ltlatex.dtx
(Lua\TeX-specific support)

David Carlisle and Joseph Wright*
2020/02/02

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 Lua\TeX version 6
  5.2 Older \LaTeX/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 function allocation 17
  5.18 Lua callback management 18

*Significant portions of the code here are adapted/simplified from the packages \latex 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 \LaTeX{} 2ε kernel level plus as a loadable file which can be used with plain \TeX{} and \LaTeX{}. 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 \texttt{\count} registers are used here for register allocation:

\begin{itemize}
\item \texttt{\e@alloc@attribute@count} Attributes (default 258)
\item \texttt{\e@alloc@ccodetable@count} Category code tables (default 259)
\item \texttt{\e@alloc@luafunction@count} Lua functions (default 260)
  \begin{itemize}
  \item \texttt{\e@alloc@whatsit@count} User whatsit (default 261)
  \item \texttt{\e@alloc@bytecode@count} Lua bytecodes (default 262)
  \item \texttt{\e@alloc@luachunk@count} Lua chunks (default 263)
  \end{itemize}
\end{itemize}

(\texttt{\count} 256 is used for \texttt{\newmarks} allocation and \texttt{\count} 257 is used for \texttt{\newXeTeXintercharclass} with Xe\TeX{}, with code defined in \texttt{ltfinal.dtx}). With any \LaTeX{} 2ε kernel from 2015 onward these registers are part of the block in the extended area reserved by the kernel (prior to 2015 the \LaTeX{} 2ε 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 \LaTeX{} format, however also extracted to the file \texttt{ltluatex.tex} which may be used with older \LaTeX{} formats, and with plain \TeX{}.

\begin{itemize}
\item \texttt{\newattribute{\texttt{attribute}}} \texttt{\newattribute{\texttt{attribute}}} \texttt{\newattribute{\texttt{attribute}}}Defines a named \texttt{\attribute}, indexed from 1 (\texttt{i.e.} \texttt{\attribute0} is never defined). Attributes initially have the marker value \texttt{~"7FFFFFFF (‘unset’) set by the engine.}
\item \texttt{\newcatcodetable{\texttt{catcodetable}}} \texttt{\newcatcodetable{\texttt{catcodetable}}}Defines a named \texttt{\catcodetable}, indexed from 1 (\texttt{\catcodetable0} is never assigned). A new catcode table will be populated with exactly those values assigned by Ini\TeX{} (as described in the Lua\TeX{} manual).
\item \texttt{\newluafunction{\texttt{function}}} \texttt{\newluafunction{\texttt{function}}}Defines a named \texttt{\luafunction}, indexed from 1. (Lua indexes tables from 1 so \texttt{\luafunction0} is not available).
\item \texttt{\newwhatstis{\texttt{whatstis}}} \texttt{\newwhatstis{\texttt{whatstis}}}Defines a custom \texttt{\whatstis}, indexed from 1.
\item \texttt{\newluabytecode{\texttt{bytecode}}} \texttt{\newluabytecode{\texttt{bytecode}}}Allocates a number for Lua bytecode register, indexed from 1.
\item \texttt{\newluachunkname{\texttt{chunkname}}} \texttt{\newluachunkname{\texttt{chunkname}}}Allocates a number for Lua chunk register, indexed from 1. Also enters the name of the register (without backslash) into the \texttt{lua.name} table to be used in stack traces.
\end{itemize}
Predefined category code tables with the obvious assignments. Note that the \latex and \atletter tables set the full Unicode range to the codes predefined by the kernel.

\setattribute{(attribute)}{(value)}
\unsetattribute{(attribute)}

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 \TeX interface

The ltluatex interface may be used with plain \TeX using \input{ltlatex}. This inputs ltluatex.tex which inputs etex.src (or etex.sty if used with \LaTeX) if it is not already input, and then defines some internal commands to allow the ltluatex interface to be defined.

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

4 Lua functionality

4.1 Allocators in Lua

\newattribute\newattribute{(attribute)}
\newattribute Returns an allocation number for the \langle attribute \rangle, indexed from 1. The attribute will be initialised with the marker value −7FFFFFFF (‘unset’). The attribute allocation sequence is shared with the \TeX code but this function does not define a token using \attributedef. The attribute name is recorded in the attributes table. A metatable is provided so that the table syntax can be used consistently for attributes declared in \TeX or Lua.

\newwhatsit\newwhatsit{(whatsit)}
\newwhatsit Returns an allocation number for the custom \langle whatsit \rangle, indexed from 1.

\newbytecode\newbytecode{(bytecode)}
\newbytecode Returns an allocation number for a bytecode register, indexed from 1. The optional \langle name \rangle argument is just used for logging.

\newchunkname\newchunkname{(chunkname)}
\newchunkname Returns an allocation number for a Lua chunk name for use with \directlua and \latelua, indexed from 1. The number is returned and also \langle name \rangle argument is added to the lua.name array at that index.

\newluafunction\newluafunction{(functionname)}
\newluafunction Returns an allocation number for a lua function for use with \luadef, \lateluafunction, and \directlua, indexed from 1. The optional \langle functionname \rangle argument is just used for logging.

These functions all require access to a named \TeX count register to manage their allocations. The standard names are those defined above for access from \TeX, e.g., \texttt{e@alloc@attribute@count}, but these can be adjusted by defining the variable \langle type \rangle.count.name before loading ltluatex.lua, for example
local attribute_count_name = "attributetracker"
require("ltluatex")

would use a \texttt{\count} (\texttt{\countdef}’d token) called \texttt{attributetracker} in place of \texttt{\e@alloc\@attribute\@count}.

4.2 \texttt{\textsc{La}}\texttt{u} access to \texttt{\textsc{T}eX} register numbers

\begin{verbatim}
registernumber \texttt{\textsc{luatexbase}.registernumber(\langle name\rangle)}
\end{verbatim}

Sometimes (notably in the case of Lua attributes) it is necessary to access a register \textit{by number} that has been allocated by \texttt{\textsc{T}eX}. This package provides a function to look up the relevant number using \textsc{Luatex}’s internal tables. After for example \texttt{\textbackslash newattribute\textbackslash myattrib}, \texttt{\textbackslash myattrib} would be defined by (say) \texttt{\textbackslash myattrib=\textbackslash attribute15}. \texttt{\textsc{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 \texttt{\textbackslash attributedef}, \texttt{\textbackslash countdef} or similar commands, the Lua value \texttt{false} is returned.

As an example, consider the input:

\begin{verbatim}
\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}
\end{verbatim}

If the demonstration code is processed with \textsc{Luatex}\texttt{\textsc{T}eX} then the following would be produced in the log and terminal output.

\begin{verbatim}
undefinedrubbish: \relax
  \texttt{bad input}
space: macro:->
  \texttt{bad input}
hbox: \texttt{hbox}
\end{verbatim}
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 \TeX\ 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 \TeX'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 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({name},metatype,{default}) Defines a user defined callback. The last argument is a default function or false.

call_callback  luatexbase.call_callback({name},...) Calls a user defined callback with the supplied arguments.

5  Implementation

1 (*2ekernel|tex|latexrelease)
2 *(2ekernel|latexrelease)\ifx\directlua\@undefined\else
3
4 \ifnum\luatexversion<60 %
5 \wlog{***************************************************}
6 \wlog{* LuaTeX version too old for ltlatex support *}
7 \wlog{***************************************************}
8 \expandafter\endinput
9 \fi
10 \fi

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 ltlatexex support *}
8 \wlog{***************************************************************************}
9 \expandafter\endinput
10 \fi

5.2 Older \LaTeX/Plain \TeX{} setup

11 (*tex)

Older \LaTeX{} 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 \if\@alloc\@undefined
14 \latex\documentclass\@undefined
15 \latex\loccount\@undefined
16 \latex\input{etex.src}\
17 \else
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}
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 L\(\text{a}\)b\(\text{T}\)X-specific code, but does not define the correct range for L\(\text{a}\)b\(\text{T}\)X.

2015-07-13 higher range in luatex.

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.
Two simple \LaTeX macros used in \texttt{ltlatex.sty}.

\begin{verbatim}
\longdef\@gobble#1{}
\longdef\@firstofone#1{#1}
\end{verbatim}

Fix up allocations not to clash with \texttt{etex.src}.

\begin{verbatim}
\expandafter\csname newcount\endcsname\e@alloc@attribute@count
\expandafter\csname newcount\endcsname\e@alloc@ccodetable@count
\expandafter\csname newcount\endcsname\e@alloc@luafunction@count
\expandafter\csname newcount\endcsname\e@alloc@whatst@count
\expandafter\csname newcount\endcsname\e@alloc@bytecode@count
\expandafter\csname newcount\endcsname\e@alloc@luachunk@count
\end{verbatim}

End of conditional setup for plain \TeX / old \LaTeX.

\begin{verbatim}
\fi
\end{verbatim}

\section*{5.3 Attributes}

\paragraph*{\texttt{\newattribute}} As is generally the case for the Lua\TeX\ registers we start here from 1. Notably, some code assumes that \texttt{\attribute0} is never used so this is important in this case.

\begin{verbatim}
\ifx\e@alloc@attribute@count\@undefined
\countdef\e@alloc@attribute@count=258
\fi
\def\newattribute#1{%
\e@alloc\attribute\attributedef
\e@alloc@attribute@count\m@ne\e@alloc@top#1%
}
\end{verbatim}

\paragraph*{\texttt{\setattribute}} Handy utilities.

\begin{verbatim}
\def\setattribute#1#2{#1=\numexpr#2\relax}
\def\unsetattribute#1{#1=-"7FFFFFFF\relax}
\end{verbatim}

\section*{5.4 Category code tables}

\paragraph*{\texttt{\newcatcodetable}} Category code tables are allocated with a limit half of that used by Lua\TeX\ 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.

\begin{verbatim}
\ifx\e@alloc@ccodetable@count\@undefined
\countdef\e@alloc@ccodetable@count=259
\fi
\def\newcatcodetable#1{%
\e@alloc\catcodetable\chardef
\e@alloc@ccodetable@count\m@ne\e@alloc@top#1%
}\end{verbatim}
Save a small set of standard tables. The Unicode data is read here in using a parser simplified from that in `load-unicode-data`: only the nature of letters needs to be detected.

```latex
\catcodetable@initex
\catcodetable@string
\catcodetable@latex
\catcodetable@atletter
\newcatcodetable\catcodetable@initex
\newcatcodetable\catcodetable@string
\newcatcodetable\catcodetable@latex
\newcatcodetable\catcodetable@atletter
\begingroup
\def\setrangecatcode#1#2#3{%
  \ifnum#1>#2 %
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {%
    \catcode#1=#3 %
    \setrangecatcode{\number\numexpr#1 + 1\relax}{#2}{#3}
  }%
%
  \@firstofone{%
    \catcodetable\catcodetable@initex
    \catcode0=12 %
    \catcode13=12 %
    \catcode37=12 %
    \setrangecatcode{65}{90}{12}%
    \setrangecatcode{97}{122}{12}%
    \catcode92=12 %
    \catcode127=12 %
    \savecatcodetable\catcodetable@string
  }%
%
  }%
}
\@firstofone{%
  \catcodetable@latex
  \catcodetable@atletter
  \begingroup
  \def\parseunicodedataI#1;#2;#3;#4\relax{% 
    \parseunicodedataII#1;#3;#2 First>elax
  }%
  \def\parseunicodedataII#1;#2;#3 First>#4\relax{%
    \ifx\relax#4\relax
      \expandafter\parseunicodedataIII
    \else
      \expandafter\parseunicodedataIV
    \fi
  }%
  \def\parseunicodedataIII#1#2#3\relax{%
    \ifx\relax#4\relax
      \expandafter\parseunicodedataIV
    \else
      \expandafter\parseunicodedataV
    \fi
  }%
  \def\parseunicodedataIV#1#2#3\relax{% 
    \ifnum 0%
      \if #21\fi
  ```
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.

\ifx\e@alloc@luafunction@count\@undefined
\countdef\e@alloc@luafunction@count=260
\fi
\def\newluafunction{\e@alloc\luafunction\e@alloc@chardef\e@alloc@luafunction@count\m@ne\e@alloc@top}
\e@alloc@luafunction@count=\z@

5.6 Custom whatsits

\newwhatsit

These are only settable from Lua but for consistency are definable here.
\ifx\e@alloc@whatsit@count\@undefined
\@firstofone
\catcode\verbatim=11 %
\catcode\verbatim=11 %
\catcode\verbatim=12 %
\chardef\verbatim=
umexpr\count16 + 1\relax
\openin\verbatim=UnicodeData.txt %
\loop
\unless\ifeof\verbatim%
\read\verbatim to \verbatimline
\ifx\verbatimline\storedpar
\expandafter\parseunicodedataI\verbatimline\relax
\fi
\repeat
\closein\verbatim
\@firstofone
\catcode\verbatim=12 %
\savecatcodetable\catcodetable@latex
\catcode\verbatim=11 %
\savecatcodetable\catcodetable@atletter
}\endgroup
5.7 Lua bytecode registers
\newluabytecode

These are only settable from Lua but for consistency are definable here.
\ifx\e@alloc@bytecode@count\@undefined
\countdef\e@alloc@bytecode@count=262
\fi
\def\newluabytecode#1{\e@alloc\luabytecode\e@alloc@chardef\e@alloc@bytecode@count\ifnum=1\e@alloc@top#1\fi}
\e@alloc@bytecode@count=\z@  

5.8 Lua chunk registers
\newluachunkname

As for bytecode registers, but in addition we need to add a string to the lua.name table to use in stack tracing. We use the name of the command passed to the allocator, with no backslash.
\ifx\e@alloc@luachunk@count\@undefined
\countdef\e@alloc@luachunk@count=263
\fi
\def\newluachunkname#1{\e@alloc\luachunk\e@alloc@chardef\e@alloc@luachunk@count\ifnum=1\e@alloc@top#1\fi{\escapechar\ifnum=1\directlua{lua.name[\the\allocationnumber]=\string#1}}}\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.
\begin{group}
\edef\attributezero=0 \chardef \charzero =0 \CountZero =0 \dimenzero =0 \mathchardef \mathcharzero =0 \muskipdef \muskipzero =0 \skipdef \skipzero =0
\end{group}

Note name change required on older luatex, for hash table access.
\toksdef \tokszero =0 %
\directlua{require("ltluatex")}
\endgroup
\newluafunction{LuaTeX}%
\newluafunction{LuaTeX}uninstall()
\EndIncludeInRelease
\IncludeInRelease{2017/01/01}%
\IncludeInRelease{0000/00/00}%
\EndIncludeInRelease
\everyjob, if luaotfload is available, load it and switch to TU.
\fontencoding{TU in everyjob}%
\fontencoding{TU}
\let\encodingdefault\f@encoding
\ifx\directlua\@undefined\else
\everyjob\expandafter{\the\everyjob}
\directlua{\if\xpcall(function ()% require('luaotfload-main')% then % local _void = luaotfload.main ()% else % texio.write_nl('Error in luaotfload: reverting to OT1')% tex.print('%\string\def\string\encodingdefault{OT1}')% end %}%\let\f@encoding\encodingdefault\let\csname ver@luaotfload.sty\endcsname\fmtversion\EndIncludeInRelease
\IncludeInRelease{0000/00/00}%
Some set up for the Lua module which is needed for all of the Lua functionality added here.

```lua
luatexbase = luatexbase or {}
```

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

```lua
local string_gsub = string.gsub
local tex_count = tex.count
local tex_setattribute = tex.setattribute
local tex_setcount = tex.setcount
local texio_write_nl = texio.write_nl
```

5.11 Lua module utilities

5.11.1 Module tracking

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

```lua
local modules = modules or {}
```

Local function to write to the log.

```lua
local function luatexbase_log(text)
  texio_write_nl("log", text)
end
```

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

```lua
local function provides_module(info)
  if not (info and info.name) then
    luatexbase_error("Missing module name for provides_module")
  end
  return text = ".. text") or ""
end
```

"Lua module: " .. info.name
.. spaced(info.date)
.. spaced(info.version)
.. spaced(info.description)
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 \LaTeX. 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.

```lua
local function msg_format(mod, msg_type, text)
  local leader = ""
  local cont
  local first_head
  if mod == "LaTeX" then
    cont = string.gsub(leader, ".", " ")
    first_head = leader .. "LaTeX: "
  else
    first_head = leader .. "Module " .. msg_type
    cont = "(" .. mod .. ")"
    .. string.gsub(first_head, ".", ", ")
    first_head = leader .. "Module " .. mod .. ", " .. msg_type .. ";"
  end
  if msg_type == "Error" then
    first_head = \n .. first_head
  end
  if string.sub(text,-1) ~= \n then
    text = text .. " 
  end
  return first_head .. " 
  .. string.gsub(\n .. tex.inputlineno, \n", \n" .. cont .. " 
 )
end
```

Write messages.

```lua
local function module_info(mod, text)
texio_write_nl("log", msg_format(mod, "Info", text))
end
luatexbase.module_info = module_info
```

```lua
local function module_warning(mod, text)
texio_write_nl("term and log",msg_format(mod, "Warning", text))
end
luatexbase.module_warning = module_warning
```

```lua
local function module_error(mod, text)
  error(msg_format(mod, "Error", text))
end
```
end
luatexbase.module_error = module_error

Dedicated versions for the rest of the code here.

function luatexbase_warning(text)
    module_warning("luatexbase", text)
end

function luatexbase_error(text)
    module_error("luatexbase", text)
end

5.12 Accessing register numbers from Lua

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

```lua
local luaregisterbasetable = { }
local registermap = {
attributezero = "assign_attr" ,
charzero = "char_given" ,
CountZero = "assign_int" ,
dimenzero = "assign_dimen" ,
mathcharzero = "math_given" ,
muskipzero = "assign_mu_skip" ,
skipzero = "assign_skip" ,
tokszero = "assign_toks" ,
}
local createtoken
if tex.luatable > 81 then
    createtoken = token.create
elseif tex.luatable > 79 then
    createtoken = newtoken.create
end
local hashtokens = tex.hashtokens()
local luatexversion = tex.luatable
for i,j in pairs (registermap) do
    if luatexversion < 80 then
        luaregisterbasetable[hashtokens[i][1]] = hashtokens[i][2]
    else
        luaregisterbasetable[j] = createtoken(i).mode
    end
end

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

```
else
  return false
end

else
  function registernumber(name)
    local nt = createtoken(name)
    if(luaregisterbasetable[nt.cmdname]) then
      return nt.mode - luaregisterbasetable[nt.cmdname]
    else
      return false
    end
  end
end

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.

local attributes=setmetatable(
  {},
  {
    __index = function(t,key)
      return registernumber(key) or nil
    end}
)

luatexbase.attributes = attributes

local attribute_count_name =
  attribute_count_name or "e@alloc@attribute@count"

local function new_attribute(name)
  tex_setcount("global", attribute_count_name,
    tex_count[attribute_count_name] + 1)
  if tex_count[attribute_count_name] > 65534 then
    luatexbase_error("No room for a new \attribute")
  end

  attributes[name]= tex_count[attribute_count_name]
  luatexbase_log("Lua-only attribute " .. name .. " = " ..
    tex_count[attribute_count_name])

  return tex_count[attribute_count_name]
end

luatexbase.new_attribute = new_attribute

5.14  Custom whatsit allocation

new_whatsit  Much the same as for attribute allocation in Lua.

local whatsis_count_name = whatsis_count_name or "e@alloc@whatsit@count"

local function new_whatsit(name)
  tex_setcount("global", whatsis_count_name,
    tex_count[whatsit_count_name] + 1)
  if tex_count[whatsit_count_name] > 65534 then
    luatexbase_error("No room for a new custom whatsit")
  end
luatexbase.log("Custom whatsit " .. (name or "") .. " = " ..
    tex_count[whatsit_count_name])
return tex_count[whatsit_count_name]
end
luatexbase.new_whatsit = new_whatsit

5.15 Bytecode register allocation

new_bytecode  Much the same as for attribute allocation in Lua. The optional ⟨name⟩ argument
is used in the log if given.

local bytecode_count_name =
  bytecode_count_name or "e@alloc@bytecode@count"
local function new_bytecode(name)
  tex_setcount("global", bytecode_count_name,
    tex_count[bytecode_count_name] + 1)
  if tex_count[bytecode_count_name] > 65534 then
    luatexbase_error("No room for a new bytecode register")
    return tex_count[bytecode_count_name]
  end
  luatexbase.log("Lua bytecode " .. (name or "") .. " = " ..
    tex_count[bytecode_count_name])
  return tex_count[bytecode_count_name]
end
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.

local chunkname_count_name =
  chunkname_count_name or "e@alloc@luachunk@count"
local function new_chunkname(name)
  tex_setcount("global", chunkname_count_name,
    tex_count[chunkname_count_name] + 1)
  local chunkname_count = tex_count[chunkname_count_name]
  chunkname_count = chunkname_count + 1
  if chunkname_count > 65534 then
    luatexbase_error("No room for a new chunkname")
  end
  lua.name[chunkname_count]=name
  luatexbase.log("Lua chunkname " .. (name or ") .. ") = " ..
    chunkname_count .. "\n")
  return chunkname_count
end
luatexbase.new_chunkname = new_chunkname

5.17 Lua function allocation

new_luafunction  Much the same as for attribute allocation in Lua. The optional ⟨name⟩ argument
is used in the log if given.

local luafunction_count_name =
  luafunction_count_name or "e@alloc@luafunction@count"
local function new_luafunction(name)
  tex_setcount("global", luafunction_count_name,
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 \texttt{func} and the identifying description as \texttt{description}. Only callbacks with a non-empty list of functions have an entry in this list.

\begin{verbatim}
lua{local callbacklist = callbacklist or { }
    local list, data, exclusive, simple, reverselist = 1, 2, 3, 4, 5
    local types = {
        list = list,
        data = data,
        exclusive = exclusive,
        simple = simple,
        reverselist = reverselist,
    }

    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
\end{verbatim}

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

\begin{verbatim}
lua{local callbacktypes = callbacktypes or {
    Section 8.2: file discovery callbacks.
    find_read_file = exclusive,
    find_write_file = exclusive,
    find_font_file = data,
\end{verbatim}

18
find_output_file  = data,
find_format_file  = data,
find_vf_file      = data,
find_map_file     = data,
find_enc_file     = data,
find_pk_file      = data,
find_data_file    = data,
find_opentype_file= data,
find_truetype_file= data,
find_type1_file   = data,
find_image_file   = data,
open_read_file    = exclusive,
read_font_file    = exclusive,
read_vf_file      = exclusive,
read_map_file     = exclusive,
read_enc_file     = exclusive,
read_pk_file      = exclusive,
read_data_file    = exclusive,
read_truetype_file= exclusive,
read_type1_file   = exclusive,
read_opentype_file= exclusive,

find_cidmap_file  = data,
read_cidmap_file  = exclusive,

Section 8.3: data processing callbacks.
process_input_buffer = data,
process_output_buffer = data,
process_jobname     = data,

Section 8.4: node list processing callbacks.
contribute_filter   = simple,
buildpage_filter    = simple,
build_page_insert   = exclusive,
pre_linebreak_filter= list,
linebreak_filter    = exclusive,
append_to_vlist_filter= exclusive,
post_linebreak_filter= reverselist,
hpack_filter        = list,
vpack_filter        = list,
hpack_quality       = list,
vpack_quality       = list,
pre_output_filter   = list,
process_rule        = exclusive,
hyphenate           = simple,
ligaturing          = simple,
kerning             = simple,
insert_local_par    = simple,
pre_mlist_to_hlist_filter = list,
mlist_to_hlist      = exclusive,
post_mlist_to_hlist_filter = reverselist,
new_graf            = simple,
Section 8.5: information reporting callbacks.

```
pre_dump = simple,
start_run = simple,
stop_run = simple,
start_page_number = simple,
stop_page_number = simple,
show_error_hook = simple,
show_warning_message = simple,
show_error_message = simple,
show_lua_error_hook = simple,
start_file = simple,
stop_file = simple,
call_edit = simple,
finish_synctex = simple,
wrapup_run = simple,
```

Section 8.6: PDF-related callbacks.

```
finish_pdf_file = data,
finish_pdf_page = data,
pages_objnum_provider = data,
page_order_index = data,
process_pdf_image_content = data,
```

Section 8.7: font-related callbacks.

```
define_font = exclusive,
glyph_info = exclusive,
glyph_not_found = exclusive,
glyph_stream_provider = exclusive,
make_extensible = exclusive,
font_descriptor_objnum_provider = exclusive,
```

```
callback.register = callback.register or callback.register
```

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:

```
callback.register(function callback()
luatexbase.callbacktypes=callbacktypes
```
**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.

```lua
local function data_handler(name)
  return function(data, ...)
    for _,i in ipairs(callbacklist[name]) do
      data = i.func(data, ...)
    end
  end
  return data
end
```

Default for user-defined **data** callbacks without explicit default.

```lua
local function data_handler_default(value)
  return value
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.

```lua
local function exclusive_handler(name)
  return function(...)
    return callbacklist[name][1].func(...) end
end
```

Handler for **list** callbacks.

```lua
local function list_handler(name)
  return function(head, ...)
    local ret
    local alltrue = true
    for _,i in ipairs(callbacklist[name]) do
      ret = i.func(head, ...)
      if ret == false then alltrue = false break end
    end
    return alltrue
  end
end
```
if ret == false then
  luatexbase_warning(
    "Function \" .. i.description .. \" returned false\n"
    .. "in callback \" .. name .. \"\n"
  )
  break
end
if ret ~= true then
  alltrue = false
  head = ret
end
return alltrue and true or head
end

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

local function list_handler_default() return true end

Handler for reverselist callbacks.

local function reverselist_handler(name)
  return function(head, ...)
    local ret = true
    local alltrue = true
    local callbacks = callbacklist[name]
    for i = #callbacks, 1, -1 do
      local cb = callbacks[i]
      ret = cb.func(head, ...)
      if ret == false then
        luatexbase_warning(
          "Function \" .. cb.description .. \" returned false\n"
          .. "in callback \" .. name .. \"\n"
        )
        break
      end
      if ret ~= true then
        alltrue = false
        head = ret
      end
    end
    return alltrue and true or head
  end
end

Handler for simple callbacks.

local function simple_handler(name)
  return function(...) for _,i in ipairs(callbacklist[name]) do i.func(...) end end

Default for user-defined simple callbacks without explicit default.
local function simple_handler_default()
  end

  Keep a handlers table for indexed access and a table with the corresponding
default functions.
local handlers = {
  [data] = data_handler,
  [exclusive] = exclusive_handler,
  [list] = list_handler,
  [reverselist] = reverselist_handler,
  [simple] = simple_handler,
}

local defaults = {
  [data] = data_handler_default,
  [exclusive] = nil,
  [list] = list_handler_default,
  [reverselist] = list_handler_default,
  [simple] = simple_handler_default,
}

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.

local user_callbacks_defaults = {
  pre_mlist_to_hlist_filter = list_handler_default,
  mlist_to_hlist = node.mlist_to_hlist,
  post_mlist_to_hlist_filter = list_handler_default,
}

create_callback  The allocator itself.

local function create_callback(name, ctype, default)
  local ctype_id = types[ctype]
  if not name or name == "" or not ctype_id
     then
      lualatexbase_error("Unable to create callback:
          " ..
          "valid callback name and type required")
  end
  if callbacktypes[name] then
    lualatexbase_error("Unable to create callback ‘" .. name ..
        "’: \ncallback is already defined")
  end
  default = default or defaults[ctype_id]
  if not default then
    lualatexbase_error("Unable to create callback ‘" .. name ..
        "’: \ndefault is required for ‘" .. ctype ..
        "’ callbacks")
    elseif type (default) ~= "function" then
      lualatexbase_error("Unable to create callback ‘" .. name ..
        "’: \ndefault is not a function")
  end
  user_callbacks_defaults[name] = default
callbacktypes[name] = ctype_id
end
luatexbase.create_callback = create_callback

call_callback Call a user defined callback. First check arguments.
local function call_callback(name,...)
  if not name or name == "" then
    luatexbase_error("Unable to create callback:
" ..
    "valid callback name required")
  end
  if user_callbacks_defaults[name] == nil then
    luatexbase_error("Unable to call callback " .. name
    .. ":\unknown or empty")
  end
  local l = callbacklist[name]
  local f
  if not l then
    f = user_callbacks_defaults[name]
  else
    f = handlers[callbacktypes[name]](name)
  end
  return f(...)
end
luatexbase.call_callback=call_callback

add_to_callback Add a function to a callback. First check arguments.
local function add_to_callback(name, func, description)
  if not name or name == "" then
    luatexbase_error("Unable to register callback:
" ..
    "valid callback name required")
  end
  if not callbacktypes[name] or
    type(func) ~= "function" or
    not description or
    description == "" then
    luatexbase_error(
    "Unable to register callback.\n\n" .. "Correct usage:\n" .. "add_to_callback(<callback>, <function>, <description>)"
  )
end

Then test if this callback is already in use. If not, initialise its list and register the
proper handler.
local l = callbacklist[name]
if l == nil then
  l = { }
callbacklist[name] = l

If it is not a user defined callback use the primitive callback register.
if user_callbacks_defaults[name] == nil then
  callback_register(name, handlers[callbacktypes[name]](name))
end
end
Actually register the function and give an error if more than one exclusive one is registered.

```lua
local f = {
    func = func,
    description = description,
}
local priority = #l + 1
if callbacktypes[name] == exclusive then
    if #l == 1 then
        luatexbase_error("Cannot add second callback to exclusive function\n" ..
                           "name .. ")
    end
end
if callbacktypes[name] == exclusive then
    if #l == 1 then
        luatexbase_error("Cannot add second callback to exclusive function\n" ..
                           "name .. ")
    end
end
```

Keep user informed.

```lua
luatexbase_log(
    "Inserting '" .. description .. "' at position " ..
    priority .. " in '" .. name .. "]
)
```

Remove a function from a callback. First check arguments.

```lua
local function remove_from_callback(name, description)
    if not name or name == "" then
        luatexbase_error("Unable to remove function from callback:\n" ..
                       "valid callback name required")
    end
    if not callbacktypes[name] or
        not description or
        description == "" then
        luatexbase_error(
            "Unable to remove function from callback:\n\n            Correct usage:\n            ".. remove_from_callback(<callback>, <description>)"
        )
    end
    local l = callbacklist[name]
    if not l then
        luatexbase_error(
            "No callback list for '" .. name .. "]"
        )
    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.

    local index = false
    for i, j in ipairs(l) do
        if j.description == description then
            index = i
            break
        end
    end
```
if not index then
  luatexbase_error(
    "No callback '" .. description .. "; registered for '" ..
    name .. 
    \n"")
end
local cb = l[index]
table.remove(l, index)
luatexbase_log(
  "Removing '" .. description .. "; from '" .. name .. ")."
)
if #l == 0 then
  callbacklist[name] = nil
  callback_register(name, nil)
end
return cb.func,cb.description
end
luatexbase.remove_from_callback = remove_from_callback
in_callback Look for a function description in a callback.
local function in_callback(name, description)
  if not name
    or name == ""
    or not callbacklist[name]
    or not callbacktypes[name]
    or not description then
    return false
  end
  for _, i in pairs(callbacklist[name]) do
    if i.description == description then
      return true
    end
  end
  return false
end
luatexbase.in_callback = in_callback
disable_callback As we subvert the engine interface we need to provide a way to access this func-
tionality.
local function disable_callback(name)
  if(callbacklist[name] == nil) then
    callback_register(name, false)
  else
    luatexbase_error("Callback list for " .. name .. " not empty")
  end
end
luatexbase.disable_callback = disable_callback
callback_descriptions List the descriptions of functions registered for the given callback.
local function callback_descriptions (name)
  local d = {}
  if not name
    or name == ""
    or not callbacklist[name]
    or not callbacktypes[name]
then
return d
else
for k, i in pairs(callbacklist[name]) do
d[k] = i.description
end
end
return d
end
luatexbase.callback_descriptions = callback_descriptions

uninstall Unlike at the T\TeX level, we have to provide a back-out mechanism here at the same time as the rest of the code. This is not meant for use by anything other than latexrelease: as such this is deliberately not documented for users!
llocal function uninstall()
module_info(
    "luatexbase",
    "Uninstalling kernel luatexbase code"
)
callback.register = callback_register
luatexbase = nil
end
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.
callback_register("mlist\_to\_hlist", function(head, display_type, need_penalties)
local current = call_callback("pre\_mlist\_to\_hlist\_filter", head, display_type, need_penalties)
if current == false then
    flush\_list(head)
    return nil
elseif current == true then
    current = head
end
current = call_callback("mlist\_to\_hlist", current, display_type, need_penalties)
local post = call_callback("post\_mlist\_to\_hlist\_filter", current, display_type, need_penalties)
if post == true then
    return current
elseif post == false then
    flush\_list(current)
    return nil
end
return post
end)

⟨/lua⟩
Reset the catcode of \@. 
⟨tex\catcode"\@=\etatcatcode\relax