ConTEXt MkIV: Going UTF

Hans Hagen
http://pragma-ade.com
http://luatex.org

1 Introduction

In this document I will keep track of the transition of ConTEXt from MkII to MkIV, the latter being the Lua aware version.

The development of LuaTEX started with a few email exchanges between me and Hartmut Henkel. I had played a bit with Lua in Scite and somehow felt that it would fit into TEX quite well. Hartmut made me a version of pdfTEX which provided a `\lua` command. After exploring this road a bit Taco Hoekwater took over and we quickly reached a point where the pdfTEX development team could agree on following this road to the future.

The development was boosted by a substantial grant from Colorado State University in the context of the Oriental TEX Project of Idris Samawi Hamid. This project aims at bringing features into TEX that will permit ConTEXt to do high quality Arabic typesetting. Due to this grant Taco could spent substantial time on development, which in turn meant that I could start playing with more advanced features.

The full MkIV document is not so much a users manual as a history of the development. Consider it a collection of articles, and some chapters — like this one — have indeed ended up in the journals of user groups. Things may evolve and the way things are done may change, but it felt right to keep track of the process this way. Keep in mind that some features may have changed while LuaTEX matured.

Just for the record: development in the LuaTEX project is done by Taco Hoekwater, Hartmut Henkel and Hans Hagen. Eventually, the stable versions will become pdfTEX version 2 and other members of the pdfTEX team will be involved in development and maintenance. In order to prevent problems due to new and maybe even slightly incompatible features, pdfTEX version 1 will be kept around as well, but no fundamentally new features will be added to it. For practical reasons we use LuaTEX as the name of the development version but also for pdfTEX 2. That way we can use both engines side by side.

Editor's note: This is the introduction and one chapter of the full *ConTEXt MkIV* document. See the 'Random comments' column on p. 377 for more publication information.

This document is also one of our test cases. Here we use traditional TEX fonts (for math), Type 1 and OpenType fonts. We use color and include test code. Taco and I always test new versions of LuaTEX (the program) and MkIV (the macros and Lua code) with this document before a new version is released. Keep tuned . . .

2 Going UTF

LuaTEX only understands input codes in the Universal Character Set Transformation Format, aka UCS Transformation Format, better known as: UTF. There is a good reason for this universal view on characters: whatever support gets hard coded into the programs, it's never enough, as 25 years of TEX history have clearly demonstrated. Macro packages often support more or less standard input encodings, as well as local standards, user adapted ones, etc.

There is enough information on the Internet and in books about what exactly is UTF. If you don't know the details yet: UTF is a multi-byte encoding. The characters with a bytecode up to 127 map onto their normal ASCII representation. A larger number indicates that the following bytes are part of the character code. Up to 4 bytes make an UTF-8 code, while UTF-16 always uses two pairs of bytes.

byte1	byte2	byte3	byte4	Unicode
192–223	128–191			0x80–0x7FF
224–239	128–191	128–191		0x800–0xFFFF
240–247	128–191	128–191	128–191	0x10000–0x1FFFF

In UTF-8 the characters in the range 128–191 are illegal as first characters. The characters 254 and 255 are completely illegal and should not appear at all since they are related to UTF-16.

Instead of providing a never-complete truckload of other input formats, LuaTEX sticks to one input encoding but at the same time provides hooks that permits users to write filters that preprocess their input into UTF.

While writing the LuaTEX code as well as the ConTEXt input handling, we experimented a lot. Right from the beginning we had a pretty clear picture of what we wanted to achieve and how it could be done, but in the end arrived at solutions that permitted fast and efficient Lua scripting as well as a simple interface.

What is involved in handling any input encoding and especially UTF? First of all, we wanted to support UTF-8 as well as UTF-16. LuaTEX implements UTF-8 rather straightforwardly: it just assumes that the input is usable UTF. This means that it does not combine characters. There is a good reason for this: any automation needs to be configurable (on/off) and the more is done in the core, the slower it gets.

In Unicode, when a character is followed by an 'accent', the standard may prescribe that these two characters are replaced by one. Of course, when characters turn into glyphs, and when no matching glyph is present, we may need to decompose any character into components and paste them together from glyphs in fonts. Therefore, as a first step, a collapser was written. In the (pre)loaded Lua tables we have stored information about what combination of characters need to be combined into another character.

So, an `a` followed by an `` ` `` becomes `à` and an `e` followed by `"` becomes `ë`. This process is repeated till no more sequences combine. After a few alternatives we arrived at a solution that is acceptably fast: mere milliseconds per average page. Experiments demonstrated that we can not gain much by implementing this in pure C, but we did gain some speed by using a dedicated loop-over-utf-string function.

A second UTF related issue is UTF-16. This coding scheme comes in two endian variants. We wanted to do the conversion in Lua, but decided to play a bit with a multi-byte file read function. After some experiments we quickly learned that hard coding such methods in TEX was doomed to be complex, and the whole idea behind LuaTEX is to make things less complex. The complexity has to do with the fact that we need some control over the different linebreak triggers, that is, (combinations of) character 10 and/or 13. In the end, the multi-byte readers were removed from the code and we ended up with a pure Lua solution, which could be sped up by using a multi-byte loop-over-string function.

Instead of hard coding solutions in LuaTEX a couple of fast loop-over-string functions were added to the Lua string function repertoire and the solutions were coded in Lua. We did extensive timing with huge UTF-16 encoded files, and are confident that fast solutions can be found. Keep in mind that reading files is never the bottleneck anyway. The only drawback of an efficient UTF-16 reader is that the file is loaded into memory, but this is hardly a problem.

Concerning arbitrary input encodings, we can be brief. It's rather easy to loop over a string and replace characters in the 0–255 range by their UTF counterparts. All one needs is to maintain conversion tables and TEX macro packages have always done that.

Yet another (more obscure) kind of remapping concerns those special TEX characters. If we use a traditional TEX auxiliary file, then we must make sure that for instance percent signs, hashes, dollars and other characters are handled right. If we set the catcode of the percent sign to 'letter', then we get into trouble when such a percent sign ends up in the table of contents and is read in under a different catcode regime (and becomes for instance a comment symbol). One way to deal with such situations is to temporarily move the problematic characters into a private Unicode area and deal with them accordingly. In that case they no longer can interfere.

Where do we handle such conversions? There are two places where we can hook converters into the input.

1. each time when we read a line from a file, i.e. we can hook conversion code into the read callbacks
2. using the special `process_input_buffer` callback which is called whenever TEX needs a new line of input

Because we can overload the standard file open and read functions, we can easily hook the UTF collapse function into the readers. The same is true for the UTF-16 handler. In ConTEXt, for performance reasons we load such files into memory, which means that we also need to provide a special reader to TEX. When handling UTF-16, we don't need to combine characters so that stage is skipped then.

So, to summarize this, here is what we do in ConTEXt. Keep in mind that we overload the standard input methods and therefore have complete control over how LuaTEX locates and opens files.

1. When we have a UTF file, we will read from that file line by line, and combine characters when collapsing is enabled.
2. When LuaTEX wants to open a file, we look into the first bytes to see if it is a UTF-16 file, in either big or little endian format. When this is the case, we load the file into memory, convert the data to UTF-8, identify lines, and provide a reader that will give back the file linewise.
3. When we have been told to recode the input (i.e. when we have enabled an input regime) we

use the normal line-by-line reader and convert those lines on the fly into valid UTF. No collapsing is needed.

Because we conduct our experiments in ConTEXt MkIV the code that we provide may look a bit messy and more complex than the previous description may suggest. But keep in mind that a mature macro package needs to adapt to what users are accustomed to. The fact that LuaTEX moved on to UTF input does not mean that all the tools that users use and the files that they have produced over decades automagically convert as well.

Because we are now living in a UTF world, we need to keep that in mind when we do tricky things with sequences of characters, for instance in processing verbatim. When we implement verbatim in pure TEX we can do as before, but when we let Lua kick in, we need to use string methods that are UTF-aware. In addition to the linked-in Unicode library, there are dedicated iterator functions added to the `string` namespace; think of:

```
for c in string.utfcharacters(str) do
    something_with(c)
end
```

Occasionally we need to output raw 8-bit code, for instance to DVI or PDF backends (specials and literals). Of course we could have cooked up a truckload of conversion functions for this, but during one of our travels to a TEX conference, we came up with the following trick.

We reserve the top 256 values of the Unicode range, starting at hexadecimal value 0x110000, for byte output. When writing to an output stream, that offset will be subtracted. So, 0x1100A9 is written out as hexadecimal byte value A9, which is the decimal value 169, which in the Latin 1 encoding is the slot for the copyright sign.