

lualatex.dtx
(LuaTeX-specific support)

David Carlisle and Joseph Wright*

2017/02/18

Contents

1	Overview	2
2	Core TeX functionality	2
3	Plain TeX interface	3
4	Lua functionality	3
4.1	Allocators in Lua	3
4.2	Lua access to TeX register numbers	4
4.3	Module utilities	5
4.4	Callback management	5
5	Implementation	6
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	11
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 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 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 kernel did not provide any functionality for the extended allocation area).

2 Core TeX functionality

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

```
\newattribute \newattribute{attribute}
Defines a named \attribute, indexed from 1 (i.e. \attribute0 is never defined).
Attributes initially have the marker value -"7FFFFFFF ('unset') set by the engine.
\newcatcodetable \newcatcodetable{catcodetable}
Defines a named \catcodetable, indexed from 1 (\catcodetable0 is never assigned).
A new catcode table will be populated with exactly those values assigned by IniTeX (as described in the LuaTeX manual).
\newluafunction \newluafunction{function}
Defines a named \luafunction, indexed from 1. (Lua indexes tables from 1 so \luafunction0 is not available).
\newwhatsit \newwhatsit{whatsit}
Defines a custom \whatsit, indexed from 1.
\newluabytecode \newluabytecode{bytecode}
Allocates a number for Lua bytecode register, indexed from 1.
\newluachunkname newluachunkname{chunkname}
Allocates a number for Lua chunk register, indexed from 1. Also enters the name of the register (without backslash) into the lua.name 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 `ltuatex` interface may be used with plain T_EX using `\input{ltuatex}`. This inputs `ltuatex.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 `ltuatex` 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 `ltuatex` code but implements a compatibility layer providing the interface of the original package.

4 Lua functionality

4.1 Allocators in Lua

<code>new_attribute</code>	<code>luatexbase.new_attribute(⟨attribute⟩)</code> Returns an allocation number for the <code>⟨attribute⟩</code> , indexed from 1. The attribute will be initialised with the marker value <code>-"7FFFFFFF</code> ('unset'). The attribute allocation sequence is shared with the T _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.

These functions all require access to a named T_EX count register to manage their allocations. The standard names are those defined above for access from T_EX, *e.g.* `"e@alloc@attribute@count`, but these can be adjusted by defining the variable `⟨type⟩_count_name` before loading `ltuatex.lua`, for example

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

would use a TeX `\count` (`\countdef`'d token) called `attributetracker` in place of `"e@alloc@attribute@count`.

4.2 Lua access to TeX register numbers

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

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

As an example, consider the input:

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

\test{undefinedrubbish}

\test{space}

\test{hbox}

\test{@MM}

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

\test{strutbox}

\test{sixt@@n}

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

If the demonstration code is processed with LuaLaTeX then the following would be produced in the log and terminal output.

```
undefinedrubbish: \relax
    bad input
space: macro:->
    bad input
hbox: \hbox
    bad input
@MM: \mathchar"4E20
    20000
@tempdima: \dimen14
```

```

14
@tempdimb: \dimen15
15
strutbox: \char"B
11
sixt@n: \char"10
16
myattr: \attribute12
12

```

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

4.3 Module utilities

`provides_module` `luatexbase.provides_module(<info>)`
This function is used by modules to identify themselves; the `info` should be a table containing information about the module. The required field `name` must contain the name of the module. It is recommended to provide a field `date` in the usual L^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_TE_X 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.

<code>callback_descriptions</code>	A list of the descriptions of functions registered to the specified callback is returned. <code>{}</code> is returned if there are no functions registered.
<code>create_callback</code>	<code>luatexbase.create_callback(<name>,metatype,<default>)</code> Defines a user defined callback. The last argument is a default function or <code>false</code> .
<code>call_callback</code>	<code>luatexbase.call_callback(<name>,...)</code> 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 ltuatex 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\sixt@@n\else\@cclvi\fi}

31 \count 270=\et@xmaxregs % locally allocates \count registers
32 \count 271=\et@xmaxregs % ditto for \dimen registers
33 \count 272=\et@xmaxregs % ditto for \skip registers
34 \count 273=\et@xmaxregs % ditto for \muskip registers
35 \count 274=\et@xmaxregs % ditto for \box registers
36 \count 275=\et@xmaxregs % ditto for \toks registers
37 \count 276=\et@xmaxregs % ditto for \marks classes
```

and 256 or 16 fam. (Done above due to plain/L^AT_EX differences in l^Auatex.)

```
38 % \outer\def\newfam{\alloc@8\fam\chardef\et@xmaxfam}
```

End of proposed changes to `etex.src`

5.2.2 luatex specific settings

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

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

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

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

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

55 \gdef\e@ch@ck#1#2#3#4{%
56   \ifnum#1<#2\else
57     \ifnum#1=#2\relax
58       #1\@cclvi
59       \ifx\count#4\advance#1 10 \fi
60     \fi
61     \ifnum#1<#3\relax
```

```

62     \else
63         \errmessage{No room for a new \string#4}%
64     \fi
65 \fi}%

Two simple LATEX macros used in lATEX.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\@alloc@attribute@count
70 \expandafter\csname newcount\endcsname\@alloc@ccodetable@count
71 \expandafter\csname newcount\endcsname\@alloc@luafunction@count
72 \expandafter\csname newcount\endcsname\@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 TEX / old LATEX.
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     \e@alloc\attribute\attributedef
82     \e@alloc@attribute@count\m@ne\@alloc@top#1%
83 }
84 \e@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     \e@alloc\catcodetable\chardef
92     \e@alloc@ccodetable@count\m@ne{"8000}#1%
93     \initcatcodetable\allocationnumber
94 }
95 \e@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\setrangeecatcode#1#2#3{%
100 \ifnum#1>#2 %
101 \expandafter\@gobble
102 \else
103 \expandafter\@firstofone
104 \fi
105 {%
106 \catcode#1=#3 %
107 \expandafter\setrangeecatcode\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 \setrangeecatcode{65}{90}{12}%
117 \setrangeecatcode{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 Lua_{TeX} 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

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

```

192 \ifx\e@alloc@bytocode@count\@undefined
193 \countdef\e@alloc@bytocode@count=262
194 \fi
195 \def\newluabytocode#1{%
196 \e@alloc\luabytocode\e@alloc@chardef
197 \e@alloc@bytocode@count\m@ne\e@alloc@top#1%
198 }
199 \e@alloc@bytocode@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 \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.

```

210 (2kernel)\everyjob\expandafter{%
211 (2kernel) \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("ltnlua")}}
222 \endgroup
223 (2kernel)}
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\@alloc@attribute@count\@undefined
230 \latexrelease\let\newattribute\@undefined
231 \latexrelease\let\setattribute\@undefined
232 \latexrelease\let\unsetattribute\@undefined
233 \latexrelease\let\@alloc@ccodetable@count\@undefined
234 \latexrelease\let\newcatcodetable\@undefined
235 \latexrelease\let\catcodetable@initex\@undefined
236 \latexrelease\let\catcodetable@string\@undefined
237 \latexrelease\let\catcodetable@latex\@undefined
238 \latexrelease\let\catcodetable@atletter\@undefined
239 \latexrelease\let\@alloc@luafunction@count\@undefined
240 \latexrelease\let\newluafunction\@undefined
241 \latexrelease\let\@alloc@luafunction@count\@undefined
242 \latexrelease\let\newwhatsit\@undefined
243 \latexrelease\let\@alloc@whatsit@count\@undefined
244 \latexrelease\let\newluabytecode\@undefined
245 \latexrelease\let\@alloc@bytecode@count\@undefined
246 \latexrelease\let\newluachunkname\@undefined
247 \latexrelease\let\@alloc@luachunk@count\@undefined
248 \latexrelease\directlua{luatexbase.uninstall()}
249 \latexrelease\EndIncludeInRelease

```

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

```

250 \latexrelease\IncludeInRelease{2017/01/01}%
251 \latexrelease          {\fontencoding}{TU in everyjob}%
252 \latexrelease\fontencoding{TU}\let\encodingdefault\f@encoding
253 \latexrelease\ifx\directlua\@undefined\else
254 (2kernel)\everyjob\expandafter{%
255 (2kernel) \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 <latexrelease>\fi
280 <2kernel> }
281 <latexrelease>\EndIncludeInRelease
282 <latexrelease>\IncludeInRelease{0000/00/00}%
283 <latexrelease>          {\fontencoding}{TU in everyjob}%
284 <latexrelease>\fontencoding{OT1}\let\encodingdefault\f@encoding
285 <latexrelease>\EndIncludeInRelease

286 <2kernel | latexrelease>\fi
287 </2kernel | 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 attribute_count_name = attribute_count_name or "e@alloc@attribute@count"
420 local function new_attribute(name)
421   tex_setcount("global", attribute_count_name,
422               tex_count[attribute_count_name] + 1)
423   if tex_count[attribute_count_name] > 65534 then
424     luatexbase_error("No room for a new \\attribute")
425   end
426   attributes[name]= tex_count[attribute_count_name]
427   luatexbase_log("Lua-only attribute " .. name .. " = " ..
428                 tex_count[attribute_count_name])
429   return tex_count[attribute_count_name]
430 end
431 luatexbase.new_attribute = new_attribute

```

5.14 Custom whatsit allocation

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

```

432 local whatsit_count_name = whatsit_count_name or "e@alloc@whatsit@count"
433 local function new_whatsit(name)

```

```

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

```

5.15 Bytecode register allocation

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

```

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

```

5.17 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.

5.17.1 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.

```
471 local callbacklist = callbacklist or { }
```

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

```
472 local list, data, exclusive, simple = 1, 2, 3, 4
473 local types = {
474   list      = list,
475   data      = data,
476   exclusive = exclusive,
477   simple    = simple,
478 }
```

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
}
```

in plain LuaTeX. (Some undocumented callbacks are omitted as they are to be removed.)

```
479 local callbacktypes = callbacktypes or {
```

Section 8.2: file discovery callbacks.

```
480   find_read_file      = exclusive,
481   find_write_file     = exclusive,
482   find_font_file      = data,
483   find_output_file    = data,
484   find_format_file    = data,
485   find_vf_file        = data,
486   find_map_file       = data,
487   find_enc_file       = data,
488   find_sfd_file       = data,
489   find_pk_file        = data,
490   find_data_file      = data,
491   find_opentype_file  = data,
492   find_truetype_file  = data,
493   find_type1_file     = data,
494   find_image_file     = data,
495   open_read_file      = exclusive,
496   read_font_file      = exclusive,
497   read_vf_file        = exclusive,
498   read_map_file       = exclusive,
499   read_enc_file       = exclusive,
```

```

500 read_sfd_file      = exclusive,
501 read_pk_file       = exclusive,
502 read_data_file     = exclusive,
503 read_truetype_file = exclusive,
504 read_type1_file    = exclusive,
505 read_opentype_file = exclusive,

```

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

```

506 find_cidmap_file  = data,
507 read_cidmap_file  = exclusive,

```

Section 8.3: data processing callbacks.

```

508 process_input_buffer = data,
509 process_output_buffer = data,
510 process_jobname      = data,

```

Section 8.4: node list processing callbacks.

```

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

```

Section 8.5: information reporting callbacks.

```

529 pre_dump             = simple,
530 start_run            = simple,
531 stop_run             = simple,
532 start_page_number    = simple,
533 stop_page_number     = simple,
534 show_error_hook      = simple,
535 show_warning_message = simple,
536 show_error_message   = simple,
537 show_lua_error_hook  = simple,
538 start_file           = simple,
539 stop_file            = simple,
540 call_edit            = simple,

```

Section 8.6: PDF-related callbacks.

```

541 finish_pdf_file      = data,
542 finish_pdf_page      = data,

```

Section 8.7: font-related callbacks.

```

543 define_font          = exclusive,

```

```

544 }
545 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.

```

546 local callback_register = callback_register or callback.register
547 function callback.register()
548   luatexbase_error("Attempt to use callback.register() directly\n")
549 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.

```

550 local function data_handler(name)
551   return function(data, ...)
552     for _,i in ipairs(callbacklist[name]) do
553       data = i.func(data,...)
554     end
555     return data
556   end
557 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.

```
558 local function exclusive_handler(name)
559   return function(...)
560     return callbacklist[name][1].func(...)
561   end
562 end
```

Handler for `list` callbacks.

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

Handler for `simple` callbacks.

```
584 local function simple_handler(name)
585   return function(...)
586     for _,i in ipairs(callbacklist[name]) do
587       i.func(...)
588     end
589   end
590 end
```

Keep a handlers table for indexed access.

```
591 local handlers = {
592   [data]      = data_handler,
593   [exclusive] = exclusive_handler,
594   [list]      = list_handler,
595   [simple]     = simple_handler,
596 }
```

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.

```
597 local user_callbacks_defaults = { }
```

create_callback The allocator itself.

```
598 local function create_callback(name, ctype, default)
599   if not name or name == ""
600   or not ctype or ctype == ""
601   then
602     luatexbase_error("Unable to create callback:\n" ..
603                       "valid callback name and type required")
604   end
605   if callbacktypes[name] then
606     luatexbase_error("Unable to create callback '" .. name ..
607                       "':\ncallback is already defined")
608   end
609   if default ~= false and type (default) ~= "function" then
610     luatexbase_error("Unable to create callback '" .. name ..
611                       "':\ndefault is not a function")
612   end
613   user_callbacks_defaults[name] = default
614   callbacktypes[name] = types[ctype]
615 end
616 luatexbase.create_callback = create_callback
```

call_callback Call a user defined callback. First check arguments.

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

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

```
639 local function add_to_callback(name, func, description)
640   if not name or name == "" then
641     luatexbase_error("Unable to register callback:\n" ..
642                       "valid callback name required")
643   end
644   if not callbacktypes[name] or
645     type(func) ~= "function" or
646     not description or
```

```

647     description == "" then
648     luatexbase_error(
649     "Unable to register callback.\n\n"
650     .. "Correct usage:\n"
651     .. "add_to_callback(<callback>, <function>, <description>)"
652     )
653 end

```

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

```

654 local l = callbacklist[name]
655 if l == nil then
656     l = { }
657     callbacklist[name] = l

```

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

```

658     if user_callbacks_defaults[name] == nil then
659         callback_register(name, handlers[callbacktypes[name]](name))
660     end
661 end

```

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

```

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

```

Keep user informed.

```

675 luatexbase_log(
676     "Inserting '" .. description .. "' at position "
677     .. priority .. " in '" .. name .. "'"")
678 )
679 end
680 luatexbase.add_to_callback = add_to_callback

```

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

```

681 local function remove_from_callback(name, description)
682     if not name or name == "" then
683         luatexbase_error("Unable to remove function from callback:\n" ..
684             "valid callback name required")
685     end
686     if not callbacktypes[name] or
687         not description or
688         description == "" then
689         luatexbase_error(

```

```

690     "Unable to remove function from callback.\n\n"
691     .. "Correct usage:\n"
692     .. "remove_from_callback(<callback>, <description>)"
693   )
694 end
695 local l = callbacklist[name]
696 if not l then
697   luatexbase_error(
698     "No callback list for '" .. name .. "'\n")
699 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.
700 local index = false
701 for i,j in ipairs(l) do
702   if j.description == description then
703     index = i
704     break
705   end
706 end
707 if not index then
708   luatexbase_error(
709     "No callback '" .. description .. "' registered for '" ..
710     name .. "'\n")
711 end
712 local cb = l[index]
713 table.remove(l, index)
714 luatexbase_log(
715   "Removing '" .. description .. "' from '" .. name .. "'."
716 )
717 if #l == 0 then
718   callbacklist[name] = nil
719   callback_register(name, nil)
720 end
721 return cb.func,cb.description
722 end
723 luatexbase.remove_from_callback = remove_from_callback

```

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

```

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

```

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

```
740 local function disable_callback(name)
741   if(callbacklist[name] == nil) then
742     callback_register(name, false)
743   else
744     luatexbase_error("Callback list for " .. name .. " not empty")
745   end
746 end
747 luatexbase.disable_callback = disable_callback
```

`callback_descriptions` List the descriptions of functions registered for the given callback.

```
748 local function callback_descriptions (name)
749   local d = {}
750   if not name
751     or name == ""
752     or not callbacklist[name]
753     or not callbacktypes[name]
754   then
755     return d
756   else
757     for k, i in pairs(callbacklist[name]) do
758       d[k]= i.description
759     end
760   end
761   return d
762 end
763 luatexbase.callback_descriptions =callback_descriptions
```

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

```
764 local function uninstall()
765   module_info(
766     "luatexbase",
767     "Uninstalling kernel luatexbase code"
768   )
769   callback.register = callback_register
770   luatexbase = nil
771 end
772 luatexbase.uninstall = uninstall
```

```
773 </lua>
```

Reset the catcode of @.

```
774 <tex>\catcode'\@=\etacatcode\relax
```