ltluatex.dtx
(LuaTeX-specific support)

David Carlisle and Joseph Wright*

2016/03/13

Contents

*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 2_ε kernel level plus as a loadable file which can be used with plain TeX and LaTeX.

This file contains code for both TeX (to be stored as part of the format) and Lua (to be loaded at the start of each job). In the Lua code, the kernel uses the namespace `luatexbase`.

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

`\e@alloc@attribute@count` Attributes (default 258)

`\e@alloc@ccodetable@count` Category code tables (default 259)

`\e@alloc@luafunction@count` Lua functions (default 260)

`\e@alloc@whatsit@count` User whatsits (default 261)

`\e@alloc@bytecode@count` Lua bytecodes (default 262)

`\e@alloc@luachunk@count` Lua chunks (default 263)

(`\count 256` is used for `\newmarks` allocation and `\count 257` is used for `\newXeTeXintercharclass` with XeTeX, with code defined in `ltfinal.dtx`). With any LaTeX 2_ε kernel from 2015 onward these registers are part of the block in the extended area reserved by the kernel (prior to 2015 the LaTeX 2_ε 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.

`\newattribute` `\newattribute{`⟨*attribute*⟩`}`
Defines a named `\attribute`, indexed from 1 (*i.e.* `\attribute0` is never defined). Attributes initially have the marker value `-"7FFFFFFF` ('unset') set by the engine.

`\newcatcodetable` `\newcatcodetable{`⟨*catcodetable*⟩`}`
Defines a named `\catcodetable`, indexed from 1 (`\catcodetable0` is never assigned). A new catcode table will be populated with exactly those values assigned by IniTeX (as described in the LuaTeX manual).

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

`\newwhatsit` `\newwhatsit{`⟨*whatsit*⟩`}`
Defines a custom `\whatsit`, indexed from 1.

`\newluabytecode` `\newluabytecode{`⟨*bytecode*⟩`}`
Allocates a number for Lua bytecode register, indexed from 1.

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

\catcodetable@initex
\catcodetable@string
\catcodetable@latex
\catcodetable@atletter
\setattribute
\unsetattribute

Predefined category code tables with the obvious assignments. Note that the `latex` and `atletter` tables set the full Unicode range to the codes predefined by the kernel.

`\setattribute{`⟨*attribute*⟩`}{`⟨*value*⟩`}`
`\unsetattribute{`⟨*attribute*⟩`}`

Set and unset attributes in a manner analogous to `\setlength`. Note that attributes take a marker value when unset so this operation is distinct from setting the value to zero.

3 Plain TEX interface

The ltluatex interface may be used with plain TEX using `\input{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

`luatexbase.new_attribute(`⟨*attribute*⟩`)`

Returns an allocation number for the ⟨*attribute*⟩, 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

`luatexbase.new_whatsit(`⟨*whatsit*⟩`)`

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

new_bytecode

`luatexbase.new_bytecode(`⟨*bytecode*⟩`)`

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

new_chunkname

`luatexbase.new_chunkname(`⟨*chunkname*⟩`)`

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.

4.2 Lua access to TEX register numbers

registernumber

`luatexbase.registernumer(`⟨*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 LuaTEX's internal tables. After for example `\newattribute\myattrib`, `\myattrib` would be defined by (say) `\myattrib=\attribute15`. `luatexbase.registernumer("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 LuaLaTeX then the following would be produced in the log and terminal output.

```
undefinedrubbish: \relax
     bad input
space: macro:->
     bad input
hbox: \hbox
     bad input
@MM: \mathchar"4E20
     20000
@tempdima: \dimen14
     14
@tempdimb: \dimen15
     15
strutbox: \char"B
     11
sixt@@n: \char"10
     16
myattr: \attribute12
     12
```

Notice how undefined commands, or commands unrelated to registers do not produce an error, just return `false` and so print `bad input` here. Note also that

commands defined by `\newbox` work and return the number of the box register even though the actual command holding this number is a `\chardef` defined token (there is no `\boxdef`).

4.3 Module utilities

`provides_module` `luatexbase.provides_module(`⟨*info*⟩`)`
This function is used by modules to identify themselves; the `info` should be a table containing information about the module. The required field `name` must contain the name of the module. It is recommended to provide a field `date` in the usual 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` `luatexbase.module_info(`⟨*module*⟩, ⟨*text*⟩`)`

`module_warning` `luatexbase.module_warning(`⟨*module*⟩, ⟨*text*⟩`)`

`module_error` `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` `luatexbase.add_to_callback(`⟨*callback*⟩, ⟨*function*⟩, ⟨*description*⟩`)` Registers the ⟨*function*⟩ into the ⟨*callback*⟩ with a textual ⟨*description*⟩ of the function. Functions are inserted into the callback in the order loaded.

`remove_from_callback` `luatexbase.remove_from_callback(`⟨*callback*⟩, ⟨*description*⟩`)` Removes the callback function with ⟨*description*⟩ from the ⟨*callback*⟩. The removed function and its description are returned as the results of this function.

`in_callback` `luatexbase.in_callback(`⟨*callback*⟩, ⟨*description*⟩`)` Checks if the ⟨*description*⟩ matches one of the functions added to the list for the ⟨*callback*⟩, returning a boolean value.

`disable_callback` `luatexbase.disable_callback(`⟨*callback*⟩`)` Sets the ⟨*callback*⟩ to `false` as described in the LuaTeX manual for the underlying `callback.register` built-in. Callbacks will only be set to false (and thus be skipped entirely) if there are no functions registered using the callback.

`callback_descriptions` A list of the descriptions of functions registered to the specified callback is returned. `{}` is returned if there are no functions registered.

`create_callback` `luatexbase.create_callback(`⟨*name*⟩,metatype,⟨*default*⟩`)` Defines a user defined callback. The last argument is a default function or `false`.

`call_callback` `luatexbase.call_callback(`⟨*name*⟩,...`)` Calls a user defined callback with the supplied arguments.

5 Implementation

1 ⟨*2ekernel | tex | latexrelease⟩

```
⟨2ekernel | latexrelease⟩\ifx\directlua\@undefined\else
```

5.1 Minimum LuaTEX version

LuaTEX has changed a lot over time. In the kernel support for ancient versions is not provided: trying to build a format with a very old binary therefore gives some information in the log and loading stops. The cut-off selected here relates to the tree-searching behaviour of `require()`: from version 0.60, LuaTEX will correctly find Lua files in the `texmf` tree without 'help'.

```
⟨latexrelease⟩\IncludeInRelease{2015/10/01}
⟨latexrelease⟩                  {\newluafunction}{LuaTeX}%
\ifnum\luatexversion<60 %
  \wlog{************************************************}
  \wlog{* LuaTeX version too old for ltluatex support *}
  \wlog{************************************************}
  \expandafter\endinput
\fi
```

5.2 Older LATEX/Plain TEX setup

```
⟨*tex⟩
```

Older LATEX formats don't have the primitives with 'native' names: sort that out. If they already exist this will still be safe.

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

In pre-2014 LATEX, or plain TEX, load `etex.{sty,src}`.

```
  \ifx\documentclass\@undefined
    \ifx\loccount\@undefined
      \input{etex.src}%
    \fi
    \catcode`\@=11 %
    \outer\expandafter\def\csname newfam\endcsname
                          {\alloc@8\fam\chardef\et@xmaxfam}
  \else
    \RequirePackage{etex}
    \expandafter\def\csname newfam\endcsname
                    {\alloc@8\fam\chardef\et@xmaxfam}
    \expandafter\let\expandafter\new@mathgroup\csname newfam\endcsname
  \fi
```

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

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

```
% 2015-07-13 higher range in luatex
\edef \et@xmaxregs {\ifx\directlua\@undefined 32768\else 65536\fi}
% luatex/xetex also allow more math fam
\edef \et@xmaxfam {\ifx\Umathchar\@undefined\sixt@@n\else\@cclvi\fi}
\count 270=\et@xmaxregs % locally allocates \count registers
\count 271=\et@xmaxregs % ditto for \dimen registers
\count 272=\et@xmaxregs % ditto for \skip registers
\count 273=\et@xmaxregs % ditto for \muskip registers
```

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

and 256 or 16 fam. (Done above due to plain/LaTeX differences in ltluatex.)

```
% \outer\def\newfam{\alloc@8\fam\chardef\et@xmaxfam}
```

End of proposed changes to `etex.src`

5.2.2 luatex specific settings

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

```
\expandafter\let\csname newcount\expandafter\expandafter\endcsname
                \csname globcount\endcsname
\expandafter\let\csname newdimen\expandafter\expandafter\endcsname
                \csname globdimen\endcsname
\expandafter\let\csname newskip\expandafter\expandafter\endcsname
                \csname globskip\endcsname
\expandafter\let\csname newbox\expandafter\expandafter\endcsname
                \csname globbox\endcsname
```

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.

```
\chardef\e@alloc@top=65535
\let\e@alloc@chardef\chardef

\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
      #1\@cclvi
      \ifx\count#4\advance#1 10 \fi
    \fi
    \ifnum#1<#3\relax
    \else
      \errmessage{No room for a new \string#4}%
    \fi
  \fi}%
```

Two simple LaTeX macros used in `ltlatex.sty`.

```
\long\def\@gobble#1{}
\long\def\@firstofone#1{#1}

% 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 TEX / old LATEX.

```
\fi
⟨/tex⟩
```

5.3 Attributes

\newattribute

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

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

\setattribute
\unsetattribute

Handy utilities.

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

5.4 Category code tables

\newcatcodetable

Category code tables are allocated with a limit half of that used by LuaTEX 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.

```
\ifx\e@alloc@ccodetable@count\@undefined
  \countdef\e@alloc@ccodetable@count=259
\fi
\def\newcatcodetable#1{%
  \e@alloc\catcodetable\chardef
    \e@alloc@ccodetable@count\m@ne{"8000}#1%
  \initcatcodetable\allocationnumber
}
\e@alloc@ccodetable@count=\z@
```

\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
      {%
        \catcode#1=#3 %
        \expandafter\setrangecatcode\expandafter
          {\number\numexpr#1 + 1\relax}{#2}{#3}
      }%
  }
  \@firstofone{%
    \catcodetable\catcodetable@initex
      \catcode0=12 %
      \catcode13=12 %
      \catcode37=12 %
      \setrangecatcode{65}{90}{12}%
      \setrangecatcode{97}{122}{12}%
      \catcode92=12 %
      \catcode127=12 %
      \savecatcodetable\catcodetable@string
    \endgroup
  }%
\newcatcodetable\catcodetable@latex
\newcatcodetable\catcodetable@atletter
\begingroup
  \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\relax{%
    \ifnum 0%
      \if L#21\fi
      \if M#21\fi
      >0 %
      \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
\fi
\def\newluafunction{%
  \e@alloc\luafunction\e@alloc@chardef
    \e@alloc@luafunction@count\m@ne\e@alloc@top
}
\e@alloc@luafunction@count=\z@
```

5.6 Custom whatsits

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

```
\ifx\e@alloc@whatsit@count\@undefined
  \countdef\e@alloc@whatsit@count=261
\fi
\def\newwhatsit#1{%
  \e@alloc\whatsit\e@alloc@chardef
    \e@alloc@whatsit@count\m@ne\e@alloc@top#1%
}
\e@alloc@whatsit@count=\z@
```

5.7 Lua bytecode registers

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

```
\ifx\e@alloc@bytecode@count\@undefined
  \countdef\e@alloc@bytecode@count=262
\fi
\def\newluabytecode#1{%
```

```
  \e@alloc\luabytecode\e@alloc@chardef
    \e@alloc@bytecode@count\m@ne\e@alloc@top#1%
}
\e@alloc@bytecode@count=\z@
```

5.8 Lua chunk registers

\newluachunkname

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

```
\ifx\e@alloc@luachunk@count\@undefined
  \countdef\e@alloc@luachunk@count=263
\fi
\def\newluachunkname#1{%
  \e@alloc\luachunk\e@alloc@chardef
    \e@alloc@luachunk@count\m@ne\e@alloc@top#1%
    {\escapechar\m@ne
    \directlua{lua.name[\the\allocationnumber]="\string#1"}}%
}
\e@alloc@luachunk@count=\z@
```

5.9 Lua loader

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

```
⟨2ekernel⟩\everyjob\expandafter{%
⟨2ekernel⟩   \the\everyjob
  \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
⟨2ekernel⟩}
⟨latexrelease⟩\EndIncludeInRelease
```

```
% \changes{v1.0b}{2015/10/02}{Fix backing out of \TeX{} code}
% \changes{v1.0c}{2015/10/02}{Allow backing out of Lua code}
⟨latexrelease⟩\IncludeInRelease{0000/00/00}
⟨latexrelease⟩                  {\newluafunction}{LuaTeX}%
⟨latexrelease⟩\let\e@alloc@attribute@count\@undefined
⟨latexrelease⟩\let\newattribute\@undefined
⟨latexrelease⟩\let\setattribute\@undefined
⟨latexrelease⟩\let\unsetattribute\@undefined
⟨latexrelease⟩\let\e@alloc@ccodetable@count\@undefined
⟨latexrelease⟩\let\newcatcodetable\@undefined
```

```
⟨latexrelease⟩\let\catcodetable@initex\@undefined
⟨latexrelease⟩\let\catcodetable@string\@undefined
⟨latexrelease⟩\let\catcodetable@latex\@undefined
⟨latexrelease⟩\let\catcodetable@atletter\@undefined
⟨latexrelease⟩\let\e@alloc@luafunction@count\@undefined
⟨latexrelease⟩\let\newluafunction\@undefined
⟨latexrelease⟩\let\e@alloc@luafunction@count\@undefined
⟨latexrelease⟩\let\newwhatsit\@undefined
⟨latexrelease⟩\let\e@alloc@whatsit@count\@undefined
⟨latexrelease⟩\let\newluabytecode\@undefined
⟨latexrelease⟩\let\e@alloc@bytecode@count\@undefined
⟨latexrelease⟩\let\newluachunkname\@undefined
⟨latexrelease⟩\let\e@alloc@luachunk@count\@undefined
⟨latexrelease⟩\directlua{luatexbase.uninstall()}
⟨latexrelease⟩\EndIncludeInRelease

⟨2ekernel | latexrelease⟩\fi
⟨/2ekernel | tex | latexrelease⟩
```

5.10 Lua module preliminaries

```
⟨*lua⟩
```

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

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

```
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 TEX. For errors we have to make some changes. Here we give the text of the error in the LATEX format then force an error from Lua to halt the run. Splitting the message text is done using \n which takes the place of \MessageBreak.

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

```
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
```

module_info
module_warning
module_error

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, LuaTEX 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
```

registernumber

Working out the correct return value can be done in two ways. For older LuaTEX releases it has to be extracted from the `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.

```
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 this end.

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

```
local function new_attribute(name)
  tex_setcount("global", "e@alloc@attribute@count",
                          tex_count["e@alloc@attribute@count"] + 1)
  if tex_count["e@alloc@attribute@count"] > 65534 then
    luatexbase_error("No room for a new \\attribute")
  end
  attributes[name]= tex_count["e@alloc@attribute@count"]
  luatexbase_log("Lua-only attribute " .. name .. " = " ..
                 tex_count["e@alloc@attribute@count"])
  return tex_count["e@alloc@attribute@count"]
end
```

```
luatexbase.new_attribute = new_attribute
```

5.14 Custom whatsit allocation

`new\_whatsit` Much the same as for attribute allocation in Lua.

```
local function new_whatsit(name)
  tex_setcount("global", "e@alloc@whatsit@count",
                         tex_count["e@alloc@whatsit@count"] + 1)
  if tex_count["e@alloc@whatsit@count"] > 65534 then
    luatexbase_error("No room for a new custom whatsit")
  end
  luatexbase_log("Custom whatsit " .. (name or "") .. " = " ..
                 tex_count["e@alloc@whatsit@count"])
  return tex_count["e@alloc@whatsit@count"]
end
luatexbase.new_whatsit = new_whatsit
```

5.15 Bytecode register allocation

`new\_bytecode` Much the same as for attribute allocation in Lua. The optional ⟨*name*⟩ argument is used in the log if given.

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

5.16 Lua chunk name allocation

`new\_chunkname` As for bytecode registers but also store the name in the `lua.name` table.

```
local function new_chunkname(name)
  tex_setcount("global", "e@alloc@luachunk@count",
                         tex_count["e@alloc@luachunk@count"] + 1)
  local chunkname_count = tex_count["e@alloc@luachunk@count"]
  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 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.

5.17.1 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 = 1, 2, 3, 4
local types = {
  list      = list,
  data      = data,
  exclusive = exclusive,
  simple    = simple,
}
```

Now, list all predefined callbacks with their current type, based on the LuaTEX manual version 0.80. 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.)

```
local callbacktypes = callbacktypes or {
```

Section 4.1.1: 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_sfd_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,
```

Section 4.1.2: file reading callbacks.

```
  open_read_file     = exclusive,
  read_font_file     = exclusive,
  read_vf_file       = exclusive,
  read_map_file      = exclusive,
  read_enc_file      = exclusive,
  read_sfd_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.

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

Section 4.1.3: data processing callbacks.

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

Section 4.1.4: node list processing callbacks.

```
  contribute_filter      = simple,
  buildpage_filter       = simple,
  pre_linebreak_filter   = list,
  linebreak_filter       = list,
  append_to_vlist_filter = list,
  post_linebreak_filter  = list,
  hpack_filter           = list,
  vpack_filter           = list,
  hpack_quality          = list,
  vpack_quality          = list,
  pre_output_filter      = list,
  process_rule           = list,
  hyphenate              = simple,
  ligaturing             = simple,
  kerning                = simple,
  insert_local_par       = simple,
  mlist_to_hlist         = list,
```

Section 4.1.5: 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,
```

Section 4.1.6: PDF-related callbacks.

```
  finish_pdffile = data,
  finish_pdfpage = data,
```

Section 4.1.7: font-related callbacks.

```
  define_font = 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
```

5.17.2 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.

Handler for `data` callbacks.

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

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

Handler for `list` callbacks.

```
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 .."’"
         )
```

```
532         break
533       end
534       if ret ~= true then
535         alltrue = false
536         head = ret
537       end
538     end
539     return alltrue and true or head
540   end
541 end
```

Handler for `simple` callbacks.

```
542 local function simple_handler(name)
543   return function(...)
544     for _,i in ipairs(callbacklist[name]) do
545       i.func(...)
546     end
547   end
548 end
```

Keep a handlers table for indexed access.

```
549 local handlers = {
550   [data]      = data_handler,
551   [exclusive] = exclusive_handler,
552   [list]      = list_handler,
553   [simple]    = simple_handler,
554 }
```

5.17.3 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.

```
555 local user_callbacks_defaults = { }
```

`create\_callback` The allocator itself.

```
556 local function create_callback(name, ctype, default)
557   if not name  or name  == ""
558   or not ctype or ctype == ""
559   then
560     luatexbase_error("Unable to create callback:\n" ..
561                      "valid callback name and type required")
562   end
563   if callbacktypes[name] then
564     luatexbase_error("Unable to create callback `" .. name ..
565                      "':\ncallback type disallowed as name")
566   end
567   if default ~= false and type (default) ~= "function" then
568     luatexbase_error("Unable to create callback `" .. name ..
569                      ":\ndefault is not a function")
570    end
571   user_callbacks_defaults[name] = default
572   callbacktypes[name] = types[ctype]
573 end
574 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:\n" ..
                     "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]
    if l == false then
   return nil
 end
  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:\n" ..
                     "valid callback name required")
  end
  if not callbacktypes[name] or
    type(func) ~= "function" or
    not description or
    description == "" then
    luatexbase_error(
      "Unable to register callback.\n\n"
        .. "Correct usage:\n"
        .. "add_to_callback(<callback>, <function>, <description>)"
    )
  end
```

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

```
  local l = callbacklist[name]
  if l == nil then
    l = { }
    callbacklist[name] = l
```

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

```
    if user_callbacks_defaults[name] == nil then
      callback_register(name, handlers[callbacktypes[name]](name))
    end
  end
```

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

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

Loop over the callback's function list until we find a matching entry. Remove it and check if the list is empty: if so, unregister the callback handler.

```
  local index = false
  for i,j in ipairs(l) do
    if j.description == description then
      index = i
      break
    end
  end
```

```
  if not index then
    luatexbase_error(
      "No callback `" .. description .. "' registered for `" ..
      name .. "'\n")
  end
  local cb = l[index]
  table.remove(l, index)
  luatexbase_log(
    "Removing  `" .. description .. "' from `" .. name .. "'."
  )
  if #l == 0 then
    callbacklist[name] = nil
    callback_register(name, nil)
  end
  return cb.func,cb.description
end
luatexbase.remove_from_callback = remove_from_callback
```

`in\_callback` Look for a function description in a callback.

```
local function in_callback(name, description)
  if not name
    or name == ""
    or not callbacklist[name]
    or not callbacktypes[name]
    or not description then
      return false
  end
  for _, i in pairs(callbacklist[name]) do
    if i.description == description then
      return true
    end
  end
  return false
end
luatexbase.in_callback = in_callback
```

`disable\_callback` As we subvert the engine interface we need to provide a way to access this 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 latexrelease: as such this is *deliberately* not documented for users!

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

⟨/lua⟩

Reset the catcode of `@`.

⟨tex⟩`\catcode‘\@=\etatcatcode\relax`