

lualatex.dtx
(LuaTeX-specific support)

David Carlisle and Joseph Wright*

2024/08/16

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 | 7 |
| 5.3 | Attributes | 9 |
| 5.4 | Category code tables | 9 |
| 5.5 | Named Lua functions | 11 |
| 5.6 | Custom whatsits | 11 |
| 5.7 | Lua bytecode registers | 12 |
| 5.8 | Lua chunk registers | 12 |
| 5.9 | Lua loader | 12 |
| 5.10 | Lua module preliminaries | 14 |
| 5.11 | Lua module utilities | 14 |
| 5.12 | Accessing register numbers from Lua | 16 |
| 5.13 | Attribute allocation | 17 |
| 5.14 | Custom whatsit allocation | 17 |
| 5.15 | Bytecode register allocation | 18 |
| 5.16 | Lua chunk name allocation | 18 |
| 5.17 | Lua function allocation | 18 |
| 5.18 | Lua callback management | 19 |

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

1 Overview

LuaTeX adds a number of engine-specific functions to TeX. Several of these require set up that is best done in the kernel or need related support functions. This file provides *basic* support for LuaTeX at the L^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).
  \newluacmd \newluacmd{function}
  Like \newluafunction, but defines the command using \luacmd instead of just assigning an integer.
\newprotectedluacmd \newluacmd{function}
  Like \newluacmd, but the defined command is not expandable.
\newwhatsit \newwhatsit{whatsit}
  Defines a custom \whatsit, indexed from 1.
\newluabytecode \newluabytecode{bytecode}
```

| | |
|-------------------------------------|---|
| | Allocates a number for Lua bytecode register, indexed from 1. |
| <code>\newluachunkname</code> | <code>newluachunkname{⟨chunkname⟩}</code> Allocates a number for Lua chunk register, indexed from 1. Also enters the name of the register (without backslash) into the <code>lua.name</code> table to be used in stack traces. |
| <code>\catcodetable@initex</code> | Predefined category code tables with the obvious assignments. Note that the |
| <code>\catcodetable@string</code> | <code>latex</code> and <code>atletter</code> tables set the full Unicode range to the codes predefined by |
| <code>\catcodetable@latex</code> | the kernel. |
| <code>\catcodetable@atletter</code> | <code>\setattribute{⟨attribute⟩}{⟨value⟩}</code> |
| <code>\setattribute</code> | <code>\unsetattribute{⟨attribute⟩}</code> |
| <code>\unsetattribute</code> | Set and unset attributes in a manner analogous to <code>\setlength</code> . Note that attributes take a marker value when unset so this operation is distinct from setting the value to zero. |

3 Plain T_EX interface

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

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

4 Lua functionality

4.1 Allocators in Lua

| | |
|------------------------------|--|
| <code>new_attribute</code> | <code>luatexbase.new_attribute(⟨attribute⟩)</code> Returns an allocation number for the <code>⟨attribute⟩</code> , indexed from 1. The attribute will be 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. |
| <code>new_luafunction</code> | <code>luatexbase.new_luafunction(⟨functionname⟩)</code> Returns an allocation number for a lua function for use with <code>\luafunction</code> , <code>\lateluafunction</code> , and <code>\luaodef</code> , indexed from 1. The optional <code>⟨functionname⟩</code> argument is just used for logging. |

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 `ltluatex.lua`, for example

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

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

4.2 Lua access to T_EX register numbers

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

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

As an example, consider the input:

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

\test{undefinedrubbish}

\test{space}

\test{hbox}

\test{@MM}

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

\test{strutbox}

\test{sixt@@n}

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

If the demonstration code is processed with LuaL^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. If the `version` begins with a digit, a `v` will be added at the start in the log.

`module_info` `luatexbase.module_info(<module>, <text>)`

`module_warning` `luatexbase.module_warning(<module>, <text>)`

`module_error` `luatexbase.module_error(<module>, <text>)`

These functions are similar to L^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 call-

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

- `in_callback` `luatexbase.in_callback($\langle callback \rangle$, $\langle description \rangle$)` Checks if the $\langle description \rangle$ matches one of the functions added to the list for the $\langle callback \rangle$, returning a boolean value.
- `disable_callback` `luatexbase.disable_callback($\langle callback \rangle$)` Sets the $\langle callback \rangle$ to `false` as described in the LuaTeX manual for the underlying `callback.register` built-in. Callbacks will only be set to `false` (and thus be skipped entirely) if there are no functions registered using the callback.
- `callback_descriptions` A list of the descriptions of functions registered to the specified callback is returned. `{}` is returned if there are no functions registered.
- `create_callback` `luatexbase.create_callback($\langle name \rangle$, $\langle type \rangle$, $\langle default \rangle$)` Defines a user defined callback. The last argument is a default function or `false`.
- `call_callback` `luatexbase.call_callback($\langle name \rangle$, ...)` Calls a user defined callback with the supplied arguments.
- `declare_callback_rule` `luatexbase.declare_callback_rule($\langle name \rangle$, $\langle first \rangle$, $\langle relation \rangle$, $\langle second \rangle$)` Adds an ordering constraint between two callback functions for callback $\langle name \rangle$.

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

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

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

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

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

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

Every call to `declare_callback_rule` with a specific callback $\langle name \rangle$ and descriptions $\langle first \rangle$ and $\langle second \rangle$ overwrites all previous calls with same callback and descriptions.

The callback functions do not have to be registered yet when the functions is called. Only the constraints for which both callback descriptions refer to callbacks registered at the time the callback is called will have an effect.

5 Implementation

```
1 <*2ekernel | tex | latexrelease>
2 <2ekernel | latexrelease> \ifx\directlua\undefined\else
```

5.1 Minimum LuaTeX version

LuaTeX has changed a lot over time. In the kernel support for ancient versions is not provided: trying to build a format with a very old binary therefore gives some

information in the log and loading stops. The cut-off selected here relates to the tree-searching behaviour of `require()`: from version 0.60, LuaTeX will correctly find Lua files in the `texmf` tree without ‘help’.

```

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

```

Two simple L^AT_EX macros from `ltdfn.dtx` have to be defined here because `ltdfn.dtx` is not loaded yet when `ltuatex.dtx` is executed.

```

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

```

5.2 Older L^AT_EX/Plain T_EX setup

```

13 <*tex>

```

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

```

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

```

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

```

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

```

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

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

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

```

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

```

`luatex/xetex` also allow more math fam.

```

30 \edef \et@xmaxfam {\ifx\Umathcode\@undefined\sixt@@n\else\@cclvi\fi}
31 \count 270=\et@xmaxregs % locally allocates \count registers
32 \count 271=\et@xmaxregs % ditto for \dimen registers
33 \count 272=\et@xmaxregs % ditto for \skip registers
34 \count 273=\et@xmaxregs % ditto for \muskip registers

```

```

35 \count 274=\et@xmaxregs % ditto for \box registers
36 \count 275=\et@xmaxregs % ditto for \toks registers
37 \count 276=\et@xmaxregs % ditto for \marks classes

    and 256 or 16 fam. (Done above due to plain/LATEX differences in ltuatex.)
38 % \outer\def\newfam{\alloc@8\fam\chardef\et@xmaxfam}
    End of proposed changes to etex.src

```

5.2.2 luatex specific settings

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

```

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

```

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

```

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

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

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

```

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

```

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

```

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

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

5.3 Attributes

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

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

`\setattribute` Handy utilities.

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

5.4 Category code tables

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

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

`\catcodetable@initex` Save a small set of standard tables. The Unicode data is read here in using a parser

`\catcodetable@string` simplified from that in `load-unicode-data`: only the nature of letters needs to

`\catcodetable@latex` be detected.

```
\catcodetable@atletter 93 \newcatcodetable\catcodetable@initex
94 \newcatcodetable\catcodetable@string
95 \begingroup
96 \def\setrangecatcode#1#2#3{%
97 \ifnum#1>#2 %
98 \expandafter\gobble
99 \else
100 \expandafter\@firstofone
101 \fi
102 {%
103 \catcode#1=#3 %
```

```

104     \expandafter\setrange\catcode\expandafter
105     {\number\numexpr#1 + 1\relax}{#2}{#3}
106     }%
107 }
108 \@firstofone{%
109   \catcodetable\catcodetable@initex
110   \catcode0=12 %
111   \catcode13=12 %
112   \catcode37=12 %
113   \setrange\catcode{65}{90}{12}%
114   \setrange\catcode{97}{122}{12}%
115   \catcode92=12 %
116   \catcode127=12 %
117   \savecatcodetable\catcodetable@string
118   \endgroup
119 }%
120 \newcatcodetable\catcodetable@latex
121 \newcatcodetable\catcodetable@atletter
122 \begingroup
123 \def\parseunicodedataI#1;#2;#3;#4\relax{%
124   \parseunicodedataII#1;#3;#2 First>\relax
125 }%
126 \def\parseunicodedataII#1;#2;#3 First>#4\relax{%
127   \ifx\relax#4\relax
128     \expandafter\parseunicodedataIII
129   \else
130     \expandafter\parseunicodedataIV
131   \fi
132   {#1}#2\relax%
133 }%
134 \def\parseunicodedataIII#1#2#3\relax{%
135   \ifnum 0%
136     \if L#21\fi
137     \if M#21\fi
138     >0 %
139     \catcode"#1=11 %
140   \fi
141 }%
142 \def\parseunicodedataIV#1#2#3\relax{%
143   \read\unicoderead to \unicodedataline
144   \if L#2%
145     \count0="#1 %
146     \expandafter\parseunicodedataV\unicodedataline\relax
147   \fi
148 }%
149 \def\parseunicodedataV#1;#2\relax{%
150   \loop
151     \unless\ifnum\count0>"#1 %
152       \catcode\count0=11 %
153       \advance\count0 by 1 %
154   \repeat
155 }%
156 \def\storedpar{\par}%
157 \chardef\unicoderead=\numexpr\count16 + 1\relax

```

```

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

```

5.5 Named Lua functions

`\newluafunction` Much the same story for allocating Lua_T_E_X functions except here they are just numbers so they are allocated in the same way as boxes. Lua indexes from 1 so once again slot 0 is skipped.

```

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

```

`\newluacmd` Additionally two variants are provided to make the passed control sequence call `\newprotectedluacmd` the function directly.

```

181 \def\newluacmd{%
182   \e@alloc\luafunction\luaodef
183   \e@alloc@luafunction@count\m@ne\e@alloc@top
184 }
185 \def\newprotectedluacmd{%
186   \e@alloc\luafunction{\protected\luaodef}
187   \e@alloc@luafunction@count\m@ne\e@alloc@top
188 }

```

5.6 Custom whatsits

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

```

189 \ifx\e@alloc@whatsit@count\@undefined
190   \countdef\e@alloc@whatsit@count=261
191   \e@alloc@whatsit@count=\z@
192 \fi
193 \def\newwhatsit#1{%
194   \e@alloc\whatsit\e@alloc@chardef
195   \e@alloc@whatsit@count\m@ne\e@alloc@top#1%
196 }

```

5.7 Lua bytecode registers

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

```
197 \ifx\@alloc@bytecode@count\undefined
198 \countdef\@alloc@bytecode@count=262
199 \@alloc@bytecode@count=\z@
200 \fi
201 \def\newluabytocode#1{%
202 \@alloc@luabytocode\@alloc@chardef
203 \@alloc@bytecode@count\m@ne\@alloc@top#1%
204 }
```

5.8 Lua chunk registers

`\newluachunkname` As for bytecode registers, but in addition we need to add a string to the `lua.name` table to use in stack tracing. We use the name of the command passed to the allocator, with no backslash.

```
205 \ifx\@alloc@luachunk@count\undefined
206 \countdef\@alloc@luachunk@count=263
207 \@alloc@luachunk@count=\z@
208 \fi
209 \def\newluachunkname#1{%
210 \@alloc@luachunk\@alloc@chardef
211 \@alloc@luachunk@count\m@ne\@alloc@top#1%
212 {\escapechar\m@ne
213 \directlua{lua.name[\the\allocationnumber]="\string#1"}}%
214 }
```

5.9 Lua loader

Lua code loaded in the format often has to be loaded again at the beginning of every job, so we define a helper which allows us to avoid duplicated code:

```
215 \def\now@and@everyjob#1{%
216 \everyjob\expandafter{\the\everyjob
217 #1%
218 }%
219 #1%
220 }
```

Load the Lua code at the start of every job. For the conversion of \TeX into numbers at the Lua side we need some known registers: for convenience we use a set of systematic names, which means using a group around the Lua loader.

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

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

```
225 \countdef \CountZero =0 %
226 \dimendef \dimenzero =0 %
227 \mathchardef \mathcharzero =0 %
228 \muskipdef \muskipzero =0 %
229 \skipdef \skipzero =0 %
```

```

230 \toksdef \tokszero =0 %
231 \directlua{require("lualatex")}
232 \endgroup
233 <2kernel>}
234 <latexrelease>\EndIncludeInRelease

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

```

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

```

258 <latexrelease>\IncludeInRelease{2017/01/01}%
259 <latexrelease> \fontencodings}{TU in everyjob}%
260 <latexrelease>\fontencodings{TU}\let\encodingdefault\f@encoding
261 <latexrelease>\ifx\directlua\@undefined\else
262 <2kernel>\everyjob\expandafter{%
263 <2kernel> \the\everyjob
264 <*2kernel, latexrelease>
265 \directlua{%
266 if xpcall(function ()%
267 \require('luaotfload-main')%
268 end, texio.write_nl) then %
269 local _void = luaotfload.main ()%
270 else %
271 texio.write_nl('Error in luaotfload: reverting to OT1')%
272 tex.print('\string\def\string\encodingdefault{OT1}')%
273 end %
274 }%
275 \let\f@encoding\encodingdefault
276 \expandafter\let\csname ver@luaotfload.sty\endcsname\fmtversion
277 </2kernel, latexrelease>
278 <latexrelease>\fi
279 <2kernel> }
280 <latexrelease>\EndIncludeInRelease
281 <latexrelease>\IncludeInRelease{0000/00/00}%

```

```

282 <latexrelease>                {\fontencoding}{TU in everyjob}%
283 <latexrelease>\fontencoding{OT1}\let\encodingdefault\f@encoding
284 <latexrelease>\EndIncludeInRelease
285 <2ekernel | latexrelease>\fi
286 </2ekernel | tex | latexrelease>

```

5.10 Lua module preliminaries

```
287 <*lua>
```

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

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

```

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

```

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

```

290 local string_gsub      = string.gsub
291 local tex_count        = tex.count
292 local tex_setcount     = tex.setcount
293 local texio_write_nl  = texio.write_nl
294 local flush_list      = node.flush_list
295 local luatexbase_warning
296 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.

```
297 local modules = modules or { }
```

provides_module Local function to write to the log.

```

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

```

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

```

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

```

```

313 )
314 modules[info.name] = info
315 end
316 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.

```

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

```

```

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

```

```

354 end
355 luatexbase.module_error = module_error

```

Dedicated versions for the rest of the code here.

```

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

```

5.12 Accessing register numbers from Lua

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

```

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

```

registernumber Working out the correct return value can be done in two ways. For older Lua \TeX releases it has to be extracted from the `hashtokens`. On the other hand, newer Lua \TeX 's have `newtoken`, and whilst `.mode` isn't currently documented, Hans Hagen pointed to this approach so we should be OK.

```

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

```

```

395     else
396         return false
397     end
398 end
399 else
400 function registernumber(name)
401     local nt = createtoken(name)
402     if(luaregisterbasetable[nt.cmdname]) then
403         return nt.mode - luaregisterbasetable[nt.cmdname]
404     else
405         return false
406     end
407 end
408 end
409 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.

```

410 local attributes=setmetatable(
411 {},
412 {
413 __index = function(t,key)
414 return registernumber(key) or nil
415 end}
416 )
417 luatexbase.attributes = attributes
418 local attribute_count_name =
419     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 =
445         bytecode_count_name or "e@alloc@bytecode@count"
446 local function new_bytecode(name)
447     tex_setcount("global", bytecode_count_name,
448                 tex_count[bytecode_count_name] + 1)
449     if tex_count[bytecode_count_name] > 65534 then
450         luatexbase_error("No room for a new bytecode register")
451     end
452     luatexbase_log("Lua bytecode " .. (name or "") .. " = " ..
453                 tex_count[bytecode_count_name])
454     return tex_count[bytecode_count_name]
455 end
456 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.

```

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

```

5.17 Lua function allocation

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

```

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

```

```

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

```

5.18 Lua callback management

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

5.18.1 Housekeeping

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

Actually there are two tables: `realcallbacklist` directly contains the entries as described above while `callbacklist` only directly contains the already sorted entries. Other entries can be queried through `callbacklist` too which triggers a resort.

Additionally `callbackrules` describes the ordering constraints: It contains two element tables with the descriptions of the constrained callback implementations. It can additionally contain a `type` entry indicating the kind of rule. A missing value indicates a normal ordering constraint.

```

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

```

```

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

```

```

562         local post_name = post_entry.value.description
563         if not remaining[post_name] then
564             remaining[post_name] = name
565         end
566     end
567 end
568 local cycle = {first_name}
569 local index = 1
570 local last_name = first_name
571 repeat
572     cycle[last_name] = index
573     last_name = remaining[last_name]
574     index = index + 1
575     cycle[index] = last_name
576 until cycle[last_name]
577 local length = index - cycle[last_name] + 1
578 table.move(cycle, cycle[last_name], index, 1)
579 for i=2, length//2 do
580     cycle[i], cycle[length + 1 - i] = cycle[length + 1 - i], cycle[i]
581 end
582 error('Cycle occurred at ' .. table.concat(cycle, ' -> ', 1, length))
583 end
584 end
585 end
586 realcallbacklist[name] = list
587 t[name] = list
588 return list
589 end
590 })

```

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

```

591 local list, data, exclusive, simple, reverselist = 1, 2, 3, 4, 5
592 local types = {
593     list      = list,
594     data      = data,
595     exclusive = exclusive,
596     simple    = simple,
597     reverselist = reverselist,
598 }

```

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.)

```

599 local callbacktypes = callbacktypes or {

```

Section 8.2: file discovery callbacks.

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

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

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

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

Section 8.3: data processing callbacks.

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

Section 8.4: node list processing callbacks.

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

```

646 % mlist_to_hlist      = exclusive,
647 new_graf              = exclusive,

```

Section 8.5: information reporting callbacks.

```

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

```

Section 8.6: PDF-related callbacks.

```

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

```

Section 8.7: font-related callbacks.

```

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

```

Sometimes multiple callbacks correspond to a single underlying engine level callback. Then the engine level callback should be registered as long as at least one of these callbacks is in use. This is implemented through a shared table which counts how many of the involved callbacks are currently in use. The engine level callback is registered iff this count is not 0.

We add `mlist_to_hlist` directly to the list to demonstrate this, but the handler gets added later when it is actually defined.

All callbacks in this list are treated as user defined callbacks.

```

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

```

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

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

5.18.2 Handlers

The handler function is registered into the callback when the first function is added to this callback's list. Then, when the callback is called, the handler takes care of running all functions in the list. When the last function is removed from the callback's list, the handler is unregistered.

More precisely, the functions below are used to generate a specialized function (closure) for a given callback, which is the actual handler.

The way the functions are combined together depends on the type of the callback. There are currently 4 types of callback, depending on the calling convention of the functions the callback can hold:

simple is for functions that don't return anything: they are called in order, all with the same argument;

data is for functions receiving a piece of data of any type except node list head (and possibly other arguments) and returning it (possibly modified): the functions are called in order, and each is passed the return value of the previous (and the other arguments untouched, if any). The return value is that of the last function;

list is a specialized variant of *data* for functions filtering node lists. Such functions may return either the head of a modified node list, or the boolean values **true** or **false**. The functions are chained the same way as for *data* except that for the following. If one function returns **false**, then **false** is immediately returned and the following functions are *not* called. If one function returns **true**, then the same head is passed to the next function. If all functions return **true**, then **true** is returned, otherwise the return value of the last function not returning **true** is used.

reverselist is a specialized variant of *list* which executes functions in inverse order.

exclusive is for functions with more complex signatures; functions in this type of callback are *not* combined: An error is raised if a second callback is registered.

Handler for *data* callbacks.

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

Default for user-defined data callbacks without explicit default.

```
698 local function data_handler_default(value)
699   return value
700 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.

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

Handler for list callbacks.

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

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

```
725 local function list_handler_default(head)
726   return head
727 end
```

Handler for reverselist callbacks.

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

```

743     head = ret
744     end
745 end
746 return head
747 end
748 end

```

Handler for simple callbacks.

```

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

```

Default for user-defined simple callbacks without explicit default.

```

756 local function simple_handler_default()
757 end

```

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

```

758 local handlers = {
759   [data] = data_handler,
760   [exclusive] = exclusive_handler,
761   [list] = list_handler,
762   [reverselist] = reverselist_handler,
763   [simple] = simple_handler,
764 }
765 local defaults = {
766   [data] = data_handler_default,
767   [exclusive] = nil,
768   [list] = list_handler_default,
769   [reverselist] = list_handler_default,
770   [simple] = simple_handler_default,
771 }

```

5.18.3 Public functions for callback management

Defining user callbacks perhaps should be in package code, but impacts on `add_to_callback`. If a default function is not required, it may be declared as `false`. First we need a list of user callbacks.

```

772 local user_callbacks_defaults = {}

```

`create_callback` The allocator itself.

```

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

```

```

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

```

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

```

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

```

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

```

817 local function add_to_callback(name, func, description)
818   if not name or name == "" then
819     luatexbase_error("Unable to register callback:\n" ..
820                     "valid callback name required")
821   end
822   if not callbacktypes[name] or
823     type(func) ~= "function" or
824     not description or
825     description == "" then
826     luatexbase_error(
827       "Unable to register callback.\n\n"
828       .. "Correct usage:\n"
829       .. "add_to_callback(<callback>, <function>, <description>)"
830     )
831   end

```

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

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

Handle count for shared engine callbacks.

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

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

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

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

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

Keep user informed.

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

declare_callback_rule Add an ordering constraint between two callback implementations

```
864 local function declare_callback_rule(name, desc1, relation, desc2)
865   if not callbacktypes[name] or
866     not desc1 or not desc2 or
867     desc1 == "" or desc2 == "" then
868     luatexbase_error(
869       "Unable to create ordering constraint. "
870       .. "Correct usage:\n"
871       .. "declare_callback_rule(<callback>, <description_a>, <description_b>)"
872     )
873   end
```

```

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

```

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

```

907 local function remove_from_callback(name, description)
908     if not name or name == "" then
909         luatexbase_error("Unable to remove function from callback:\n" ..
910             "valid callback name required")
911     end
912     if not callbacktypes[name] or
913         not description or
914         description == "" then
915         luatexbase_error(
916             "Unable to remove function from callback.\n\n"
917             .. "Correct usage:\n"
918             .. "remove_from_callback(<callback>, <description>)"
919         )
920     end
921     local l = realcallbacklist[name]
922     if not l then
923         luatexbase_error(
924             "No callback list for '" .. name .. "'\n")
925     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.

```

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

```

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

```

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

```

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

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

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

```
983 local function callback_descriptions (name)
984   local d = {}
985   if not name
986     or name == ""
987     or not realcallbacklist[name]
988     or not callbacktypes[name]
989   then
990     return d
991   else
992     for k, i in pairs(callbacklist[name]) do
993       d[k]= i.description
994     end
995   end
996   return d
997 end
998 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!

```
999 local function uninstall()
1000   module_info(
1001     "luatexbase",
1002     "Uninstalling kernel luatexbase code"
1003   )
1004   callback.register = callback_register
1005   luatexbase = nil
1006 end
1007 luatexbase.uninstall = uninstall
```

`mlist_to_hlist` To emulate these callbacks, the “real” `mlist_to_hlist` is replaced by a wrapper calling the wrappers before and after.

```
1008 create_callback('pre_mlist_to_hlist_filter', 'list')
1009 create_callback('mlist_to_hlist', 'exclusive', node.mlist_to_hlist)
1010 create_callback('post_mlist_to_hlist_filter', 'reverselist')
1011 function shared_callbacks.mlist_to_hlist.handler(head, display_type, need_penalties)
1012   local current = call_callback("pre_mlist_to_hlist_filter", head, display_type, need_penalties)
1013   if current == false then
1014     flush_list(head)
1015     return nil
```

```
1016 end
1017 current = call_callback("mlist_to_hlist", current, display_type, need_penalties)
1018 local post = call_callback("post_mlist_to_hlist_filter", current, display_type, need_penal
1019 if post == false then
1020     flush_list(current)
1021     return nil
1022 end
1023 return post
1024 end

1025  $\langle$ /lua $\rangle$ 

Reset the catcode of @.
1026  $\langle$ tex $\rangle$ \catcode'\@=\etatcatcode\relax
```