ltluatex.dtx
(LuaTEX-specific support)

David Carlisle and Joseph Wright*
2020/08/10

Contents

1 Overview 2

2 Core T\TeX functionality 2

3 Plain T\TeX interface 3

4 Lua functionality 3
  4.1 Allocators in Lua 3
  4.2 Lua access to T\TeX register numbers 4
  4.3 Module utilities 5
  4.4 Callback management 5

5 Implementation 6
  5.1 Minimum LuaT\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 17
  5.15 Bytecode register allocation 17
  5.16 Lua chunk name allocation 17
  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 \LaTeXe kernel level plus as a loadable file which can be used with plain \TeX and \LaTeXe.

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 Xe\TeX, with code defined in `ltfinal.dtx'). With any \LaTeXe kernel from 2015 onward these registers are part of the block in the extended area reserved by the kernel (prior to 2015 the \LaTeXe \LaTeXe 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 \LaTeXe format, however also extracted to the file `ltluatex.tex' which may be used with older \LaTeX 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 `\texttt{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 Ini\TeX (as described in the Lua\TeX manual).

- `\newluafunction' `\newluafunction{(function)}'
  Defines a named `\luafunction', indexed from 1. (Lua indexes tables from 1 so \luafunction0 is not available).

- `\newwhatsit' `\newwhatsit{(whatsit)}'
  Defines a custom `\whatsit', indexed from 1.

- `\newluabytecode' `\newluabytecode{(bytecode)}'
  Allocates a number for Lua bytecodes register, indexed from 1.

- `\newluachunkname' `\newluachunkname{(chunkname)}'
  Allocates a number for Lua chunk register, indexed from 1. Also enters the name of the register (without backslash) into the `lua.name' table to be used in stack traces.
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.

\begin{verbatim}
\setattribute{⟨attribute⟩}{⟨value⟩}
\unsetattribute{⟨attribute⟩}
\end{verbatim}

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.

\section{Plain \TeX interface}

The \ltlatex interface may be used with plain \TeX using \input{ltlatex}. This inputs \ltlatex.tex which inputs \etext src (or \etext sty if used with \LaTeX) if it is not already input, and then defines some internal commands to allow the \ltlatex 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 \ltlatex code but implements a compatibility layer providing the interface of the original package.

\section{Lua functionality}

\subsection{Allocators in Lua}

\begin{verbatim}
new_attribute luatexbase.new_attribute(⟨attribute⟩)
\end{verbatim}

Returns an allocation number for the ⟨attribute⟩, indexed from 1. The attribute will be initialised with the marker value \texttt{\textasciitilde"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. Ametatable is provided so that the table syntax can be used consistently for attributes declared in \TeX or Lua.

\begin{verbatim}
new_whatsit luatexbase.new_whatsit(⟨whatsit⟩)
\end{verbatim}

Returns an allocation number for the custom ⟨whatsit⟩, indexed from 1.

\begin{verbatim}
new_bytecode luatexbase.new_bytecode(⟨bytecode⟩)
\end{verbatim}

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

\begin{verbatim}
new_chunkname luatexbase.new_chunkname(⟨chunkname⟩)
\end{verbatim}

Returns an allocation number for a Lua chunk name for use with \directlua and \latelua, indexed from 1. The number is returned and also ⟨name⟩ argument is added to the lua.name array at that index.

\begin{verbatim}
new_luafunction luatexbase.new_luafunction(⟨functionname⟩)
\end{verbatim}

Returns an allocation number for a lua function for use with \luafunction, \lateluafunction, and \luadef, indexed from 1. The optional ⟨functionname⟩ 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{\textasciitilde@alloc@attribute@count}, but these can be adjusted by defining the variable ⟨type⟩.count.name before loading \ltlatex.lua, for example
local attribute_count_name = "attributetracker"
require("ltluatex")

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

4.2 Lua access to \TeX register numbers

\registernumber luatexbase.registernumber(⟨name⟩)

Sometimes (notably in the case of Lua attributes) it is necessary to access a register by number that has been allocated by \TeX. This package provides a function to look up the relevant number using Lua\TeX’s internal tables. After for example \newattribute\myattrib, \myattrib would be defined by (say) \myattrib=\attribute15. \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\space\space\space\space\space\space\space\space\space\space\space\space\space\space\space\space\space\space\space\space\space\space
\directlua{tex.write(luatexbase.registernumber("#1") or "bad input")}%
}%
\test{undefinedrubbish}
\test{space}
\test{hbox}
\test{@MM}
\test{@tempdima}
\test{@tempdimb}
\test{strutbox}
\test{sixt@n}
\attributedef\myattr=12
\myattr=200
\test{myattr}

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

undefinedrubbish: \relax
bad input
space: macro:->
bad input
hbox: \hbox

4
Notice how undefined commands, or commands unrelated to registers do not produce an error, just return \texttt{false} and so print \texttt{bad input} here. Note also that commands defined by \texttt{\newbox} work and return the number of the box register even though the actual command holding this number is a \texttt{\chardef} defined token (there is no \texttt{\boxdef}).

### 4.3 Module utilities

- **provides\_module**
  
  \texttt{luatexbase.provides\_module(⟨info⟩)}
  
  This function is used by modules to identify themselves; the \texttt{info} should be a table containing information about the module. The required field \texttt{name} must contain the name of the module. It is recommended to provide a field \texttt{date} in the usual \LaTeX\ format \texttt{yyyy/mm/dd}. Optional fields \texttt{version} (a string) and \texttt{description} may be used if present. This information will be recorded in the log. Other fields are ignored.

- **module\_info**
  
  \texttt{luatexbase.module\_info(⟨module⟩, ⟨text⟩)}

- **module\_warning**
  
  \texttt{luatexbase.module\_warning(⟨module⟩, ⟨text⟩)}

- **module\_error**
  
  \texttt{luatexbase.module\_error(⟨module⟩, ⟨text⟩)}

  These functions are similar to \LaTeX's \texttt{\PackageError}, \texttt{\PackageWarning} and \texttt{\PackageInfo} in the way they format the output. No automatic line breaking is done, you may still use \texttt{\n} as usual for that, and the name of the package will be prepended to each output line.

  Note that \texttt{luatexbase.module\_error} raises an actual Lua error with \texttt{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**
  
  \texttt{luatexbase.add\_to\_callback(⟨callback⟩, ⟨function⟩, ⟨description⟩)}
  
  Registers the \texttt{(function)} into the \texttt{(callback)} with a textual \texttt{(description)} of the function. Functions are inserted into the callback in the order loaded.

- **remove\_from\_callback**
  
  \texttt{luatexbase.remove\_from\_callback(⟨callback⟩, ⟨description⟩)}
  
  Removes the callback function with \texttt{(description)} from the \texttt{(callback)}. The removed function and its description are returned as the results of this function.

- **in\_callback**
  
  \texttt{luatexbase.in\_callback(⟨callback⟩, ⟨description⟩)}
  
  Checks if the \texttt{(description)} matches one of the functions added to the list for the \texttt{(callback)}, returning a boolean value.
disable_callback \texttt{luatexbase.disable_callback(\langle\textit{callback}\rangle)} Sets the \langle\textit{callback}\rangle to \texttt{false} as described in the \textsc{LuaTEX} 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.

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

call_callback \texttt{luatexbase.call_callback(\langle\textit{name}\rangle,...)} Calls a user defined callback with the supplied arguments.

5 Implementation

5.1 Minimum \textsc{LuaTEX} version

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

\begin{verbatim}
\ifnum\luatexversion<60 \wlog{***************************************************}
\wlog{* LuaTeX version too old for ltluatex support *}
\wlog{***************************************************}
\expandafter\endinput
\fi
\end{verbatim}

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

\begin{verbatim}
\long\def\@gobble#1{}
\long\def\@firstofone#1{#1}
\end{verbatim}

5.2 Older \textsc{LTEx}/Plain \TeX{} setup

\begin{verbatim}
\(*\text{tex}\)
\end{verbatim}

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

\begin{verbatim}
\directlua{tex.enableprimitives("",tex.extraprimitives(\"luatex\")\})
\end{verbatim}

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

\begin{verbatim}
\(*\text{tex}\)
\end{verbatim}
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.

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.
5.3 Attributes

\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.

\setattribute \unsetattribute

Handy utilities.

5.4 Category code tables

\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.
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.

\begin{verbatim}
\catcode@initex \catcode@string \catcode@latex \catcode@atletter
\catcodetable@initex\catcodetable@initex
\catcodetable@string\catcodetable@string
\catcodetable@latex\catcodetable@latex
\catcodetable@atletter\catcodetable@atletter
\begin{group}
\def\setrangecatcode#1#2#3{% 
  \ifnum#1>#2 % 
  \expandafter\@gobble 
  \else 
  \expandafter\@firstofone 
  \fi 
  {\catcode#1=#3 % 
  \expandafter\setrangecatcode\expandafter
  \{\number\numexpr#1 + 1\relax\{#2\{#3\}
  %
  }%
  \@firstofone{% 
  \catcodetable@initex
  \catcode0=12 %
  \catcode13=12 %
  \catcode37=12 %
  \setrangecatcode{65}{90}{12}%
  \setrangecatcode{97}{122}{12}%
  \catcode92=12 %
  \catcode127=12 %
  \savecatcodetable\catcodetable@string
  \endgroup
  }%
  \newcatcodetable\catcodetable@latex
  \newcatcodetable\catcodetable@atletter
  \begin{group}
  \def\parseunicodedataI#1;#2;#3;#4\relax{%
  \parseunicodedataII#1;#3;#2 First>\relax
  }%
  \def\parseunicodedataII#1;#2;#3 First>#4\relax{% 
  \ifx\relax#4\relax
  \expandafter\parseunicodedataIII
  \else 
  \expandafter\parseunicodedataIV 
  \fi 
  {#1}#2\relax%
  }%
  \def\parseunicodedataIII#1;#2;#3 First>#4\relax{% 
  \ifx\relax#4\relax
  \expandafter\parseunicodedataIV
  \else 
  \expandafter\parseunicodedataV 
  \fi 
  {#1}{#2}\relax%
  }%
  \def\parseunicodedataIV#1;#2#3\relax{% 
  \ifnum 0% 
  \if L#21\fi 
  \if H#21\fi 
  >0 %
  }
\end{verbatim}
\catcode"#1=11 \fi
\def\parseunicodedataIV#1#2#3\relax{%}
\read\unicoderead to \unicodedataline
\if L#2%
\count0="#1 \expandafter\parseunicodedataV\unicodedataline\relax
\fi
\def\parseunicodedataV#1;#2\relax{%}
\loop
\unless\ifnum\count0>"#1 \catcode\count0=11 \advance\count0 by 1 \repeat
\def\storedpar\par}
\chardef\unicoderead=\numexpr\count16 + 1\relax
\openin\unicoderead=UnicodeData.txt \loop\unless\ifeof\unicoderead\read\unicoderead to \unicodedataline
\unless\ifx\unicodedataline\storedpar
\expandafter\parseunicodedataI\unicodedataline\relax
\fi\repeat
\closein\unicoderead
@firstofone{%
\catcode64=12 
\savecatcodetable\catcodetable@latex
\catcode64=11 
\savecatcodetable\catcodetable@atletter
}
endgroup

5.5 Named Lua functions

\newluafunction

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

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

5.6 Custom whatsits

\newwhatsit

These are only settable from Lua but for consistency are definable here.
\ifx\e@alloc@whatsit@count\@undefined
\edef\e@alloc@whatsit=\e@alloc@chardef
\edef\e@alloc@whatsit@count=\z@
\fi
\def\newwhatsit{%
\e@alloc@luafunction\e@alloc@chardef
\e@alloc@luafunction@count=m@ne\e@alloc@top
}
5.7 Lua bytecode registers
\newluabytecode
These are only settable from Lua but for consistency are definable here.

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.

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

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.
\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
\Begin\Kernel
\\endgroup
\Kernel
\IncludeInRelease{2017/01/01}{\newluafunction}{LuaTeX}%
\IncludeInRelease{0000/00/00}{\Let\e@alloc\attribute\count\@undefined}
\IncludeInRelease{0000/00/00}{\Let\new\attribute\@undefined}
\IncludeInRelease{0000/00/00}{\Let\set\attribute\@undefined}
\IncludeInRelease{0000/00/00}{\Let\unset\attribute\@undefined}
\IncludeInRelease{0000/00/00}{\Let\e@alloc\catcodetable\count\@undefined}
\IncludeInRelease{0000/00/00}{\Let\new\catcodetable\@undefined}
\IncludeInRelease{0000/00/00}{\Let\catcodetable\initex\@undefined}
\IncludeInRelease{0000/00/00}{\Let\catcodetable\string\@undefined}
\IncludeInRelease{0000/00/00}{\Let\catcodetable\latex\@undefined}
\IncludeInRelease{0000/00/00}{\Let\catcodetable\atletter\@undefined}
\IncludeInRelease{0000/00/00}{\Let\e@alloc\luafunction\count\@undefined}
\IncludeInRelease{0000/00/00}{\Let\new\luafunction\@undefined}
\IncludeInRelease{0000/00/00}{\Let\e@alloc\whatsit\count\@undefined}
\IncludeInRelease{0000/00/00}{\Let\new\luabytecode\@undefined}
\IncludeInRelease{0000/00/00}{\Let\new\luachunkname\@undefined}
\IncludeInRelease{0000/00/00}{\Let\e@alloc\luachunk\count\@undefined}
\IncludeInRelease{0000/00/00}{\Directlua{\tex@kernbase.uninstall()}}
\IncludeInRelease{0000/00/00}{\Let\encodingdefault{\f@encoding}}
\Ifx\directlua\@undefined\else
\Kernel\Everyjob\Expandafter{\The\Everyjob}
\Kernel\\directlua{\If\xpcall\{function ()\%
\require\("luaotfload-main")\%
\tex\io.write\_nl\(\)\then\%
\local\_\textit{\void}=\luaotfload.main\(\)\%
\else\%
\tex\io.write\_nl\("Error in luaotfload: reverting to OT1\")\%
\tex.print\("\texttt{\string\def\string\\encodingdefault{\string OT1}}")\%
\end{verbatim}

In \everyjob, if luaotfload is available, load it and switch to TU.
\IncludeInRelease{2017/01/01}{\Let\encodingdefault{\f@encoding}}
\If\fontencoding\{\texttt{\string\encodingdefault{TU \ string \ everyjob}}\}\then\%
\Else\%
\Kernel\Everyjob\Expandafter{\%
\Kernel\The\Everyjob}
\Kernel\{"\Kernel,\\\Kernel,\\\Kernel,\\\Kernel\}
5.10 Lua module preliminaries

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.

luatexbase = luatexbase or {}
local luatexbase = luatexbase

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

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 luatexbase_warning
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.

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

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

modules[info.name] = info
end

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 \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 \texttt{\LaTeXbreak}.

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, ".", " ")
        first_head = leader .. "Module " .. mod .. " " .. msg_type .. "::"
    end
    if msg_type == "Error" then
        first_head = \LaTeXbreak .. first_head
    end
    if string.sub(text,-1) ~= \LaTeXbreak then
        text = text .. "\LaTeXbreak"
    end
    return first_head .. " 
    .. string.gsub(
        text
        .. "on input line 
        .. tex.inputlineno, \LaTeXbreak, \LaTeXbreak .. cont .. " 
    )
    .. \LaTeXbreak"
end

local function module_info(mod, text)
    Write messages.
end

local function module_warning(mod, text)
    Write messages.
end

local function module_error(mod, text)
    Write messages.
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, Lua\TeX{} 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 = nil
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[i] = createtoken(i).mode
    end
end
```

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

```lua
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

5.13 Attribute allocation

`new_attribute` As attributes are used for Lua manipulations its useful to be able to assign from
```
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
```

16
5.14 Custom whatsit allocation

new_whatsit

Much the same as for attribute allocation in Lua.

```lua
424 local whatsit_count_name = whatsit_count_name or "e@alloc@whatsit@count"
425 local function new_whatsit(name)
426   tex_setcount("global", whatsit_count_name,
427     tex_count[whatsit_count_name] + 1)
428   if tex_count[whatsit_count_name] > 65534 then
429     luatexbase_error("No room for a new custom whatsit")
430   end
431   luatexbase_log("Custom whatsit " .. (name or ") .. ",
432     tex_count[whatsit_count_name])
433   return tex_count[whatsit_count_name]
434 end
435 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.

```lua
436 local bytecode_count_name =
437   bytecode_count_name or "e@alloc@bytecode@count"
438 local function new_bytecode(name)
439   tex_setcount("global", bytecode_count_name,
440     tex_count[bytecode_count_name] + 1)
441   if tex_count[bytecode_count_name] > 65534 then
442     luatexbase_error("No room for a new bytecode register")
443   end
444   luatexbase_log("Lua bytecode " .. (name or ") .. ",
445     tex_count[bytecode_count_name])
446   return tex_count[bytecode_count_name]
447 end
448 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.

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

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

487 local callbacktypes = callbacktypes or {
  Section 8.2: file discovery callbacks.
  488  find_read_file = exclusive,  
  489  find_write_file = exclusive,  
  490  find_font_file = data,  
  491  find_output_file = data,  
  492  find_format_file = data,  
  493  find_vf_file = data,  
  494  find_map_file = data,  
  495  find_enc_file = data,  
  496  find_pk_file = data,  
  497  find_data_file = data,  
  498  open_read_file = exclusive,  
  499  find_opentype_file = data,  
  500  find_truetype_file = data,  
  501  find_type1_file = data,  
  502  open_read_file = exclusive,  
  503  find_cidmap_file = data,  
  504  read_font_file = exclusive,  
  505  find_opentype_file = data,  
  506  read_vf_file = exclusive,  
  507  find_cidmap_file = data,  
  508  read_map_file = exclusive,  
  509  find_data_file = data,  
  510  read_enc_file = exclusive,  
  511  find_truetype_file = data,  
  512  read_pk_file = exclusive,  
  513  find_type1_file = data,  
  514  read_data_file = exclusive,  
  515  find_image_file = data,  
  516  read_opentype_file = exclusive,

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

518  find_cidmap_file = data,  
519  read_cidmap_file = exclusive,

Section 8.3: data processing callbacks.

521  process_input_buffer = data,  
522  process_output_buffer = data,  
523  process_jobname = data,  

Section 8.4: node list processing callbacks.

525  contribute_filter = simple,  
526  buildpage_filter = simple,  
527  build_page_insert = exclusive,  
528  pre_linebreak_filter = list,  
529  linebreak_filter = exclusive,  
530  append_to_vlist_filter = exclusive,  
531  post_linebreak_filter = reverse_list,  
532  pre_output_filter = list,  
533  hpack_filter = list,  
534  vpack_filter = list,  
535  hpack_quality = list,  
536  vpack_quality = list,  
537  pre_output_filter = list,  
538  process_rule = exclusive,  
539  hyphenate = 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_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_pdffile = data,
finish_pdfpage = data,
page_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

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.

callback.register = callback.register or callback.register

function callback.register()
  lua_texbase.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
```

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
                luatexbase_warning(
                    "Function " .. i.description .. " returned false\n" .. "in callback " .. name .. ","
                )
                break
            end
            if ret ~= true then
                alltrue = false
                head = ret
            end
        end
        return alltrue and true or head
    end
end

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

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

Handler for reverselist callbacks.

```lua
local function reverselist_handler(name)
    return function(head, ...)
        local ret
        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 .. ","
                )
                break
            end
            if ret ~= true then
                alltrue = false
                head = ret
            end
        end
        return alltrue and true or head
    end
end
```

Handler for simple callbacks.

```lua
local function simple_handler(name)
```
return function(...) 
  for _,i in ipairs(callbacklist[name]) do 
    i.func(...) 
  end 
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 
  or not default then
    luatexbase_error("Unable to create callback:\n" .. 
    "valid callback name and type required")
  end 
  if callbacktypes[name] then
    luatexbase_error("Unable to create callback '.. name .. 
    '\"\ncallback is already defined")
  end 
  default = default or defaults[ctype_id] 
  if not default then
luatexbase_error("Unable to create callback '\".. name ..
"\n\n\":\ndefault is required for '\".. ctype ..
"\n\n\":\ncallbacks")
elseif type (default) ~= "function" then
  luatexbase_error("Unable to create callback '\".. name ..
"\n\n\":\ndefault is not a function")
end
user_callbacks_defaults[name] = default
callbacktypes[name] = ctype_id
end
luatexbase.create_callback = create_callback

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
" .. ":\nunknown 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

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" .. ":\nadd_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

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

```lua
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.

```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'\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
  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

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.

```lua
callback_descriptions (name)
```

```lua
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
```

```lua
luatexbase.callback_descriptions = callback_descriptions
```

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

```lua
local function uninstall()
    module_info("luatexbase", "Uninstalling kernel luatexbase code")
    callback.register = callback_register
    luatexbase = nil
end
```

```lua
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
    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)
```
Reset the catcode of \@.

\texttt{\catcode'\@=\etcatcode\relax}