LuaTEX

Taco Hoekwater
http://luatex.org

Abstract

LuaTEX is an extended version of TEX that uses Lua as an embedded scripting language. The main objective of the LuaTEX project is to provide an open and configurable variant of TEX while at the same time offering downward compatibility. This paper gives a broad overview of the project.

1 Introduction

The LuaTEX source code of course includes Lua and the latest versions of pdfTEX and Aleph, but it also contains a few other more or less distinguishable parts:

- Some specialized TEX extensions
- A set of Lua module libraries
- The font reading code from FontForge
- Some of the font writing code from xdvipdfmx
- C source code to glue all of this together

If `\pdfoutput` is not set, LuaTEX is a lot like Aleph with additional support for the microtypography pdfTEX is known for. And if `\pdfoutput` is set, then it is like pdfTEX with the much better directionality support provided by Aleph.

If needed, Lua code is used to apply input reencoding, instead of I/O translation OTPs (Aleph) or tcx files (pdfTEX). Also, some experimental features of both programs were removed since the original problems can be better dealt with using Lua.

2 Time-line

The first informal start of LuaTEX was around TUG 2005. After an initial period of playing around and experimenting, the project gained momentum in the spring of 2006, when funding from Colorado State University and TUG (via the Oriental TEX Project) was acquired.

Soon after that, a public repository was set up and a mailing list and website were started. After a year of continuous work, the first beta was released at the TUG 2007 conference in San Diego, USA. The offered functionality is not completely finalized yet, so the interfaces are likely to change a bit still. A stable release is planned for the next TUG conference, in Cork, Ireland, 2008.

3 Features

The new functionality of LuaTEX falls into a few broad categories that are explained briefly in the next paragraphs.

3.1 Unicode support

LuaTEX uses UTF-8 encoded Unicode throughout the system. That means input and output files are Unicode, and also that the hyphenation patterns are expected to express hyphenation points of Unicode characters (instead of the traditional font glyphs).

Commands like `\char` and `\catcode` are extended to accept the full Unicode range, and used fonts can (but do not have to) be Unicode encoded.

3.2 TEX extensions

In the process of extending TEX for Unicode support and the cleanup required for interfacing to Lua code, some other extensions were also added. Here we briefly describe the most interesting of these.

The startup processing is altered to allow the document (via a Lua script) to have access to the command line.

A new feature called '`\catcode` tables' allows switching of all category codes in a single statement.

A new set of registers called attribute is added. Attributes can be used as extra counter values, but their usefulness comes mostly from the fact that all the 'set' attributes are automatically attached to all typesetting nodes created within their active scope. These node attributes can then be queried from any Lua code that deals with node processing.

The single internal memory heap that traditional TEX uses for tokens and nodes is split into two separate arrays, and each of these will grow dynamically when needed. The same is true for the input line buffer and the string pool size. All font memory is allocated on a per-font basis. Some less important arrays are still statically allocated, but eventually all memory allocation will become dynamic.

There is no separate pool file any more; all strings from that file are embedded during the final phase of the compilation of the `luatex` executable.

The format files are passed through zlib, allowing them to shrink to roughly half of the size they would have had in uncompressed form.

3.3 Extended font subsystem

The font system of LuaTEX is totally configurable through optional Lua code. If you do nothing, LuaTEX handles both TEX (TFM) and Omega (OFM) fonts as well as the related virtual font formats.

With some user-supplied Lua code, LuaTEX can also happily use OpenType and TrueType fonts. In addition, it is possible to build up encodings and virtual fonts totally in-memory.

The handling of 'virtualness' takes place at a different level in LuaTEX, meaning every single character can be either virtual or real, instead of this being handled at the font level. Inside a virtual character, it is possible to use arbitrary typeset data as 'character contents'.

3.4 Lua execution

Execution of Lua code within a document is handled by two new primitives: the expandable `\directlua` command, and the non-expandable `\latelua` command. The latter creates a node with Lua code that will be executed inside the `\output` routine, just like the traditional `\write`.

There can be more than one Lua interpreter state active at the same time. Some modules that are normally external to Lua are statically linked in with LuaTEX, because they offer useful functionality. This is one of the areas where future change is likely, but at the moment the list comprises:

- slnunicode, from the Selene libraries, `luaforge.net/projects/sln` (v1.1)
- luazip, from the kepler project, `www.keplerproject.org/luazip/` (v1.2.1)
- luafilesystem, also from the kepler project, `www.keplerproject.org/luafilesystem/` (v1.2)
- lpeg, by Roberto Ierusalimschy, `www.inf.puc-rio.br/~roberto/lpeg.html` (v0.6)
- lzlib, by Tiago Dionizio, `mega.ist.utl.pt/~tngd/lua/` (v0.2)
- md5, by Roberto Ierusalimschy, `www.inf.puc-rio.br/~roberto/md5/md5-5/md5.html`
- fontforge, a partial binding to the FontForge font editor by George Williams, `fontforge.sf.net`

It is also possible to make LuaTEX behave like the standalone Lua interpreter or the Lua bytecode compiler.

3.5 Lua interface libraries

LuaTEX would not be very useful if the Lua code did not have a way to communicate with the TEX internals. For this purpose, a set of Lua modules is defined:

- tex (general TEX access)
- pdf (routines related to pdf output)
- lua (lua bytecode registers)
- texio (writing to the log and terminal)
- font (accessing font internals)
- status (LuaTEX status information)
- kpse (file searching)
- callback (setting up callback hooks)
- token (handling TEX tokens)
- node (handling typeset nodes)

3.6 Callbacks

A callback is a hook into the internal processing of LuaTEX. Using callbacks, you can make LuaTEX run a Lua function you have defined instead of (or on top of) a bit of the core functionality.

It is easiest to think of it this way: callbacks offer a way to define something equivalent to compiled executable code. They have no connection to the TEX input language at all. Because of this they are very different from the argument to `\directlua`.

There are a few dozen callback hooks already defined, with many more to come later. There are callbacks for a wide variety of tasks, for instance: finding files, reading and preprocessing textual input, defining fonts, token creation, node list handling, and information display.

Here is a short example of defining a callback:

```
\directlua0{
function read_tfm (name)
 archive = zip.open('texmf-fonts.zip')
 if archive then
   tfmfile = archive:open(name .. '.tfm')
   if tfmfile then
     data = tfmfile:read('*all')
     return true, data, \string#data
   end
 end
 return false, nil, 0
end
callback.register('read_font_file',read_tfm)
}
```

4 Contact

The LuaTEX project is currently run by:

- Hans Hagen (general overview and website)
- Hartmut Henkel (pdf backend)
- Taco Hoekwater (coding and manual)

With help from:

- Arthur Reutenauer (binaries and testing)
- Martin Schröder (release support)