

lualatex.dtx

(LuaTeX-specific support)

David Carlisle and Joseph Wright*

2017/01/20

Contents

1 Overview	2
2 Core TeX functionality	2
3 Plain TeX interface	3
4 Lua functionality	3
4.1 Allocators in Lua	3
4.2 Lua access to TeX register numbers	3
4.3 Module utilities	5
4.4 Callback management	5
5 Implementation	5
5.1 Minimum LuaTeX version	6
5.2 Older L ^A TeX/Plain TeX setup	6
5.3 Attributes	8
5.4 Category code tables	8
5.5 Named Lua functions	10
5.6 Custom whatsits	10
5.7 Lua bytecode registers	10
5.8 Lua chunk registers	11
5.9 Lua loader	11
5.10 Lua module preliminaries	13
5.11 Lua module utilities	13
5.12 Accessing register numbers from Lua	15
5.13 Attribute allocation	16
5.14 Custom whatsit allocation	16
5.15 Bytecode register allocation	17
5.16 Lua chunk name allocation	17
5.17 Lua callback management	17

*Significant portions of the code here are adapted/simplified from the packages `luatex` and `luatexbase` written by Heiko Oberdiek, Élie Roux, Manuel Pégourié-Gonnar and Philipp Gesang.

1 Overview

LuaTeX adds a number of engine-specific functions to TeX. Several of these require set up that is best done in the kernel or need related support functions. This file provides *basic* support for LuaTeX at the L^AT_ΕX 2_ε kernel level plus as a loadable file which can be used with plain TeX and L^AT_ΕX.

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

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

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

(`\count 256` is used for `\newmarks` allocation and `\count 257` is used for `\newXeTeXintercharclass` with XeTeX, with code defined in `ltfinal.dtx`). With any L^AT_ΕX 2_ε kernel from 2015 onward these registers are part of the block in the extended area reserved by the kernel (prior to 2015 the L^AT_ΕX 2_ε kernel did not provide any functionality for the extended allocation area).

2 Core TeX functionality

The commands defined here are defined for possible inclusion in a future L^AT_ΕX format, however also extracted to the file `ltluatex.tex` which may be used with older L^AT_ΕX formats, and with plain TeX.

<code>\newattribute</code>	<code>\newattribute{<attribute>}</code> Defines a named <code>\attribute</code> , indexed from 1 (<i>i.e.</i> <code>\attribute0</code> is never defined). Attributes initially have the marker value <code>-7FFFFFFF</code> ('unset') set by the engine.
<code>\newcatcodetable</code>	<code>\newcatcodetable{<catcodetable>}</code> Defines a named <code>\catcodetable</code> , indexed from 1 (<code>\catcodetable0</code> is never assigned). A new catcode table will be populated with exactly those values assigned by IniTeX (as described in the LuaTeX manual).
<code>\newluafunction</code>	<code>\newluafunction{<function>}</code> Defines a named <code>\luafunction</code> , indexed from 1. (Lua indexes tables from 1 so <code>\luafunction0</code> is not available).
<code>\newwhatsit</code>	<code>\newwhatsit{<whatsit>}</code> Defines a custom <code>\whatsit</code> , indexed from 1.
<code>\newluabytecode</code>	<code>\newluabytecode{<bytecode>}</code> Allocates a number for Lua bytecode register, indexed from 1.
<code>\newluachunkname</code>	<code>newluachunkname{<chunkname>}</code> Allocates a number for Lua chunk register, indexed from 1. Also enters the name of the register (without backslash) into the <code>lua.name</code> table to be used in stack traces.

<code>\catcodetable@initex</code>	Predefined category code tables with the obvious assignments. Note that the
<code>\catcodetable@string</code>	<code>latex</code> and <code>atletter</code> tables set the full Unicode range to the codes predefined by
<code>\catcodetable@latex</code>	the kernel.
<code>\catcodetable@attribute</code>	<code>\setattribute{⟨attribute⟩}{⟨value⟩}</code>
<code>\unsetattribute</code>	<code>\unsetattribute{⟨attribute⟩}</code>

Set and unset attributes in a manner analogous to `\setlength`. Note that attributes take a marker value when unset so this operation is distinct from setting the value to zero.

3 Plain T_EX interface

The `luatex` interface may be used with plain T_EX using `\input{luatex}`. This inputs `luatex.tex` which inputs `etex.src` (or `etex.sty` if used with L^AT_EX) if it is not already input, and then defines some internal commands to allow the `luatex` interface to be defined.

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

4 Lua functionality

4.1 Allocators in Lua

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

4.2 Lua access to T_EX register numbers

<code>registernumber</code>	<code>luatexbase.registernumber(⟨name⟩)</code> Sometimes (notably in the case of Lua attributes) it is necessary to access a register <i>by number</i> that has been allocated by T _E X. This package provides a function to look up the relevant number using LuaT _E X's internal tables. After for example <code>\newattribute\myattrib</code> , <code>\myattrib</code> would be defined by (say) <code>\myattrib=\attribute15</code> . <code>luatexbase.registernumber("myattrib")</code>
-----------------------------	---

would then return the register number, 15 in this case. If the string passed as argument does not correspond to a token defined by `\attributedef`, `\countdef` or similar commands, the Lua value `false` is returned.

As an example, consider the input:

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

\test{undefinedrubbish}

\test{space}

\test{hbox}

\test{@MM}

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

\test{strutbox}

\test{sixt@@n}

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

If the demonstration code is processed with Lua^AT_EX 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^AT_EX 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^AT_EX's `\PackageError`, `\PackageWarning` and `\PackageInfo` in the way they format the output. No automatic line breaking is done, you may still use `\n` as usual for that, and the name of the package will be prepended to each output line.

Note that `luatexbase.module_error` raises an actual Lua error with `error()`, which currently means a call stack will be dumped. While this may not look pretty, at least it provides useful information for tracking the error down.

4.4 Callback management

`add_to_callback` `luatexbase.add_to_callback(<callback>, <function>, <description>)` Registers the *<function>* into the *<callback>* with a textual *<description>* of the function. Functions are inserted into the callback in the order loaded.

`remove_from_callback` `luatexbase.remove_from_callback(<callback>, <description>)` Removes the callback function with *<description>* from the *<callback>*. The removed function and its description are returned as the results of this function.

`in_callback` `luatexbase.in_callback(<callback>, <description>)` Checks if the *<description>* matches one of the functions added to the list for the *<callback>*, returning a boolean value.

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

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

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

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

5 Implementation

¹ `<*2ekernel | tex | latexrelease>`

```
2 (2ekernel | latexrelease) \ifx\directlua \@undefined \else
```

5.1 Minimum LuaTeX version

LuaTeX has changed a lot over time. In the kernel support for ancient versions is not provided: trying to build a format with a very old binary therefore gives some information in the log and loading stops. The cut-off selected here relates to the tree-searching behaviour of `require()`: from version 0.60, LuaTeX will correctly find Lua files in the `texmf` tree without ‘help’.

```
3 (latexrelease) \IncludeInRelease{2015/10/01}
4 (latexrelease) \newluafunction{LuaTeX}%
5 \ifnum\luatexversion<60 %
6   \wlog{*****}
7   \wlog{* LuaTeX version too old for ltluatex support *}
8   \wlog{*****}
9   \expandafter\endinput
10 \fi
```

5.2 Older L^AT_EX/Plain T_EX setup

```
11 (*tex)
```

Older L^AT_EX formats don’t have the primitives with ‘native’ names: sort that out. If they already exist this will still be safe.

```
12 \directlua{tex.enableprimitives("",tex.extraprimitives("luatex"))}
13 \ifx\@alloc \@undefined
```

In pre-2014 L^AT_EX, or plain T_EX, load `etex.{sty,src}`.

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

```
27 % 2015-07-13 higher range in luatex
28 \edef \et@xmaxregs {\ifx\directlua \@undefined 32768\else 65536\fi}
29 % luatex/xetex also allow more math fam
30 \edef \et@xmaxfam {\ifx\Umathchar \@undefined \sift@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\@ccclvi
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}%

```

Two simple L^AT_EX macros used in `lAlATeX.sty`.

```

66 \long\def\@gobble#1{}
67 \long\def\@firstofone#1{#1}

68 % Fix up allocations not to clash with /etex.src/.

69 \expandafter\csname newcount\endcsname\e@alloc@attribute@count
70 \expandafter\csname newcount\endcsname\e@alloc@ccodetable@count
71 \expandafter\csname newcount\endcsname\e@alloc@luafunction@count
72 \expandafter\csname newcount\endcsname\e@alloc@whatsit@count

```

```

73 \expandafter\csname newcount\endcsname\@alloc@bytecode@count
74 \expandafter\csname newcount\endcsname\@alloc@luachunk@count

```

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

```

75 \fi
76 \</tex>

```

5.3 Attributes

`\newattribute` As is generally the case for the LuaT_EX registers we start here from 1. Notably, some code assumes that `\attribute0` is never used so this is important in this case.

```

77 \ifx\@alloc@attribute@count\@undefined
78   \countdef\@alloc@attribute@count=258
79 \fi
80 \def\newattribute#1{%
81   \@alloc@attribute\attributedef
82   \@alloc@attribute@count\m@ne\@alloc@top#1%
83 }
84 \@alloc@attribute@count=\z@

```

`\setattribute` Handy utilities.

```

\unsetattribute 85 \def\setattribute#1#2{#1=\numexpr#2\relax}
86 \def\unsetattribute#1{#1=-"7FFFFFFF\relax}

```

5.4 Category code tables

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

```

87 \ifx\@alloc@ccodetable@count\@undefined
88   \countdef\@alloc@ccodetable@count=259
89 \fi
90 \def\newcatcodetable#1{%
91   \@alloc@catcodetable\chardef
92   \@alloc@ccodetable@count\m@ne{"8000}#1%
93   \initcatcodetable\allocationnumber
94 }
95 \@alloc@ccodetable@count=\z@

```

`\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 96 \newcatcodetable\catcodetable@initex
97 \newcatcodetable\catcodetable@string
98 \begingroup
99   \def\setrangecatcode#1#2#3{%
100     \ifnum#1>#2 %
101       \expandafter\@gobble
102     \else
103       \expandafter\@firstofone

```



```

104     \fi
105     {%
106         \catcode#1=#3 %
107         \expandafter\setrange\catcode\expandafter
108         {\number\numexpr#1 + 1\relax}{#2}{#3}
109     }%
110 }
111 \@firstofone{%
112     \catcodetable\catcodetable@initex
113     \catcode0=12 %
114     \catcode13=12 %
115     \catcode37=12 %
116     \setrange\catcode{65}{90}{12}%
117     \setrange\catcode{97}{122}{12}%
118     \catcode92=12 %
119     \catcode127=12 %
120     \savecatcodetable\catcodetable@string
121     \endgroup
122 }%
123 \newcatcodetable\catcodetable@latex
124 \newcatcodetable\catcodetable@atletter
125 \begingroup
126     \def\parseunicodedataI#1;#2;#3;#4\relax{%
127         \parseunicodedataII#1;#3;#2 First>\relax
128     }%
129     \def\parseunicodedataII#1;#2;#3 First>#4\relax{%
130         \ifx\relax#4\relax
131             \expandafter\parseunicodedataIII
132         \else
133             \expandafter\parseunicodedataIV
134         \fi
135         {#1}#2\relax%
136     }%
137     \def\parseunicodedataIII#1#2#3\relax{%
138         \ifnum 0%
139             \if L#21\fi
140             \if M#21\fi
141             >0 %
142             \catcode"#1=11 %
143         \fi
144     }%
145     \def\parseunicodedataIV#1#2#3\relax{%
146         \read\unicoderead to \unicodedataline
147         \if L#2%
148             \count0="#1 %
149             \expandafter\parseunicodedataV\unicodedataline\relax
150         \fi
151     }%
152     \def\parseunicodedataV#1;#2\relax{%
153         \loop
154             \unless\ifnum\count0>"#1 %
155                 \catcode\count0=11 %
156                 \advance\count0 by 1 %
157         \repeat

```

```

158 }%
159 \def\storedpar{\par}%
160 \chardef\unicoderead=\numexpr\count16 + 1\relax
161 \openin\unicoderead=UnicodeData.txt %
162 \loop\unless\ifeof\unicoderead %
163   \read\unicoderead to \unicodedataline
164   \unless\ifx\unicodedataline\storedpar
165     \expandafter\parseunicodedataI\unicodedataline\relax
166   \fi
167 \repeat
168 \closein\unicoderead
169 \@firstofone{%
170   \catcode64=12 %
171   \savecatcodetable\catcodetable@latex
172   \catcode64=11 %
173   \savecatcodetable\catcodetable@atletter
174 }
175 \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.

```

176 \ifx\e@alloc@luafunction@count\@undefined
177   \countdef\e@alloc@luafunction@count=260
178 \fi
179 \def\newluafunction{%
180   \e@alloc@luafunction\e@alloc@chardef
181   \e@alloc@luafunction@count\m@ne\e@alloc@top
182 }
183 \e@alloc@luafunction@count=\z@

```

5.6 Custom whatsits

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

```

184 \ifx\e@alloc@whatsit@count\@undefined
185   \countdef\e@alloc@whatsit@count=261
186 \fi
187 \def\newwhatsit#1{%
188   \e@alloc@whatsit\e@alloc@chardef
189   \e@alloc@whatsit@count\m@ne\e@alloc@top#1%
190 }
191 \e@alloc@whatsit@count=\z@

```

5.7 Lua bytecode registers

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

```

192 \ifx\e@alloc@bytecode@count\@undefined
193   \countdef\e@alloc@bytecode@count=262
194 \fi
195 \def\newluabytcode#1{%

```

```

196 \e@alloc\luabytecode\e@alloc@chardef
197 \e@alloc@bytecode@count\m@ne\e@alloc@top#1%
198 }
199 \e@alloc@bytecode@count=\z@

```

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.

```

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

```

5.9 Lua loader

Load the Lua code at the start of every job. For the conversion of T_EX 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.

```

210 <2ekernel>\everyjob\expandafter{%
211 <2ekernel> \the\everyjob
212 \begingroup
213 \attributedef\attributezero=0 %
214 \chardef \charzero =0 %

```

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

```

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

225 % \changes{v1.0b}{2015/10/02}{Fix backing out of \TeX{ } code}
226 % \changes{v1.0c}{2015/10/02}{Allow backing out of Lua code}
227 <latexrelease>\IncludeInRelease{0000/00/00}
228 <latexrelease> {\newluafunction}{LuaTeX}%
229 <latexrelease>\let\e@alloc@attribute@count\@undefined
230 <latexrelease>\let\newattribute\@undefined
231 <latexrelease>\let\setattribute\@undefined
232 <latexrelease>\let\unsetattribute\@undefined
233 <latexrelease>\let\e@alloc@ccodetable@count\@undefined
234 <latexrelease>\let\newcatcodetable\@undefined

```

```

235 \let\catcodetable@initex\@undefined
236 \let\catcodetable@string\@undefined
237 \let\catcodetable@latex\@undefined
238 \let\catcodetable@atletter\@undefined
239 \let\@alloc@luafunction@count\@undefined
240 \let\newluafunction\@undefined
241 \let\@alloc@luafunction@count\@undefined
242 \let\newwhatsit\@undefined
243 \let\@alloc@whatsit@count\@undefined
244 \let\newluabytecode\@undefined
245 \let\@alloc@bytecode@count\@undefined
246 \let\newluachunkname\@undefined
247 \let\@alloc@luachunk@count\@undefined
248 \directlua{luatexbase.uninstall()}
249 \EndIncludeInRelease

```

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

```

250 \IncludeInRelease{2017/01/01}%
251 \fontencoding{TU} in everyjob}%
252 \fontencoding{TU}\let\encodingdefault\f@encoding
253 \ifx\directlua\@undefined\else
254 \everyjob\expandafter{%
255 \the\everyjob
256 \*2kernel, latexrelease)
257 \directlua{%
258 %% Horrible hack, locally reset the luatex version number
259 %% This is not required for the source version of luaotfload
260 %% but is required due to an error in the version check in the
261 %% public version (January 2017)
262 %% https://github.com/lualatex/luaotfload/issues/387
263 %% It is expected that this will be removed before TeXLive 2017
264 local tmp_version=tex.luatexversion %
265 tex.luatexversion=199 %
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 tex.luatexversion=tmp_version%
275 }%
276 \let\f@encoding\encodingdefault
277 \expandafter\let\csname ver@luaotfload.sty\endcsname\fmtversion
278 \*2kernel, latexrelease)
279 \fi
280 \*2kernel}
281 \EndIncludeInRelease
282 \IncludeInRelease{0000/00/00}%
283 \fontencoding{TU} in everyjob}%
284 \fontencoding{OT1}\let\encodingdefault\f@encoding
285 \EndIncludeInRelease
286 \*2kernel | latexrelease)\fi

```

287 `</2ekernel | tex | latexrelease>`

5.10 Lua module preliminaries

288 `<*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.

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

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

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

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_EX level into a Lua table: from version 0.80, LuaT_EX makes that easy.

```

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

```

registernumber Working out the correct return value can be done in two ways. For older LuaT_EX releases it has to be extracted from the `hashtokens`. On the other hand, newer LuaT_EX'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 function new_attribute(name)
420   tex_setcount("global", "e@alloc@attribute@count",
421               tex_count["e@alloc@attribute@count"] + 1)
422   if tex_count["e@alloc@attribute@count"] > 65534 then
423     luatexbase_error("No room for a new \\attribute")
424   end
425   attributes[name]= tex_count["e@alloc@attribute@count"]
426   luatexbase_log("Lua-only attribute " .. name .. " = " ..
427                 tex_count["e@alloc@attribute@count"])
428   return tex_count["e@alloc@attribute@count"]
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 function new_whatsit(name)
432   tex_setcount("global", "e@alloc@whatsit@count",
433               tex_count["e@alloc@whatsit@count"] + 1)
434   if tex_count["e@alloc@whatsit@count"] > 65534 then
435     luatexbase_error("No room for a new custom whatsit")
436   end
437   luatexbase_log("Custom whatsit " .. (name or "") .. " = " ..
438                 tex_count["e@alloc@whatsit@count"])
439   return tex_count["e@alloc@whatsit@count"]
440 end
441 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.

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

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

5.17 Lua callback management

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

5.17.1 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.

```
467 local callbacklist = callbacklist or { }
```

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

```
468 local list, data, exclusive, simple = 1, 2, 3, 4
469 local types = {
470   list      = list,
```

```

471 data      = data,
472 exclusive = exclusive,
473 simple    = simple,
474 }

```

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.)

```

475 local callbacktypes = callbacktypes or {

```

Section 8.2: file discovery callbacks.

```

476 find_read_file      = exclusive,
477 find_write_file     = exclusive,
478 find_font_file      = data,
479 find_output_file    = data,
480 find_format_file    = data,
481 find_vf_file        = data,
482 find_map_file       = data,
483 find_enc_file       = data,
484 find_sfd_file       = data,
485 find_pk_file        = data,
486 find_data_file      = data,
487 find_opentype_file  = data,
488 find_truetype_file  = data,
489 find_type1_file     = data,
490 find_image_file     = data,

491 open_read_file      = exclusive,
492 read_font_file      = exclusive,
493 read_vf_file        = exclusive,
494 read_map_file       = exclusive,
495 read_enc_file       = exclusive,
496 read_sfd_file       = exclusive,
497 read_pk_file        = exclusive,
498 read_data_file      = exclusive,
499 read_truetype_file  = exclusive,
500 read_type1_file     = exclusive,
501 read_opentype_file  = exclusive,

```

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

```

502 find_cidmap_file    = data,
503 read_cidmap_file    = exclusive,

```

Section 8.3: data processing callbacks.

```

504 process_input_buffer = data,
505 process_output_buffer = data,
506 process_jobname       = data,

```

Section 8.4: node list processing callbacks.

```

507 contribute_filter    = simple,
508 buildpage_filter     = simple,
509 build_page_insert    = exclusive,
510 pre_linebreak_filter  = list,
511 linebreak_filter     = list,
512 append_to_vlist_filter = list,
513 post_linebreak_filter = list,
514 hpack_filter         = list,
515 vpack_filter         = list,
516 hpack_quality        = list,
517 vpack_quality        = list,
518 pre_output_filter     = list,
519 process_rule         = list,
520 hyphenate            = simple,
521 ligaturing           = simple,
522 kerning              = simple,
523 insert_local_par     = simple,
524 mlist_to_hlist       = list,

```

Section 8.5: information reporting callbacks.

```

525 pre_dump             = simple,
526 start_run            = simple,
527 stop_run             = simple,
528 start_page_number    = simple,
529 stop_page_number     = simple,
530 show_error_hook      = simple,
531 show_warning_message = simple,
532 show_error_message   = simple,
533 show_lua_error_hook  = simple,
534 start_file           = simple,
535 stop_file            = simple,
536 call_edit            = simple,

```

Section 8.6: PDF-related callbacks.

```

537 finish_pdffile = data,
538 finish_pdfpage = data,

```

Section 8.7: font-related callbacks.

```

539 define_font = exclusive,

```

```

540 }

```

```

541 luatexbase.callbacktypes=callbacktypes

```

callback.register Save the original function for registering callbacks and prevent the original being used. The original is saved in a place that remains available so other more sophisticated code can override the approach taken by the kernel if desired.

```

542 local callback_register = callback_register or callback.register
543 function callback.register()
544   luatexbase_error("Attempt to use callback.register() directly\n")
545 end

```

5.17.2 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.

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.

```
546 local function data_handler(name)
547   return function(data, ...)
548     for _,i in ipairs(callbacklist[name]) do
549       data = i.func(data,...)
550     end
551     return data
552   end
553 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.

```
554 local function exclusive_handler(name)
555   return function(...)
556     return callbacklist[name][1].func(...)
557   end
558 end
```

Handler for **list** callbacks.

```
559 local function list_handler(name)
```

```

560 return function(head, ...)
561     local ret
562     local alltrue = true
563     for _,i in ipairs(callbacklist[name]) do
564         ret = i.func(head, ...)
565         if ret == false then
566             luatexbase_warning(
567                 "Function '" .. i.description .. "' returned false\n"
568                 .. "in callback '" .. name .. "'"
569             )
570             break
571         end
572         if ret ~= true then
573             alltrue = false
574             head = ret
575         end
576     end
577     return alltrue and true or head
578 end
579 end

```

Handler for simple callbacks.

```

580 local function simple_handler(name)
581     return function(...)
582         for _,i in ipairs(callbacklist[name]) do
583             i.func(...)
584         end
585     end
586 end

```

Keep a handlers table for indexed access.

```

587 local handlers = {
588     [data]      = data_handler,
589     [exclusive] = exclusive_handler,
590     [list]      = list_handler,
591     [simple]     = simple_handler,
592 }

```

5.17.3 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.

```

593 local user_callbacks_defaults = { }

```

`create_callback` The allocator itself.

```

594 local function create_callback(name, ctype, default)
595     if not name or name == ""
596     or not ctype or ctype == ""
597     then
598         luatexbase_error("Unable to create callback:\n" ..
599             "valid callback name and type required")
600     end
601     if callbacktypes[name] then

```

```

602     luatexbase_error("Unable to create callback '" .. name ..
603                       "':\ncallback is already defined")
604   end
605   if default ~= false and type (default) ~= "function" then
606     luatexbase_error("Unable to create callback '" .. name ..
607                       "':\ndefault is not a function")
608   end
609   user_callbacks_defaults[name] = default
610   callbacktypes[name] = types[ctype]
611 end
612 luatexbase.create_callback = create_callback

```

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

```

613 local function call_callback(name,...)
614   if not name or name == "" then
615     luatexbase_error("Unable to create callback:\n" ..
616                       "valid callback name required")
617   end
618   if user_callbacks_defaults[name] == nil then
619     luatexbase_error("Unable to call callback '" .. name
620                       .. "':\nunknown or empty")
621   end
622   local l = callbacklist[name]
623   local f
624   if not l then
625     f = user_callbacks_defaults[name]
626     if l == false then
627       return nil
628     end
629   else
630     f = handlers[callbacktypes[name]](name)
631   end
632   return f(...)
633 end
634 luatexbase.call_callback=call_callback

```

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

```

635 local function add_to_callback(name, func, description)
636   if not name or name == "" then
637     luatexbase_error("Unable to register callback:\n" ..
638                       "valid callback name required")
639   end
640   if not callbacktypes[name] or
641     type(func) ~= "function" or
642     not description or
643     description == "" then
644     luatexbase_error(
645       "Unable to register callback.\n\n"
646       .. "Correct usage:\n"
647       .. "add_to_callback(<callback>, <function>, <description>)"
648     )
649   end

```

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

```

650 local l = callbacklist[name]
651 if l == nil then
652     l = { }
653     callbacklist[name] = l

```

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

```

654     if user_callbacks_defaults[name] == nil then
655         callback_register(name, handlers[callbacktypes[name]](name))
656     end
657 end

```

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

```

658 local f = {
659     func      = func,
660     description = description,
661 }
662 local priority = #l + 1
663 if callbacktypes[name] == exclusive then
664     if #l == 1 then
665         luatexbase_error(
666             "Cannot add second callback to exclusive function\n'" ..
667             name .. "'"
668         )
669     end
670     table.insert(l, priority, f)

```

Keep user informed.

```

671     luatexbase_log(
672         "Inserting '" .. description .. "' at position "
673         .. priority .. " in '" .. name .. "'"
674     )
675 end
676 luatexbase.add_to_callback = add_to_callback

```

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

```

677 local function remove_from_callback(name, description)
678     if not name or name == "" then
679         luatexbase_error("Unable to remove function from callback:\n" ..
680             "valid callback name required")
681     end
682     if not callbacktypes[name] or
683         not description or
684         description == "" then
685         luatexbase_error(
686             "Unable to remove function from callback.\n\n"
687             .. "Correct usage:\n"
688             .. "remove_from_callback(<callback>, <description>)"
689         )
690     end
691     local l = callbacklist[name]
692     if not l then
693         luatexbase_error(
694             "No callback list for '" .. name .. "'\n")
695     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.

```

696 local index = false
697 for i,j in ipairs(l) do
698   if j.description == description then
699     index = i
700     break
701   end
702 end
703 if not index then
704   luatexbase_error(
705     "No callback '" .. description .. "' registered for '" ..
706     name .. "'\n")
707 end
708 local cb = l[index]
709 table.remove(l, index)
710 luatexbase_log(
711   "Removing '" .. description .. "' from '" .. name .. "'."
712 )
713 if #l == 0 then
714   callbacklist[name] = nil
715   callback_register(name, nil)
716 end
717 return cb.func,cb.description
718 end
719 luatexbase.remove_from_callback = remove_from_callback

```

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

```

720 local function in_callback(name, description)
721   if not name
722     or name == ""
723     or not callbacklist[name]
724     or not callbacktypes[name]
725     or not description then
726     return false
727   end
728   for _, i in pairs(callbacklist[name]) do
729     if i.description == description then
730       return true
731     end
732   end
733   return false
734 end
735 luatexbase.in_callback = in_callback

```

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

```

736 local function disable_callback(name)
737   if(callbacklist[name] == nil) then
738     callback_register(name, false)
739   else
740     luatexbase_error("Callback list for " .. name .. " not empty")
741   end
742 end

```



```

743 luatexbase.disable_callback = disable_callback

callback\_descriptions List the descriptions of functions registered for the given callback.
744 local function callback_descriptions (name)
745   local d = {}
746   if not name
747     or name == ""
748     or not callbacklist[name]
749     or not callbacktypes[name]
750   then
751     return d
752   else
753     for k, i in pairs(callbacklist[name]) do
754       d[k] = i.description
755     end
756   end
757   return d
758 end
759 luatexbase.callback_descriptions = callback_descriptions

uninstall Unlike at the TEX level, we have to provide a back-out mechanism here at the
same time as the rest of the code. This is not meant for use by anything other
than latexrelease: as such this is deliberately not documented for users!
760 local function uninstall()
761   module_info(
762     "luatexbase",
763     "Uninstalling kernel luatexbase code"
764   )
765   callback.register = callback_register
766   luatexbase = nil
767 end
768 luatexbase.uninstall = uninstall

769 </lua>

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

```