 1
577 table.move(cycle, cycle[last_name], index, 1)
578 for i=2, length//2 do
579     cycle[i], cycle[length + 1 - i] = cycle[length + 1 - i], cycle[i]
580 end
581 error('Cycle occurred at ' .. table.concat(cycle, ' -> ', 1, length))
582 end
583 end
584 end
585 realcallbacklist[name] = list
586 t[name] = list
587 return list
588 end
589 })

```

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

```

590 local list, data, exclusive, simple, reverselist = 1, 2, 3, 4, 5
591 local types = {
592     list = list,
593     data = data,
594     exclusive = exclusive,
595     simple = simple,
596     reverselist = reverselist,
597 }

```

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.)

```

598 local callbacktypes = callbacktypes or {

```

Section 8.2: file discovery callbacks.

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

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

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

```
623 find_cidmap_file   = data,
624 read_cidmap_file    = exclusive,
```

Section 8.3: data processing callbacks.

```
625 process_input_buffer = data,
626 process_output_buffer = data,
627 process_jobname      = data,
```

Section 8.4: node list processing callbacks.

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

```

645 % mlist_to_hlist      = exclusive,
646 new_graf              = exclusive,

```

Section 8.5: information reporting callbacks.

```

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

```

Section 8.6: PDF-related callbacks.

```

661 finish_pdffile        = data,
662 finish_pdfpage        = data,
663 page_objnum_provider  = data,
664 page_order_index      = data,
665 process_pdf_image_content = data,

```

Section 8.7: font-related callbacks.

```

666 define_font           = exclusive,
667 glyph_info            = exclusive,
668 glyph_not_found       = exclusive,
669 glyph_stream_provider = exclusive,
670 make_extensible       = exclusive,
671 font_descriptor_objnum_provider = exclusive,
672 input_level_string    = exclusive,
673 provide_charproc_data = exclusive,
674 }
675 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.

```

676 local shared_callbacks = {
677   mlist_to_hlist = {
678     callback = "mlist_to_hlist",
679     count = 0,
680     handler = nil,
681   },
682 }
683 shared_callbacks.pre_mlist_to_hlist_filter = shared_callbacks.mlist_to_hlist
684 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.

```
685 local callback_register = callback_register or callback.register
686 function callback.register()
687   luatexbase_error("Attempt to use callback.register() directly\n")
688 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.

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



```

695 end
696 end

```

Default for user-defined data callbacks without explicit default.

```

697 local function data_handler_default(value)
698   return value
699 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.

```

700 local function exclusive_handler(name)
701   return function(...)
702     return callbacklist[name][1].func(...)
703   end
704 end

```

Handler for list callbacks.

```

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

```

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

```

724 local function list_handler_default(head)
725   return head
726 end

```

Handler for reverselist callbacks.

```

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

```

```

740     end
741     if ret ~= true then
742         head = ret
743     end
744 end
745 return head
746 end
747 end

```

Handler for simple callbacks.

```

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

```

Default for user-defined simple callbacks without explicit default.

```

755 local function simple_handler_default()
756 end

```

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

```

757 local handlers = {
758     [data]      = data_handler,
759     [exclusive] = exclusive_handler,
760     [list]      = list_handler,
761     [reverselist] = reverselist_handler,
762     [simple]     = simple_handler,
763 }
764 local defaults = {
765     [data]      = data_handler_default,
766     [exclusive] = nil,
767     [list]      = list_handler_default,
768     [reverselist] = list_handler_default,
769     [simple]     = simple_handler_default,
770 }

```

### 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.

```

771 local user_callbacks_defaults = {}

```

`create_callback` The allocator itself.

```

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

```

```

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

```

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

```

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

```

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

```

816 local function add_to_callback(name, func, description)
817     if not name or name == "" then
818         luatexbase_error("Unable to register callback:\n" ..
819             "valid callback name required")
820     end
821     if not callbacktypes[name] or
822         type(func) ~= "function" or
823         not description or
824         description == "" then
825         luatexbase_error(
826             "Unable to register callback.\n\n"
827             .. "Correct usage:\n"
828             .. "add_to_callback(<callback>, <function>, <description>)"
829         )

```

```
830 end
```

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

```
831 local l = realcallbacklist[name]
832 if l == nil then
833     l = { }
834     realcallbacklist[name] = l
```

Handle count for shared engine callbacks.

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

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

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

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

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

Keep user informed.

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

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

```
863 local function declare_callback_rule(name, desc1, relation, desc2)
864     if not callbacktypes[name] or
865         not desc1 or not desc2 or
866         desc1 == "" or desc2 == "" then
867         luatexbase_error(
868             "Unable to create ordering constraint. "
869             .. "Correct usage:\n"
870             .. "declare_callback_rule(<callback>, <description_a>, <description_b>)"
871         )
```

```

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

```

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

```

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

```

```

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

```

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

```

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

```

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

```

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

```

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

```

982 local function callback_descriptions (name)
983   local d = {}
984   if not name
985     or name == ""
986     or not realcallbacklist[name]
987     or not callbacktypes[name]
988   then
989     return d
990   else
991     for k, i in pairs(callbacklist[name]) do
992       d[k] = i.description
993     end
994   end
995   return d
996 end
997 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!

```

998 local function uninstall()
999   module_info(
1000     "luatexbase",
1001     "Uninstalling kernel luatexbase code"
1002   )
1003   callback.register = callback_register
1004   luatexbase = nil
1005 end
1006 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.

```

1007 create_callback('pre_mlist_to_hlist_filter', 'list')
1008 create_callback('mlist_to_hlist', 'exclusive', node.mlist_to_hlist)
1009 create_callback('post_mlist_to_hlist_filter', 'reverselist')
1010 function shared_callbacks.mlist_to_hlist.handler(head, display_type, need_penalties)
1011   local current = call_callback("pre_mlist_to_hlist_filter", head, display_type, need_penalties)
1012   if current == false then
1013     flush_list(head)
1014     return nil

```

```

1015 end
1016 current = call_callback("mlist_to_hlist", current, display_type, need_penalties)
1017 local post = call_callback("post_mlist_to_hlist_filter", current, display_type, need_penal
1018 if post == false then
1019     flush_list(current)
1020     return nil
1021 end
1022 return post
1023 end

1024  $\langle$ /lua $\rangle$ 

Reset the catcode of @.
1025  $\langle$ tex $\rangle$ \catcode'\@=\etatcatcode\relax

```