

Content

Introduction	4
1 The first decade	6
2 Plug mode, an application of ffi	8
3 Variable fonts	22
4 Emoji again	32
5 Children of T_EX	46
6 Performance	48
7 Editing	66
8 Advertising T_EX	72
9 Tricky fences	74
10 From 5.2 to 5.3	76

Introduction

With L^AT_EX version 1.0 being released it's not time to move on to a next stage in the development. The first four stages were discussed in 'mk', 'hybrid', 'about' and 'still'. Much in there ended up as article in user group journals. some was just a wrap-up of something I ran into or played with. Also, some of it could be seen as a kind of manual for a specific aspect of L^AT_EX and/or C^ON^TE^XT.

In this document we continue this kind of reporting. Maybe it's useful for others to read about it but in the first place it serves me to wrap up experiences occasionally.

Some chapters were meant for publications in user groups journals so they are made public afterwards. I like to thank Karl Berry for correcting many of my mistakes and improving the content. Because Luigi Scarso and I spend quite some time on L^AT_EX development, we also share many of the experiences described in this document. Without his patience with me this would not be possible.

Hans Hagen
Hasselt NL
2016 onward

<http://www.luatex.org>
<http://www.pragma-ade.com>

1 The first decade

When writing this it's hard to believe that we're already a decade working on L^AT_EX and about the same time on M_KIV. The question is, did we achieve the objectives? The answer can easily be "yes" because we didn't start with objectives, just with some experiments with a LUA extension interface. However, it quickly became clear that this was the way to go. Already in an early stage we took a stand in what direction we had to move.

How did we end up with LUA and not one of the other popular scripting languages? The C_{ON}T_EX_T macro package always came with a runner. Not only did the runner manage the (often) multiple runs, it also took care of sorting the index and other inter-job activities. Additional helpers were written for installing fonts, managing (and converting) images, job control, etc. First they were binaries (written in MODULA 2), but successive implementations used PERL and RUBY. When I found out that the SCITE editor I switched to had an extension mechanism using LUA, I immediately liked that language. It's clean, not bloated, relatively stable, evolves in an academic environment and is not driven by commerce and/or short term success, and above all, the syntax makes the code look good. So, it was the most natural candidate for extending T_EX.

Already for along time, T_EX is a stable program and whatever we do with it, we should not break it. There has been frontend extensions, like ϵ -T_EX, and backend extensions, like PDF_TE_X, and experiments like OMEGA and ALEPH and we could start from there. So, basically we took PDF_TE_X, after all, that was what we used for the first experiments, and merged some ALEPH directional code in it. A tremendous effort was undertaken (thanks to funding by the Oriental T_EX project) to convert the code base from PASCAL to C.

It is hard to get an agreement over what needs to be added and it's a real waste of time to enter that route by endless discussions: every T_EX user has different demands and macro packages differ in philosophy. So, in the spirit of the extension language LUA we stuck to concept of "If you want it better, write it in LUA". As a consequence we had to provide access to the internals with efficient and convenient methods, something that happened stepwise. We did extend the engine with a few features that make live easier but tried to limit ourselves. On the other hand, due to developments with fonts and languages we generalized these concepts so that extending and controlling them is easier. And, due to developments in math font technology we also added alternative code paths to the math renderer.

All these matters have been presented and discussed at meetings, in user group journals and in documents that are part of the C_{ON}T_EX_T suite. And during this decade the C_{ON}T_EX_T users have been patient testers of whatever we threw at them in the M_KIV version of this macro package.

It's kind of interesting to note that in the T_EX community it takes a while before version 1 of programs becomes available. Some programs never (seem to) reach that state. However, for us version 1.0 marks the moment that we consider the interfaces to be stable. Of course we move on so a version 2.0 can divert and provide more or even

less interfaces, provide new functionality or drop obsolete features. The intermediate versions (up to version one) were always quite useable in production. In 2005 the first prototype of L^AT_EX was demonstrated at the TUG conference, and in 2007 at the TUG conference we had a whole day on L^AT_EX. At that time C^ON^TE^XT M^KI^V evolved fast and we already had decent O^PE^NT_E^PE support as part of the oriental T_EX project. It was in those years that the major reorganization of the code base took place but in successive years many subsystems were opened and cleaned up. There were some occasions where an interface was changed for the better but adapting was not that hard. It might have helped that much of C^ON^TE^XT M^KI^V is written in L^UA. What also helped is that most C^ON^TE^XT users quickly switched to M^KI^V, if only because M^KI^I was frozen. And, thanks to those users, we were able to root out bugs and bottlenecks. It was interesting to see that the approach of mixing T_EX, M^ET^AP^OST and L^UA caught on quite well.

By the end of September 2016, at the 10th C^ON^TE^XT meeting we released what we call the first long term stable version of L^AT_EX. This version performs quite well but we might still add a few things here and there and the code will be further cleaned up and documented. In the meantime L^AT_EX is also used in other macro packages. It will not replace P^DF_T_EX (at least not soon) because that engine does the job for most of the publications done in T_EX: articles. As they are mostly in English and use traditional fonts, there is no need to switch to the more flexible but somewhat slower L^AT_EX. In a similar fashion X_EL_AT_EX serves those who want the benefits of P^DF_T_EX, hard-coded font support and token juggling at the T_EX level. We will support those engines with M^KI^I but as mentioned, we will not develop new code for. We strongly advice C^ON^TE^XT users to use L^AT_EX but there the advertisements stop. Personally I haven't used P^DF_T_EX (which made T_EX survive in the evolving world of electronic documents) for a decade and I never really used X_EL_AT_EX (which opened up the T_EX world to modern fonts). At least for the coming decade I hope that L^AT_EX can serve us well.

2 Plug mode, an application of ffi

A while ago, at an NTG meeting, Kai Eigner and Ivo Geradts demonstrated how to use the Harfbuzz (hb) library for processing OPENTYPE fonts. The main motivation for them playing with that was that it provides a way to compare the LUA based font machinery with other methods. They also assumed that it would give a better performance for complex fonts and/or scripts.

One of the guiding principles of L^AT_EX development is that we don't provide hard coded solutions. For that reason we opened up the internals so that one can provide solutions written in pure LUA, but, of course, one can cooperate with libraries via LUA code as well. Hard coding solutions makes no sense as there are often several solutions possible, depending on one's need. Although development is closely related to CON_TE_XT, the development of the L^AT_EX engine is generic. We try to be macro package agnostic. Already in an early stage we made sure that the CON_TE_XT font handler could be used in other packages as well, but one can easily dream up light weight variants for specific purposes. The standard T_EX font handling was kept and is called **base** mode in CON_TE_XT. The LUA variant is tagged **node** mode because it operates on the node list. Later we will refer to these modes.

With the output of X_YL^AT_EX for comparison, the first motive mentioned for looking into support for such a library is not that strong. And when we want to test against the standard, we can use MS-Word. A minimal CON_TE_XT MkIV installation one only has the L^AT_EX engine. Maintaining several renderers simultaneously might give rise to unwanted dependencies.

The second motive could be more valid for users because, for complex fonts, there is—or at least was—a performance hit with the LUA variant. Some fonts use many lookup steps or are inefficient even in using their own features. It must be said that till now I haven't heard CON_TE_XT users complain about speed. In fact, the font handling became many times faster the last few years, and probably no one even noticed. Also, when using alternatives to the built in methods, in the end, you will loose functionality and/or interactions with other mechanisms that are built into the current font system. Any possible gain in speed is lost, or even becomes negative, when a user wants to use additional functionality that requires additional processing.¹

Just kicking in some alternative machinery is not the whole story. We still need to deal with the way T_EX sees text, and that, in practice, is as a sequence of glyph nodes—mixed with discretionaries for languages that hyphenate, glue, kern, boxes, math, and more. It's the discretionary part that makes it a bit complex. In contextual analysis as well as positioning one needs to process up to three additional cases: the pre, post and replace texts—either or not linked backward and forward. And as applied features accumulate one ends up winding and unwinding these snippets. In the process one also needs

¹ In general we try to stay away from libraries. For instance, graphics can be manipulated with external programs, and caching the result is much more efficient than recreating it. Apart from SQL support, where integration makes sense, I never felt the need for libraries. And even SQL can efficiently be dealt with via intermediate files.

to keep an eye on spaces as they can be involved in lookups. Also, when injecting or removing glyphs one needs to deal with attributes associated with nodes. Of course something hard codes in the engine might help a little, but then one ends up with the situation where macro packages have different demands (and possible interactions) and no solution is the right one. Using LUA as glue is a way to avoid that problem. In fact, once we go along that route, it starts making sense to come up with a stripped down L^AT_EX that might suit C^ON^TE^XT better, but it's not a route we are eager to follow right now.

Kai and Ivo are plain T_EX users so they use a font definition and switching environment that is quite different from C^ON^TE^XT. In an average C^ON^TE^XT run the time spent on font processing is measurable but not the main bottleneck because other time consuming things happen. Sometimes the load on the font subsystem can be higher because we provide additional features normally not found in O^PE^NT^EY^E. Add to that a more dynamic font model and it will be clear that comparing performance between situations that use different macro packages is not that trivial (or relevant).

More reasons why we follow a LUA route are that we: support (run time generated) virtual fonts, are able to kick in additional features, can let the font mechanism cooperate with other functionality, and so on. In the upcoming years more trickery will be provided in the current mechanisms. Because we had to figure out a lot of these O^PE^NT^EY^E things a decade ago when standards were fuzzy quite some tracing and visualization is available. Below we will see some timings, It's important to keep in mind that in C^ON^TE^XT the O^PE^NT^EY^E font handler can do a bit more if requested to do so, which comes with a bit of overhead when the handler is used in C^ON^TE^XT—something we can live with.

Some time after Kai's presentation he produced an article, and that was the moment I looked into the code and tried to replicate his experiments. Because we're talking libraries, one can understand that this is not entirely trivial, especially because I'm on another platform than he is—Windows instead of OSX. The first thing that I did was rewrite the code that glues the library to T_EX in a way that is more suitable for C^ON^TE^XT. Mixing with existing modes (`base` or `node` mode) makes no sense and is asking for unwanted interferences, so instead a new `plug` mode was introduced. A sort of general text filtering mechanism was derived from the original code so that we can plug in whatever we want. After all, stability is not the strongest point of today's software development, so when we depend on a library, we need to be prepared for other (library based) solutions—for instance, if I understood correctly, X_EL_AT_EX switched a few times.

After redoing the code the next step was to get the library running and I decided that the `ffi` route made most sense.² Due to some expected functions not being supported, my efforts in using the library failed. At that time I thought it was a matter of interfacing, but I could get around it by piping into the command line tools that come with the library, and that was good enough for testing. Of course it was dead slow, but the main objective was comparison of rendering so it doesn't matter that much. After that I just quit and moved on to something else.

² One can think of a intermediate layer but I'm pretty sure that I have different demands than others, but `ffi` sort of frees us from endless discussions.

At some point Kai's article came close to publishing, and I tried the old code again, and, surprise, after some messing around, the library worked. On my system the one shipped with Inkscape is used, which is okay as it frees me from bothering about installations. As already mentioned, we have no real reason in `CONTEXT` for using fonts libraries, but the interesting part was that it permitted me to play with this so called `ffi`. At that moment it was only available in `LUAJITTEX` because that creates a nasty dependency, after a while, Luigi Scarso and I managed to get a similar library working in stock `LUATEX` which is of course the reference. So, I decided to give it a second try, and in the process I rewrote the interfacing code. After all, there is no reason not to be nice for libraries and optimize the interface where possible.

Now, after a decade of writing `LUA` code, I dare to claim that I know a bit about how to write relatively fast code. I was surprised to see that where Kai claimed that the library was faster than the `LUA` code. I saw that it really depends on the font. Sometimes the library approach is actually slower, which is not what one expects. But remember that one argument for using a library is for complex fonts and scripts. So what is meant with complex?

Most Latin fonts are not complex—ligatures and kerns and maybe a little bit of contextual analysis. Here the `LUA` variant is the clear winner. It runs up to ten times faster. For more complex Latin fonts, like `EBgaramond`, that resolves ligatures in a different way, the library catches up, but still the `LUA` handler is faster. Keep in mind that we need to juggle discretionary nodes in any case. One difference between both methods is that the `LUA` handler runs over all the lists (although it has to jump over fonts not being processed then), while the library gets snippets. However, tests show that the overhead involved in that is close to zero and can be neglected. Already long ago we saw that when we compared `MKIVLUATEX` and `MKIIXTEX`, the `LUA` based font handler is not that slow at all. This makes sense because the problem doesn't change, and maybe more importantly because `LUA` is a pretty fast language. If one or the other approach is less than two times faster the gain will probably go unnoticed in real runs. In my experience a few bad choices in macro or style writing is more harmful than a bit slower font machinery. Kick in some additional node processing and it might make comparison of a run even harder. By the way, one reason why font handling has been sped up over the years is because our workflows sometimes have a high load, and, for instance, processing a set of 5 documents remotely has to be fast. Also, in an edit workflow you want the runtime to be a bit comfortable.

Contrary to Latin, a pure Arabic text (normally) has no discretionary nodes, and the library profits most of this. Some day I have to pick up the thread with Idris about the potential use of discretionary nodes in Arabic typesetting. Contrary to Arabic, Latin text has not many replacements and positioning, and, therefore, the `LUA` variant gets the advantage. Some of the additional features that the `LUA` variant provides can, of course, be provided for the library variant by adding some pre- and postprocessing of the list, but then you quickly lose any gain a library provides. So, Arabic has less complex node lists with no branches into discretinaries, but it definitely has more replacements, positioning and contextual lookups due to the many calls to helpers in the `LUA` code. Here the library should win because it can (I assume) use more optimized datastructures.

In Kai's prototype there are some cheats for right-to-left rendering and special scripts like Devanagari. As these tweaks mostly involve discretionary nodes; there is no real need for them. When we don't hyphenate no time is wasted anyway. I didn't test Devanagari, but there is some preprocessing needed in the LUA variant (provided by Kai and Ivo) that I might rewrite from scratch once I understand what happens there. But still, I expect the library to perform somewhat better there but I didn't test it. Eventually I might add support for some more scripts that demand special treatments, but so far there has not been any request for it.

So what is the processing speed of non-Latin scripts? An experiment with Arabic using the frequently used Arabtype font showed that the library performs faster, but when we use a mixed Latin and Arabic document the differences become less significant. On pure Latin documents the LUA variant will probably win. On pure Arabic the library might be on top. On average there is little difference in processing speed between the LUA and library engines when processing mixed documents. The main question is, does one want to lose functionality provided by the LUA variant? Of course one can depend on functionality provided by the library but not by the LUA variant. In the end the user decides.

How did we measure? The baseline measurement is the so called `none` mode: nothing is done there. It's fast but still takes a bit of time as it is triggered by a general mode identifying pass. That pass determines what font processing modes are needed for a list. `Base` mode only makes sense for Latin and has some limitations. It's fast and, basically, its run time can be neglected. That's why, for instance, `PDFTEX` is faster than the other engines, but it doesn't do UNICOD well. `Node` mode is the fancy name for the LUA font handler. So, in order of increasing run time we have: `none`, `base` and `node`. If we compare `node` mode with `plug` mode (in our case using the `hb` library), we can subtract `none` mode. This gives a cleaner (more distinctive) comparison but not a real honest one because the identifying pass always happens.

We also tested with and without hyphenation, but in practice that makes no sense. Only verbatim is typeset that way, and normally we typeset that in `none` mode anyway. On the other hand mixing fonts does happen. All the tests start with forced garbage collection in order to get rid of that variance. We also pack into horizontal boxes so that the par builder (with all kind of associated callbacks) doesn't kick in, although the `node` mode should compensate that.

Keep in mind that the tests are somewhat dumb. There is no overhead in handling structure, building pages, adding color or whatever. I never process raw text. As a reference it's no problem to let `CONTEXT` process hundreds of pages per second. In practice a moderate complex document like the `metafun` manual does some 20 pages per second. In other words, only a fraction of the time is spent on fonts. The timings for `LUATEX` are as follows:

luatex latin

modern	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.48	0.04	-0.75	0.39	0.05
context node	1.23	0.79	0.00	1.00	1.00
context none	0.44	0.00	-0.79	0.36	0.00
harfbuzz native	5.06	4.62	3.83	4.12	5.86
harfbuzz uniscribe	5.24	4.80	4.02	4.27	6.10

pagella	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.50	0.03	-0.77	0.39	0.04
context node	1.27	0.80	0.00	1.00	1.00
context none	0.47	0.00	-0.80	0.37	0.00
harfbuzz native	4.96	4.49	3.69	3.89	5.58
harfbuzz uniscribe	5.49	5.02	4.22	4.31	6.24

dejavu	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.46	0.04	-1.21	0.28	0.03
context node	1.68	1.25	0.00	1.00	1.00
context none	0.43	0.00	-1.25	0.25	0.00
harfbuzz native	4.50	4.07	2.82	2.68	3.26
harfbuzz uniscribe	4.79	4.37	3.12	2.86	3.49

cambria	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.44	0.02	-1.67	0.21	0.01
context node	2.11	1.69	0.00	1.00	1.00
context none	0.43	0.00	-1.69	0.20	0.00
harfbuzz native	4.59	4.16	2.47	2.17	2.47
harfbuzz uniscribe	5.03	4.60	2.91	2.38	2.73

ebgaramond	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.50	0.06	-1.86	0.21	0.03
context node	2.36	1.92	0.00	1.00	1.00
context none	0.43	0.00	-1.92	0.18	0.00
harfbuzz native	4.96	4.52	2.60	2.10	2.35
harfbuzz uniscribe	5.17	4.74	2.81	2.19	2.46

lucidaot	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.48	0.01	-0.45	0.52	0.02
context node	0.93	0.45	0.00	1.00	1.00
context none	0.47	0.00	-0.45	0.51	0.00
harfbuzz native	4.28	3.81	3.35	4.62	8.42
harfbuzz uniscribe	4.68	4.21	3.76	5.06	9.32

luatex arabic

arabtype	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.42	0.00	-14.75	0.03	0.00
context node	15.17	14.76	0.00	1.00	1.00
context none	0.41	0.00	-14.76	0.03	0.00
harfbuzz native	7.14	6.73	-8.02	0.47	0.46
harfbuzz uniscribe	7.68	7.27	-7.49	0.51	0.49

husayni	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.45	-0.01	-25.63	0.02	-0.00
context node	26.08	25.62	0.00	1.00	1.00
context none	0.46	0.00	-25.62	0.02	0.00
harfbuzz native	10.50	10.04	-15.58	0.40	0.39
harfbuzz uniscribe	18.96	18.50	-7.12	0.73	0.72

luatex mixed

arabtype	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.68	-0.01	-7.18	0.09	-0.00
context node	7.85	7.17	0.00	1.00	1.00
context none	0.69	0.00	-7.17	0.09	0.00
harfbuzz native	5.82	5.13	-2.03	0.74	0.72
harfbuzz uniscribe	6.21	5.53	-1.64	0.79	0.77

husayni	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.72	0.05	-11.20	0.06	0.00
context node	11.92	11.25	0.00	1.00	1.00
context none	0.67	0.00	-11.25	0.06	0.00
harfbuzz native	6.93	6.25	-4.99	0.58	0.56
harfbuzz uniscribe	9.85	9.18	-2.07	0.83	0.82

The timings for L^AJIT_TE_X are, of course, overall better. This is because the virtual machine is faster, but at the cost of some limitations. We seldom run into these limitations, but fonts with large tables can't be cached unless we rewrite some code and sacrifice clean solutions. Instead, we perform a runtime conversion which is not that noticeable when it's just a few fonts. The numbers below are not influenced by this as the test stays away from these rare cases.

luajittex latin

modern	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.42	0.03	-0.36	0.54	0.09
context node	0.77	0.39	0.00	1.00	1.00
context none	0.38	0.00	-0.39	0.50	0.00

harfbuzz native	3.07	2.69	2.30	3.98	6.90
harfbuzz uniscribe	3.05	2.67	2.28	3.94	6.84

pagella	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.44	0.02	-0.37	0.54	0.05
context node	0.80	0.39	0.00	1.00	1.00
context none	0.42	0.00	-0.39	0.52	0.00
harfbuzz native	3.02	2.61	2.22	3.77	6.74
harfbuzz uniscribe	3.01	2.59	2.20	3.74	6.69

dejavu	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.40	0.04	-0.59	0.41	0.06
context node	0.98	0.62	0.00	1.00	1.00
context none	0.36	0.00	-0.62	0.37	0.00
harfbuzz native	3.02	2.66	2.04	3.07	4.28
harfbuzz uniscribe	2.97	2.60	1.98	3.01	4.19

cambria	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.38	0.02	-0.79	0.33	0.02
context node	1.17	0.80	0.00	1.00	1.00
context none	0.37	0.00	-0.80	0.31	0.00
harfbuzz native	2.91	2.54	1.74	2.48	3.16
harfbuzz uniscribe	2.86	2.50	1.69	2.45	3.11

ebgaramond	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.43	0.05	-0.89	0.33	0.05
context node	1.32	0.94	0.00	1.00	1.00
context none	0.38	0.00	-0.94	0.29	0.00
harfbuzz native	3.00	2.62	1.68	2.27	2.78
harfbuzz uniscribe	2.98	2.60	1.66	2.25	2.77

lucidaot	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.41	-0.01	-0.21	0.66	-0.04
context node	0.63	0.20	0.00	1.00	1.00
context none	0.42	0.00	-0.20	0.67	0.00
harfbuzz native	2.61	2.18	1.98	4.16	10.71
harfbuzz uniscribe	2.59	2.17	1.97	4.14	10.65

luajittex arabic

arabtype	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.32	-0.00	-6.85	0.04	-0.00
context node	7.17	6.84	0.00	1.00	1.00
context none	0.32	0.00	-6.84	0.04	0.00

harfbuzz native	4.63	4.31	-2.54	0.65	0.63
harfbuzz uniscribe	4.67	4.35	-2.50	0.65	0.64
husayni	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.35	-0.00	-11.90	0.03	-0.00
context node	12.25	11.90	0.00	1.00	1.00
context none	0.35	0.00	-11.90	0.03	0.00
harfbuzz native	15.28	14.93	3.03	1.25	1.25
harfbuzz uniscribe	15.25	14.90	3.00	1.25	1.25

luajittex mixed

arabtype	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.57	-0.03	-3.47	0.14	-0.01
context node	4.04	3.44	0.00	1.00	1.00
context none	0.60	0.00	-3.44	0.15	0.00
harfbuzz native	3.69	3.09	-0.35	0.91	0.90
harfbuzz uniscribe	3.69	3.08	-0.35	0.91	0.90
husayni	t	$t - t_{\text{none}}$	$t - t_{\text{node}}$	t/t_{node}	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.62	0.04	-5.33	0.10	0.01
context node	5.94	5.37	0.00	1.00	1.00
context none	0.57	0.00	-5.37	0.10	0.00
harfbuzz native	7.19	6.62	1.25	1.21	1.23
harfbuzz uniscribe	7.11	6.53	1.17	1.20	1.22

A few side notes. Since a library is an abstraction, one has to live with what one gets. In my case that was a crash in UTF-32 mode. I could get around it, but one advantage of using LUA is that it's hard to crash—if only because as a scripting language it manages its memory well without user interference. My policy with libraries is just to wait till things get fixed and not bother with the why and how of the internals.

Although CONTEXT will officially support the `plug` model, it will not be actively used by me, or in documentation, so for support users are on their own. I didn't test the `plug` mode in real documents. Most documents that I process are Latin (or a mix), and redefining feature sets or adapting styles for testing makes no sense. So, can one just switch engines without looking at the way a font is defined? The answer is—not really, because (even without the user knowing about it) virtual fonts might be used, additional features kicked in and other mechanisms can make assumptions about how fonts are dealt with too.

The useability of `plug` mode probably depends on the workflow one has. We use CONTEXT in a few very specific workflows where, interestingly, we only use a small subset of its functionality. Most of which is driven by users, and tweaking fonts is popular and has resulted in all kind of mechanisms. So, for us it's unlikely that we will use it. If you process (in bursts) many documents in succession, each demanding a few runs, you don't want to sacrifice speed.

Of course timing can (and likely will) be different for plain $\text{T}_{\text{E}}\text{X}$ and $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ usage. It depends on how mechanisms are hooked into the callbacks, what extra work is done or not done compared to $\text{C}_{\text{O}}\text{N}\text{T}_{\text{E}}\text{X}\text{T}$. This means that my timings for $\text{C}_{\text{O}}\text{N}\text{T}_{\text{E}}\text{X}\text{T}$ for sure will differ from those of other packages. Timings are a snapshot anyway. And as said, font processing is just one of the many things that goes on. If you are not using $\text{C}_{\text{O}}\text{N}\text{T}_{\text{E}}\text{X}\text{T}$ you probably will use Kai's version because it is adapted to his use case and well tested.

A fundamental difference between the two approaches is that—whereas the $\text{L}_{\text{U}}\text{A}$ variant operates on node lists only, the `plug` variant generates strings that get passed to a library where, in the $\text{C}_{\text{O}}\text{N}\text{T}_{\text{E}}\text{X}\text{T}$ variant of hb support, we use UTF-32 strings. Interesting, a couple of years ago I considered using a similar method for $\text{L}_{\text{U}}\text{A}$ but eventually decided against it, first of all for performance reasons, but mostly because one still has to use some linked list model. I might pick up that idea as a variant, but because all this $\text{T}_{\text{E}}\text{X}$ related development doesn't really pay off and costs a lot of free time it will probably never happen.

I finish with a few words on how to use the plug model. Because the library initializes a default set of features,³ all you need to do is load the plugin mechanism:

```
\usemodule [fonts-plugins]
```

Next you define features that use this extension:

```
\definefontfeature
  [hb-native]
  [mode=plug,
   features=harfbuzz,
   shaper=native]
```

After this you can use this feature set when you define fonts. Here is a complete example:

```
\usemodule [fonts-plugins]

\starttext

  \definefontfeature
    [hb-library]
    [mode=plug,
     features=harfbuzz,
     shaper=native]

  \definedfont [Serif*hb-library]

  \input ward \par

  \definefontfeature
```

³ Somehow passing features to the library fails for Arabic. So when you don't get the desired result, just try with the defaults.

```
[hb-binary]
[mode=plug,
 features=harfbuzz,
 method=binary,
 shaper=uniscribe]
```

```
\definedfont [Serif*hb-binary]
```

```
\input ward \par
```

```
\stoptext
```

The second variant uses the `hb-shape` binary which is, of course, pretty slow, but does the job and is okay for testing.

There are a few trackers available too:

```
\enabletrackers [fonts.plugins.hb.colors]
\enabletrackers [fonts.plugins.hb.details]
```

The first one colors replaced glyphs while the second gives lot of information about what is going on. If you want to know what gets passed to the library you can use the `text` plugin:

```
\definefontfeature [test] [mode=plug, features=text]
\start
  \definedfont [Serif*test]
  \input ward \par
\stop
```

This produces something:

```
otf plugin > text > start run 3
otf plugin > text > 001 : [-] The [+] -> U+00054 U+00068 U+00065
otf plugin > text > 002 : [+] Earth, [+] -> U+00045 U+00061 U+00072 ...
otf plugin > text > 003 : [+] as [+] -> U+00061 U+00073
otf plugin > text > 004 : [+] a [+] -> U+00061
otf plugin > text > 005 : [+] habi- [-] -> U+00068 U+00061 U+00062 ...
otf plugin > text > 006 : [-] tat [+] -> U+00074 U+00061 U+00074
otf plugin > text > 007 : [+] habitat [+] -> U+00068 U+00061 U+00062 ...
otf plugin > text > 008 : [+] for [+] -> U+00066 U+0006F U+00072
otf plugin > text > 009 : [+] an- [-] -> U+00061 U+0006E U+0002D
```

You can see how hyphenation of `habi-tat` results in two snippets and a whole word. The font engine can decide to turn this word into a disc node with a pre, post and replace text. Of course the machinery will try to retain as many hyphenation points as possible. Among the tricky parts of this are lookups across and inside discretionary nodes resulting in (optional) replacements and kerning. You can imagine that there is some trade off between performance and quality here. The results are normally acceptable, especially because `TeX` is so clever in breaking paragraphs into lines.

Using this mechanism (there might be variants in the future) permits the user to cook up special solutions. After all, that is what L^AT_EX is about—the traditional core engine with the ability to plug in your own code using L^AU_A. This is just an example of it.

I'm not sure yet when the plugin mechanism will be in the C^ON^TE^XT distribution, but it might happen once the `ffi` library is supported in L^AT_EX. At the end of this document the basics of the test setup are shown, just in case you wonder what the numbers apply to.

Just to put things in perspective, the current (February 2017) M^ET^AF^UN manual has 424 pages. It takes L^AT_EX 18.3 seconds and L^AU_AJIT_EX 14.4 seconds on my Dell 7600 laptop with 3840QM mobile i7 processor. Of this 6.1 (4.5) seconds is used for processing 2170 M^ET^AP^OST graphics. Loading the 15 fonts used takes 0.25 (0.3) seconds, which includes also loading the outline of some. Font handling is part of the, so called, hlist processing and takes around 1 (0.5) second, and attribute backend processing takes 0.7 (0.3) seconds. One problem in these timings is that font processing often goes too fast for timing, especially when we have lots of small snippets. For example, short runs like titles and such take no time at all, and verbatim needs no font processing. The difference in runtime between L^AT_EX and L^AU_AJIT_EX is significant so we can safely assume that we spend some more time on fonts than reported. Even if we add a few seconds, in this rather complete document, the time spent on fonts is still not that impressive. A five fold increase in processing (we use mostly Pagella and DejaVu) is a significant addition to the total run time, especially if you need a few runs to get cross referencing etc. right.

The test files are the familiar ones present in the distribution. The `tufte` example is a good torture test for discretionary processing. We preload the files so that we don't have the overhead of `\input`.

```
\edef\tufte{\cldloadfile{tufte.tex}}
\edef\khatt{\cldloadfile{khatt-ar.tex}}
```

We use six buffers for the tests. The Latin test uses three fonts and also has a paragraph with mixed font usage. Loading the fonts happens once before the test, and the local (re)definition takes no time. Also, we compensate for general overhead by subtracting the `none` timings.

```
\startbuffer[latin-definitions]
\definefont[TestA][Serif*test]
\definefont[TestB][SerifItalic*test]
\definefont[TestC][SerifBold*test]
\stopbuffer
```

```
\startbuffer[latin-text]
\TestA \tufte \par
\TestB \tufte \par
\TestC \tufte \par
\dorecurse {10} {%
  \TestA Fluffy Test Font A
  \TestB Fluffy Test Font B
```

```

    \TestC Fluffy Test Font C
}\par
\stopbuffer

```

The Arabic tests are a bit simpler. Of course we do need to make sure that we go from right to left.

```

\startbuffer[arabic-definitions]
\definedfont[Arabic*test at 14pt]
\setupinterlinespace[line=18pt]
\setupalign[r2l]
\stopbuffer

```

```

\startbuffer[arabic-text]
\dorecurse {10} {
    \khatt\space
    \khatt\space
    \khatt\blank
}
\stopbuffer

```

The mixed case use a Latin and an Arabic font and also processes a mixed script paragraph.

```

\startbuffer[mixed-definitions]
\definefont[TestL][Serif*test]
\definefont[TestA][Arabic*test at 14pt]
\setupinterlinespace[line=18pt]
\setupalign[r2l]
\stopbuffer

```

```

\startbuffer[mixed-text]
\dorecurse {2} {
    {\TestA\khatt\space\khatt\space\khatt}
    {\TestL\lefttoright\tufte}
    \blank
    \dorecurse{10}{%
        {\TestA      }
        {\TestL\lefttoright A snippet text that makes no sense.}
    }
}
\stopbuffer

```

The related font features are defined as follows:

```

\definefontfeature
[test-none]
[mode=none]

```

```

\definefontfeature
  [test-base]
  [mode=base,
   liga=yes,
   kern=yes]

\definefontfeature
  [test-node]
  [mode=node,
   script=auto,
   autoscript=position,
   autolanguage=position,
   ccmp=yes,liga=yes,clig=yes,
   kern=yes,mark=yes,mkmk=yes,
   curs=yes]

\definefontfeature
  [test-text]
  [mode=plug,
   features=text]

\definefontfeature
  [test-native]
  [mode=plug,
   features=harfbuzz,
   shaper=native]

\definefontfeature
  [arabic-node]
  [arabic]

\definefontfeature
  [arabic-native]
  [mode=plug,
   features=harfbuzz,
   script=arab,language=dflt,
   shaper=native]

```

The timings are collected in LUA tables and typeset afterwards, so there is no interference there either.

The timings are as usual a snapshot and just indications. The relative times can differ over time depending on how binaries are compiled, libraries are improved and LUA code evolves. In node mode we can have experimental trickery that is not yet optimized. Also, especially with complex fonts like Husayni, not all shapers give the same result, although node mode and Uniscribe should be the same in most cases. A future (public) version of Husayni will play more safe and use less complex sequences of features.

3 Variable fonts

Introduction

History shows the tendency to recycle ideas. Often quite some effort is made by historians to figure out what really happened, not just long ago, when nothing was written down and we have to do with stories or pictures at most, but also in recent times. Descriptions can be conflicting, puzzling, incomplete, partially lost, biased, . . .

Just as language was invented (or evolved) several times, so were scripts. The same might be true for rendering scripts on a medium. Semaphores came and went within decades and how many people know now that they existed and that encryption was involved? Are the old printing presses truly the old ones, or are older examples simply gone? One of the nice aspects of the internet is that one can now more easily discover similar solutions for the same problem, but with a different (and independent) origin.

So, how about this “new big thing” in font technology: variable fonts. In this case, history shows that it’s not that new. For most T_EX users the names METAFONT and METAPOST will ring bells. They have a very well documented history so there is not much left to speculation. There are articles, books, pictures, examples, sources, and more around for decades. So, the ability to change the appearance of a glyph in a font depending on some parameters is not new. What probably *is* new is that creating variable fonts is done in the natural environment where fonts are designed: an interactive program. The METAFONT toolkit demands quite some insight in programming shapes in such a way that one can change look and feel depending on parameters. There are not that many meta fonts made and one reason is that making them requires a certain mind- and skill set. On the other hand, faster computers, interactive programs, evolving web technologies, where real-time rendering and therefore more or less real-time tweaking of fonts is a realistic option, all play a role in acceptance.

But do interactive font design programs make this easier? You still need to be able to translate ideas into usable beautiful fonts. Taking the common shapes of glyphs, defining extremes and letting a program calculate some interpolations will not always bring good results. It’s like morphing a picture of your baby’s face into yours of old age (or that of your grandparent): not all intermediate results will look great. It’s good to notice that variable fonts are a revival of existing techniques and ideas used in, for instance, multiple master fonts. The details might matter even more as they can now be exaggerated when some transformation is applied.

There is currently (March 2017) not much information about these fonts so what I say next may be partially wrong or at least different from what is intended. The perspective will be one from a T_EX user and coder. Whatever you think of them, these fonts will be out there and for sure there will be nice examples circulating soon. And so, when I ran into a few experimental fonts, with POSTSCRIPT and TRUETYPE outlines, I decided to have a look at what is inside. After all, because it’s visual, it’s also fun to play with. Let’s stress that at the moment of this writing I only have a few simple fonts available, fonts

that are designed for testing and not usage. Some recommended tables were missing and no complex OPEN`TYPE` features are used in these fonts.

The specification

I'm not that good at reading specifications, first of all because I quickly fall asleep with such documents, but most of all because I prefer reading other stuff (I do have lots of books waiting to be read). I'm also someone who has to play with something in order to understand it: trial and error is my *modus operandi*. Eventually it's my intended usage that drives the interface and that is when everything comes together.

Exploring this technology comes down to: locate a font, get the OPEN`TYPE` 1.8 specification from the MICROSOFT website, and try to figure out what is in the font. When I had a rough idea the next step was to get to the shapes and see if I could manipulate them. Of course it helped that in CON`TEXT` we already can load fonts and play with shapes (using METAP`OST`). I didn't have to install and learn other programs. Once I could render them, in this case by creating a virtual font with inline PDF literals, a next step was to apply variation. Then came the first experiments with a possible user interface. Seeing more variation then drove the exploration of additional properties needed for typesetting, like features.

The main extension to the data packaged in a font file concerns the (to be discussed) axis along which variable fonts operate and deltas to be applied to coordinates. The `gdef` table has been extended and contains information that is used in `gpos` features. There are new `hvar`, `vvar` and `mvar` tables that influence the horizontal, vertical and general font dimensions. The `gvar` table is used for TRUE`TYPE` variants, while the `cff2` table replaces the `cff` table for OPEN`TYPE` POST`SCRIPT` outlines. The `avar` and `stat` tables contain some meta-information about the axes of variations.

It must be said that because this is new technology the information in the standard is not always easy to understand. The fact that we have two rendering techniques, POST`SCRIPT` `cff` and TRUE`TYPE` `ttf`, also means that we have different information and perspectives. But this situation is not much different from OPEN`TYPE` standards a few years ago: it takes time but in the end I will get there. And, after all, users also complain about the lack of documentation for CON`TEXT`, so who am I to complain? In fact, it will be those CON`TEXT` users who will provide feedback and make the implementation better in the end.

Loading

Before we discuss some details, it will be useful to summarize what the font loader does when a user requests a font at a certain size and with specific features enabled. When a font is used the first time, its binary format is converted into a form that makes it suitable for use within CON`TEXT` and therefore L`UA`T`E`X. This conversion involves collecting properties of the font as a whole (official names, general dimensions like x-height and em-width, etc.), of glyphs (dimensions, UN`ICODE` properties, optional math properties), and all kinds of information that relates to (contextual) replacements of glyphs (small

caps, oldstyle, scripts like Arabic) and positioning (kerning, anchoring marks, etc.). In the `CONTEXT` font loader this conversion is done in `LUA`.

The result is stored in a condensed format in a cache and the next time the font is needed it loads in an instant. In the cached version the dimensions are untouched, so a font at different sizes has just one copy in the cache. Often a font is needed at several sizes and for each size we create a copy with scaled glyph dimensions. The feature-related dimensions (kerning, anchoring, etc.) are shared and scaled when needed. This happens when sequences of characters in the node list get converted into sequences of glyphs. We could do the same with glyph dimensions but one reason for having a scaled copy is that this copy can also contain virtual glyphs and these have to be scaled beforehand. In practice there are several layers of caching in order to keep the memory footprint within reasonable bounds.⁴

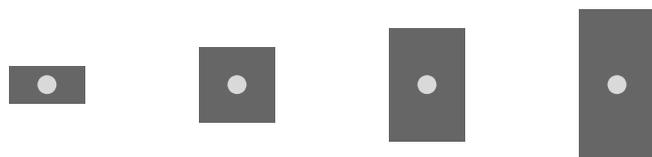
When the font is actually used, interaction between characters is resolved using the feature-related information. When for instance two characters need to be kerned, a lookup results in the injection of a kern, scaled from general dimensions to the current size of the font.

When the outlines of glyphs are needed in `METAFUN` the font is also converted from its binary form to something in `LUA`, but this time we filter the shapes. For a `cff` this comes down to interpreting the `charstrings` and reducing the complexity to `moveto`, `lineto` and `curveto` operators. In the process subroutines are inlined. The result is something that `METAPOST` is happy with but that also can be turned into a piece of a PDF.

We now come to what a variable font actually is: a basic design which is transformed along one or more axes. A simple example is wider shapes:



We can also go taller and retain the width:



Here we have a linear scaling but glyphs are not normally done that way. There are font collections out there with lots of intermediate variants (say from light to heavy) and it's more profitable to sell each variant independently. However, there is often some logic behind it, probably supported by programs that designers use, so why not build that logic into the font and have one file that represents many intermediate forms. In fact, once we have multiple axes, even when the designer has clear ideas of the intended

⁴ In retrospect one can wonder if that makes sense; just look at how much memory a browser uses when it has been open for some time. In the beginning of `LUATEX` users wondered about caching fonts, but again, just look at what amounts browsers cache: it gets pretty close to the average amount of writes that a SSD can handle per day within its guarantee.

usage, nothing will prevent users from tinkering with the axis properties in ways that will fulfil their demands but hurt the designers eyes. We will not discuss that dilemma here.

When a variable font follows the route described above, we face a problem. When you load a TRUETYPE font it will just work. The glyphs are packaged in the same format as static fonts. However, a variable font has axes and on each axis a value can be set. Each axis has a minimum, maximum and default. It can be that the default instance also assumes some transformations are applied. The standard recommends adding tables to describe these things but the fonts that I played with each lacked such tables. So that leaves some guesswork. But still, just loading a TRUETYPE font gives some sort of outcome, although the dimensions (widths) might be weird due to lack of a (default) axis being applied.

An OPENTYPE font with POSTSCRIPT outlines is different: the internal `cff` format has been upgraded to `cff2` which on the one hand is less complicated but on the other hand has a few new operators — which results in programs that have not been adapted complaining or simply quitting on them.

One could argue that a font is just a resource and that one only has to pass it along but that's not what works well in practice. Take L^AT_EX. We can of course load the font and apply axis vales so that we can process the document as we normally do. But at some point we have to create a PDF. We can simply embed the TRUETYPE files but no axis values are applied. This is because, even if we add the relevant information, there is no way in current PDF formats to deal with it. For that, we should be able to pass all relevant axis-related information as well as specify what values to use along these axes. And for TRUETYPE fonts this information is not part of the shape description so then we in fact need to filter and pass more. An OPENTYPE POSTSCRIPT font is much cleaner because there we have the information needed to transform the shape mostly in the glyph description. There we only need to carry some extra information on how to apply these so-called blend values. The region/axis model used there only demands passing a relatively simple table (stripped down to what we need). But, as said above, `cff2` is not backward-compatible so a viewer will (currently) simply not show anything.

Recalling how we load fonts, how does that translate with variable changes? If we have two characters with glyphs that get transformed and that have a kern between them, the kern may or may not transform. So, when we choose values on an axis, then not only glyph properties change but also relations. We no longer can share positional information and scale afterwards because each instance can have different values to start with. We could carry all that information around and apply it at runtime but because we're typesetting documents with a static design it's more convenient to just apply it once and create an instance. We can use the same caching as mentioned before but each chosen instance (provided by the font or made up by user specifications) is kept in the cache. As a consequence, using a variable font has no overhead, apart from initial caching.

So, having dealt with that, how do we proceed? Processing a font is not different from what we already had. However, I would not be surprised if users are not always satisfied with, for instance, kerning, because in such fonts a lot of care has to be given to this

by the designer. Of course I can imagine that programs used to create fonts deal with this, but even then, there is a visual aspect to it too. The good news is that in `CONTEXT` we can manipulate features so in theory one can create a so-called font goodie file for a specific instance.

Shapes

For `OPENTYPE POSTSCRIPT` shapes we always have to do a dummy rendering in order to get the right bounding box information. For `TRUETYPE` this information is already present but not when we use a variable instance, so I had to do a bit of coding for that. Here we face a problem. For `TEX` we need the width, height and depth of a glyph. Consider the following case:



The shape has a bounding box that fits the shape. However, its left corner is not at the origin. So, when we calculate a tight bounding box, we cannot use it for actually positioning the glyph. We do use it (for horizontal scripts) to get the height and depth but for the width we depend on an explicit value. In `OPENTYPE POSTSCRIPT` we have the width available and how the shape is positioned relative to the origin doesn't much matter. In a `TRUETYPE` shape a bounding box is part of the specification, as is the width, but for a variable font one has to use so-called phantom points to recalculate the width and the test fonts I had were not suitable for investigating this.

At any rate, once I could generate documents with typeset text using variable fonts it became time to start thinking about a user interface. A variable font can have predefined instances but of course a user also wants to mess with axis values. Take one of the test fonts: Adobe Variable Font Prototype. It has several instances:

extralight	It looks like this!	<code>weight=0.0 contrast=0.0</code>
light	It looks like this!	<code>weight=150.0 contrast=0.0</code>
regular	It looks like this!	<code>weight=394.0 contrast=0.0</code>
semibold	It looks like this!	<code>weight=600.0 contrast=0.0</code>
bold	It looks like this!	<code>weight=824.0 contrast=0.0</code>
black high contrast	It looks like this!	<code>weight=1000.0 contrast=100.0</code>
black medium contrast	It looks like this!	<code>weight=1000.0 contrast=50.0</code>
black	It looks like this!	<code>weight=1000.0 contrast=0.0</code>

Such an instance is accessed with:

```
\definefont
  [MyLightFont]
  [name:adobevariablefontprototypelight*default]
```

The Avenir Next variable demo font (currently) provides:

regular	It looks like this!	weight=400.0 width=100.0
medium	It looks like this!	weight=500.0 width=100.0
bold	It looks like this!	weight=700.0 width=100.0
heavy	It looks like this!	weight=900.0 width=100.0
condensed	It looks like this!	weight=400.0 width=75.0
medium condensed	It looks like this!	weight=500.0 width=75.0
bold condensed	It looks like this!	weight=700.0 width=75.0
heavy condensed	It looks like this!	weight=900.0 width=75.0

Before we continue I will show a few examples of variable shapes. Here we use some METAFUN magic. Just take these definitions for granted.

```
\startMPcode
draw outlinetext.b
  ("\definedfont[name:adobevariablefontprototypeextralight]foo@bar")
  (withcolor "gray")
  (withcolor red withpen pencircle scaled 1/10)
  xsized .45TextWidth ;
\stopMPcode
```

```
\startMPcode
draw outlinetext.b
  ("\definedfont[name:adobevariablefontprototype]foo@bar")
  (withcolor "gray")
  (withcolor red withpen pencircle scaled 1/10)
  xsized .45TextWidth ;
\stopMPcode
```

```
\startMPcode
draw outlinetext.b
  ("\definedfont[name:adobevariablefontprototypebold]foo@bar")
  (withcolor "gray")
  (withcolor red withpen pencircle scaled 1/10)
  xsized .45TextWidth ;
\stopMPcode
```

```
\startMPcode
draw outlinetext.b
  ("\definefontfeature[whatever][axis={weight:350}]%
  \definedfont[name:adobevariablefontprototype*whatever]foo@bar")
  (withcolor "gray")
  (withcolor red withpen pencircle scaled 1/10)
  xsized .45TextWidth ;
\stopMPcode
```

The results are shown in figure 3.1. What we see here is that as long as we fill the shape everything will look as expected but using an outline only won't. The crucial (control) points are moved to different locations and as a result they can end up inside the shape. Giving up outlines is the price we evidently need to pay. Of course this is not

unique for variable fonts although in practice static fonts behave better. To some extent we're back to where we were with METAFONT and (for instance) Computer Modern: because these originate in bitmaps (and probably use similar design logic) we also can have overlap and bits and pieces pasted together and no one will notice that. The first outline variants of Computer Modern also had such artifacts while in the static Latin Modern successors, outlines were cleaned up.

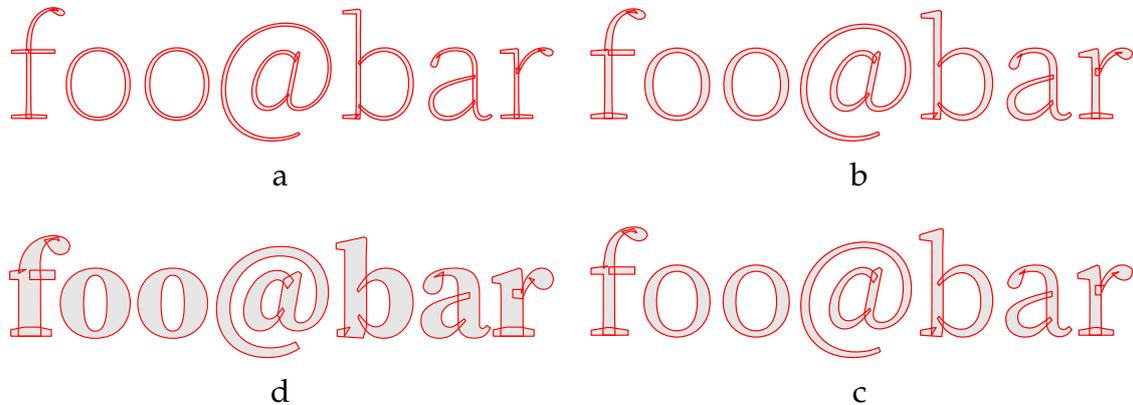


Figure 3.1 Four variants

The fact that we need to preprocess an instance but only know how to do that when we have gotten the information about axis values from the font means that the font handler has to be adapted to keep caching correct. Another definition is:

```
\definefontfeature
  [lightdefault]
  [default]
  [axis={weight:230,contrast:50}]

\definefont
  [MyLightFont]
  [name:adobevariablefontprototype*lightdefault]
```

Here the complication is that where normally features are dealt with after loading, the axis feature is part of the preparation (and caching). If you want the virtual font solution you can do this:

```
\definefontfeature
  [inlinelightdefault]
  [default]
  [axis={weight:230,contrast:50},
   variablesapes=yes]

\definefont
  [MyLightFont]
  [name:adobevariablefontprototype*inlinelightdefault]
```

When playing with these fonts it was hard to see if loading was done right. For instance not all values make sense. It is beyond the scope of this article, but axes like weight, width, contrast and italic values get applied differently to so-called regions (subspaces).

So say that we have an x coordinate with value 50. This value can be adapted in, for instance, four subspaces (regions), so we actually get:

$$x' = x + s_1 \times x_1 + s_2 \times x_2 + s_3 \times x_3 + s_4 \times x_4$$

The (here) four scale factors s_n are determined by the axis value. Each axis has some rules about how to map the values 230 for weight and 50 for contrast to such a factor. And each region has its own translation from axis values to these factors. The deltas x_1, \dots, x_4 are provided by the font. For a POSTSCRIPT-based font we find sequences like:

```
1 <setvstore>
120 [10 -30 40 -60] 1 <blend> ... <operator>
100 120 [10 -30 40 -60] [30 -10 -30 20] 2 <blend> .. <operator>
```

A store refers to a region specification. From there the factors are calculated using the chosen values on the axis. The deltas are part of the glyphs specification. Officially there can be multiple region specifications, but how likely it is that they will be used in real fonts is an open question.

For TRUETYPE fonts the deltas are not in the glyph specification but in a dedicated `gvar` table.

```
apply x deltas [10 -30 40 -60] to x 120
apply y deltas [30 -10 -30 20] to y 100
```

Here the deltas come from tables outside the glyph specification and their application is triggered by a combination of axis values and regions.

The following two examples use Avenir Next Variable and demonstrate that kerning is adapted to the variant.

```
\definefontfeature
  [default:shaped]
  [default]
  [axis={width:10}]

\definefont
  [SomeFont]
  [file:avenirnextvariable*default:shaped]
```

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

Hermann Zapf

```
\definefontfeature
  [default:shaped]
  [default]
```

```
[axis={width:100}]
```

```
\definefont  
[SomeFont]  
[file:avenirnextvariable*default:shaped]
```

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

Hermann Zapf

Embedding

Once we're done typesetting and a PDF file has to be created there are three possible routes:

- We can embed the shapes as PDF images (inline literal) using virtual font technology. We cannot use so-called xforms here because we want to support color selectively in text.
- We can wait till the PDF format supports such fonts, which might happen but even then we might be stuck for years with viewers getting there. Also documents need to get printed, and when printer support might arrive is another unknown.
- We can embed a regular font with shapes that match the chosen values on the axis. This solution is way more efficient than the first.

Once I could interpret the right information in the font, the first route was the way to go. A side effect of having a converter for both outline types meant that it was trivial to create a virtual font at runtime. This option will stay in `CONTEXT` as pseudo-feature `variableshapes`.

When trying to support variable fonts I tried to limit the impact on the backend code. Also, processing features and such was not touched. The inclusion of the right shapes is done via a callback that requests the blob to be injected in the `cff` or `glyf` table. When implementing this I actually found out that the `LUATEX` backend also does some juggling of charstrings, to serve the purpose of inlining subroutines. In retrospect I could have learned a few tricks faster by looking at that code but I never realized that it was there. Looking at the code again, it strikes me that the whole inclusion could be done with LUA code and some day I will give that a try.

Conclusion

When I first heard about variable fonts I was confident that when they showed up they could be supported. Of course a specimen was needed to prove this. A first implementation demonstrates that indeed it's no big deal to let `CONTEXT` with `LUATEX` handle

such fonts. Of course we need to fill in some gaps which can be done once we have complete fonts. And then of course users will demand more control. In the meantime the helper script that deals with identifying fonts by name has been extended and the relevant code has been added to the distribution. At some point the CONTEX Garden will provide the L^AT_EX binary that has the callback.

I end with a warning. On the one hand this technology looks promising but on the other hand one can easily get lost. Probably most such fonts operate over a well-defined domain of values but even then one should be aware of complex interactions with features like positioning or replacements. Not all combinations can be tested. It's probably best to stick to fonts that have all the relevant tables and don't depend on properties of a specific rendering technology.

Although support is now present in the core of CONTEX the official release will happen at the CONTEX meeting in 2017. By then I hope to have tested more fonts. Maybe the interface has also been extended by then because after all, T_EX is about control.

4 Emoji again

Because at the CONTEXT 2016 meeting color fonts⁵ were on the agenda, some time was spent on emoji (these colorful small picture glyphs). When possible I bring kids to the BachoTEX conference so for the 2017 BachoTUG I decided to do something with emoji that, after all, are mostly used by those younger than I am. So, I had to take a look at the current state. Here are some observations.

The UNICODE standard defines a whole lot of emoji and if mankind manages to survive for a while one can assume that a lot more will be added. After all, icons as well as variants keep evolving. There are several ways to organize these symbols in groups but I will not give grouping a try. Just visit emojipedia.org and you get served well. For this story I only mention that:

- There are quite some shapes and nearly all of them are in color. The yellow ones, smilies and such, are quite prominently present but there are many more.
- A special subset is fulfilled by persons: man, woman, girl, boy and recently a baby.
- The grown ups can be combined in loving couples (either or not kissing) and then can form families, but only upto 2 young kids or gender neutral babies.
- All persons can be flagged with one of five skin tones so that not all persons (or heads) look bright yellow.
- Interesting is that girls and boys are still fond of magenta (pinkish) and cyan (blueish) cloths and ornaments. Also haircuts are rather specific to the gender.

For rendering color emojis we have a few color related OPENTYPE font properties available: bitmaps, SVG and stacked glyphs. Now, if you think of the combinations that can be made with skin tones, you realize that fonts can become pretty large if each combination results in a glyph. In the first half of 2017 MICROSOFT released an update for its emoji font and the company took the challenge to provide not only mixed skin tone couples, but also supported skin tones for the kids, including a baby.

This recent addition already adds over 25.000 additional glyphs⁶ so imagine what will happen in the future. But, instead of making a picture for each variant, a different solution has been chosen. For coloring this `seguiemj` font uses the (very flexible) stacking technology: a color shape is an overlay of colored symbols. The colors are organized in pallets and it's no big deal to add additional pallets if needed. Instead of adding pre-composed shapes (as is needed with bitmaps and SVG) snippets are used to build alternative glyphs and these can be combined into new shapes by substitution and positioning (for that kerns, mark anchoring and distance compensation is used).

⁵ For that occasion the cowfont, a practical joke concerning Dutch 'koeieletters', were turned into a color font and presented at the meeting.

⁶ That is the amount I counted when I added all combinations runtime but the emojipedia mentions twice that amount. Currently in CONTEXT we resolve such combinations when requested.

So, a family can be constructed of composed shapes (man, woman, etc) that each are composed of snippets (skull, hair, mouth, eyes). So, effectively a family of four is a bunch of maybe 25 small glyphs overlaid and colored. In figure 4.1 we see how a shape is constructed out of separate glyphs. Figure 4.2 shows how they can be overlaid with colors (we use a dedicated color set).

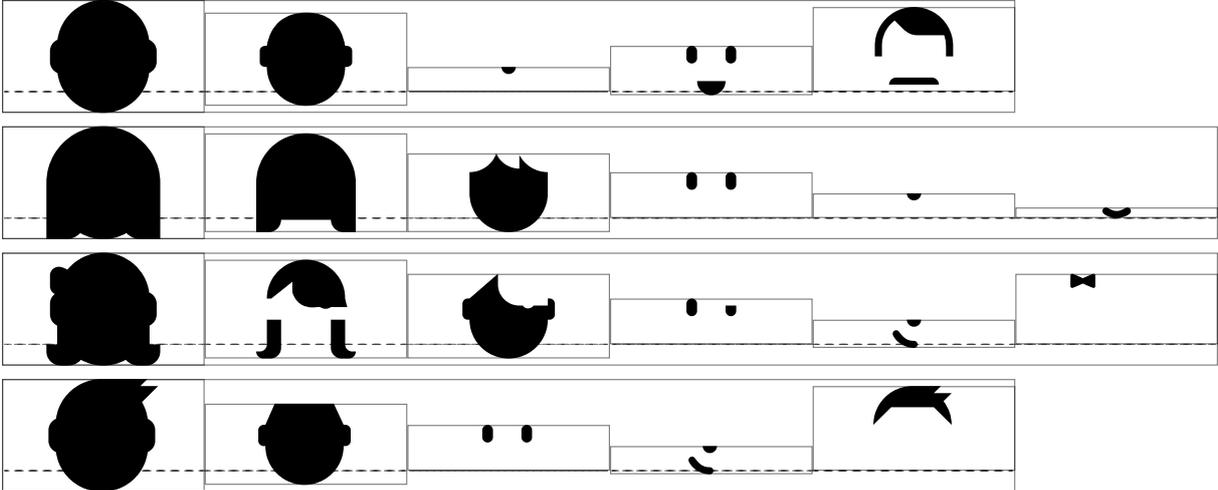


Figure 4.1 Emoji snippets.

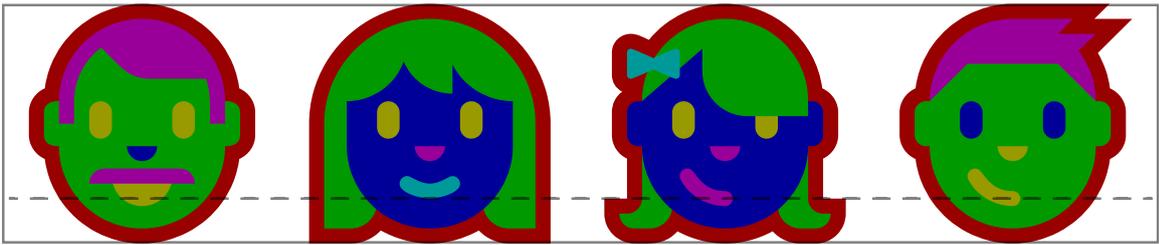


Figure 4.2 Emoji snippets overlaid.

When a font supports it, a sequence of emoji can be turned into a more compact representation. In figure 4.3 we see how skin tones are applied in such combinations. Figure 4.4 shows the small snippets.

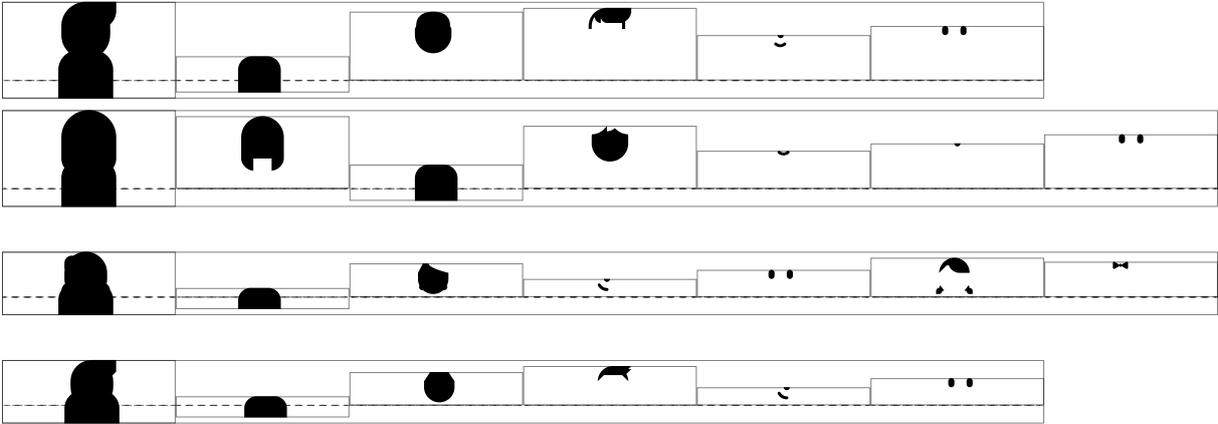
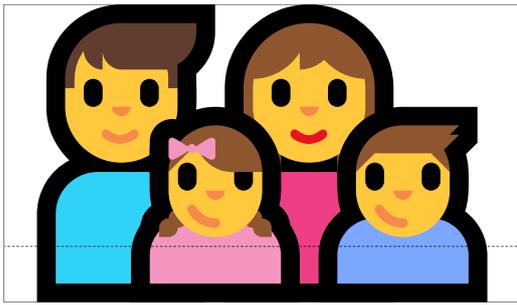
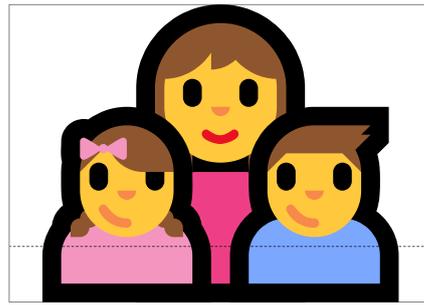


Figure 4.4 Emoji glyphs.

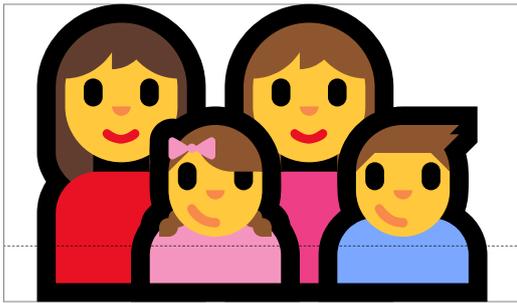
When we have to choose a font we need to take the following criteria into account:



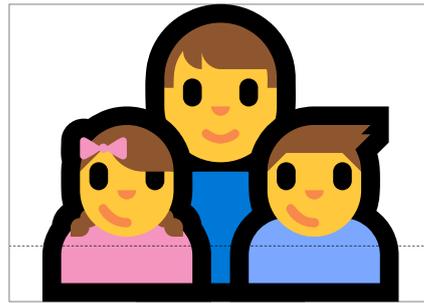
family man woman girl boy



family woman girl boy



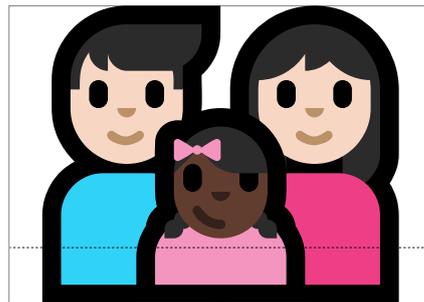
family woman woman girl boy



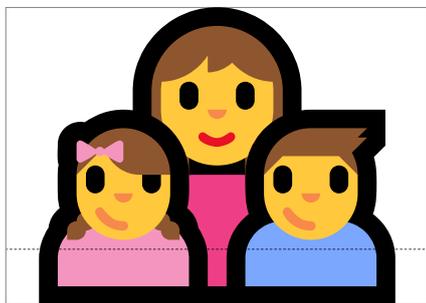
family man girl boy



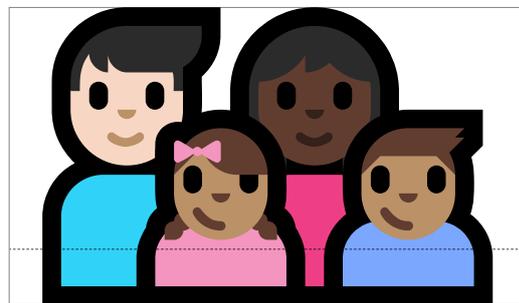
family man dark skin
tone woman girl baby



family man light skin
tone woman light skin
tone girl dark skin
tone



family woman girl boy



family man light skin tone woman
dark skin tone girl medium skin
tone boy medium skin tone

Figure 4.3 Emoji families and such with skin tones.

- What is the quality of the shapes? For sure, outlines are best if you want to scale too.
- How efficient is a shape constructed. In that respect a bitmap or SVG image is just one entity.
- How well can (semi) arbitrary combinations of emoji be provided. Here the glyph approach wins.
- Are all skin colors for all human relates shapes supported? Actually it opens the possibility for racist fonts.
- Are all reasonable combinations of persons supported? It looks like (depending on time and version) kissing men or women can be missing, maybe because of social political reasons.
- Are black and white shapes provided alongside color shapes.

Maybe an SVG or bitmap image can have a lot of detail compared to a stacked glyph but, when we're just using pictographic representations, the later is the best choice.

When I was playing a bit with the skin tone variants and other combinations that should result in some composed shape, I used the UNICODE test files but I got the impression that there are some errors in the test suite, for instance with respect to modifiers. Maybe the fonts are just doing the wrong thing or maybe some implement these sequences a bit inconsistent. This will probably improve over time but the question is if we should intercept issues. I'm not in favour of this because it adds more and more fuzzy code that not only wastes cycles (energy) but is also a conceptual horror. So, when testing, imperfection has to be accepted for now. This is no big deal as until now no one ever asked for emoji support in CONTEXT.

When no combined shape is provided, the original sequence shows up. A side effect can be that zero-width- joiners and modifiers become visible. This depends on the fonts. Users probably don't care that much about it. Now how do we suppose that users enter these emoji (sequences) in a document source? One can imagine a pop up in the editor but TEXies are often using commands for special cases.

We already showed some combined shapes. The reader might appreciate the outcome but getting there from the input takes a bit of work. For instance a two person `man light skin tone woman medium skin tone girl medium-light skin tone baby medium-light skin tone` involves this:

```
font          49: seguiemj.ttf @ 12.0pt
features      [basic: ccmp=yes, dist=yes, mark=yes, mkmk=yes,
              script=dflt, tlig=yes, trep=yes] [extra: analyze=yes,
              autolanguage=position, autoscript=position, checkmarks=yes,
              colr=yes, curs=yes, devanagari=yes, dummies=yes,
              extensions=yes, extrafeatures=yes, extraprivates=yes,
              kern=yes, liga=yes, mathkerns=yes, mathrules=yes,
              mode=node, spacekern=yes]
```

step 1  [+TLT] U+1F468: U+1F3FB:
 U+200D: ↑ U+1F469: U+1F3FD: U+200D: ↑ U+1F467:
 U+1F3FC: U+200D: ↑ U+1F476: U+1F3FC:

feature 'ccmp', type 'gsub_ligature', lookup 's_s_0',
 replacing U+1F468 upto U+1F3FB by ligature U+F01C5
 case 2

feature 'ccmp', type 'gsub_ligature', lookup 's_s_0',
 replacing U+1F469 upto U+1F3FD by ligature U+F01D2
 case 2

feature 'ccmp', type 'gsub_ligature', lookup 's_s_0',
 replacing U+1F467 upto U+1F3FC by ligature U+F01BC
 case 2

feature 'ccmp', type 'gsub_ligature', lookup 's_s_0',
 replacing U+1F476 upto U+1F3FC by ligature U+F021A
 case 2

step 2  [+TLT] U+F01C5: U+200D: ↑ U+F01D2:
 U+200D: ↑ U+F01BC: U+200D: ↑ U+F021A:

feature 'ccmp', type 'gsub_contextchain', chain lookup
 's_s_2', replacing single U+F01C5 by U+F15C4

step 3  [+TLT] U+F15C4: U+200D: ↑ U+F01D2: U+200D: ↑
 U+F01BC: U+200D: ↑ U+F021A:

feature 'ccmp', type 'gsub_contextchain', chain lookup
 's_s_3', index 1, replacing character U+200D upto
 U+F01D2 by ligature U+F15EC case 4

step 4  [+TLT] U+F15C4: U+F15EC: U+200D: ↑ U+F01BC:
 U+200D: ↑ U+F021A:

feature 'ccmp', type 'gsub_contextchain', chain lookup
 's_s_5', index 1, replacing character U+200D upto
 U+F01BC by ligature U+F15B9 case 4

step 5  [+TLT] U+F15C4: U+F15EC: U+F15B9: U+200D: ↑
 U+F021A:

feature 'ccmp', type 'gsub_contextchain', chain lookup
 's_s_6', index 1, replacing character U+200D upto
 U+F021A by ligature U+F1607 case 4

step 6  [+TLT] U+F15C4: U+F15EC: U+F15B9: U+F1607:

feature 'ccmp', type 'gsub_contextchain', chain lookup
 's_s_7', replacing single U+F15B9 by U+F15AC

step 7  [+TLT] U+F15C4: U+F15EC: U+F15AC: U+F1607:

feature 'dist', type 'gpos_single', lookup 'p_s_0',
shifting single U+F15C4 by single xy (1.5pt,0pt) and
wh (0pt,0pt)

step 8  [+TLT] U+F15C4: U+F15EC: U+F15AC: U+F1607: 

feature 'dist', type 'gpos_single', lookup 'p_s_1',
shifting single U+F15EC by single xy (0pt,0pt) and wh
(1.5pt,0pt)

step 9  [+TLT] U+F15C4: U+F15EC: [kern] U+F15AC:
U+F1607: 

feature 'dist', type 'gpos_contextchain', chain lookup
'p_s_2', shifting single U+F15C4 by single (0pt,0pt)
and correction (1.5pt,0pt)

step 10  [+TLT] [kern] U+F15C4: U+F15EC: [kern] U+F15AC:
U+F1607: 

feature 'dist', type 'gpos_contextchain', chain lookup
'p_s_3', shifting single U+F15EC by single
(5.71289pt,0pt) and correction (0pt,0pt)

feature 'dist', type 'gpos_contextchain', chain lookup
'p_s_3', shifting single U+F15EC by single (0pt,0pt)
and correction (9.92578pt,0pt)

step 11  [+TLT] [kern] U+F15C4: [kern] U+F15EC: [kern]
U+F15AC: U+F1607: 

feature 'dist', type 'gpos_contextchain', chain lookup
'p_s_5', shifting single U+F15C4 by single (0pt,0pt)
and correction (-5.71289pt,0pt)

step 12  [+TLT] [kern] U+F15C4: [kern] [kern] U+F15EC:
[kern] U+F15AC: U+F1607: 

feature 'mark', type 'gpos_mark2base', lookup 'p_s_27',
bound 1, anchoring mark U+F15AC to basechar U+F15EC
=> (7.59375pt,0pt)

step 13  [+TLT] [kern] U+F15C4: [kern] [kern] U+F15EC:
[kern] U+F15AC: U+F1607: 

feature 'mark', type 'gpos_mark2base', lookup 'p_s_28',
bound 2, anchoring mark U+F1607 to basechar U+F15EC
=> (0.01172pt,0pt)

result  [+TLT] [kern] U+F15C4: [kern] [kern] U+F15EC:
[kern] U+F15AC: U+F1607: 

A black and white example is the following **family woman girl**:

```

font      52: seguiemj.ttf @ 12.0pt

features  [basic: ccmp=yes, dist=yes, mark=yes, mkmk=yes,
          script=dflt, tlig=yes, trep=yes] [extra: analyze=yes,
          autolanguage=position, autoscript=position, checkmarks=yes,
          curs=yes, devanagari=yes, dummies=yes, extensions=yes,
          extrafeatures=yes, extraprivates=yes, kern=yes,
          liga=yes, mathkerns=yes, mathrules=yes, mode=node,
          spacekern=yes]

step 1    👤👤 [+TLT] U+1F469:👤 U+200D: † U+1F467:👤

          feature 'ccmp', type 'gsub_contextchain', chain lookup
            's_s_4', replacing single U+1F469 by U+F15E2

step 2    👤👤 [+TLT] U+F15E2:👤 U+200D: † U+1F467:👤

          feature 'ccmp', type 'gsub_contextchain', chain lookup
            's_s_5', index 1, replacing character U+200D upto
            U+1F467 by ligature U+F15B0 case 4

step 3    👤 [+TLT] U+F15E2:👤 U+F15B0: 👤

          feature 'dist', type 'gpos_single', lookup 'p_s_1',
            shifting single U+F15E2 by single xy (0pt,0pt) and wh
            (1.5pt,0pt)

step 4    👤 [+TLT] U+F15E2:👤 [kern] U+F15B0: 👤

          feature 'dist', type 'gpos_contextchain', chain lookup
            'p_s_4', shifting single U+F15E2 by single
            (1.5pt,0pt) and correction (0pt,0pt)

          feature 'dist', type 'gpos_contextchain', chain lookup
            'p_s_4', shifting single U+F15E2 by single (0pt,0pt)
            and correction (5.71289pt,0pt)

step 5    👤👤 [+TLT] [kern] U+F15E2:👤 [kern] U+F15B0: 👤

          feature 'mark', type 'gpos_mark2base', lookup 'p_s_28',
            bound 1, anchoring mark U+F15B0 to basechar U+F15E2
            => (0.01172pt,0pt)

result    👤 [+TLT] [kern] U+F15E2:👤 [kern] U+F15B0: 👤

```

I will not show all emoji, just the subset that contains the word **woman** in the description. As you can see the persons in the sequences are separated by a zero-width-joiner. There are some curious ones, for instance a **woman wearing turban** which in terms of UNICODE input is a female combine with a turban wearing man becomes a beardless woman wearing a turban. Woman vampires and zombies are not supported so these are male properties.

		woman getting haircut
		woman getting massage
		woman golfing
		woman guard
		woman hat
		woman health worker
		woman in lotus position
		woman in steamy room
		woman judge
		woman juggling
		woman lifting weights
		woman mage
		woman mechanic
		woman mountain biking
		woman office worker
		woman pilot
		woman playing handball
		woman playing water polo
		woman police officer
		woman pouting
		woman raising hand
		woman rowing boat
		woman running
		woman sandal
		woman scientist
		woman shrugging
		woman singer
		woman student
		woman surfing
		woman swimming
		woman teacher
		woman technologist
		woman tipping hand
		woman vampire
		woman walking
		woman wearing turban
		woman with headscarf
		woman zombie

So what if you don't like these colors? Because we're dealing with T_EX you can assume that if there is some way around the fixed color sets, then it will be provided. So, when you use C_ON_TE_XT, here is away to overload them:

```

\definecolor[emoji-red] [r=.4]
\definecolor[emoji-green] [g=.4]
\definecolor[emoji-blue] [b=.4]
\definecolor[emoji-yellow] [r=.4,g=.4]
\definecolor[emoji-gray] [s=1,t=.5,a=1]

```

```

\definefontcolorpalette
  [emoji-s]
  [black,emoji-gray]

\definefontcolorpalette
  [emoji-r]
  [emoji-red,emoji-gray]

\definefontcolorpalette
  [emoji-g]
  [emoji-green,emoji-gray]

\definefontcolorpalette
  [emoji-b]
  [emoji-blue,emoji-gray]

\definefontcolorpalette
  [emoji-y]
  [emoji-yellow,emoji-gray]

\definefontfeature[seguiemj-s] [ccmp=yes,dist=yes,colr=emoji-s]
\definefontfeature[seguiemj-r] [ccmp=yes,dist=yes,colr=emoji-r]
\definefontfeature[seguiemj-g] [ccmp=yes,dist=yes,colr=emoji-g]
\definefontfeature[seguiemj-b] [ccmp=yes,dist=yes,colr=emoji-b]
\definefontfeature[seguiemj-y] [ccmp=yes,dist=yes,colr=emoji-y]

\definefont [MyEmojiS] [seguiemj*seguiemj-s]
\definefont [MyEmojiR] [seguiemj*seguiemj-r]
\definefont [MyEmojiG] [seguiemj*seguiemj-g]
\definefont [MyEmojiB] [seguiemj*seguiemj-b]
\definefont [MyEmojiY] [seguiemj*seguiemj-y]

```

In figure 4.5 we see how this is applied. You can provide as many colors as needed but when you don't provide enough the last one is used. This way we get the overlaid transparent colors in the examples. By using transparency we don't obscure shapes.

The emojiopedia mentions "Asked about the design, MICROSOFT told emojiopedia that one of the reasons for the thick stroke was to allow each emoji to be easily read on any background color." The first glyph in the stack seems to do the trick, so just make sure that it doesn't become white. And, before I read that remark, while preparing a presentation with a colored background, I had already noticed that using a background was no problem. This font definitely sets the standard.

How do we know what colors are used? The next table shows the first color palette of `seguiemj`. There are quite some colors so defining your own definitely involved some studying.

1	1	2	3	4	5	6	7	8	9	10	11	12
	13	14	15	16	17	18	19	20	21	22	23	24



Figure 4.5 Overloading colors by plugging in a sequence of alternate colors.

25	26	27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72
73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132
133	134	135	136	137	138	139	140	141	142	143	144
145	146	147	148	149	150	151	152	153	154	155	156
157	158	159	160	161	162	163	164	165	166	167	168
169	170	171	172	173	174	175					

Normally special symbols are accessed in CONTEXT with the `symbol` command where symbols are organized in symbol sets. This is a rather old mechanism and dates from the time that fonts were limited in coverage and symbols were collected in special fonts. The emoji are accessed by their own command: `\emoji`. The font used has the font synonym `emoji` so you need to set that one first:

```
\definefontsynonym[emoji][seguiemj*seguiemj-cl]
```

Here is an example:

```
\emoji{woman light skin tone}\quad
\emoji{woman scientist}\quad
{\bfd bigger \emoji{man health worker}}
```

or typeset:   **bigger** 

The emoji symbol scales with the normal running font. When you ask for a family with skin toned members the lookup can result in another match (or no match) because one never knows to what extent a font supports it.

```
\expandedemoji the sequence constructed from the given string
\resolvedemoji a protected sequence constructed from the given string
\checkedemoji an typeset sequence with unresolved modifiers and joiners removed
\emoji a typeset resolved sequence using the emoji font synonym
\robustemoji a typeset checked sequence using the emoji font synonym
```

In case you wonder how some of the details above were typeset, there is a module `fonts-emoji` that provides some helpers for introspection.

```
\ShowEmoji show all the emoji in the current font
\ShowEmojiSnippets show the snippets of a given emoji
\ShowEmojiSnippetsOverlay show the overlaid snippets of a given emoji
\ShowEmojiGlyphs show the snippets of a typeset emoji
\ShowEmojiPalettes show the color pallets in the current font
```

Examples of usage are:

```
\ShowEmojiSnippets[family man woman girl boy]
\ShowEmojiGlyphs  [family man woman baby girl]
\ShowEmoji        [^man]
\ShowEmoji
\ShowEmojiPalettes
\ShowEmojiPalettes[1]
```

A good source of information about emoji is the mentioned emojipedia.org website. There you find not only details about all these symbols but also has some history. It compares updates in fonts too. It mentions for instance that in the creative update of Windows 10, some persons grew beards in the `seguiemj` font and others lost an eye. Now, if you look at the snippets shown before, you can wonder if that eye is really gone. Maybe the color is wrong or the order of stacking is not right. I decided not to waste time looking into that.

Another quote: “Support for color emoji presentation on MS WINDOWS is limited. Many applications on MS WINDOWS display emojis with a black and white text presentation instead of their color version.” Well, we can do better with \TeX , but as usual not that many people really cares about that. But it’s fun anyway.

We end with a warning. When you use ‘ligatures’ like this, you really need to check the outcome. For instance, when MICROSOFT updated the font end 2017, same gender couples got different hair style for the individuals so that one can still distinguish them. However, kissing couples and couples in love (indicated by a heart) seem to be removed. Who knows how and when politics creep into fonts: is public mixed couple kissing permitted, do we support families with any mix of gender, is associating pink with girls okay or not, how do we distinguish male and female anyway? In figure 4.6 we see the same combination twice, the early 2017 rendering versus the late 2017 rendering. Can you notice the differences?

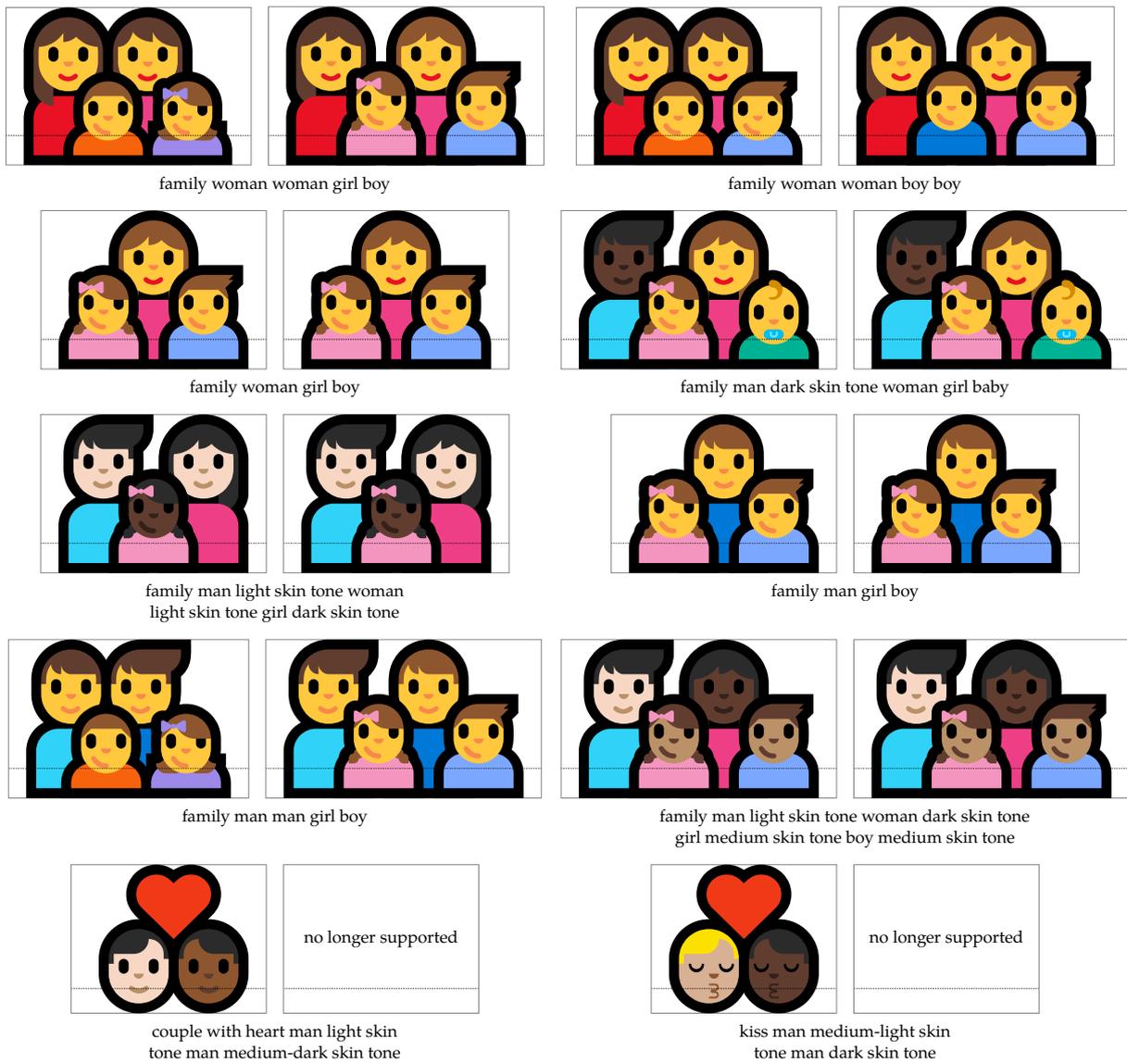


Figure 4.6 Incompatible updates.

5 Children of T_EX

First published in user group magazines.

6 Performance

6.1 Introduction

This chapter is about performance. Although it concerns L^AT_EX this text is only meant for C^ON^TE^XT users. This is not because they ever complain about performance, on the contrary, I never received a complain from them. No, it's because it gives them some ammunition against the occasionally occurring nagging about the speed of L^AT_EX (somewhere on the web or at some meeting). My experience is that in most such cases those complaining have no clue what they're talking about, so effectively we could just ignore them, but let's, for the sake of our users, waste some words on the issue.

6.2 What performance

So what exactly does performance refer to? If you use C^ON^TE^XT there are probably only two things that matter:

- How long does one run take.
- How many runs do I need.

Processing speed is reported at the end of a run in terms of seconds spent on the run, but also in pages per second. The runtime is made up out of three components:

- start-up time
- processing pages
- finishing the document

The startup time is rather constant. Let's take my 2013 Dell Precision with i7-3840QM as reference. A simple

```
\starttext  
\stoptext
```

document reports 0.4 seconds but as we wrap the run in an `mtxrun` management run we have an additional 0.3 overhead (auxiliary file handling, PDF viewer management, etc). This includes loading the Latin Modern font. With L^UA_{JIT}T_EX these times are below 0.3 and 0.2 seconds. It might look like much overhead but in an edit-preview runs it feels snappy. One can try this:

```
\stoptext
```

which bring down the time to about 0.2 seconds for both engines but as it doesn't do anything useful that is is no practice.

Finishing a document is not that demanding because most gets flushed as we go. The more (large) fonts we use, the longer it takes to finish a document but on the average that time is not worth noticing. The main runtime contribution comes from processing the pages.

Okay, this is not always true. For instance, if we process a 400 page book from 2500 small XML files with multiple graphics per page, there is a little overhead in loading the files and constructing the XML tree as well as in inserting the graphics but in such cases one expects a few seconds more runtime. The METAFUN manual has some 450 pages with over 2500 runtime generated METAPOST graphics. It has color, uses quite some fonts, has lots of font switches (verbatim too) but still one run takes only 18 seconds in stock L^AT_EX and less than 15 seconds with L^AJIT_EX. Keep these numbers in mind if a non-CON_TE_XT users barks against the performance tree that his few page mediocre document takes 10 seconds to compile: the content, styling, quality of macros and whatever one can come up with all plays a role. Personally I find any rate between 10 and 30 pages per second acceptable, and if I get the lower rate then I normally know pretty well that the job is demanding in all kind of aspects.

Over time the CON_TE_XT–L^AT_EX combination, in spite of the fact that more functionality has been added, has not become slower. In fact, some subsystems have been sped up. For instance font handling is very sensitive for adding functionality. However, each version so far performed a bit better. Whenever some neat new trickery was added, at the same time improvements were made thanks to more insight in the matter. In practice we're not talking of changes in speed by large factors but more by small percentages. I'm pretty sure that most CON_TE_XT users never noticed. Recently a 15–30% speed up (in font handling) was realized (for more complex fonts) but only when you use such complex fonts and pages full of text you will see a positive impact on the whole run.

There is one important factor I didn't mention yet: the efficiency of the console. You can best check that by making a format (`context --make en`). When that is done by piping the messages to a file, it takes 3.2 seconds on my laptop and about the same when done from the editor (SciTE), maybe because the L^AT_EX run and the log pane run on a different thread. When I use the standard console it takes 3.8 seconds in Windows 10 Creative update (in older versions it took 4.3 and slightly less when using a console wrapper). The powershell takes 3.2 seconds which is the same as piping to a file. Interesting is that in Bash on Windows it takes 2.8 seconds and 2.6 seconds when piped to a file. Normal runs are somewhat slower, but it looks like the 64 bit Linux binary is somewhat faster than the 64 bit mingw version.⁷ Anyway, it demonstrates that when someone yells a number you need to ask what the conditions where.

At a CON_TE_XT meeting there has been a presentation about possible speed-up of a run for instance by using a separate syntax checker to prevent a useless run. However, the use case concerned a document that took a minute on the machine used, while the same document took a few seconds on mine. At the same meeting we also did a comparison of speed for a I^AT_EX run using P_DF_TE_X and the same document migrated to CON_TE_XT M_KI_V using L^AT_EX (Harald Königs XML torture and compatibility test). Contrary to what one might expect, the CON_TE_XT run was significantly faster; the resulting document was a few gigabytes in size.

⁷ Long ago we found that L^AT_EX is very sensitive to for instance the CPU cache so maybe there are some differences due to optimization flags and/or the fact that bash runs in one thread and all file IO in the main windows instance. Who knows.

6.3 Bottlenecks

I will discuss a few potential bottlenecks next. A complex integrated system like `CONTEXT` has lots of components and some can be quite demanding. However, when something is not used, it has no (or hardly any) impact on performance. Even when we spend a lot of time in `LUA` that is not the reason for a slow-down. Sometimes using `LUA` results in a speedup, sometimes it doesn't matter. Complex mechanisms like natural tables for instance will not suddenly become less complex. So, let's focus on the "aspects" that come up in those complaints: fonts and `LUA`. Because I only use `CONTEXT` and occasionally test with the plain `TEX` version that we provide, I will not explore the potential impact of using truckloads of packages, styles and such, which I'm sure of plays a role, but one neglected in the discussion.

Fonts

According to the principles of `LUATEX` we process (`OPENTYPE`) fonts using `LUA`. That way we have complete control over any aspect of font handling, and can, as to be expected in `TEX` systems, provide users what they need, now and in the future. In fact, if we didn't had that freedom in `CONTEXT` I'd probably already quit using `TEX` a decade ago and found myself some other (programming) niche.

After a font is loaded, part of the data gets passed to the `TEX` engine so that it can do its work. For instance, in order to be able to typeset a paragraph, `TEX` needs to know the dimensions of glyphs. Once a font has been loaded (that is, the binary blob) the next time it's fetched from a cache. Initial loading (and preparation) takes some time, depending on the complexity or size of the font. Loading from cache is close to instantaneous. After loading the dimensions are passed to `TEX` but all data remains accessible for any desired usage. The `OPENTYPE` feature processor for instance uses that data and `CONTEXT` for sure needs that data (fast accessible) for different purposes too.

When a font is used in so called base mode, we let `TEX` do the ligaturing and kerning. This is possible with simple fonts and features. If you have a critical workflow you might enable base mode, which can be done per font instance. Processing in node mode takes some time but how much depends on the font and script. Normally there is no difference between `CONTEXT` and generic usage. In `CONTEXT` we also have dynamic features, and the impact on performance depends on usage. In addition to base and node we also have plug mode but that is only used for testing and therefore not advertised.

Every `\hbox` and every paragraph goes through the font handler. Because we support mixed modes, some analysis takes place, and because we do more in `CONTEXT`, the generic analyzer is more light weight, which again can mean that a generic run is not slower than a similar `CONTEXT` one.

Interesting is that added functionality for variable and/or color fonts had no impact on performance. Runtime added user features can have some impact but when defined well it can be neglected. I bet that when you add additional node list handling yourself, its impact on performance is larger. But in the end what counts is that the job gets done and the more you demand the higher the price you pay.

LUA

The second possible bottleneck when using L^AT_EX can be in using LUA code. However, using that as argument for slow runs is laughable. For instance C^ON^TE^XT MKIV can easily spend half its time in LUA and that is not making it any slower than MKII using P^DF^TE^X doing equally complex things. For instance the embedded M^ET^AP^OST library makes MKIV way faster than MKII, and the built-in X^ML processing capabilities in MKIV can easily beat MKII X^ML handling, apart from the fact that it can do more, like filtering by path and expression. In fact, files that take, say, half a minute in MKIV, could as well have taken 15 minutes or more in MKII (and imagine multiple runs then).

So, for C^ON^TE^XT using LUA to achieve its objectives is mandate. The combination of T_EX, M^ET^AP^OST and LUA is pretty powerful! Each of these components is really fast. If T_EX is your bottleneck, review your macros! When LUA seems to be the bad, go over your code and make it better. Much of the LUA code I see flying around doesn't look that efficient, which is okay because the interpreter is really fast, but don't blame LUA beforehand, blame your coding (style) first. When M^ET^AP^OST is the bottleneck, well, sometimes not much can be done about it, but when you know that language well enough you can often make it perform better.

For the record: every additional mechanism that kicks in, like character spacing (the ugly one), case treatments, special word and line trickery, marginal stuff, graphics, line numbering, underlining, referencing, and a few dozen more will add a bit to the processing time. In that case, in C^ON^TE^XT, the font related runtime gets pretty well obscured by other things happening, just that you know.

6.4 Some timing

Next I will show some timings related to fonts. For this I use stock L^AT_EX (second column) as well as L^AU^AJIT_TE_X (last column) which of course performs much better. The timings are given in 3 decimals but often (within a set of runs) and as the system load is normally consistent in a set of test runs the last two decimals only matter in relative comparison. So, for comparing runs over time round to the first decimal. Let's start with loading a bodyfont. This happens once per document and normally one has only one bodyfont active. Loading involves definitions as well as setting up math so a couple of fonts are actually loaded, even if they're not used later on. A setup normally involves a serif, sans, mono, and math setup (in C^ON^TE^XT).⁸

bodyfont

modern	0.023	0.019
pagella	0.127	0.079
termes	0.128	0.087
cambria	0.180	0.123
dejavu	0.140	0.092

⁸ The timing for Latin Modern is so low because that font is loaded already.

ebgaramond	0.142	0.093
lucidaot	0.146	0.120

There is a bit difference between the font sets but a safe average is 150 milli seconds and this is rather constant over runs.

An actual font switch can result in loading a font but this is a one time overhead. Loading four variants (regular, bold, italic and bold italic) roughly takes the following time:

bodyfont switch and 4 style changes (first time)

modern	0.028	0.028
pagella	0.035	0.031
termes	0.036	0.069
cambria	0.052	0.047
dejavu	0.091	0.069
ebgaramond	0.022	0.016
lucidaot	0.017	0.031

Using them again later on takes no time:

bodyfont switch and 4 style changes (follow up)

modern	0.000	0.000
pagella	0.001	0.000
termes	0.000	0.001
cambria	0.000	0.000
dejavu	0.001	0.000
ebgaramond	0.000	0.000
lucidaot	0.000	0.000

Before we start timing the font handler, first a few baseline benchmarks are shown. When no font is applied and nothing else is done with the node list we get:

100 hboxes with 4 texts and no font handling

baseline	0.142	2.343
----------	-------	-------

A simple monospaced, no features applied, run takes a bit more:

100 hboxes with 4 texts and no features

baseline	0.275	0.220
----------	-------	-------

Now we show a one font typesetting run. As the two benchmarks before, we just typeset a text in a `\hbox`, so no par builder interference happens. We use the `sapolsky` sample text and typeset it 100 times 4 (either of not with font switches).

100 hboxes with 4 texts using one font

modern	0.933	0.591
pagella	1.027	0.660
termes	1.032	0.604
cambria	1.483	0.862
dejavu	1.009	0.581
ebgaramond	3.240	1.774
lucidaot	0.699	0.444

Much more runtime is needed when we typeset with four font switches. The garamond is most demanding. Actually we're not doing 4 fonts there because it has no bold, so the numbers are a bit lower than expected for this example. One reason for it being demanding is that it has lots of (contextual) lookups. The only comment I can make about that is that it also depends on the strategies of the font designer. Combining lookups saves space and time so complexity of a font is not always a good predictor for performance hits.

If we typeset paragraphs we get this:

100 times 4 texts on pages

modern	1.377	0.904
pagella	1.523	0.961
termes	1.453	0.898
cambria	1.901	1.138
dejavu	1.437	0.917
ebgaramond	3.714	2.133
lucidaot	1.117	0.767

We're talking of some 275 pages here.

100 times 4 texts on pages using 4 styles

modern	2.074	1.307
pagella	2.155	1.338
termes	2.153	1.373
cambria	3.349	2.012
dejavu	2.408	1.453
ebgaramond	4.368	2.512
lucidaot	1.682	1.056

There is of course overhead in handling paragraphs and pages:

100 paragraphs with 4 texts and no features

baseline 0.825 0.559

Before I discuss these numbers in more details two more benchmarks are shown. The next table concerns a paragraph with only a few (bold) words.

100 texts on pages with [1,2,4] bold font switches

modern	0.409	0.263
pagella	0.445	0.281
termes	0.432	0.300
Cambria	0.606	0.368
dejavu	0.465	0.295
ebgaramond	0.922	0.530
lucidaot	0.345	0.220

The following table has paragraphs with a few mono spaced words typeset using `\type`.

100 texts on pages with [1,2,4] word verbatim switches

modern	0.380	0.255
pagella	0.396	0.266
termes	0.384	0.278
Cambria	0.535	0.355
dejavu	0.366	0.247
ebgaramond	0.939	0.533
lucidaot	0.322	0.216

When a node list (hbox or paragraph) is processed, each glyph is looked at. One important property of `LUATEX` (compared to `PDFTEX`) is that it hyphenates the whole text, not only the most feasible spots. For the `sapolsky` snippet this results in 200 potential breakpoints, registered in an equal number of discretionary nodes. The snippet has 688 characters grouped into 125 words and because it's an English quote we're not hampered with composed characters or complex script handling. And, when we mention 100 runs then we actually mean 400 ones when font switching and bodyfonts are compared

Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. Robert M. Sapolsky

automated flows batch mode is your friend. The rate is not much worse for a document in a language with a bit more complex character handling, take accents or ligatures. Of course PDFTEX is faster on such a dumb document but kick in some more functionality and the advantage quickly disappears. So, if someone complains that LUALATEX needs 10 or more seconds for a simple few page document . . . you can bet that when the fonts are seen as reason, that the setup is pretty bad. Personally I'd not waste time on such a complaint.

6.5 Valid questions

Here are some reasonable questions that you can ask when someone complains to you about the slowness of LUALATEX:

What engines do you compare?

If you come from PDFTEX you come from an 8 bit world: input and font handling are based on bytes and hyphenation is integrated into the par builder. If you use UTF-8 in PDFTEX, the input is decoded by TEX macros which carries a speed penalty. Because in the wide engines macro names can also be UTF sequences, construction of macro names is less efficient too.

When you try to use wide fonts, again there is a penalty. Now, if you use XeTEX or LUALATEX your input is UTF-8 which becomes something 32 bit internally. Fonts are wide so more resources are needed, apart from these fonts being larger and in need of more processing due to feature handling. Where XeTEX uses a library, LUALATEX uses its own handler. Does that have a consequence for performance? Yes and no. First of all it depends on how much time is spent on fonts at all, but even then the difference is not that large. Sometimes XeTEX wins, sometimes LUALATEX. One thing is clear: LUALATEX is more flexible as we can roll out our own solutions and therefore do more advanced font magic. For CONTEXTEXT it doesn't matter as we use LUALATEX exclusively and rely on the flexible font handler, also for future extensions. If really needed you can kick in a library based handler but it's (currently) not distributed as we loose other functionality which in turn would result in complaints about that fact (apart from conflicting with the strive for independence).

There is no doubt that PDFTEX is faster but for CONTEXTEXT it's an obsolete engine. The hard coded solutions engine XeTEX is also not feasible for CONTEXTEXT either. So, in practice CONTEXTEXT users have no choice: LUALATEX is used, but users of other macro packages can use the alternatives if they are not satisfied with performance. The fact that CONTEXTEXT users don't complain about speed is a clear signal that this is no issue. And, if you want more speed you can use LUALAJITTEX.⁹ In the last section the different engines will be compared in more detail.

⁹ In plug mode we can actually test a library and experiments have shown that performance on the average is much worse but it can be a bit better for complex scripts, although a gain gets unnoticed in normal documents. So, one can decide to use a library but at the cost of much other functionality that CONTEXTEXT offers, so we don't support it.

Just that you know, when we do the four switches example in plain TEX on my laptop I get a rate of 40 pages per second, and for one font 180 pages per second. There is of course a bit more going on in $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$ in page building and so, but the difference between plain and $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$ is not that large.

What macro package is used?

If the answer is that when plain TEX is used, a follow up question is: what variant? The $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$ distribution ships with `luatex-plain` and that is our benchmark. If there really is a bottleneck it is worth exploring. But keep in mind that in order to be plain, not that much can be done. The $\text{L}\text{U}\text{A}\text{T}\text{E}\text{X}$ part is just an example of an implementation. We already discussed $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$, and for $\text{L}\text{A}\text{T}\text{E}\text{X}$ I don't want to speculate where performance hits might come from. When we're talking fonts, $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$ can actually be a bit slower than the generic (or $\text{L}\text{A}\text{T}\text{E}\text{X}$) variant because we can kick in more functionality. Also, when you compare macro packages, keep in mind that when node list processing code is added in that package the impact depends on interaction with other functionality and depends on the efficiency of the code. You can't compare mechanisms or draw general conclusions when you don't know what else is done!

What do you load?

Most $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$ modules are small and load fast. Of course there can be exceptions when we rely on third party code; for instance loading `tikz` takes a bit of time. It makes no sense to look for ways to speed that system up because it is maintained elsewhere. There can probably be gained a bit but again, no user complained so far.

If $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$ is not used, one probably also uses a large TEX installation. File lookup in $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$ is done differently and can be faster. Even loading can be more efficient in $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$, but it's hard to generalize that conclusion. If one complains about loading fonts being an issue, just try to measure how much time is spent on loading other code.

Did you patch macros?

Not everyone is a TEX pert. So, coming up with macros that are expanded many times and/or have inefficient user interfacing can have some impact. If someone complains about one subsystem being slow, then honestly demands to complain about other subsystems as well. You get what you ask for.

How efficient is the code that you use?

Writing super efficient code only makes sense when it's used frequently. In $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$ most code is reasonable efficient. It can be that in one document fonts are responsible for most runtime, but in another document table construction can be more demanding while yet another document puts some stress on interactive features. When `hz` or `protrusion` is enabled then you run substantially slower anyway so when you are willing to sacrifice 10% or more runtime don't complain about other components. The same is true for enabling `SYNCTEX`: if you are willing to add more than 10% runtime for that, don't wither about the same amount for font handling.¹⁰

¹⁰ In $\text{C}\text{O}\text{N}\text{T}\text{E}\text{X}\text{T}$ we use a `SYNCTEX` alternative that is somewhat faster but it remains a fact that enabling more and more functionality will make the penalty of for instance font processing relatively small.

How efficient is the styling that you use?

Probably the most easily overseen optimization is in switching fonts and color. Although in `CONTEXT` font switching is fast, I have no clue about it in other macro packages. But in a style you can decide to use inefficient (massive) font switches. The effects can easily be tested by commenting bit and pieces. For instance sometimes you need to do a full bodyfont switch when changing a style, like assigning `\small\bf` to the `style` key in `\setuphead`, but often using e.g. `\tfd` is much more efficient and works quite as well. Just try it.

Are fonts really the bottleneck?

We already mentioned that one can look in the wrong direction. Maybe once someone is convinced that fonts are the culprit, it gets hard to look at the real issue. If a similar job in different macro packages has a significant different runtime one can wonder what happens indeed.

It is good to keep in mind that the amount of text is often not as large as you think. It's easy to do a test with hundreds of paragraphs of text but in practice we have whitespace, section titles, half empty pages, floats, itemize and similar constructs, etc. Often we don't mix many fonts in the running text either. So, in the end a real document is the best test.

If you use LUA, is that code any good?

You can gain from the faster virtual machine of `LUAJITTEX`. Don't expect wonders from the jitting as that only pays off for long runs with the same code used over and over again. If the gain is high you can even wonder how well written your LUA code is anyway.

What if they don't believe you?

So, say that someone finds `LUATEX` slow, what can be done about it? Just advice him or her to stick to tool used previously. Then, if arguments come that one also wants to use UTF-8, `OPENTYPE` fonts, a bit of `METAPOST`, and is looking forward to using LUA runtime, the only answer is: take it or leave it. You pay a price for progress, but if you do your job well, the price is not that large. Tell them to spend time on learning and maybe adapting and bark against their own tree before barking against those who took that step a decade ago. Most `CONTEXT` users took that step and someone still using `LUATEX` after a decade can't be that stupid. It's always best to first wonder what one actually asks from `LUATEX`, and if the benefit of having LUA on board has an advantage. If not, one can just use another engine.

Also think of this. When a job is slow, for me it's no problem to identify where the problem is. The question then is: can something be done about it? Well, I happily keep the answer for myself. After all, some people always need room to complain, maybe if only to hide their ignorance or incompetence. Who knows.

6.6 Comparing engines

The next comparison is to be taken with a grain of salt and concerns the state of affairs mid 2017. First of all, you cannot really compare `MKII` with `MKIV`: the later has more

functionality (or a more advanced implementation of functionality). And as mentioned you can also not really compare PDFTEX and the wide engines. Anyway, here are some (useless) tests. First a bunch of loads. Keep in mind that different engines also deal differently with reading files. For instance MKIV uses L^AT_EX callbacks to normalize the input and has its own readers. There is a bit more overhead in starting up a L^AT_EX run and some functionality is enabled that is not present in MKII. The format is also larger, if only because we preload a lot of useful font, character and script related data.

```
\starttext
  \dorecurse {#1} {
    \input knuth
    \par
  }
\stoptext
```

When looking at the numbers one should realize that the times include startup and job management by the runner scripts. We also run in batchmode to avoid logging to influence runtime. The average is calculated from 5 runs.

engine	50	500	2500
pdftex	0.43	0.77	2.33
xetex	0.85	2.66	10.79
luatex	0.94	2.50	9.44
luajittex	0.68	1.69	6.34

The second example does a few switches in a paragraph:

```
\starttext
  \dorecurse {#1} {
    \tf \input knuth
    \bf \input knuth
    \it \input knuth
    \bs \input knuth
    \par
  }
\stoptext
```

engine	50	500	2500
pdftex	0.58	2.10	8.97
xetex	1.47	8.66	42.50
luatex	1.59	8.26	38.11
luajittex	1.12	5.57	25.48

The third examples does a few more, resulting in multiple subranges per style:

```
\starttext
  \dorecurse {#1} {
    \tf \input knuth \it knuth
    \bf \input knuth \bs knuth
  }
\stoptext
```

```

        \it \input knuth \tf knuth
        \bs \input knuth \bf knuth
    \par
}
\stoptext

```

engine	50	500	2500
pdftex	0.59	2.20	9.52
xetex	1.49	8.88	43.85
luatex	1.64	8.91	41.26
luajittex	1.15	5.91	27.15

The last example adds some color. Enabling more functionality can have an impact on performance. In fact, as MkIV uses a lot of LUA and is also more advanced than MkII, one can expect a performance hit but in practice the opposite happens, which can also be due to some fundamental differences deep down at the macro level.

```
\setupcolors[state=start] % default in MkIV
```

```

\starttext
  \dorecurse {#1} {
    {\red \tf \input knuth \green \it knuth}
    {\red \bf \input knuth \green \bs knuth}
    {\red \it \input knuth \green \tf knuth}
    {\red \bs \input knuth \green \bf knuth}
  \par
}
\stoptext

```

engine	50	500	2500
pdftex	0.61	2.36	10.33
xetex	1.53	9.25	45.59
luatex	1.65	8.91	41.32
luajittex	1.15	5.93	27.34

In these measurements the accuracy is a few decimals but a pattern is visible. As expected PDFTEX wins on simple documents but starts losing when things get more complex. For these tests I used 64 bit binaries. A 32 bit XETEX with MkII performs the same as LUAJITEX with MkIV, but a 64 bit XETEX is actually quite a bit slower. In that case the mingw cross compiled LUATEX version does pretty well. A 64 bit PDFTEX is also slower (it looks) than a 32 bit version. So in the end, there are more factors that play a role. Choosing between LUATEX and LUAJITEX depends on how well the memory limited LUAJITEX variant can handle your documents and fonts.

Because in most of our recent styles we use OPENTYPE fonts and (structural) features as well as recent METAFUN extensions only present in MkIV we cannot compare engines using such documents. The mentioned performance of LUATEX (or LUAJITEX)

and MKIV on the METAFUN manual illustrate that in most cases this combination is a clear winner.

```
\starttext
  \dorecurse {#1} {
    \null \page
  }
\stoptext
```

This gives:

engine	50	500	2500
pdftex	0.46	1.05	3.72
xetex	0.73	1.80	6.56
luatex	0.84	1.44	4.07
luajittex	0.61	1.10	3.33

That leaves the zero run:

```
\starttext
  \dorecurse {#1} {
    % nothing
  }
\stoptext
```

This gives the following numbers. In longer runs the difference in overhead is negligible.

engine	50	500	2500
pdftex	0.36	0.36	0.36
xetex	0.57	0.57	0.59
luatex	0.74	0.74	0.74
luajittex	0.53	0.53	0.54

It will be clear that when we use different fonts the numbers will also be different. And if you use a lot of runtime METAPost graphics (for instance for backgrounds), the MKIV runs end up at the top. And when we process XML it will be clear that going back to MKII is no longer a realistic option. It must be noted that I occasionally manage to improve performance but we've now reached a state where there is not that much to gain. Some functionality is hard to compare. For instance in CONTEXT we don't use much of the PDF backend features because we implement them all in LUA. In fact, even in MKII already a done in T_EX, so in the end the speed difference there is not large and often in favour of MKIV.

For the record I mention that shipping out the about 1250 pages has some overhead too: about 2 seconds. Here L_UAJIT_TE_X is 20% more efficient which is an indication of quite some L_UA involvement. Loading the input files has an overhead of about half a second. Starting up L_UA_TE_X takes more time than P_DF_TE_X and X_E_TE_X, but that disadvantage disappears with more pages. So, in the end there are quite some factors that

blur the measurements. In practice what matters is convenience: does the runtime feel reasonable and in most cases it does.

If I would replace my laptop with a reasonable comparable alternative that one would be some 35% faster (single threads on processors don't gain much per year). I guess that this is about the same increase in performance that `CONTEXT MkIV` got in that period. I don't expect such a gain in the coming years so at some point we're stuck with what we have.

6.7 Summary

So, how “slow” is `LUATEX` really compared to the other engines? If we go back in time to when the first wide engines showed up, `OMEGA` was considered to be slow, although I never tested that myself. Then, when `XFTEX` showed up, there was not much talk about speed, just about the fact that we could use `OPENTYPE` fonts and native `UTF` input. If you look at the numbers, for sure you can say that it was much slower than `PDFTEX`. So how come that some people complain about `LUATEX` being so slow, especially when we take into account that it's not that much slower than `XFTEX`, and that `LUAJITTEX` is often faster than `XFTEX`. Also, computers have become faster. With the wide engines you get more functionality and that comes at a price. This was accepted for `XFTEX` and is also acceptable for `LUATEX`. But the price is not that high if you take into account that hardware performs better: you just need to compare `LUATEX` (and `XFTEX`) runtime with `PDFTEX` runtime 15 years ago.

As a comparison, look at games and video. Resolution became much higher as did color depth. Higher frame rates were in demand. Therefore the hardware had to become faster and it did, and as a result the user experience kept up. No user will say that a modern game is slower than an old one, because the old one does 500 frames per second compared to some 50 for the new game on the modern hardware. In a similar fashion, the demands for typesetting became higher: `UNICODE`, `OPENTYPE`, graphics, `XML`, advanced `PDF`, more complex (niche) typesetting, etc. This happened more or less in parallel with computers becoming more powerful. So, as with games, the user experience didn't degrade with demands. Comparing `LUATEX` with `PDFTEX` is like comparing a low res, low frame rate, low color game with a modern one. You need to have up to date hardware and even then, the writer of such programs need to make sure it runs efficient, simply because hardware no longer scales like it did decades ago. You need to look at the larger picture.

7 Editing

7.1 Introduction

Some users like the `synctex` feature that is built in the `TeX` engines. Personally I never use it because it doesn't work well with the kind of documents I maintain. If you have one document source, and don't shuffle around (reuse) text too much it probably works out okay but that is not our practice. Here I will describe how you can enable a more `CONTEXT` specific `synctex` support so that aware PDF viewers can bring you back to the source.

7.2 The premise

Most of the time we provide our customers with an authoring workflow consisting of:

- the typesetting engine `CONTEXT`
- the styles to generate the desired PDF files
- the text editor `SCITE`
- the `SUMATRAPDF` viewer

For the `MATHML` we advice the `MATHTYPE` editor and we provide them with a customized `MATHML` translator for the copy & paste actions. When `ASCIIMATH` is used to code math no special tools are needed.

What people operate this workflow? Sometimes it's an author, but most of the time they are editors with a background in copy-editing. We call them `XML` editors, because they are maintaining the large (sets of) `XML` documents and edit directly in the `XML` sources.

Maybe you'll ask yourself "Can they do that? Can they edit directly in the `XML` resource?" The answer is yes, because after they have hit the processing key they are rewarded with a publishable PDF document in a demanding layout.

The `XML` sources have a dual purpose. They form the basis for:

- all folio products that are generated in `XML` to `PDF` workflow(s)
- the digital web product(s)

The `XML` editors do their proofing chapter-wise. Sometimes a chapter is one big `XML` file (10.000 lines is no exception when the chapter contains hundreds of bloated `MATHML` snippets). In other projects they have to deal with chapters that are made up of hundreds (100 upto 500) of smaller `XML` files.

7.3 The problem

Let's keep it simple: there's a typo. Here's what an `XML` editor will do:

- start `SCITE`
- open a file

- correct the typo
- generate the PDF
- proof the PDF and see if his alteration has some undesired side effects like text flow of image floating

So far so good. When the editor dealing with one big XML file there's no problem. Hopefully the filename will indicate the specific chapter. He or she opens the file and searches for the typo. And then correction happens. But what if there are hundreds of small XML files. How does the editor know in which file the typo can be found?

First, let's give a few statistics based on two projects that are in a revision stage.

project	chapters	# of files	average # of lines
A	16	16	11000
B	132	16000 ¹¹	100

The XML resource passes three stages: a raw, a semi final and a final version. The raw XML version originates from a web authoring tool that is used by the author. Then the PDF is proofread and the XML editor goes to work.

workflow	# edit locations and adaptations	# runs ¹²
raw to semifinal	75	105
semifinal to final	35	55

Keep in mind that altering text may cause text to flow and images to float in a way that an XML editor will have to finetune and needs multiple runs for one correction.

Just to give an idea of the work involved. A typical semi final needs some 50 runs where each run takes 20 seconds (assuming 3 runs to get all cross referencing right). The numbers of explicit pagebreaks is about 5, and (related to formulas) explicit linebreaks around 8. It takes some 2 hours to get everything right, which includes checking in detail, fixing some things and if needed moving content a bit around.

Now we broaden the earlier question into: how can we make the work of an XML editor as easy and efficient as possible?

7.4 Enhancing efficiency

Since it is easier to proof content for folio and web via PDF documents we generate proof PDF files in which the complete content is shown. The proof can be a massive document. A normal 40 page chapter can explode to 140 pages visualizing all the content that is coded in the XML file(s).

The content in the proof is shown in an effective way and a functional order. Let's give a few examples of how we enhance the XML editors effectiveness:

¹¹ 132 chapters consisting of ±120 files.

¹² Maybe you can now see why we put quite some effort in keeping CONTEXT working at a comfortable speed.

- By default the proof PDF file is interactive which serves testing the tocs and the register.
- The web hyperlinks are active so their destination can be tested.
- The questions and their answers are displayed in each others proximity. This sounds logical but in folio they are two separate products (theory and answer books).
- Medium specific content (web or folio) is typographically highlighted. For example by colored backgrounds.
- When spelling mode is on the XML editor can easily pick out the colored misspelled words.
- Images can be active areas although this is of no interest to XML editors. Clicking the image results in opening the image file in its corresponding application for maintenance.
- For practical reasons the filenames and paths of the XML files are displayed. The filenames are active links and clicking them results in opening the destination XML file in SCITE.

Okay. The last option is a nice feature. However, the destination file is opened at the top of the file and you still have to find the typo or whatever incorrect issue you are looking for.

So a further enhancement in efficiency would be to jump to the typo's corresponding line in the XML source. This is where SYNCTEX comes into view. This feature, present in the T_EX engines, provides a way to go from PDF to source by using a secondary file with positions. Unfortunately that mechanism is hardly useable for CONTEX_T because it assumes a page and file handling model different from what we use. However, as CONTEX_T uses L_AT_EX, it can also provide it's own alternative.

7.5 What we want

The SYNCTEX method roughly works as follows. Internally T_EX constricts linked lists of glyphs, kerns, glue, boxes, rules etc. These elements are called nodes. Some nodes carry information about the file and line where they were created. In the backend this information gets somehow translated in a (sort of) verbose tree that describes the makeup in terms of boxes, glue and kerns. From that information the SYNCTEX parser library, hooked into a PDF viewer, can go back from a position on the screen to a line in a file. One would expect this to be a relative simple rectangle based model, but as far as I can see it's way more complex than that. There are some comments that CONTEX_T is not supported well because it has a layered page model, which indicates that there are some assumptions about how macro packages are supposed to work. Also the used heuristics not only involve some specific spot (location) but also involve the corners and edges. It is therefore not so much a (simple) generic system but a mechanism geared for a macro package like L_AT_EX.

Because we have a couple of users who need to edit complex sets of documents, coded in T_EX or XML, I decided to come up with a variant that doesn't use the SYNCTEX machinery

but manipulates the few `SYNCTEX` fields directly¹³ and eventually outputs a straightforward file for the editor. Of course we need to follow some rules so that the editor can deal with it. It took a bit of trial and error to get the right information in the support file needed by the viewer but we got there.

The prerequisites of a decent `CONTEXT` “click on preview and goto editor” are the following:

- It only makes sense to click on text in the text flow. Headers and footers are often generated from structure, and special typographic elements can originate in macros hooked into commands instead of in the source.
- Users should not be able to reach environments (styles) and other files loaded from the (normally read-only) `TEX` tree, like modules. We don’t want accidental changes in such files.
- We not only have `TEX` files but also `XML` files and these can normally flush in rather arbitrary ways. Although the concept of lines is sort of lost in such a file, there is still a relation between lines and the snippets that make out the content of an `XML` node.
- In the case of `XML` files the overhead related to preserving line numbers should be minimal and have no impact on loading and memory when these features are not used.
- The overhead in terms of an auxiliary file size and complexity as well as producing that file should be minimal. It should be easy to turn on and off these features. (I’d never turn them on by default.)

It is unavoidable that we get more run time but I assume that for the average user that is no big deal. It pays off when you have a workflow when a book (or even a chapter in a book) is generated from hundreds of small `XML` files. There is no overhead when `SYNCTEX` is not used.

In `CONTEXT` we don’t use the built-in `SYNCTEX` features, that is: we let filename and line numbers be set but often these are overloaded explicitly. The output file is not compressed and constructed by `CONTEXT`. There is no benefit in compression and the files are probably smaller than default `SYNCTEX` anyway.

7.6 Commands

Although you can enable this mechanism with directives it makes sense to do it using the following command.

```
\setupsynctex[state=start]
```

¹³ This is something that in my opinion should have been possible right from the start but it’s too late now to change the system and it would not be used beyond `CONTEXT` anyway.

The advantage of using an explicit command instead of some command line option is that in an editor it's easier to disable this trickery. Commenting that line will speed up processing when needed. This command can also be given in an environment (style). On the command line you can say

```
context --synctex somefile.tex
```

A third method is to put this at the top of your file:

```
% synctex=yes
```

Often an XML files is very structured and although probably the main body of text is flushed as a stream, specific elements can be flushed out of order. In educational documents flushing for instance answers to exercises can happen out of order. In that case we still need to make sure that we go to the right spot in the file. It will never be 100% perfect but it's better than nothing. The above command will also enable XML support.

If you don't want a file to be accessed, you can block it:

```
\blocksynctexfile[foo.tex]
```

Of course you need to configure the viewer to respond to the request for editing. In Sumatra combined with SciTE the magic command is:

```
c:\data\system\scite\wscite\scite.exe "%f" "-goto:%l"
```

Such a command is independent of the macro package so you can just consult the manual or help info that comes with a viewer, given that it supports this linking back to the source at all.

If you enable tracing (see next section) you can what has become clickable. Instead of words you can also work with ranges, which not only gives less runtime but also much smaller `.synctex` files. Use

```
\setupsynctex[state=start,method=min]
```

to get words clickable and

```
\setupsynctex[state=start,method=max]
```

if you want somewhat more efficient ranges. The overhead for `min` is about 10 percent while `max` slows down around 5 percent.

7.7 Tracing

In case you want to see what gets synced you can enable a tracker:

```
\enabletrackers[system.synctex.visualize]  
\enabletrackers[system.synctex.visualize=real]
```

The following tracker outputs some status information about XML flushing. Such trackers only make sense for developers.

```
\enabletrackers[system.synctex.xml]
```

7.8 Warning

Don't turn on this feature when you don't need it. This is one of those mechanism that hits performance badly.

Depending on needs the functionality can be improved and/or extended. Of course you can always use the traditional SYNCTEX method but don't expect it to behave as described here.

8 Advertising T_EX

First published in user group magazines.

9 Tricky fences

First published in user group magazines.

10 From 5.2 to 5.3

Maybe first published in user group magazines.

