ltlatex.dtx
(Luatex-specific support)

David Carlisle and Joseph Wright*
2021/04/18

Contents

1 Overview 2

2 Core \TeX\ functionality 2

3 Plain \TeX\ interface 3

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

5 Implementation 6
  5.1 Minimum Lua\TeX\ version 6
  5.2 Older \BibTeX/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 whatstil 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 \latex{} and \luatexbase{} written by Heiko Oberdiek, Élie Roux, Manuel Pégourié-Gonnar and Philipp Gesang.
1 Overview

Lua\TeX{} 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 Lua\TeX{} at the \LaTeX{} \texttt{2e} 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 \texttt{luatexbase}.

The following \count registers are used here for register allocation:

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

(\count 256 is used for \texttt{newmarks} allocation and \count 257 is used for
\texttt{newXeTeXintercharclass} with Xe\TeX{}, with code defined in \texttt{ltfinal.dtx}).
With any \LaTeX{} \texttt{2e} kernel from 2015 onward these registers are part of the block
in the extended area reserved by the kernel (prior to 2015 the \LaTeX{} \texttt{2e} 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{1tluatex.tex} which may be used with
older \LaTeX{} formats, and with plain \TeX{}.

\begin{verbatim}
\newattribute \newattribute{\texttt{(attribute)}}
Defines a named \texttt{attribute}, indexed from 1 (i.e. \texttt{attribute0} is never defined).
Attributes initially have the marker value \texttt{~7FFFFFFF} ("unset") set by the engine.
\newcatcodetable \newcatcodetable{\texttt{(catcodetable)}}
Defines a named \texttt{catcodetable}, indexed from 1 \texttt{(catcodetable0} is never as-
signed). A new catcode table will be populated with exactly those values assigned
by Ini\TeX{} (as described in the Lua\TeX{} manual).
\newluafunction \newluafunction{\texttt{(function)}}
Defines a named \texttt{luafunction}, indexed from 1. (Lua indexes tables from 1 so
\texttt{luafunction0} is not available).
\newwhatshit \newwhatshit{\texttt{(whatshit)}}
Defines a custom \texttt{whatshit}, indexed from 1.
\newluabytecode \newluabytecode{\texttt{(bytecode)}}
Allocates a number for Lua bytecode register, indexed from 1.
\newluachunkname \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{verbatim}
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.

\catcodetable\initex
\catcodetable\string
\catcodetable\latex
\catcodetable\atletter
\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{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`  
\begin{verbatim}
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 = "FFFFFFFF ('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`  
\begin{verbatim}
luatexbase.new_whatsit(⟨whatsit⟩)
\end{verbatim}

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

`new_bytecode`  
\begin{verbatim}
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.

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

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

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

Returns an allocation number for a lua function for use with `\luafunction`, `\latexluafunction`, 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., `e@alloc@attribute@count`, but these can be adjusted by defining the variable ⟨type⟩.count.name before loading `ltluatex.lua`, for example.
local attribute_count_name = "attributetracker"
require("ltluatex")

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

4.2 Lua access to \TeX register numbers

\registernumber\luatexbase.registernumber(\langle name\rangle)

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. \luatexbase.registernumber("myattrib") would then return the register number, 15 in this case. If the string passed as argument does not correspond to a token defined by \attributedef, \countdef or similar commands, the Lua value false is returned.

As an example, consider the input:

\newcommand\test[1]{%\typeout{#1: \expandafter\meaning\csname#1\endcsname^^J\space\space\space\space\directlua{tex.write(luatexbase.registernumber("#1") or "bad input")}%
  }%
  }\test{undefinedrubbish}\test{space}\test{hbox}\test{@MM}\test{@tempdima}\test{@tempdimb}\test{strutbox}\test{sixt@@n}\attrbutedef\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
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**
\[\text{luatexbase.provides_module(⟨info⟩)}\]
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 \LATEX 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**
\[\text{luatexbase.module_info(⟨module⟩, ⟨text⟩)}\]

**module_warning**
\[\text{luatexbase.module_warning(⟨module⟩, ⟨text⟩)}\]

**module_error**
\[\text{luatexbase.module_error(⟨module⟩, ⟨text⟩)}\]

These functions are similar to \LATEX’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**
\[\text{luatexbase.add_to_callback(⟨callback⟩, ⟨function⟩, ⟨description⟩)}\]
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**
\[\text{luatexbase.remove_from_callback(⟨callback⟩, ⟨description⟩)}\]
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**
\[\text{luatexbase.in_callback(⟨callback⟩, ⟨description⟩)}\]
Checks if the (description) matches one of the functions added to the list for the (callback), returning a boolean value.
disable_callback  \texttt{luatexbase.disable_callback(\langle callback\rangle)} Sets the \langle callback\rangle to \texttt{false} as described in the \LaTeX\ manual for the underlying \texttt{callback.register} built-in. Callbacks will only be set to \texttt{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 name\rangle,\texttt{metatype},\langle default\rangle)} Defines a user defined callback. The last argument is a default function or \texttt{false}.

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

5 Implementation

5.1 Minimum \LaTeX\ version

\LaTeX\ 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, \LaTeX\ will correctly find Lua files in the \texttt{texmf} tree without ‘help’.

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.

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

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

\setattribute \unsetattribute Handy utilities.

\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 initialization 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.
\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 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
 \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
 \z@
\fi
\def\newwhatsit{\@firstofone{\
 \catcode64=12 \%
 \savecatcodetable\catcodetable@latex
 \catcode64=11 \%
 \savecatcodetable\catcodetable@atletter
}}
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 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 \%
\countdef \CountZero =0 \%
\dimendef \dimenzero =0 \%
\mathchardef \mathcharzero =0 \%
\muskipdef \muskipzero =0 \%
\skipdef \skipzero =0 \%
\toksdef \tokszero =0 \%
\directlua{require("ltluatex")}
\endgroup
\langle \laatexrelease \rangle \EndIncludeInRelease
\langle \laatexrelease \rangle \IncludeInRelease{0000/00/00} \%
\langle \laatexrelease \rangle \IncludeInRelease{2017/01/01} \%
\langle \laatexrelease \rangle \fontencoding{TU in everyjob} \%
\langle \laatexrelease \rangle \fontencoding{TU} \let\encodingdefault\f@encoding
\langle \laatexrelease \rangle \ifx\directlua\@undefined\else
\langle \ekernelf \rangle \everyjob\expandafter{\%
\langle \ekernelf \rangle \the\everyjob
\langle * \ekernelf , \laatexrelease \rangle \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}')\%
\endlua}
\langle \laatexrelease \rangle \EndIncludeInRelease

In \everyjob, if luaotfload is available, load it and switch to TU.
Some set up for the Lua module which is needed for all of the Lua functionality added here.

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

Some Lua module utilities

5.11.1 Module tracking

Local function to write to the log.

Modelled on \ProvidesPackage, we store much the same information but with a little more structure.
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 \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, ".", " ")
        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(
        text
        .. "on input line "
        .. tex.inputlineno, "\n", "\n" .. cont .. " "
    )
    .. "\n"
end

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, Lua\TeX{} 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()
local 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

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

As attributes are used for Lua manipulations its useful to be able to assign from this end.

```
5.14 Custom whatsit allocation

```latex
local whatsit_count_name = whatsit_count_name or "e@alloc@whatsit@count"
local function new_whatsit(name)
    tex_setcount("global", whatsit_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

```latex
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")
    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

```latex
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

```
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 { }
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 LuaTeX. (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 find_opentype_file = data,
499 find_truetype_file = data,
500 find_type1_file = data,
501 find_image_file = data,
502 open_read_file = exclusive,
503 read_font_file = exclusive,
504 read_vf_file = exclusive,
505 read_map_file = exclusive,
506 read_enc_file = exclusive,
507 read_pk_file = exclusive,
508 read_data_file = exclusive,
509 read_truetype_file = exclusive,
510 read_type1_file = exclusive,
511 read_opentype_file = exclusive,

Not currently used by luatex but included for completeness. may be used by a
font handler.
512 find_cidmap_file = data,
513 read_cidmap_file = exclusive,

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

Section 8.4: node list processing callbacks.
517 contribute_filter = simple,
518 buildpage_filter = simple,
519 build_page_insert = exclusive,
520 pre_linebreak_filter = list,
521 linebreak_filter = exclusive,
522 append_to_vlist_filter = exclusive,
523 post_linebreak_filter = reverse_list,
524 hpack_filter = list,
525 vpack_filter = list,
526 hpack_quality = list,
527 vpack_quality = list,
528 pre_output_filter = list,
529 process_rule = exclusive,
530 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 = exclusive,

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_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,
input_level_string = exclusive,
}
} luatexbase.callbacktypes=callbacktypes

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

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
```

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 .. "',"
      )
      return false
    end
    if ret ~= true then
      alltrue = false
      head = ret
    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 .. "',"
      )
      return false
    end
    if ret ~= true then
      alltrue = false
      head = ret
    end
    return alltrue and true or head
  end
end
```

Handler for simple callbacks.

```lua
local function simple_handler(name)
```

22
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
    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 ..
    '\n\ncallback 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.

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:
  .. \‘\‘\‘Correct usage:\n\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

24
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
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
        luaerror(base_error("Cannot add second callback to exclusive function\n" ..
            name .. ":")
    end
end
local l = callbacklist[name]
if not l then
    luaerror(base_error("No callback list for " .. name .. ":")
end
```

Remove a function from a callback. First check arguments.

```lua
local function remove_from_callback(name, description)
    if not name or name == "" then
        luaerror("Unable to remove function from callback:\n" ..
            "valid callback name required")
    end
    if not callbacktypes[name] or
        not description or
        description == "" then
        luaerror("Unable to remove function from callback:\n" ..
            "Correct usage:\n" ..
            "remove_from_callback(<callback>, <description>)")
    end
    local l = callbacklist[name]
    if not l then
        luaerror("No callback list for " .. name .. ":")
    end
    table.remove(l, priority, f)
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, description, name)
end
local cb = l[index]
table.remove(l, index)
luatexbase_log("Removing '{}' description '{}' from '{}'.",
               description, name, name)
if #l == 0 then
    callbacklist[name] = nil
    if user_callbacks_defaults[name] == nil then
        callback_register(name, nil)
    end
end
else
    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 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
luatexbase.disable_callback = disable_callback

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

```lua
local function callback_descriptions (name)
    local d = {}
    if not name then
        return d
    else
        for k, i in pairs(callbacklist[name]) do
            d[k] = i.description
        end
        return d
    end
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 \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
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 \@.
\catcode'\@=\etcatcode\relax