ltluatex.dtx
(LuatEX-specific support)

David Carlisle and Joseph Wright*

2021/12/27

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 ETeX/Plain TeX setup . . . . . . . . . . . . . . . . . . . . . 6
5.3 Attributes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
5.4 Category code tables . . . . . . . . . . . . . . . . . . . . . . . . . . 8
5.5 Named Lua functions . . . . . . . . . . . . . . . . . . . . . . . . . . 10
5.6 Custom whatsits . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
5.7 Lua bytecode registers . . . . . . . . . . . . . . . . . . . . . . . . . 11
5.8 Lua chunk registers . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
5.9 Lua loader . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
5.10 Lua module preliminaries . . . . . . . . . . . . . . . . . . . . . . . . 13
5.11 Lua module utilities . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
5.12 Accessing register numbers from Lua . . . . . . . . . . . . . . . . . 15
5.13 Attribute allocation . . . . . . . . . . . . . . . . . . . . . . . . . . 16
5.14 Custom whatsit allocation . . . . . . . . . . . . . . . . . . . . . . . 17
5.15 Bytecode register allocation . . . . . . . . . . . . . . . . . . . . . 17
5.16 Lua chunk name allocation . . . . . . . . . . . . . . . . . . . . . . 18
5.17 Lua function allocation . . . . . . . . . . . . . . . . . . . . . . . . 18
5.18 Lua callback management . . . . . . . . . . . . . . . . . . . . . . . 18

*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 \LaTeX\ 2ε kernel level plus as a loadable
file which can be used with plain TeX and \LaTeX.\n
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.\n
The following \count registers are used here for register allocation:

\begin{itemize}
  \item \e@alloc@attribute@count \ Attribute counts (default 258)
  \item \e@alloc@ccodetable@count \ Category code table counts (default 259)
  \item \e@alloc@luafunction@count \ Lua function counts (default 260)
    \begin{itemize}
      \item \e@alloc@whatsit@count \ User whatsit counts (default 261)
      \item \e@alloc@bytecode@count \ Lua bytecode counts (default 262)
      \item \e@alloc@luachunk@count \ Lua chunk counts (default 263)
    \end{itemize}
\end{itemize}

(\count 256 is used for \newmarks allocation and \count 257 is used for
\newXeTeXintercharclass with XeTeX, with code defined in 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 ltluatex.tex which may be used with
older \LaTeX\ formats, and with plain TeX.

\begin{itemize}
  \item \newattribute \newattribute{\langle attribute \rangle}
    \begin{itemize}
      \item Defines a named attribute, indexed from 1 (i.e. \attribute0 is never defined).
      \item Attributes initially have the marker value \texttt{-\textasciitilde7FFFFFFF} (‘unset’) set by the engine.
    \end{itemize}
  \item \newcatcodetable \newcatcodetable{\langle catcodetable \rangle}
    \begin{itemize}
      \item Defines a named \catcodetable, indexed from 1 (\catcodetable0 is never as-
        signed). A new catcode table will be populated with exactly those values assigned
        by IniTeX (as described in the LuaTeX manual).
    \end{itemize}
  \item \newluafunction \newluafunction{\langle function \rangle}
    \begin{itemize}
      \item Defines a named \luafunction, indexed from 1. (Lua indexes tables from 1 so
        \luafunction0 is not available).
    \end{itemize}
  \item \newluacmd \newluacmd{\langle function \rangle}
    \begin{itemize}
      \item Like \newluafunction, but defines the command using \luadef instead of just
        assigning an integer.
    \end{itemize}
  \item \newprotectedluacmd \newprotectedluacmd{\langle function \rangle}
    \begin{itemize}
      \item Like \newluacmd, but the defined command is not expandable.
    \end{itemize}
  \item \newwhatsit \newwhatsit{\langle whatsit \rangle}
    \begin{itemize}
      \item Defines a custom \whatst, indexed from 1.
    \end{itemize}
  \item \newluabytecode \newluabytecode{\langle bytecode \rangle}
\end{itemize}
\newluachunkname\langle chunkname \rangle

Allocates a number for Lua bytecode register, indexed from 1.

\newluachunkname\{\langle chunkname \rangle\}

Allocates a number for Lua chunk register, indexed from 1. Also enters the name of the register (without backslash) into the lua.name table to be used in stack traces.

\setattribute\{\langle attribute \rangle\}\{\langle value \rangle\}

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\{\langle attribute \rangle\}

\unsetattribute\{\langle attribute \rangle\}

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{ltluatex}. 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

\new_attribute\ltexbase.new_attribute\{\langle attribute \rangle\}

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.

\new_whatsit\ltexbase.new_whatsit\{\langle whatsit \rangle\}

Returns an allocation number for the custom \langle whatsit \rangle, indexed from 1.

\new_bytecode\ltexbase.new_bytecode\{\langle bytecode \rangle\}

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

\new_chunkname\ltexbase.new_chunkname\{\langle chunkname \rangle\}

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.

\new_luafunction\ltexbase.new_luafunction\{\langle functionname \rangle\}

Returns an allocation number for a lua function for use with \luafunction, \lateluafuncti and \luadef, 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. “@alloc@attribute@count, but these can be adjusted by defining the variable \texttt{(type)\_count\_name} before loading \texttt{ltluatex.lua}, for example

\begin{verbatim}
local attribute_count_name = "attributetracker"
require("ltluatex")
\end{verbatim}

would use a \TeX\ \texttt{\count\_\countdef’\,d\ token} called \texttt{attributetracker} in place of \texttt{\@alloc@attribute@count}.

### 4.2 Lua access to \TeX\ register numbers

\texttt{registernumber luatexbase.registernumber(\texttt{(name)})}

Sometimes (notably in the case of Lua attributes) it is necessary to access a register by number that has been allocated by \TeX. This package provides a function to look up the relevant number using Lua\TeX’s internal tables. After for example \texttt{\newattribute\myattrib, \myattrib would be defined by (say) \texttt{\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 \texttt{\attributedef, \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 Lua\TeX then the following would be produced in the log and terminal output.
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 call-
back function with \textit{(description)} from the \textit{(callback)}. The removed function and its description are returned as the results of this function.

\textbf{in\_callback} \texttt{luatexbase.in\_callback\((\textit{callback}), \textit{(description)}\)} Checks if the \textit{(description)} matches one of the functions added to the list for the \textit{(callback)}, returning a boolean value.

\textbf{disable\_callback} \texttt{luatexbase.disable\_callback\((\textit{callback})\)} Sets the \textit{(callback)} to \texttt{false} as described in the Lua\TeX{} manual for the underlying \texttt{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.

\textbf{callback\_descriptions} A list of the descriptions of functions registered to the specified callback is returned. {} is returned if there are no functions registered.

\textbf{create\_callback} \texttt{luatexbase.create\_callback\((\textit{name}),\textit{metatype},\textit{(default)})} Defines a user defined callback. The last argument is a default function or \texttt{false}.

\textbf{call\_callback} \texttt{luatexbase.call\_callback\((\textit{name}), . . .\)} Calls a user defined callback with the supplied arguments.

5 Implementation

5.1 Minimum Lua\TeX{} version

Lua\TeX{} 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 \texttt{require()}: from version 0.60, Lua\TeX{} will correctly find Lua files in the \texttt{texmf} tree without 'help'.

\texttt{\ifnum\luatexversion<60 \%}
\texttt{\wlog{***************************************************}}
\texttt{\wlog{* LuaTeX version too old for ltluatex support *}}
\texttt{\wlog{***************************************************}}
\texttt{\expandafter\endinput}
\texttt{\fi}

Two simple \LaTeX{} macros from \texttt{ltdefns.dtx} have to be defined here because \texttt{ltdefns.dtx} is not loaded yet when \texttt{ltluatex.dtx} is executed.

5.2 Older \LaTeX/Plain \TeX{} setup

Older \LaTeX{} formats don’t have the primitives with ‘native’ names: sort that out. If they already exist this will still be safe.

\texttt{\directlua{tex.enableprimitives('',tex.extraprimitives('luatex'))}}
\texttt{\ifx\e@alloc\@undefined}

In pre-2014 \LaTeX{}, or plain \TeX{}, load \texttt{etex.{sty,src}}.

\texttt{\ifx\documentclass\@undefined}
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.
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.
Define\e@alloc as in latex (the existing macros in etex.src hard to extend to further register types as they assume specific 26x and 27x count range. For compatibility the existing register allocation is not changed.
\def\e@alloc#1#2#3#4#5#6{%  
\global\advance#3\@ne
\e@ch@ck{#3}{#4}{#5}#1%
\allocationnumber#3\relax
\global#2#6\allocationnumber
\wlog{\string#6=\string#1\the\allocationnumber}}%
\gdef\e@ch@ck#1#2#3#4{%  
\ifnum#1<#2\else
\ifnum#1=#2\relax
\ifx\count#4\advance#1 10 \fi
\fi
\ifnum#1<#3\relax
\else
\errmessage{No room for a new \string#4}%
\fi
\fi
\fi}%

Fix up allocations not to clash with \etex\src.
\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@whatsit@count
\expandafter\csname newcount\endcsname\e@alloc@bytecode@count
\expandafter\csname newcount\endcsname\e@alloc@luachunk@count

End of conditional setup for plain \texttt{\LaTeX} / old \texttt{\LaTeX}.
\fi
⟨\tex⟩

\section{Attributes}
\texttt{\newattribute} As is generally the case for the Lua\texttt{\LaTeX} registers we start here from 1. Notably, some code assumes that \texttt{\attribute0} is never used so this is important in this case.
\ifx\e@alloc@attribute@count\@undefined
\countdef\e@alloc@attribute@count=258
\e@alloc@attribute@count=\z@
\fi
\def\newattribute#1{%  \e@alloc\attribute\attributedef
\e@alloc@attribute@count\m@ne\e@alloc@top#1%
}
\setattribute \unsetattribute Handy utilities.
\def\setattribute#1#2{#1=\numexpr#2\relax}
\def\unsetattribute#1{#1=-"7FFFFFFF\relax}

\section{Category code tables}
\texttt{\newcatcodetable} Category code tables are allocated with a limit half of that used by Lua\texttt{\LaTeX} 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.
\ifx \e@alloc@ccodetable@count \@undefined
\countdef \e@alloc@ccodetable@count = 259
\e@alloc@ccodetable@count = \z@
\fi
\def \newcatcodetable#1 {%
\e@alloc \catcodetable \chardef \e@alloc@ccodetable@count \m@ne \eighthyphen "8000" #1 %
\initcatcodetable \allocationnumber}

\catcodetable@initex
\catcodetable@string
\catcodetable@latex
\catcodetable@atletter
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.
\newcatcodetable \catcodetable@initex
\newcatcodetable \catcodetable@string
\begingroup
\def \setrangecatcode #1 #2 #3 {%
\ifnum #1 > #2 %
\expandafter \@gobble
\else
\expandafter \@firstofone
\fi
\expandafter \setrangecatcode \expandafter
{\number \numexpr #1 + 1 \relax}{#2}{#3} %
}\@firstofone
\catcodetable \catcodetable@atletter
\begin{document}
\def \parseunicodedataI #1; #2; #3; #4 \relax {}% 
\def \parseunicodedataII #1; #3 First > #4 \relax % 
\def \parseunicodedataIII #1; #2 First > #4 \relax % 
\ifx \relax \relax
\.expandafter \parseunicodedataIII
\else
\expandafter \parseunicodedataIV
\fi
\end{document}
5.5 Named Lua functions

\newluafunction Much the same story for allocating LuaT\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.
Additionally two variants are provided to make the passed control sequence call the function directly.

5.6 Custom whatsits

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

5.7 Lua bytecode registers

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

5.8 Lua chunk registers

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.
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:

\def\now@and@everyjob#1{%
  \everyjob\expandafter{\the\everyjob #1%}
  #1%
}

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{verbatim}
\begingroup
\attributedef\attributezero=0 \\
\chardef \charzero =0 \\
Note name change required on older luatex, for hash table access.
\countdef \CountZero =0 \\
\dimendef \dimenzero =0 \\
\mathchardef \mathcharzero =0 \\
\muskipdef \muskipzero =0 \\
\skipdef \skipzero =0 \\
\toksdef \tokszero =0 \\
\directlua{require("ltluatex")}
\endgroup
\end{verbatim}

Note name change required on older luatex, for hash table access.
In \everyjob, if luaotfload is available, load it and switch to TU.

\begin{verbatim}
\ifx\directlua\@undefined\else
\directlua{
if xpcall(function ()
require('luaotfload-main')
end,texio.write_nl) 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
\expandafter\let\csname ver@luaotfload.sty\endcsname\fmtversion
\fi
\end{verbatim}

\subsection*{5.10 Lua module preliminaries}

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

\begin{verbatim}
luatexbase = luatexbase or { }
local luatexbase = luatexbase

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
local flush_list = node.flush_list

local luatexbase_warning
local luatexbase_error
\end{verbatim}

\begin{verbatim}
luatexbase = luatexbase or { }
local luatexbase = luatexbase

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
local flush_list = node.flush_list

local luatexbase_warning
local luatexbase_error
\end{verbatim}

\subsection*{5.10 Lua module preliminaries}

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

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

\begin{verbatim}
luatexbase = luatexbase or { }
local luatexbase = luatexbase

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
local flush_list = node.flush_list

local luatexbase_warning
local luatexbase_error
\end{verbatim}

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

local modules = modules or { }

provides_module
Local function to write to the log.

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.

local function provides_module(info)
  if not (info and info.name) then
    luatexbase_error("Missing module name for provides_module")
  end
  local function spaced(text)
    return text and (" ". text) or ""
  end
  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, ",", " ")
    end
    first_head = leader .. "Module " .. mod .. " ". msg_type .. ":"
  end

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 \MessageBreak.

First an auxiliary for the formatting: this measures up the message leader so
we always get the correct indent.

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, ",", " ")
  end
  first_head = leader .. "Module " .. mod .. " ". msg_type .. ":"
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(
    text
    .. "on input line 
    .. tex.inputlineno, "\n", "\n" .. cont .. " 
)
.. "\n"
end

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

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

local function module_error(mod, text)
error(msg_format(mod, "Error", text))
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, \TeX{}\LaTeX{} makes that easy.

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.luatexversion > 81 then
  createtoken = token.create
elseif tex.luatexversion > 79 then
  createtoken = newtoken.create
end
local hashtokens = tex.hashtokens()
luatexversion = tex.luatexversion
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

local registernumber
if luatexversion < 80 then
  function registernumber(name)
    local nt = hashtokens[name]
    if (nt and luaregisterbasetable[nt[1]]) then
      return nt[2] - luaregisterbasetable[nt[1]]
    else
      return false
    end
  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

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

5.13 Attribute allocation

As attributes are used for Lua manipulations its useful to be able to assign from this end.
418 luatexbase.attributes = attributes
419 local attribute_count_name =
420 attribute_count_name or "e@alloc@attribute@count"
421 local function new_attribute(name)
422 tex_setcount("global", attribute_count_name,
423 tex_count[attribute_count_name] + 1)
424 if tex_count[attribute_count_name] > 65534 then
425 luatexbase_error("No room for a new \attribute")
426 end
427 attributes[name] = tex_count[attribute_count_name]
428 luatexbase_log("Lua-only attribute " .. name .. " = " ..
429 tex_count[attribute_count_name])
430 return tex_count[attribute_count_name]
431 end
432 luatexbase.new_attribute = new_attribute

5.14 Custom whatsit allocation

new_whatsit Much the same as for attribute allocation in Lua.
433 local whatsit_count_name = whatsit_count_name or "e@alloc@whatsit@count"
434 local function new_whatsit(name)
435 tex_setcount("global", whatsit_count_name,
436 tex_count[whatsit_count_name] + 1)
437 if tex_count[whatsit_count_name] > 65534 then
438 luatexbase_error("No room for a new custom whatsit")
439 end
440 luatexbase_log("Custom whatsit " .. (name or ") .. " = " ..
441 tex_count[whatsit_count_name])
442 return tex_count[whatsit_count_name]
443 end
444 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
445 is used in the log if given.
446 local bytecode_count_name =
447 bytecode_count_name or "e@alloc@bytecode@count"
448 local function new_bytecode(name)
449 tex_setcount("global", bytecode_count_name,
450 tex_count[bytecode_count_name] + 1)
451 if tex_count[bytecode_count_name] > 65534 then
452 luatexbase_error("No room for a new bytecode register")
453 end
454 luatexbase_log("Lua bytecode " .. (name or ") .. " = " ..
455 tex_count[bytecode_count_name])
456 return tex_count[bytecode_count_name]
457 end
458 luatexbase.new_bytecode = new_bytecode

17
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, 
    tex_count[luafunction_count_name] + 1)
    if tex_count[luafunction_count_name] > 65534 then
        luatexbase_error("No room for a new luafunction register")
    end
    luatexbase_log("Lua function " .. (name or "") .. " = 
    tex_count[luafunction_count_name])
    return tex_count[luafunction_count_name]
end
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.

```
local callbacklist = callbacklist or { }
```
Numerical codes for callback types, and name-to-value association (the table keys are strings, the values are numbers).

```lua
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

```lua\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.)

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

Not currently used by luatex but included for completeness. may be used by a font handler.
Section 8.3: data processing callbacks.

```plaintext
find_cidmap_file = data,
read_cidmap_file = exclusive,
```

Section 8.4: node list processing callbacks.

```plaintext
process_input_buffer = data,
process_output_buffer = data,
process_jobname = data,
```

Section 8.5: information reporting callbacks.

```plaintext
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.

```plaintext
finish_pdffile = data,
finish_pdfpage = data,
page_objnum_provider = data,
page_order_index = data,
process_pdf_image_content = data,
```

Section 8.7: font-related callbacks.

```plaintext
define_font = exclusive,
```
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.

```lua
local callback_register = callback_register or callback.register

function callback.register()
    luatexbase_error("Attempt to use callback.register() directly\n")
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.

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

Default for user-defined data callbacks without explicit default.

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

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

```lua
local function list_handler_default(head)
  return head
end
end
```

Handler for reverselist callbacks.

```lua
local function reverselist_handler(name)
  return function(head, ...)
```

22
local ret
callbacks = callbacklist[name]
for i = #callbacks, 1, -1 do
cb = callbacks[i]
ret = cb.func(head, ...)
if ret == false then
  luatexbase_warning("Function " .. cb.description .. " returned false\n" .. "in callback " .. name .. "")
  return false
end
if ret ~= true then
  head = ret
end
end
return head
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

Package code management

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]
    or not name or name == ""
    then
        luatexbase_error("Unable to create callback:
            "valid callback name and type required")
    end
    if callbacktypes[name] then
        luatexbase_error("Unable to create callback '
            "callback is already defined")
    end
    default = default or defaults[ctype_id]
    if not default then
        luatexbase_error("Unable to create callback '
            "default is required for "callbacks")
    elseif type (default) ~= "function" then
        luatexbase_error("Unable to create callback '
            "default 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 '
            "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.

```lua
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
  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.
  end
  if user_callbacks_defaults[name] == nil then
    callback_register(name, handlers[callbacktypes[name]](name))
  end
  actually register the function and give an error if more than one exclusive one
  is registered.
  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 \n        add_to_callback(<callback>, <function>, <description>)"
      )
    end
  end
  table.insert(l, priority, f)
  keep user informed.
  luatexbase_log(
    "Inserting " .. description .. ", at position 
    .. priority .. " in " .. name .. ","
  )
end
luatexbase.add_to_callback = add_to_callback
```

remove_from_callback Remove a function from a callback. First check arguments.
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 .. '!\n"
        )
    end
    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 .. '!\n"
    )
    if #l == 0 then
        callbacklist[name] = nil
        if user_callbacks_defaults[name] == nil then
            callback_register(name, nil)
        end
    end
    return cb.func, cb.description
end

luatexbase.remove_from_callback = remove_from_callback

local function in_callback(name, description)
    if not name or name == "" then
        luatexbase_error("Unable to find function in callback:
                        " .. 
                        "valid callback name required")
    end
    if not callbacklist[name] or
        not description or
        description == "" then
        luatexbase_error(  
            "Unable to find function in callback.\n\n" ..  
            "Correct usage: \n" ..  
            "in_callback(<callback>, <description>)"
        )
    end
    local index = false
    for i,j in ipairs(callbacklist[name]) do
        if j.description == description then
            index = i
            break
        end
    end
    if not index then
        luatexbase_error(  
            "No function ' .. description .. ' registered in callback for ' ..  
            name .. '!\n"
        )
    end
    local cb = callbacklist[name][index]
    table.remove(callbacklist[name], index)
    luatexbase_log(  
        "Removing ' .. description .. ' from ' .. name .. '!\n"
    )
    if #callbacklist[name] == 0 then
        callbacklist[name] = nil
        if user_callbacks_defaults[name] == nil then
            callback_register(name, nil)
        end
    end
    return cb.func, cb.description
end

in_callback Look for a function description in a callback.
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

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

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 \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 \texttt{latexrelease}: as such this is \textit{deliberately} not documented for users!

local function uninstall()
  module_info(uninstall(),
   "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.

```lua
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
    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 == false then
        flush_list(current)
        return nil
    end
    return post
end)

Reset the catcode of @.
```

```tex\catcode'@=\etatcatcode\relax
```