FOLLOWING UP

ConTEXt lmtx

Table of contents

1 Introduction

This document, the fifth in a series, describes the follow up project on ConTEXt MkIV & LuaTEX which carries the working title ConTEXt LMTX. This four letter acronym represents Lua, MetaPost and TEX, and if you want you can see the last character representing xml, as that has been an integral part of ConTEXt for a long time. But the 'x' can also be found in 'experimental', 'extreme', 'experience' and 'extravagant', so take your choice.

Of course ConTEXt is and will be a typesetting system using the TEX language and typesetting core, but a rather substantial amount of the functionality is a hybrid of TEX macros and Lua code. The built-in graphic support is driven by MetaPost, but there we also use Lua as an extension language. The Lua machinery is used for alternative input and handling data too. The same is true for xml, sql, csv, json, etc.

The output from ConTEXt is normally pdf and MkIV doesn't even enable dvi output. Mid 2018 I started experimenting with a backend that no longer used the one provided by the engine. After all, we only used page stream building, font embedding and bitmap inclusion and all other features were always done in Lua. The experiments also concerned a MetaPost and Lua backend. Those familiar with ConTEXt know that there is already an export feature which till now runs in parallel with the ConTEXt pdf backend (it started as a kind of joke but in the end was seen as relevant and kept and maybe so some point I will rewrite that code).

The idea behind ConTEXt LMTX is that we will use a minimalist engine. Being minimalist also means that probably only ConTEXt will use it and therefore no other package will be affected by further experiments, although at some point a sort of general low level layer might be provided. The frontend is mostly the same as LuaTEX 1.1 but the backend and related code is gone and/or different. Libraries have (and are) being cleaned up and reorganized too. At least for a while, ConTEXt will work on LuaTEX 1.1 (stable) as well as its (experimental) follow up, where the follow up will evolve over a few years and be tested in the usual ConTEXt (garden) beta setting. The next chapters will explain this in more detail.

Just to be clear I repeat: LuaTEX 1.1 will be supported by ConTEXt and maintained as usual, including binaries generated on the ConTEXt garden. We've invested many years in it and it serves its purpose well, but our experiments will happen in its follow up, so that it doesn't affect stable workflows. Of course there have been (and probably are) bugs in LuaTEX but the engine could be used pretty well right from the start with ConTEXt. The same will be true for the follow up.

One of the ideas of the follow up is to provide a combination of a stable engine independent of libraries with a relative simple compilation setup and a macro package that has

proven to exploit a mix of TEX, MetaPost and Lua. As a side effect I can explore some postponed ideas. Of course there can be valid reasons to move to the successor sooner. In that case we might create a stable snapshot of MkIV as was done with MkII. As to be expected in ConTEXt, the user interfaces won't change nor will the functionality, but there will be two code paths, one for MkIV and one for LMTX. There will also be new functionality in ConTEXt that is only available in LMTX. So, eventually we expect all users to migrate.

In the beginning of december 2018 most of the work was done and users involved in development could start testing. By the end of the year a reasonable stable state was reached. In 2019 the code base was further overhauled and libraries got upgraded. The code base became smaller and compilation easier, smoother and much faster. Eventually the source code (now some 11MB uncompressed and 3MB compressed) will be part of the ConTEXt distribution, so that we have a complete package (also in the archival sense).

The next chapters discuss the process and choices that were made. The chapters were written in order so later chapters can amend earlier ones. Consider it a history, and one cannot cheat by patching history. In some cases footnotes were added to earlier chapters when writing later ones. It's not a manual! Reported typos (for sure there are many) will be fixed but changes in later versions of the follow discussed here will not end up in this document.

This document is dedicated to Wolfgang Schuster, who has been instrumental in the transition from MkII to MkIV, and often baffles me with his knowledge of the (even obscure bits) of the ConTEXt internals. Without him checking the code base, fundamental changes like those that are and might get introduced in this follow up are impossible.

I want to thank Alan Braslau who accompanies me on this journey and patiently compiles the lot for some platforms. He, Thomas Schmitz and Aditya Mahajan are examples of power users who also are early adopters of something new like this and are willing to take the risks. And of course there is Mojca Miklavec without whose enthusiasm and optimism developments like this would never take place. In the meantime Luigi Scarso made sure that the (frozen) LuaTEX code base served existing users. It is hard to tell how users experience the transition: there are no that many issues reported which can be a good or bad sign. We will see.

Hans Hagen
PRAGMA ADE, Hasselt NL
August 2018 – May 2019

2 Evolution

2.1 Introduction

The original idea behind TEX is that of a relatively small kernel with (either or not system dependent) extensions. One such extension is the dvi backend, and later pdfTEX added a pdf backend. Other extensions are 'writing to files' and 'writing to the output medium' using so called specials. This extension mechanism permits TEX to support, for instance, color and image inclusion.

The LuaTEX project started from pdfTEX, including its extensions like font expansion, and combined that with (bi)directional typesetting from the, at that moment, stable Omega variant Aleph. During the more than a decade development we integrated expansion in a more efficient way and limited directions to the four that made sense. The assumption that Unicode has the future lead to utf8 being used all over the place.

The LuaTEX variant opens up the internals using the Lua extension language. The idea was (and still is) that instead if adding more and more hard coded solutions, one can use Lua to do it on demand. So, for instance OpenType fonts are supported by providing a font file reader but the implementation of features is up to Lua. From pdfTEX the graphic inclusions were inherited but an image and pdf reading library provided a few more possibilities, for instance for querying properties. An important integral part of LuaTEX is the MetaPost library, but apart from that one, the amount of libraries is kept at a minimum. That way we're free of dependencies and compilation hassles.

With version 1.0 the functionality became official and with version 1.1 the functionality became more of less frozen. The main reason for this is that further extensions would violate the principle of using Lua instead of hard coding solutions. Another reason is that at some point you have to provide a stable machinery for macro packages so that backward as well as forward compatibility over a longer period is possible. Also, because one can use TEX in (unattended) workflows sudden changes become undesirable.

2.2 What next?

Does it stop here? We have reached a reasonable stable state with ConTEXt MkIV and can basically do what we want to do. However, during the more than a decade development of this MkII follow up, the idea surfaced that we can go more minimal in the engine. Basically we can go back to where TEX started: a core plus extension mechanism. What does that mean? First of all, there is the very efficient frontend: scanning macros, expanding them and constructing node lists, all within a powerful grouping mechanism. There is no reason to reconsider that. The core of the interface is also well documented,

for instance in the TEX book. We added some primitives to LuaTEX, but most of them are of no real importance to users; they make more sense to macro package writers.

Original TEX has a dvi backend which is a simple representation of a page: characters and rules positioned on some grid. A separate program has to convert that into something for a printer. There is a basic extension mechanism that permits injection of so called specials that get passed to the external program so that for instance an image can be included. Given that LuaTEX is mostly used to generate pdf, using so called wide fonts in a Unicode universe, a dvi backend is not that useful. In fact, one can then better use the faster pdfTEX program or just ε-TEX or TEX: use the best tool available for the job.

The backend however can be left out and can be implemented in Lua instead. In fact, most of the backend related code in ConTEXt doesn't really use the LuaTEX backend features at all. The backend is only used to convert the page stream to a pdf content stream, include images, include fonts and manage low level objects. Everything specific to pdf is already done in Lua. Of course this has a performance penalty but given the overhead already present in ConTEXt it is bearable.

Alongside the frontend the MetaPost library plays an important role in ConTEXt: integration between TEX, MetaPost and Lua is pretty tight and a unique property of ConTEXt. But, for instance the font reader library is no longer used. Also the interfacing to the TEX Directory Structure was done in Lua, originally for performance reasons as it reduced startup time by more that a second. For some of the frontend code (like hyphenation and par building) we can kick in Lua variants too but there is not much to gain there. (I know that some users use them with success.)

So, traditional TEX can be summarized as:

```
tex core + dvi backend + tex extensions
```

where the extension interface provide a few goodies. If we would have to summarize LuaTEX we could say:

```
tex core + dvi & pdf backend + tex extensions + lua callbacks
```

The core interprets the input and does the typesetting. In order to be able to typeset TEX only needs the dimensions of characters and information about spacing (which in principle are sort of independent) in math mode a few more properties are needed, like snippets that make large symbols. In text mode ligature and kerning information can be used too. However, in LuaTEX, where normally OpenType fonts are used, that information is provided from Lua. This means that one can also think of:

```
tex core + basic font data + tex extensions + lua callbacks
```

Compared to regular TEX this is not that different, and it's what ConTEXt can do with. So, it will be no surprise that when I wondered what LuaTEX 2.0 could be that a more minimalistic approach was considered: back to the basics.

2.3 Roadmap

Before I continue it is good to mention the following. One of the burdens that ConTEXt users (and developers) carry is that the outside world likes putting labels on ConTEXt, like "A macro package depending on pdfTEX" in a time that we supported dvi at the same level using a more of less generic driver model. The same is true for MkIV, e.g. "ConTEXt uses a lot of Lua and moves away from TEX" while in fact we provide a hybrid tool: you can use TEX input (which most users do) but also Lua (which can be handy) or xml (which some publishers demand and definitely seems to be used by some ConTEXt power users). A special one is "ConTEXt is kind of plain TEX, so you have to program all yourself." Reality is that ConTEXt is an integrated system, where TEX and MetaPost work together to provide a lot of integrated functionality. Because of LuaTEX development and the relation between an updated engine and the beta version of ConTEXt, the impression can be that we have an unstable system. This strategy of parallel adaptation is the only way to really test of things work as expected. Because we have a rather fast update cycle normally users don't suffer that much from it.

The core of whatever we follow up with is and remains TEX, just because I like it. So, when I talk about a small core, I actually still talk about TEX. The main reason is that it's way easier (and readable) to code some solutions in this hybrid fashion. A pure Lua solution is no fun, maybe even a pain, and I have no use for it, but a pure TEX solution can be cumbersome too. And TEX input is just very convenient and for that one needs a TEX interpreter. I would already have dropped out when TEX was not part of the game: an intriguing, puzzling and powerful toy. And MetaPost and Lua add even more fun. So, I settle for a mix between three interesting languages. And, because I seldom run into professional demand for LuaTEX related support (or high end, high performance rendering), the fun factor has always been the driving force.

All that said, for practical reasons, when we explore a follow up in the perspective of ConTEXt, we will use the working title LuaMetaTEX instead. LuaMetaTEX has the current LuaTEX frontend, some Lua libraries, but no backend. Gone are the font reader, image inclusion, dvi and pdf backend (including font inclusion) and the interface to the tds. Can that work? As mentioned, the font reader was already not used in ConTEXt for quite a while. An alternative page stream builder was also in good working condition in ConTEXt when LuaTEX 1.08 was released and around LuaTEX 1.09 image inclusion was replaced (pdf inclusion was already accompanied for a while by a Lua variant). Currently (fall 2018) ConTEXt is able to completely construct the pdf file which also meant font inclusion. However, it didn't make much sense to release that code yet because after all,

there was minimal gain when using it with a full blown LuaTEX. Also, switching to this variant involved some runtime adaption of code which might confuse users. But above all, it needed more testing, and releasing something before an upcoming TEXLive code freeze is a bad idea.

During LuaTEX development a few times we got suggestions for additional features but merely looking at them already made clear that what works for someone in a particular case, can introduce side effects that make (for instance) ConTEXt fail. And, how many folks keep ConTEXt in mind? So, when LuaTEX goes into maintenance mode, specific distributions could accept patches outside our control, which has the danger that a binary (suggesting to be LuaTEX) doesn't work with ConTEXt. Of course we cannot change something ourselves either without looking around. And I'm not even bringing possible negative side effects on performance into the discussion here.

When developing LuaTEX some ideas were dropped or delayed and these can now be explored without the danger of messing up the stable version. It has always been relatively easy to adapt ConTEXt to changes so an (at least for now) experimental follow up can be dealt with too, but this time the concept of 'experimental' is really bound to ConTEXt. When something is found useful (or can be improved) it can always (after testing it for a while) be fed back into LuaTEX, as long as it doesn't break something. I'll decide on that later.

In the documentation of TEX, when discussing the extension mechanism, Donald Knuth says:

> "The goal of a TEX extender should be to minimize alterations to the standard parts of the program, and to avoid them completely if possible. He or she should also be quite sure that there's no easy way to accomplish the desired goals with the standard features that TEX already has. "Think thrice before extending", because that may save a lot of work, and it will also keep incompatible extensions of TEX from proliferating."

With the in the next chapters discussed reduction of backend and some frontend code, combined with hooks that can trigger callbacks, we try to come close to this objective. Now, the last sentence of this quote relates to stability and this is also a reason why we enter this new thread: the smaller the core is, the less subjected we are to change. Think of this: I haven't used ConTEXt MkII in over a decade. A pdfTEX format still gets generated but I have no clue if the engine has been changed in ways that make some code behave differently (it could also be the ecosystem related to that engine), but I assume it's still behaving the same. The same has to become true for stock LuaTEX and MkIV and for ConTEXt it can even become more true with LuaMetaTEX. We'll see.

2.4 Experiments

This (still sort of) prototype of what LuaMetaTEX could be boils down to a much smaller binary, and not that much more Lua code on top of what we already have. There are no longer dependencies on third party code, apart from Lua (`pplib` is tuned for LuaTEX and permanent part of the code base). Performance wise the backend of the experimental version makes a run upto 5% slower than when using a native backend (on processing the LuaTEX manual) but history has learned that we can gain some of that back in due time. Performance also depends a bit on the properties of the document. Interesting is that better control over the output showed that pdf output of the mentioned manual was a bit smaller (but that might change).[1]

The experiments actually started already years ago with no longer using the font loader. It sort of went this way:

- Stepwise ConTEXt functionality started using a combination of TEX and Lua code and we got an idea of what was needed. The most demanding part was support for fonts.
- Font handling was done in Lua because it's flexible which is what TEXies are accustomed to. The OpenType and pdf standards would not be called standards if some implementation was impossible and so far we're ok. (Some more script support will be provided in future versions.)
- We stopped using the fontforge font loader but use one written in Lua instead. One reason for this was that when variable fonts showed up we wanted to support it in ConTEXt right from the start (not that there has been much demand). The same is true for fonts using color (like emoji). Also, fighting the built-in FontForge heuristics was hard.
- The (large and dependent on C++) poppler library used for pdf embedding has been replaced by a small lightweight library in pure C. This was triggered at a chat during a bachoTEX meeting.
- The hard coded pdf inclusion can be swapped with a Lua based one so that we can for instance filter the page stream. We already had a hybrid solution in ConTEXt anyway for other reasons (merging annotations, layers, bookmarks, etc.).
- The page stream constructor got a (shipout and xforms) by a Lua variant, but I decided not to make that an independent option in stock LuaTEX with ConTEXt MkIV, although for a while I had the option `--lmtx` for activating that experimental code.

[1] In the meantime the experimental version can process the LuaTEX manual 5–10% faster and the result is still smaller.

- Then of course bitmap image inclusion had to be done by Lua code, in order to see if we can get rid of another external dependency as some of these libraries get frequent updates while in practice we only use a very small subset of functionality. Indeed this was possible.[2]

- With some effort (deciphering specs and such) the font inclusion could also be done by a Lua. This was made possible by the fact that we already had support for variable fonts. More tricks are possible and will be explored.

- Finally the pdf file construction and pdf object management had to be implemented. This was actually the easiest part.

Performance wise the Lua font loader is faster than the built in one. The same is true for pdf inclusion but in practice that is unnoticeable. Bitmap inclusion is currently slower for interlaced images (seldom used in print) and just as efficient for other types. The page stream constructor is definitely slower but this is compensated by the faster font inclusion and pdf file construction. Of course it all depends on the kind of content, but these are the observation as of fall 2018. Anyway, they were enough reason to continue this experiment.

One thing to keep in mind is that the smaller the binary and the less code paths we have, the better future performance might be. Computers are not becoming much faster for single thread processes like TEX, so the less we jump around code space (memory) the better it probably is for cpu caching (as caches are not growing much either).

2.5 Conclusion

Normally when writing this kind of code I make sure that I can enable such new mechanisms on top of others but at some point one has to decide how to really integrate them. For instance, we can do font inclusion independent of pdf generation or page stream construction independent of pdf generation and/or font inclusion but in the end that doesn't make sense and makes the code base a bit of a mess. So, this is how it will go.

Stock LuaTEX with MkIV will use the normal backend but probably there might be an option to overload the built-in image inclusion so that one can avoid the abortion of a run in case of problematic images. Complete pdf file construction, which then also includes page stream construction, font embedding and object management might be available as option for MkIV with LuaTEX 1.10 (for a while) but will be default when using LuaMetaTEX. When we move on LMTX support might evolve in more sophisticated trickery.[3]

[2] I have a pure Lua parser for pdf too, so at some point that might get included in the ConTEXt code base.
[3] A few months later I decided that this made no sense, and that it was cleaner to just leave that approach

Once tested a bit in real documents experimental code will end up in the distribution. That code can then be turned into production code (read: cleaned up and reshuffled a bit). We can streamline the engine code base: strip the components that are not needed any more, remove some obsolete features, optimize the code, strip some functions from Lua libraries, rename some helpers, and finally add some documentation. There are some plans to extend MetaPost so also things can get added. Concerning the Lua interface it means that `slunicode` is removed, the embedded socket related Lua code goes external (but the library stays), the font loader gets removed, the `img` library goes away, no longer png libraries are embedded, synctex is stripped out (but the fields in nodes stay or get extended).[4] The resulting binary will be much smaller and the code base more independent and smaller too. In the process LuaJIT support might be dropped as well, simply because it no longer is in sync with stock Lua, but that also depends on how complex long term maintenance becomes.[5]

Because such a stripped down binary is no longer what got presented as LuaTEX version 1, it will basically become LuaTEX version 2, but then we have the problem that its binary name clashes with the original. This is why it will be run as `luametatex`. For ConTEXt it's not that relevant as it will run on both LuaTEX 1.10 and its lean and mean successor. I might also provide a plain TEX (read: generic) version but that is to be decided because it probably doesn't make much sense to spend time on it. As usual we will test this within the ConTEXt beta program. The good thing is that it doesn't interact with LuaTEX, so that other macro packages are not affected. Another side effect can be that we uncover issues with LuaTEX 1.10 and that we can experiment with some improvements that we feed back into the parent.

At the ConTEXt end of this there are some plans to extend the export, maybe improve already present pdf tagging (if found useful), add some more input (xml) manipulations, and maybe extend (virtual) font handling a bit, now that we no longer are bound to the currently used packet model. Contrary to what one might expect this is not really dependent on the engine.

How do we proceed? As with the transition from MkII to MkIV, it will all happen stepwise. This means that for a while the code base will be a bit hybrid but at some point it might be partially split to make things cleaner, not that I expect many fundamental differences (certainly not in the front-end). This dualistic approach means more work but also makes that we keep a working ConTEXt. We also need to keep an eye on for instance generic commands as used in tikz: we can't drop them so we emulate them (so far with success). As the time of this writing, begin November 2018, the ConTEXt test suite can

for LMTX only. So, now both engines use different code exclusively.

[4] Much later I also decided to remove the zip file reader library.

[5] As we will see in following chapters, indeed support for LuaJIT has been dropped while Lua got upgraded to 5.4.

be processed in LMTX mode without problems so I'm confident that it will work out ok. The next chapter describes the results of how we did the above in more detail.

3 Stripping

3.1 Introduction

Normally I need a couple of iterations to reach the implementation that I like (an average of three rewrites is rather normal). So, I sat down and started stripping the engine and did so a few times in order to get an idea of how to proceed. One drawback of going public too soon (and we ran into that with LuaTEX) is that as soon as there are more users, one gets stuck into the situation that a different approach is not really possible. This is why from now on experimental is really experimental, even if that means: it works ok in ConTEXt (even for production) but we can change interfaces be better, e.g. more consistent (although we're also stuck with existing TEX terminology). Anyway, let's proceed.

3.2 The binary

In 2014 the LuaTEX binary was some 10.9 MB large. The version 1.09 binary of October 2018 was about 6.8MB, and the reduction was due to removing the bitmap generation from mplib as well as replacing poppler by pplib. As an exercise I decided to see how easy it was to make a small version suitable for ConTEXt LMTX, and as expected the binary shrunk to below 3MB (plus a Lua and kpse dll). This is a reasonable size given what is still present.

There is hardly any file related code left because in practice the backend used the most different file types. That also meant that we could remove kpse related code and keep all that in the library part. In principle one can load that library and hook it into the few callbacks that relate to loading files. Once we're stable I'll probably write some code for that.[6] Launching the binary with a startup script can deal with all matters needed, because the command line arguments are available.

We could actually go even smaller by removing the built-in tfm and vf readers. For instance it made not much sense to read and store information that is never used anyway, like virtual font data: as long as the backend has access to what it needs it's fine. By removing unused code and stripping no longer used fields in the internal font tables (which is also good for memory consumption), and cleaning up a bit here and there the experimental binary ended up at a bit above 2.5MB (plus a Lua dll).[7]

[6] In the meantime I think it makes not much sense to do that.

[7] Mid January we were just below 2.7 MB with a static, all inclusive, binary. In March the static ended up at 2.9 MB on MS Windows and 2.6 MB in Unix.

3.3 Functionality

There is no real reason to change much in the functionality of the frontend but as we have no backend now, some primitives are gone. These have to be implemented as part of creating a backend.

```
\dviextension \dvivariable \dvifeedback
\pdfextension \pdfvariable \pdffeedback
```

The already obsolete related dimensions are also removed:

```
\pageleftoffset \pagerightoffset
\pagetopoffset  \pagebottomoffset
```

And we no longer need the page dimensions because they are just registers that are normally used in the backend. So, we got rid of:

```
\pageheight
\pagewidth
```

Some font related inheritances from pdfTEX have also been dropped:

```
\letterspacefont
\copyfont
\expandglyphsinfont
\ignoreligaturesinfont
\tagcode
```

Internally all backend whatsits are gone, but generic `literal`, `save`, `restore` and `setmatrix` nodes can still be created. Under consideration is to let them be so called user nodes but for testing it made sense to keep them around for a while.[8]

The resource relates primitives are backend dependent so the primitives have been removed. As with other backend related primitives, their arguments depend on the implementation. So, no more:

```
\saveboxresource
\useboxresource
\lastsavedboxresourceindex
```

and:

[8] Don't take this as a reference: later we will see that more was changed.

```
\saveimageresource
\useimageresource
\lastsavedimageresourceindex
\lastsavedimageresourcepages
```

Of course the rule nodes subtypes are still there, so the typesetting machinery will handle them fine. It is no big deal to define a pseudo-primitive that provides the functionality at the TEX level.

The position related primitives are also backend dependent so again they were removed.[9]

```
\savepos
\lastxpos
\lastypos
```

We could have kept `\savepos` but better is to be consistent. We no longer need these:

```
\outputmode
\draftmode
\synctex
```

These could go because we no longer have a backend and if one needs it it's easy to define a meaningful variable and listen to that.

The `\shipout` primitive does no ship out but just flushes the content of the box, if that hasn't happened already.

Because we have Lua on board, and because we can now use the token scanners to implement features, we no longer need the hard coded randomizer extensions. In fact, also the MetaPost should now use the Lua randomizer, so that we are consistent. Anyway, removed are:

```
\randomseed
\setrandomseed
\normaldeviate
\uniformdeviate
```

[9] There was some sentimental element in this. Long ago, even before pdfTEX showed up, ConTEXt already had a positional mechanism. It worked by using specials in combination with a program that calculated the positions from the dvi file. At some point that functionality was integrated into pdfTEX. For me it always was a nice example of demonstrating that complaints like "TEX is limited because we don't know the position of an element in the text." make no sense: TEX can do more than one thinks, given that one thinks the right way.

plus the helpers in the `tex` library.

3.4 Fonts

Fonts are sort of special. We need the data at the Lua end in order to process OpenType fonts and the backend code needs the virtual commands. The par builder also needs to access font properties, as does the math renderer, but here is no real reason to carry virtual font information around (which involves packing and unpacking virtual packets). So, in the end it made much sense to also delegate the tfm and vf loading to Lua as well. And, as a consequence dumping and undumping font information could go away too, which is okay, as we didn't preload fonts in ConTEXt anyway. The saving in binary bytes is not impressive but keeping unused code around neither. In principle we can get rid of the internal representation if we fetch relevant data from the Lua tables but that might be unwise from the perspective of performance. By removing the no longer needed fields the memory footprint became somewhat smaller and font loading (passing from Lua to TEX) more efficient.

3.5 File IO

What came next? A program like LuaTEX interacts with its environment and one of the nice things about TEX is that it has a standard ecosystem, organized as the "TEX Directory Structure". There is library that interfaces with this structure: kpse, but in ConTEXt MkIV we implement its functionality in Lua. The primary reason for this was performance. When we started with LuaTEX the startup on my machine (MS Windows) and a few servers (linux) of a TEX engine took seconds and most fo that was due to loading the rather large file databases, because a TEX Live installation was a gigabyte adventure. With the Lua variant I could bring that down to milliseconds, because I could pre-hash the database and limit it to files relevant for ConTEXt (still a lot, as fonts made up most). Nowadays we have ssd disks and plenty of memory for caching, so these things are less urgent, but on network shares it still matters.

So, as we don't use kpse, we can remove that library. By doing that we simplify compilation a lot as then all dependencies are in the engine's source tree, and we're no longer dependent on updates. One can argue that we then sacrifice too much, but already for a decade we don't use it and the Lua variant does the job well within the tds ecosystem. Also, in our by now stripped down engine, there is not that much lookup going on anyway: we're already in Lua when we do fonts. But on the other hand, some generic usage could benefit from the library to be present, so we face a choice. The choice is made even more difficult by the fact that we can remove all kind of tweaks once we delegate for instance control over command execution to Lua completely. But, we might provide kpse as loadable Lua module so that when needed one can use a stub to start the pro-

gram with a Lua script that as first action loads this library that then can take care of further file management. As command line arguments are available in Lua, one can also implement the relevant extra switches (and even more if needed).

Now, the interesting thing is that because we have a Lua interface to kpse we can actually drop some hard coded solutions. This means that we can have a binary without kpse, in which case one has to cook up callbacks that do what this library does. But in a version with kpse embedded one also has to define some file related callbacks although they can be rather simple. By keeping a handful of file related callbacks the code base could be simplified a lot. In the process the recorder option went away (not that we ever used it). It is relatively easy to support this in the 'find' related callbacks and one has to deal with other files (like images and fonts) also, so keeping this feature was a cheat anyway.

At this point it is important to notice that while we're dropping some command line options, they can still be passed and intercepted at the Lua end. So, providing compatible (or alternative solution) is no big deal. For instance, execution of (shell) programs is a Lua activity and can be managed from there.

3.6 Callbacks

Callbacks can be organized in groups. First there are those related to io. We only have to deal with a few types: all kind of TEX files (data files), format files and Lua modules (but these to are on the list of potentially dropped files as this can be programmed in Lua).

```
find_write_file
find_data_file open_data_file read_data_file
find_format_file find_lua_file find_clua_file
```

The callbacks related to errors stay:[10]

```
show_error_hook show_lua_error_hook,
show_error_message show_warning_message
```

The management hooks were kept (but the edit one might go):[11]

```
process_jobname
call_edit
start_run stop_run wrapup_run
pre_dump
start_file stop_file
```

[10] Some more error handling was added later, as was intercepting user input related to it.
[11] And indeed, that one went away.

Of course the typesetting callbacks remain too as they are the backbone of the opening up:

```
buildpage_filter hpack_filter vpack_filter
hyphenate ligaturing kerning
pre_output_filter contribute_filter build_page_insert
pre_linebreak_filter linebreak_filter post_linebreak_filter
insert_local_par append_to_vlist_filter new_graf
hpack_quality vpack_quality
mlist_to_hlist make_extensible
```

Finally we mention one of the important callbacks:

```
define_font
```

Without that one defined not much will happen with respect to typesetting. I could actually remove the `\font` primitive but that would be a bit weird as other font related commands stay. Also, it's one of the fundamental frontend primitives, so removal was never really considered.

3.7 Bits and pieces

In the process some helpers and status queries were removed. From the summary above you can deduce that this concerns images, backend, and file management. Also not used variables (some inherited from the past and predecessors) were removed. These and other changes are the reason why there is a separate manual for LuaMetaTEX.[12]

One of my objectives was to see how lean and mean the code base could be. But even if we don't use that many files, the rather complex build system makes that we need to have (make and configure) files in the tree that are not really used but even then omitting them aborts a build. I played a bit with that but the problem is that it needs to be dealt with upstream in order to prevent repetitive work. So, this is something to sort out later. Eventually it would be nice to be able to compile with a minimal set of source files, also because other programs (all kind of TEX variants) that are checked for but not compiled depend on libraries that we don't need (and therefore want) to have in the stripped down source tree.[13]

For now we also brought down the number of catcode tables (to 256)[14], and the number

[12] Relatively late in the project I decided to be more selective in what got initialized in Lua only mode.
[13] In the end, the source tree was redesigned completely.
[14] As with math families, and if more tables are needed one should wonder about the TEX code used.

of languages (to 8192)[15] as that saves some initially allocated memory.

3.8 What's next

Basically the experiment ends here. A next step is to create a stable code base, make compilation easy and consider the way the code is packages. Then some cleanup can take place. Also, as it's a window to the outside world, `ffi` support will move to the code base and be integral to LuaMetaTEX. And of course the decision about LuaJIT support has to be made some day soon. The same is true for Lua 5.4: in LuaTEX for now we stick to 5.3 but experimenting with 5.4 in LuaMetaTEX can't harm us.[16]

To what extend the ConTEXt code base will have a special files for LMTX is yet to be decided, but we have some ideas about new features that might make that desirable from the perspective of maintenance. The main question is: do I want to have hybrid files or clean files for each variant (stock MkIV and LMTX).

For the record: at the time of wrapping this up, processing the LuaTEX manual of 294 pages took 13.5 seconds using stock LuaTEX while using the stripped down binary, where Lua takes over some tasks, took 13.9 seconds.[17] The LuajitTEX variant needed 10.9 and 10.8 seconds. So, there is no real reason to not explore this route, although ... the pdf file size shrinks from 1.48MB to 1.18MB (and optionally we can squeeze out more) but one can wonder if I didn't make big mistakes. It is good to realize that there is not much performance to gain in the engine simply because most code is already pretty well optimized. The same is true for the ConTEXt code: there might be a few places where we can squeeze out a few milliseconds but probably it will go unnoticed.

On the todo list went removal of `\primitive` which we never use (need) and the possible introduction of a way to protect primitives and macros against redefinition, but on the other hand, it might impact performance and be not worth the trouble. In the end it is a macro package issue anyway and we never really ran into users redefining primitives.[18]

[15] This is already a lot and because languages are loaded run time, we can go much lower than this.

[16] The choice has been made: LuaMetaTEX will not have a LuaJIT based companion.

[17] In the meantime we're down to around 11.6MB. These are all rough numbers and mostly indicate relative speeds at some point.

[18] Indeed this primitive has been removed.

20 Stripping

4 Bitmap images

4.1 Introduction

In TEX image inclusion is traditionally handled by specials. Think of a signal added someplace in the page stream that says:

```
\special{image: foo.png 2000 3000}
```

Here the number for instance indicate a scale factor to be divided by 1000. Because TEX has no floating point numbers, normally one uses an integer and the magic multiplier 1000 representing 1.000. Such a special is called a 'whatsit' and is one reason why TEX is so flexible and adaptive.

In pdfTEX instead of a `\special` the command `\pdfximage` and its companions are used. In LuaTEX this concept has been generalized to `\useimageresource` which internally is not a so called whatsit (an extension node) but a special kind of rule. This makes for nicer code as now we don't need to check if a certain whatsit node is actually one with dimensions, while rules already are part of calculating box dimensions, so no extra overhead in checking for whatsits is added. In retrospect this was one of the more interesting conceptual changes in LuaTEX.

In LuaMetaTEX we don't have such primitives but we do have these special rule nodes; we're talking of subtypes and the frontend doesn't look at those details. Depending on what the backend needs one can easily define a scanner that implements a primitive. We already did that in ConTEXt. More important is that inclusion is not handled by the engine simply because there is no backend. This means that we need to do it ourselves. There are two steps involved in this that we will discuss below.

4.2 Identifying

There is only a handful of image formats that makes sense in a typesetting workflow. Because pdf inclusion is supported (but not discussed here) one can actually take any format as long as it converts to pdf, and tools like graphic magic do a decent job on that.[19] The main bitmap formats that we care about are jpeg, jpeg2000, and png. We could deal with jbig files but I never encountered them so let's forget about them for now.

[19] Although one really need to check a converted image. When we moved to pplib, I found out that lots of converted images in a project had invalid pdf objects, but apart from a warning nothing bad resulted from this because those objects were not used.

One of the problems with a built-in analyzer (and embedder) is that it can crash or just abort the engine. The main reason is that when the used libraries run into some issue, the engine is not always able to recover from it: a converter just aborts which then cleans up (potentially messed up) memory. In LuaTEX we also abort, simply because we have no clue to what extend further on the libraries are still working as expected. We play safe. For the average user this is quite ok as it signals that an image has to be fixed.

In a workflow that runs unattended on a server and where users push images to a resource tree, there is a good change that a TEX job fails because of some problem with images. A crash is not really an option then. This is one reason why converting bitmaps to pdf makes much sense. Another reason is that some color profiling might be involved. Runtime manipulations make no sense, unless there is only one typesetting run.

Because in LMTX we do the analyzing ourselves[20] we can recover much easier. The main reason is of course that because we use Lua, memory management and garbage collection happens pretty well controlled. And crashing Lua code can easily be intercepted by a `pcall`.

Most (extensible) file formats are based on tables that gets accessed from an index of names and offsets into the file. This means that filtering for instance metadata like dimensions and resolutions is no big deal (we always did that). I can extend analyzing when needed without a substantial change in the engine that can affect other macro packages. And Lua is fast enough (and often faster) for such tasks.

4.3 Embeding

Once identified the frontend can use that information for scaling and (if needed) reuse of the same image. Embedding of the image resource happens when a page is shipped out. For jpeg images this is actually quite simple: we only need to create a dictionary with the right information and push the bitmap itself into the associated stream.

For png images it's a bit different. Unfortunately pdf only supports certain formats, for instance masks are separated and transparency needs to be resolved. This means that there are two routes: either pass the bitmap blob to the stream, or convert it to a suitable format supported by pdf. In LuaTEX that is normally done by the backend code, which uses a library for this. It is a typical example of a dependency of something much larger than actually needed. In LuaTEX the original poppler library used for filtering objects from a pdf file as well as the png library also have tons of code on board that relates to manipulating (writing) data. But we don't need those features. As a side note: this is

[20] Actually, in MkIV this was also possible but not widely advertised, but we now exclusively keep this for LMTX.

something rather general. You decide to use a small library for a simple task only to find out after a decade that it has grown a lot offering features and having extra dependencies that you really don't want. Even worse: you end up with constant updates due to fixed security (read: bug) fixes.

Passing the png blob unchanged in itself to the pdf file is trivial, but massaging it into an acceptable form when it doesn't suit the pdf specification takes a bit more code. In fact, pdf does not really support png as format, but it supports png compression (aka filters).

Trying to support more complex png files is a nice way to test if you can transform a public specification into a program as for instance happens with pdf, OpenType, and font embedding in ConTEXt. So this again was a nice exercise in coding. After a while I was able to process the png test suite using Lua. Optimizing the code came with understanding the specification. However, for large images, especially interlaced ones, runtime was definitely not to be ignored. It all depended on the tasks at hand:

- A png blob is compressed with zip compression, so first it needs to be decompressed. This takes a bit of time (and in the process we found out that the `zlib` library used in LuaTEX had a bug that surfaced when a mostly zero byte image was uncompressed and we can then hit a filled up buffer condition.

- The resulting uncompressed stream is itself compressed with a so called filter. Each row starts with a filter byte that indicates how to convert bytes into other bytes. The most commonly used methods are deltas with preceding pixels and/or pixels on a previous row. When done the filter bytes can go away.

- Sometimes an image uses 1, 2 or 4 bits per pixel, in which case the rows needs to be expanded. This can involve a multiplication factor per pixel (it can also be an index in a palette).

- An image can be interlaced which means that there are seven parts of the image that stepwise build up the whole. In professional workflows with high res images interlacing makes no sense as transfer over the internet is not an issue and the overhead due to reassembling the image and the potentially larger file size (due to independent compression of the seven parts) are not what we want either.

- There can be an image mask that needs to be separated from the main blob. A single byte gray scale image then has two bytes per pixel, and a double byte pixel has four bytes of information. An rgb image has three bytes per pixel plus an alpha byte, and in the case of double byte pixels we get eight bytes per pixel.

- Finally the resulting blob has to be compressed again. The current amount of time involved in that suggests that there is room for improvement.

The process is controlled by number of rows and columns, the number of bytes per pixel (one or two) and the color space which effectively means one or three bytes. These numbers get fed into the filter, deinterlacer, expander and/or mask separator. In order to speed up the embedding these basic operations can be assisted by a helpers written in C. Because Lua is quite good with strings, we pass strings and get back strings. So, most of the logic stays at the Lua end.

4.4 Conclusion

Going for a library-less solution for bitmap inclusion is quite doable and in most cases as efficient. Because we have a pure Lua implementation for testing and an optimized variant for production, we can experiment as we like. A positive side effect is that we can more robustly intercept bad images and inject a placeholder instead.

5 Logging

5.1 Introduction

In ConTEXt we have quite some logging enabled by default and even more when you enable trackers. Most logging is done with Lua, which is quite efficient. Information from the TEX machinery follows a different path and one reason for that is that it often happens on a character (or small strings) basis.

The runtime of a job is, in spite of what one may expect, also dependent on the speed of the console: what fonts are used (there can be font features being applied), is the output buffered, and with what delays, how large is the history, etc. When more complex fonts arrived I found out that on os-x generating a format was impacted by seconds. When on MS Windows the normal console was used its character-by-character flushing made it sluggish, and on linux it depended on the font, kind of console, delays, etc. Lucky me, the SciTE editors log pane beats them all.[21]

At the TEX end a few decades of coding has made the system also complex.[22] Each string goes through a mechanism that checks with line ending to apply and where to cut off lines exceeding a preset maximum length, where LuaTEX also needs to take utf into account. Some characters can (optionally) be escaped with `^^` and occasionally the line length gets reset by explicit newline commands.

In ConTEXt already for a long time we always used an (at least) 10K line length and disabled output escaping. We have consoles that can handle long lines and live in an utf world so escaping makes no sense. And, when OpenType features get applied random line breaks can interfere badly. Just in case one wonders what happens with so called `null` characters: as all goes through C anyway, such a character just terminates a string. Therefore the line length limitations have been removed and the line-ending substitution be optimized. In principle this gives simpler codes and less overhead.

The log is not always compatible with LuaTEX. For instance we output more details about node lists. This is natural because we have more subtypes and these can provide additional information (clues) when debugging TEX code.

In LuaTEX the error handling is already such that some can be delegated to Lua, and later I will look into more isolation. But, error handling is quite interwoven in the code and I don't want to mess up the original concept too much.[23]

---

[21] I use the linux subsystem on MS Windows for cross compiling LuaTEX, and with the advent of that subsystem the regular console was also rewritten so most of the delays are gone now.

[22] Interfaces like that are only partly defined by TEX and left to the implementation.

[23] Indeed the error handling was redone in such a way that we now have an even better isolation.

6 Directions

6.1 Introduction

In LuaTEX the directional model taken from Omega has been upgraded a bit. For instance in addition to the `\*dir` commands we have `\*direction` commands that take a number instead of a keyword. This is a bit more efficient and consistent as using these keywords was kind of un-TEX. Internally direction related nodes (text directions) are not whatsits but first class nodes. We also use a subtype that indicates the push or pop state.

The LuaTEX directional model provides four directions which is a subset of the many that Omega provided, indicated by three letters, like `TRT` and `LTT`. In the beginning we had them all fixed[24] and thereby implemented but being in doubt about their usefulness we dropped most of them, just four were kept. However, in practice only right-to-left makes sense. Going from top to bottom in Japanese or Mongolian can also involve glyph rotation, which actually is not implemented in the engine at all. Spacing and inter-character breaks have to be implemented and in the end one has to combine the results into a page body. So, in practice you end up with juggling node list and macro magic in the page builder. The `LTL` (number 2) and `RTT` (number 3) directions are not used for serious work. Therefore, in LuaMetaTEX the model has been adapted. In the end, it was not entirely clear anyway what the three letters were indicating in each direction property (page, body, par, text, math) as most had no real meaning.

As a side note: if you leave the (not really working well) vertical directions out of the picture, directional typesetting is not that hard to deal with and has hardly any consequences for the code. This is because horizontal dimensions are not affected by direction, only the final ship out is: when a run (wrapped in an hbox) goes the other way, the backend effectively has to skip the width and then with each component goes back. Not much more is involved. This means that a bidirectional engine is rather simple. The complications are more in the way a macro package deals with it, in relation to the input as well as the layout. The backend has to do the real work.[25]

6.2 Two directions

We now have only two directions left: the default left-to-right (l2r) and right-to-left (r2l). They work the same as before and in the backend we can get rid of the fuzzy parallel and rotation (which actually was just stacking nodes) heuristics.

[24] This was doen by Hartmut by rigorously checking all possible combinations

[25] Of course when one hooks in Lua code taking care of direction can be needed!

Reducing the lot to two directions simplifies some code in the engine. This is because when calculating dimensions a change in horizontal direction doesn't influence the width, height and depth in an orthogonal way. Because there are no longer top-down items we don't need to swap the height and or depth with the width. This also means that we don't need to keep much track of direction changes. Technically an hpack doesn't need to know its own direction and we can set it to any value afterwards if we want because the calculation are not influenced by it; so that also simplified matters.

The `\bodydir` and `\pagedir` already didn't make much sense, and in ConTEXt we actually intercepted them, so now they are removed. The body direction is always left-to-right and the page direction was only consulted in the backend code which we no longer have. Another side effect of going with only two directions is that rules no longer need to carry the direction property: there is no flipping of width with height and depth needed.

6.3 Four orientations

Instead of the top-bottom variants we now have four orientations plus a bunch of anchoring options. Of course one could use the backend save, restore and matrix whatsits but a natural feature makes more sense. Let's start with what happens normally:

This line has height and depth. We can rotate this sentence by 180 degrees around the baseline in which case the depth and height are flipped.

or we flip part:

or flip nested:

but we're talking boxes, so the above examples are defined as:

```
This is a \LUAMETATEX\ goodie.

\hbox orientation 2{This is a \LUAMETATEX\ goodie.}

This is a \hbox orientation 2{\LUAMETATEX} goodie.

\hbox orientation 2{This is a \hbox orientation 002{\LUAMETATEX} goodie.}
```

The orientation keyword does the magic here. There are four such orientations with zero being the default. We saw that two rotates over 180 degrees, so one and three are left for up and down.

This is codes as:

```
\hbox orientation 0 {\TEX} and
\hbox orientation 1 {\TEX} and
\hbox orientation 2 {\TEX} and
\hbox orientation 3 {\TEX}
```

The landscape and seascape variants both sit on top of the baseline while the flipped variant has its depth swapped with the height. Although this would be enough a bit more control is possible. The number is actually a three byte hex number:

```
0x<X><Y><O>
```

or in TEX syntax

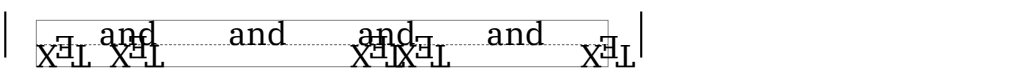
```
"<X><Y><O>
```

We saw that the last byte regulates the orientation. The first and second one deal with anchoring horizontally and vertically. The vertical options of the horizontal variants anchor on the baseline, lower corner, upper corner or center.

```
\hbox orientation "002 {\TEX} and
\hbox orientation "012 {\TEX} and
\hbox orientation "022 {\TEX} and
\hbox orientation "032 {\TEX}
```

The horizontal options of the horizontal variants anchor in the center, left, right, halfway left and halfway right.

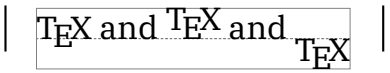
```
\hbox orientation "002 {\TEX} and
\hbox orientation "102 {\TEX} and
\hbox orientation "202 {\TEX} and
\hbox orientation "302 {\TEX} and
\hbox orientation "402 {\TEX}
```



All combinations will be shown on the next pages, so we suffice with telling that for the vertical variants we can vertically anchor on the baseline, top, bottom or center, while horizontally we center, hang left or right, halfway left or right, and in addition align on the (rotated) baseline left or right.

The orientation has consequences for the dimensions so they are dealt with in the expected way in constructing lines, paragraphs and pages, but the anchoring is virtual. As a bonus, we have two extra variants for orientation zero: on top of baseline or below, with dimensions taken into account.

```
\hbox orientation "000 {\TEX} and
\hbox orientation "004 {\TEX} and
\hbox orientation "005 {\TEX}
```



The anchoring can look somewhat confusing but you need to keep in mind that it is normally only used in very controlled circumstances and not in running text. Wrapped in macros users don't see the details. We're talking boxes here, so or instance:

```
test\quad
\hbox orientation 3 \bgroup
    \strut test\hbox orientation "002 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
    \strut test\hbox orientation "002 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
    \strut test\hbox orientation "012 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
    \strut test\hbox orientation "022 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
    \strut test\hbox orientation "032 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
    \strut test\hbox orientation "042 \bgroup\strut test\egroup test%
\egroup
\quad test
```

gives:

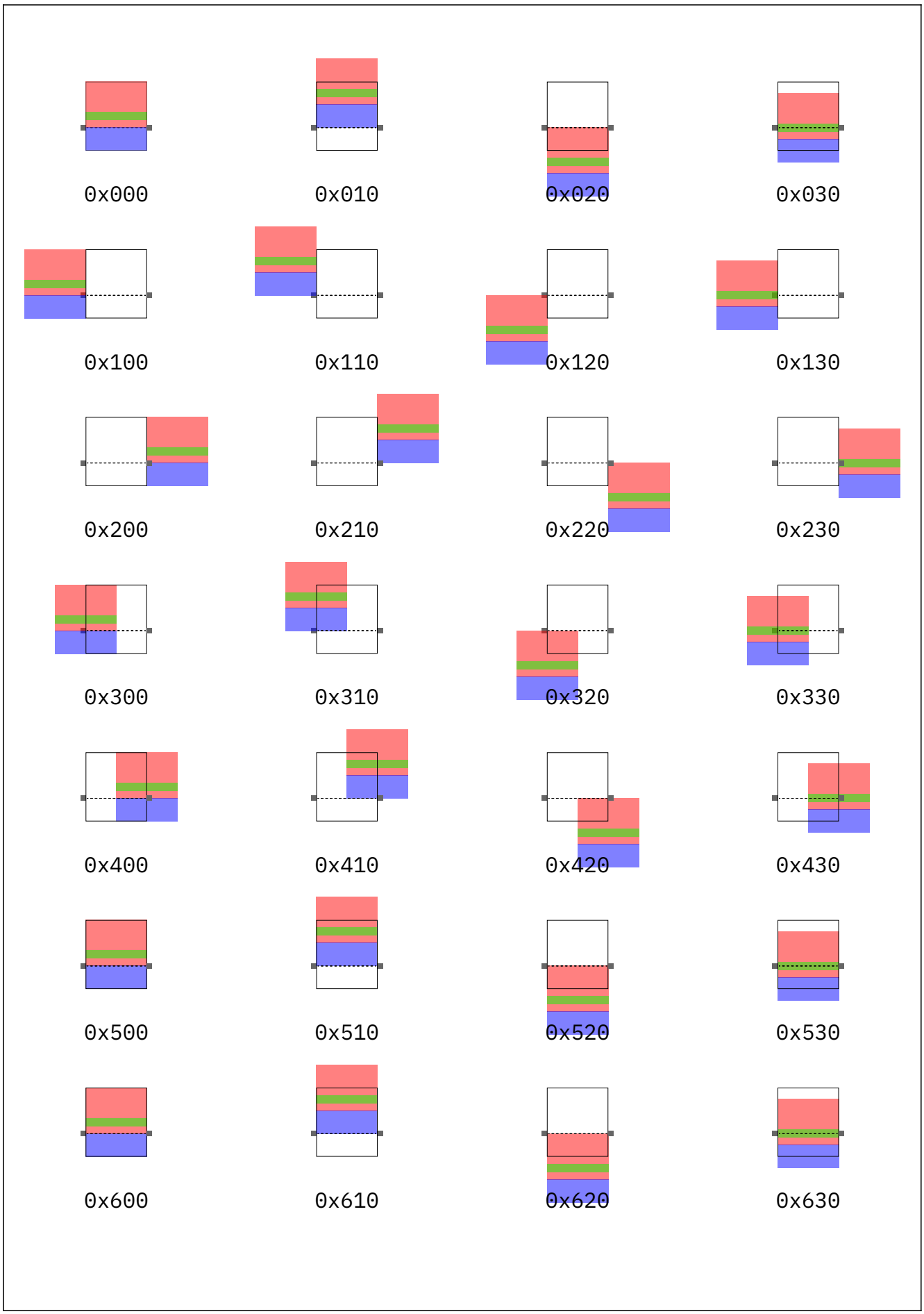


Figure 6.1 orientation 0

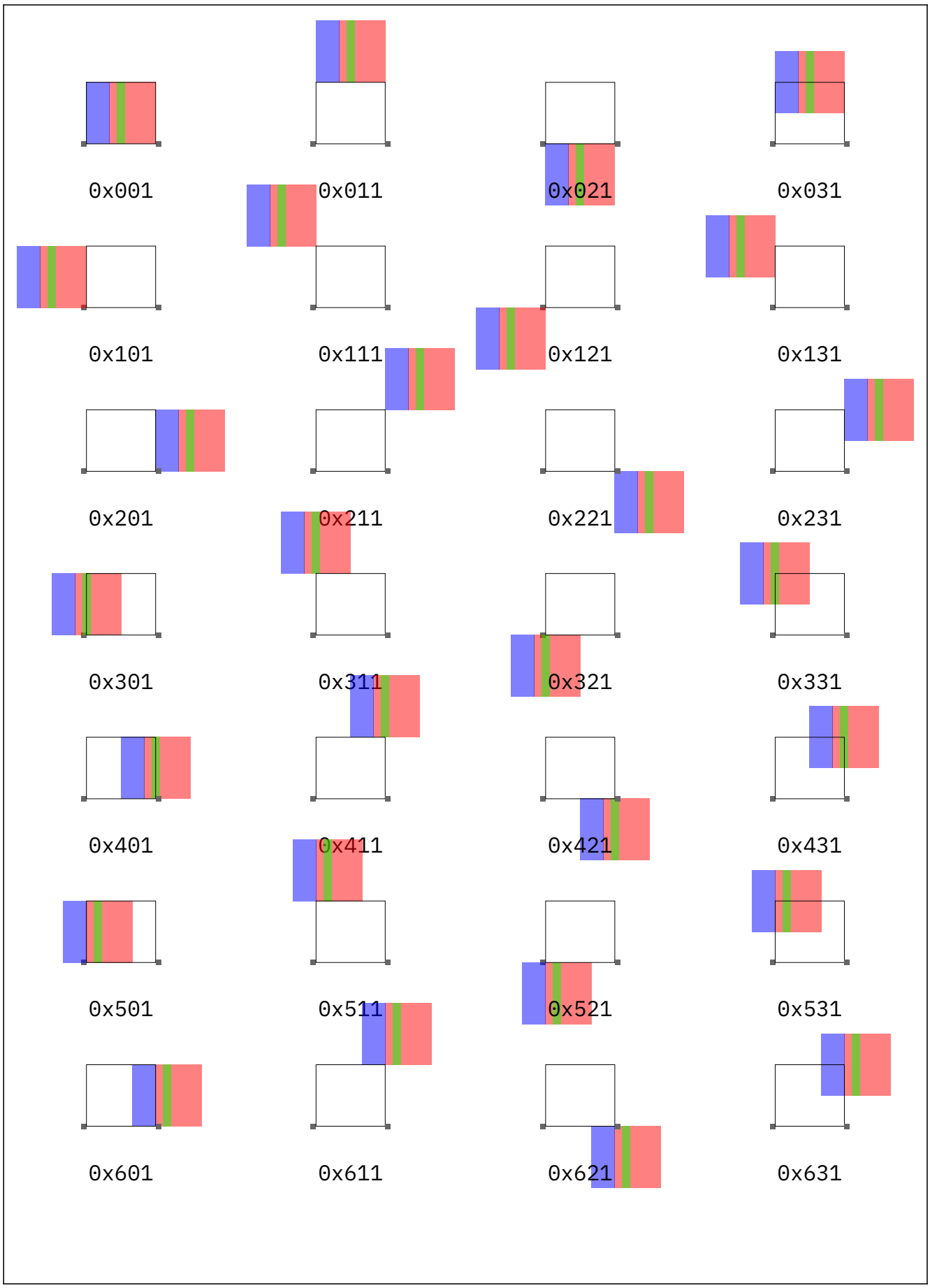


Figure 6.2 orientation 1

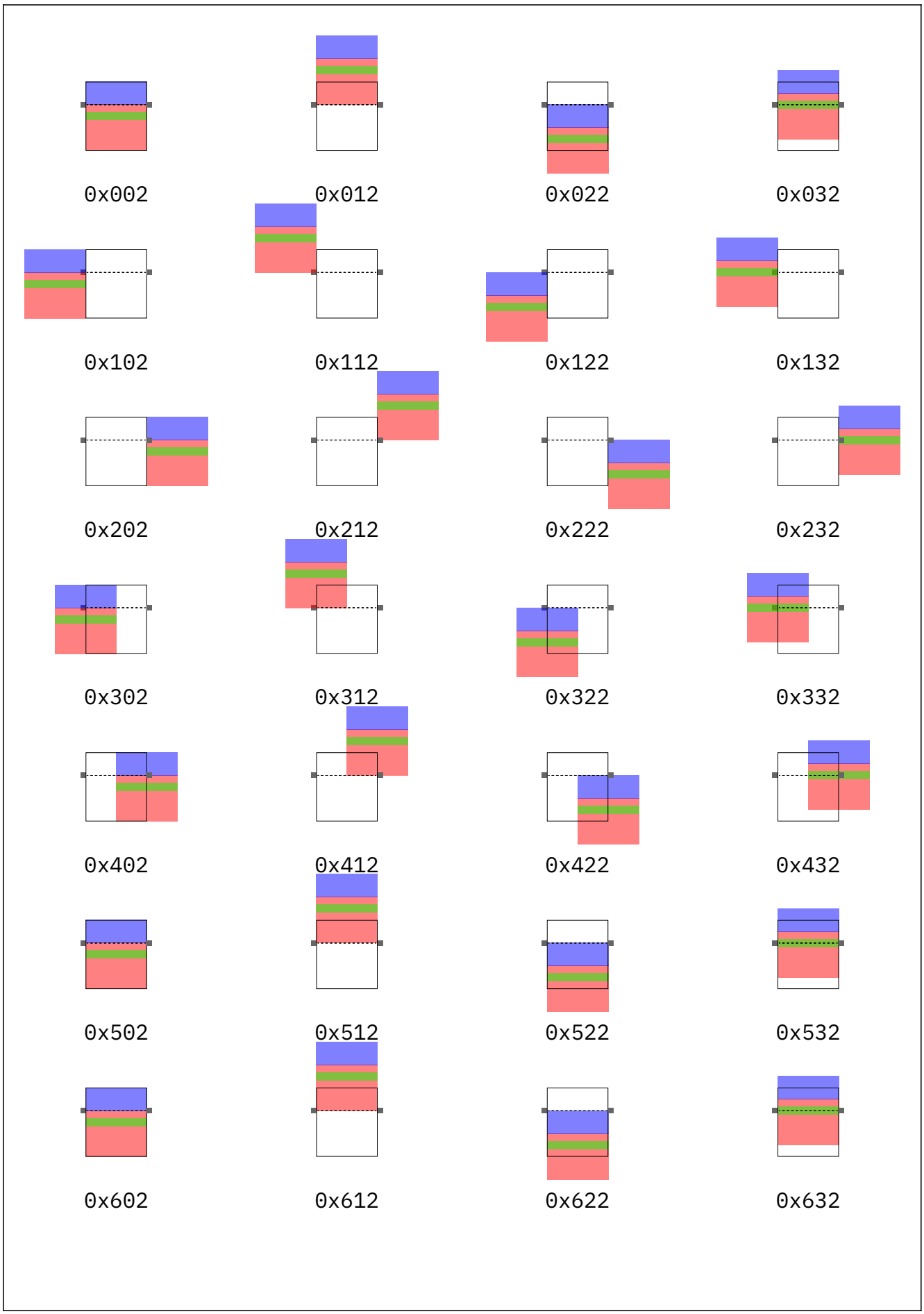


Figure 6.3 orientation 2

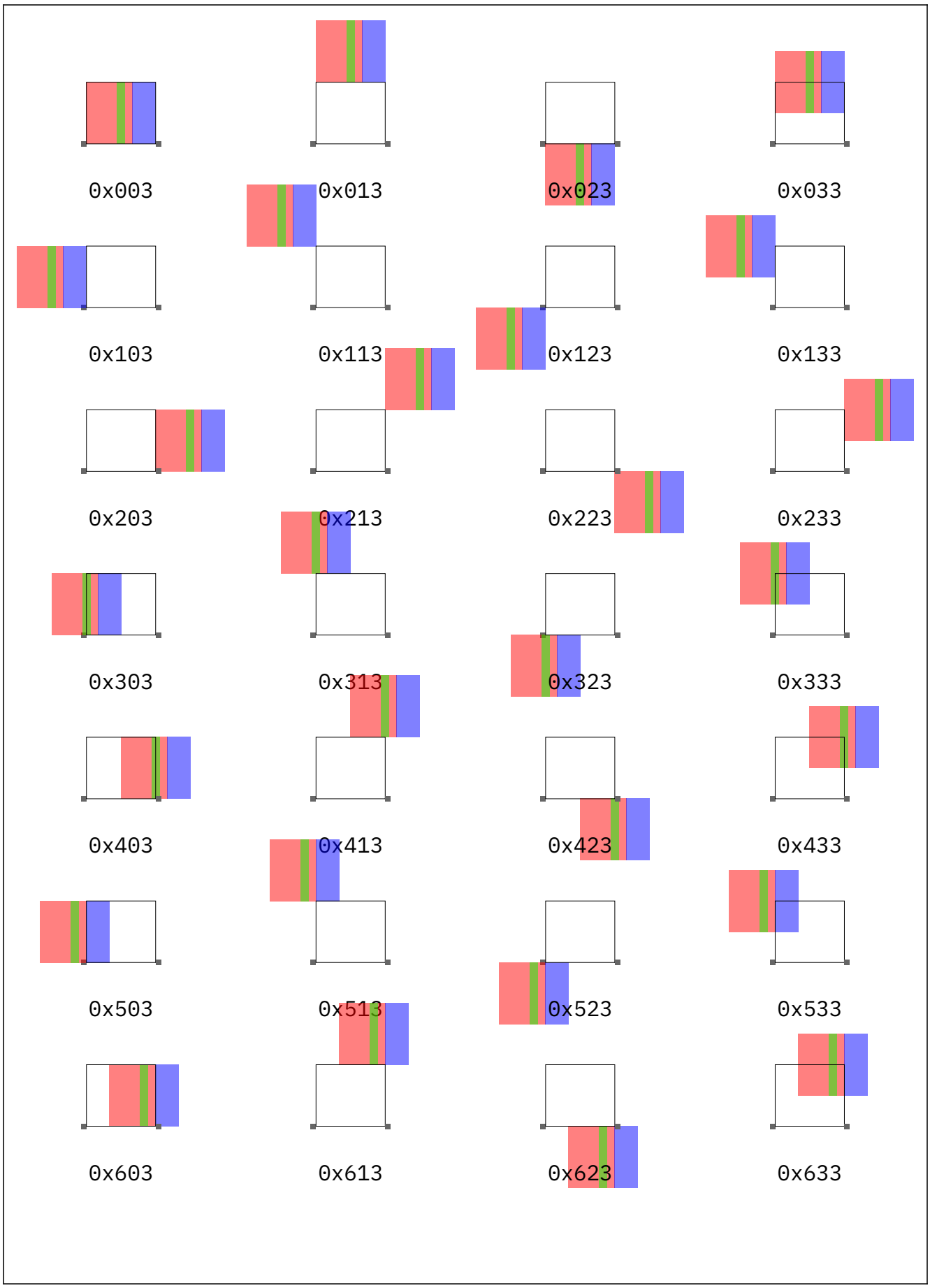


Figure 6.4 orientation 3

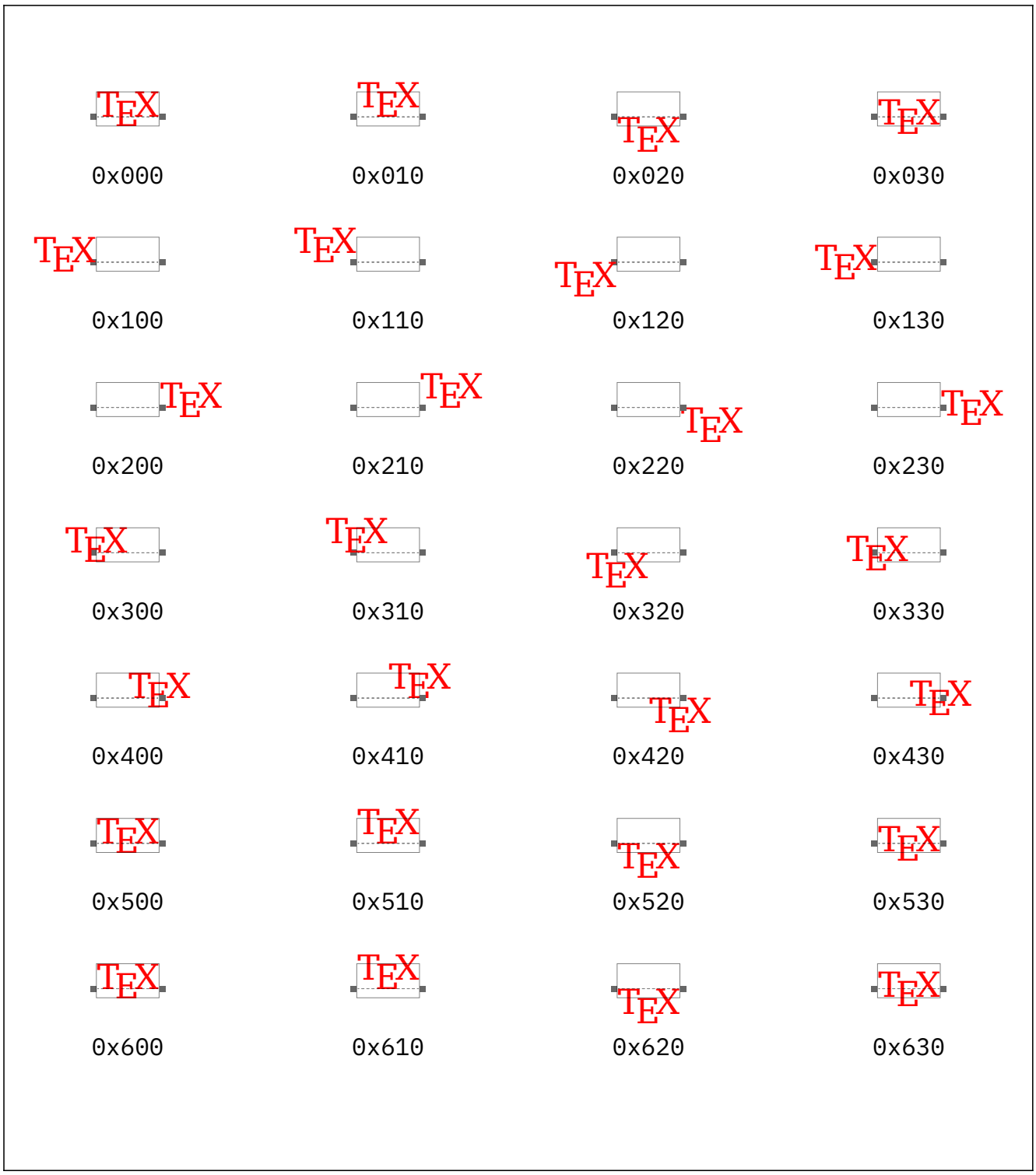
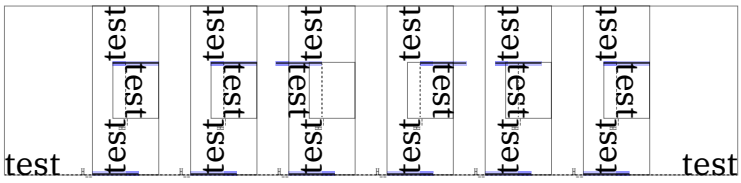


Figure 6.5 orientation 0



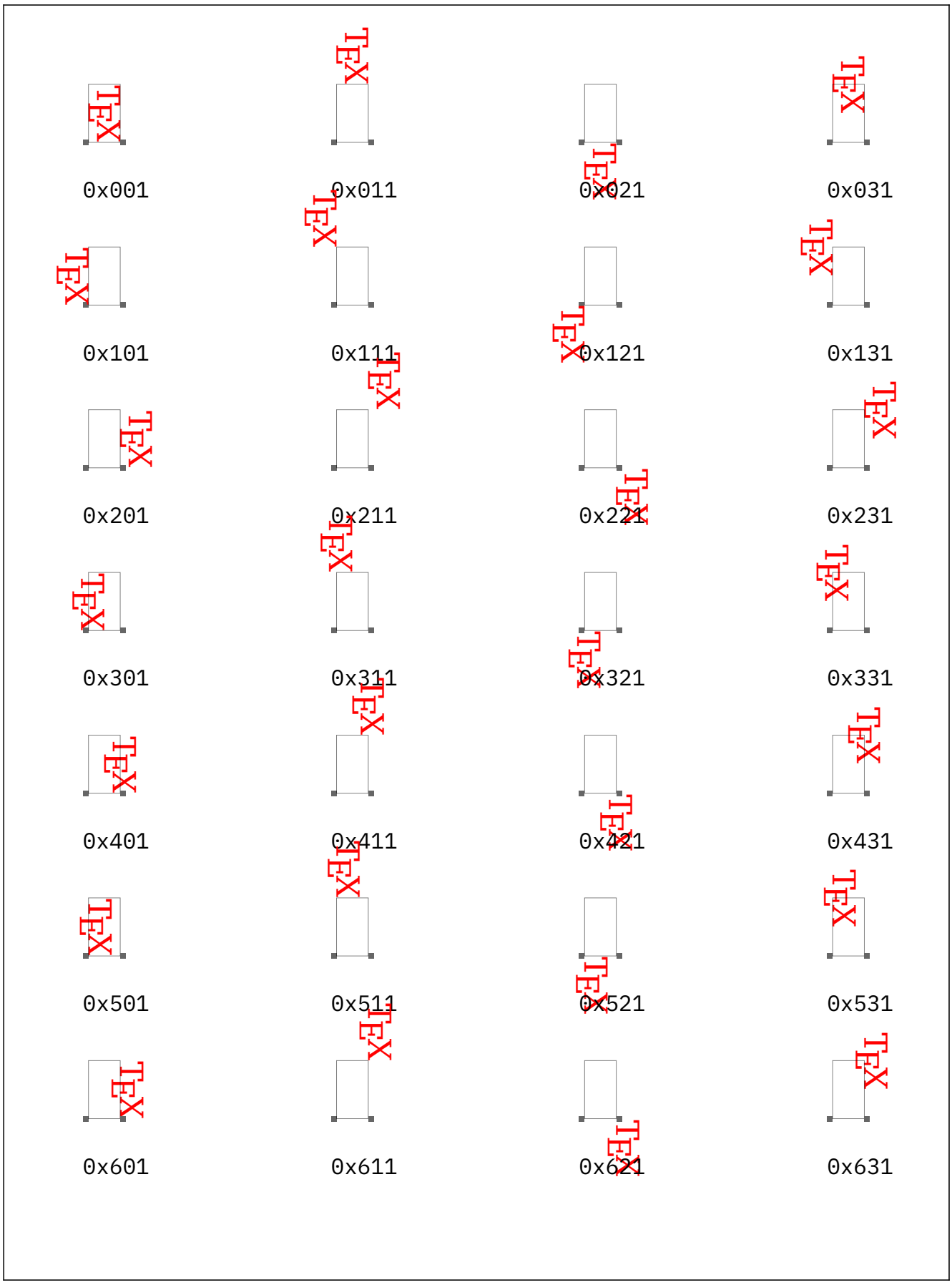


Figure 6.6 orientation 1

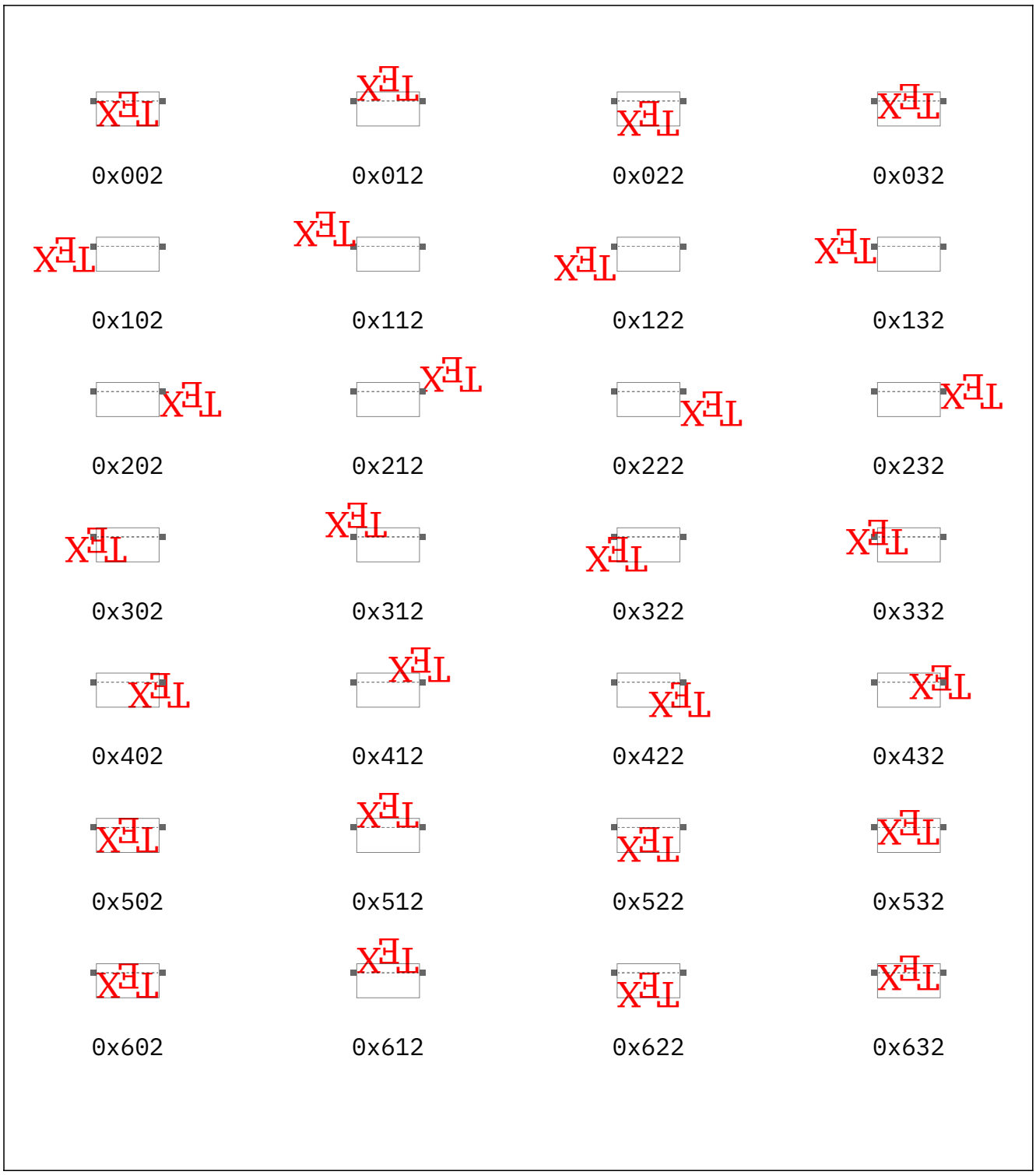


Figure 6.7 orientation 2

6.4 Right-to-left typesetting

Another aspect to keep in mind when we transform is the already mentioned right-to-left direction. We show some examples where we do things like this:

```
\hbox{\hbox
    orientation #1
```

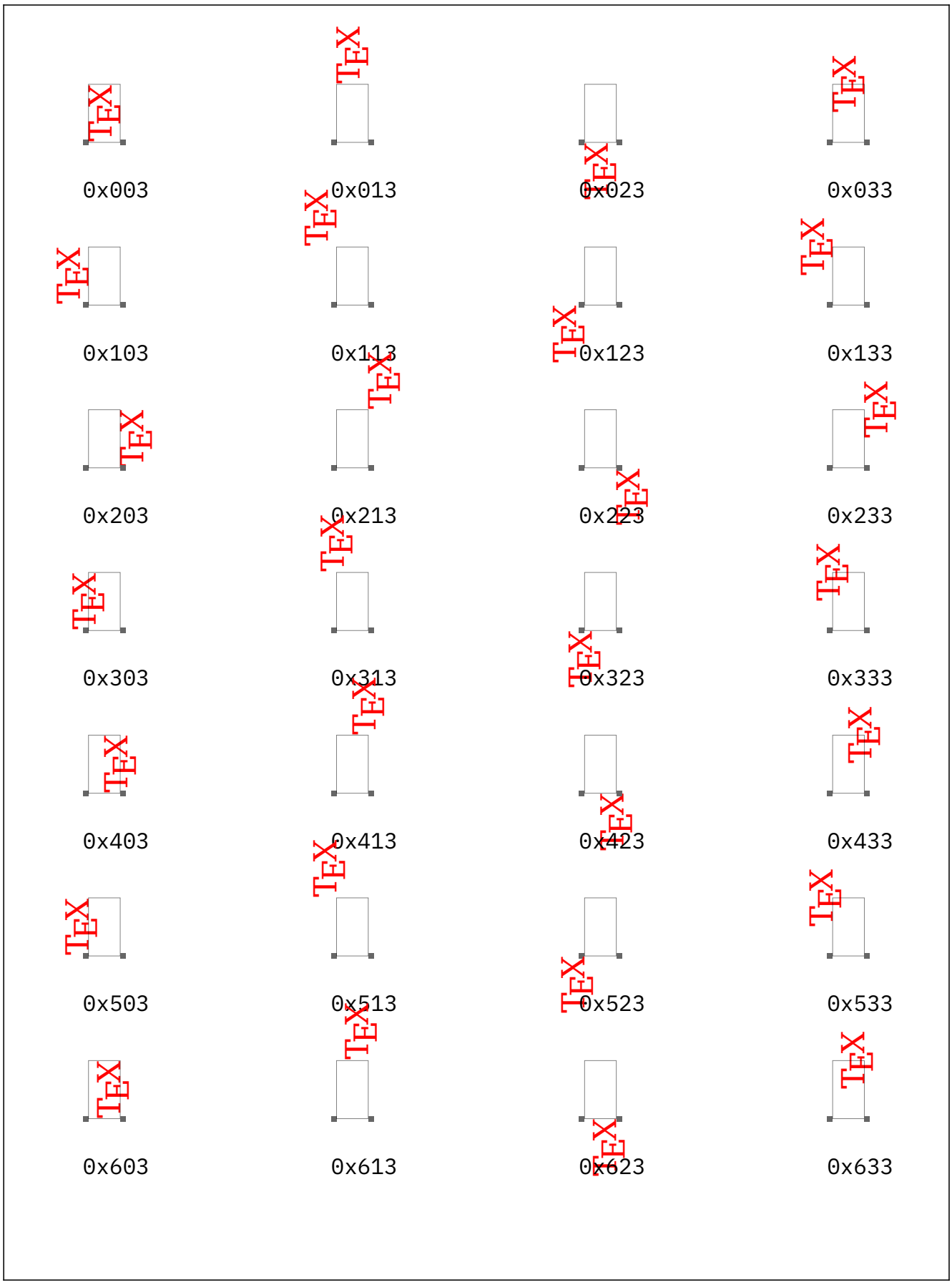


Figure 6.8 orientation 3

```
    {\strut abcd}}
\hbox{\hbox
    orientation #1
    to 15mm
    {\strut abcd}}
\hbox{\hbox
    orientation #1
    direction 1
    {\righttoleft\strut abcd}}
\hbox{\hbox
    orientation #1
    direction 1
    to 15mm {\righttoleft\strut abcd}}
```

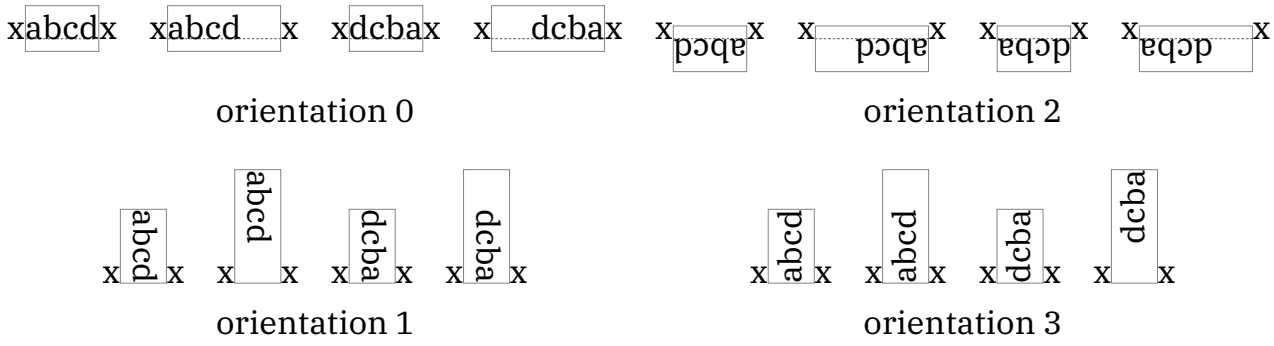


Figure 6.9 Horizontal boxes.

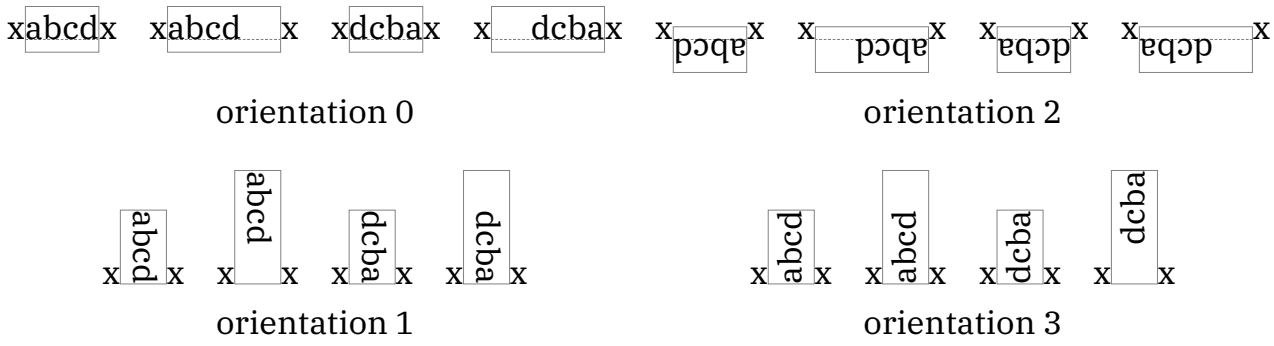


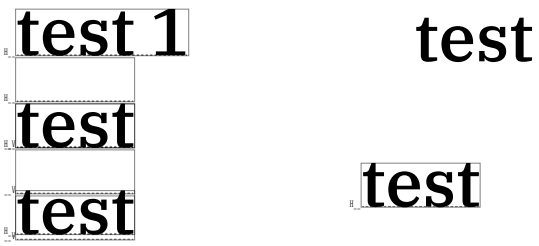
Figure 6.10 Vertical boxes.

6.5 Vertical typesetting

I'm no expert on vertical typesetting and have no application for it either. But from what I've seen vertically positioned glyphs are normally used in rather straightforward situations. Here I will just give some examples of how transformations can be used to achieve certain effects. It is no big deal to make macros or use Lua to apply magic to node lists but it is beyond this description to discuss that.

Before we fine tune this example we have to discuss another feature. When a `orientation` keyword is given optionally `xoffset` and `yoffset` can be specified. These offsets are *not* taken into account when calculating dimensions. This is different from the offsets (at the Lua end) used in glyphs because there the vertical offset is taken into account. Here are some examples of offsets in packaged lists:

```
\hbox
    {test 1}
\hbox
    orientation 0
    yoffset  15pt
    xoffset 150pt
    {test}
\vbox
    orientation 0
    {\hbox{test}}
\vbox
    orientation 0
    yoffset  -5pt
    xoffset 130pt
    {\hbox{test}}
\vbox
    orientation 0
    yoffset 2pt
    {\hbox{test}}
```



In order to demonstrate some hacking, we first define a font that supports chinese glyphs:

```
\definefont[NotoCJK][NotoSansCJKtc-Regular*default @ 24pt]
```

We put some text in a horizontal box; it doesn't show up in verbatim but you get the idea nevertheless:

```
\hbox{\NotoCJK }
```

通用规范汉字表

Let's now rotate this line of text:

```
\hbox orientation 1 {\NotoCJK }
```

The result is shown in a while. Because we also need to rotate the glyphs we deconstruct the box.

```
\hbox orientation 1 \bgroup \NotoCJK %
    \vbox {\hbox {}}%
    \vbox {\hbox {}}%
    \vbox {\hbox {}}%
    \vbox {\hbox {test}}%
    \vbox {\hbox {}}%
    \vbox {\hbox {}}%
    \vbox {\hbox {}}%
    \vbox {\hbox {}}%
\egroup
```

Next we rotate the glyphs.

```
\hbox orientation 1 \bgroup \NotoCJK %
    \vbox orientation 3 {\hbox {}}%
    \vbox orientation 3 {\hbox {}}%
    \vbox orientation 3 {\hbox {}}%
    \vbox orientation 0 {\hbox {test}}%
    \vbox orientation 3 {\hbox {}}%
    \vbox orientation 3 {\hbox {}}%
    \vbox orientation 3 {\hbox {}}%
    \vbox orientation 3 {\hbox {}}%
\egroup
```

This still looks bad so we kick in some offsets and glue:

```
\dontleavehmode\hbox orientation 1 \bgroup \NotoCJK
    \vbox
        orientation 0 yoffset -.1ex
        {\hbox orientation 3 {}}\hskip.2ex
    \vbox
        orientation 0 yoffset -.1ex
        {\hbox orientation 3 {}}\hskip.2ex
    \vbox
```

```
        orientation 0 yoffset -.1ex
        {\hbox orientation 3 {}}\hskip.6ex
    \vbox
        {\hbox             {test}}\hskip.2ex
    \vbox
        orientation 0 yoffset -.1ex
        {\hbox orientation 3 {}}\hskip.2ex
    \vbox
        orientation 0 yoffset -.1ex
        {\hbox orientation 3 {}}\hskip.2ex
    \vbox
        orientation 0 yoffset -.1ex
        {\hbox orientation 3 {}}\hskip.2ex
    \vbox
        orientation 0 yoffset -.1ex
        {\hbox orientation 3 {}}\hskip.2ex
\egroup
```

Now we're ready to compare the results

This could of course also be done with traditional kerns, raising and/or lowering and messing around with dimensions. It's just that when manipulating such rather complex constructs a little help (and efficiency) makes a difference, also at the Lua end. Of course one can argue the result but all is programmable in the end.

6.6 Considerations

Just in case you wonder if using these offsets is better than using normal kerning and shifting, in practice offsets are not more efficient. Let's compare the alternatives. We go from most to least efficient.

```
\setbox\scratchbox\hpack{}
\boxxoffset\scratchbox\scratchdimen
\boxyoffset\scratchbox\scratchdimen
```

This sets the offsets and by setting them we also trigger the transform. Scanning is fast and so is setting them. One million times expanding this takes (as reference) 0.73 seconds on my current machine.

```
\setbox\scratchbox\hpack
  orientation \zerocount
  xoffset     \scratchdimen
  yoffset     \scratchdimen
  {}
```

This takes a bit more time, 1.11 seconds, because the keywords have to be scanned which happens on a token by token base.

```
\setbox\scratchbox\hpack{}
\scratchheight\ht\scratchbox
\scratchdepth\dp\scratchbox
\setbox\scratchbox\hpack
  {\kern\scratchdimen
   \raise\scratchdimen\box\scratchbox
   \kern\scratchdimen}
\ht\scratchbox\scratchheight
\dp\scratchbox\scratchdepth
```

Now we're up to 1.69 seconds for the million expansions. Not only do we have some parsing going on, but we also have assignments and extra packing, which means calculations taking place.

```
\setbox\scratchbox\hpack{}
\scratchwidth\wd\scratchbox
\scratchheight\ht\scratchbox
\scratchdepth\dp\scratchbox
\setbox\scratchbox\hpack
  {\kern\scratchdimen
```

```
   \raise\scratchdimen\box\scratchbox}
\wd\scratchbox\scratchwidth
\ht\scratchbox\scratchheight
\dp\scratchbox\scratchdepth
```

This variant is about as fast, as I measured 1.72 seconds. So, compared to the 0.73 seconds for the first variant, is this better? Does it help when we look at our existing macros and adapt them?

Normally we don't have an empty box and normally we use `\hbox` because we want the content to be processed. And a million times building a list and processing content (which means runs over the list) will make the differences in timing become noise. Add to that garbage collection (in Lua) and memory management (in TEX) and it even becomes unpredictable. Seeing differences of a factor two in such timings is no exception.

Another aspect is the parsing. When these commands are wrapped in macros we're talking expanding tokens which is pretty fast. When it comes from the input file a conversion to tokens has to happen too. And we will never see millions of such sequences in a source file.

The backend also plays a role. Handling a kern or shift is more efficient than analyzing transforms (and offsets) especially in a Lua variant. But on the other hand, we don't have an extra wrapping in a box so that actually saves work.

So, before a ConTEXt user thinks “Let's update macros and change policy.”, just consider staying with proven good old TEX approaches. These features are mostly meant for efficient low level manipulations as discussed in relation to for instance handling scripts. In the rather large ConTEXt code base there are really only a few places where it will make code look nicer, but there I don't expect an impact on performance.

6.7 Integration

How these mechanisms are used depends on ones needs and the macro package used. It makes no sense to cook up generic solutions because integration in a macro package is too different. But anyhow we'll give an example of some (definitely non optimized) Lua magic.

```
\startluacode
local glyph_id = node.id("glyph")
local fontdata = fonts.hashes.identifiers -- assumes generic font
loader
```

```
local function is_vertical(c)
  -- more ranges matter but this will do here
  return c >= 0x04E00 and c <= 0x09FFF
end

function document.go_vertical(boxnumber)
  local box = tex.getbox(boxnumber)
  local n   = box.list
  while n do
    if n.id == glyph_id and is_vertical(n.char) then
      local o = .2 * fontdata[n.font].parameters.xheight
      local prev, next = n.prev, n.next
      n.next, n.prev = nil, nil
      local l = nodes.new("hlist")
      l.list = n
      local w, h, d = n.width, n.height, n.depth
      if prev then
          prev.next, l.prev = l, prev
      else
          box.list = l
      end
      if next then
          l.next, next.prev = next, l
      end
      l.width, l.height, l.depth  = h + d + o, w, 0
      l.orientation = 0x003
      l.xoffset, l.yoffset = o/2, -o/2
      l.hoffset, l.doffset = h, d - o
      n = next
    else
      n = n.next
    end
  end
end
\stopluacode
```

We will use some other magic that we won't discuss here which relates to handling scripts. For Hangul one needs to inject breakpoints and if needed also glue between characters. The script environment does this. We also need to bump the interline spacing. First we define a regular text helper and an auxiliary box.

```
\unexpanded\def\stripe#1%
```

```
  {\hbox orientation 0 yoffset .2\exheight{\strut #1}}

\newbox\MyVerticalBox
```

Next we fill that box with some mix of text (I have no clue what, as I just copied it from some web page).

```
\setbox\MyVerticalBox\hbox \bgroup
    \NotoCJK
    \startscript[hangul]%
    \dorecurse{20}{ \stripe{test #1} }%
    \unskip % remove last space
    \stopscript
\egroup
```

We then apply the Lua magic to the result:

```
\ctxlua{document.go_vertical(\number\MyVerticalBox)}
```

and finally assemble the result:

```
\ruledvbox orientation 1 to \textwidth \bgroup
    \setupinterlinespace[40pt]
    \hsize .95\textheight
    \unhbox\MyVerticalBox
    \vfill
\egroup
```

The result is shown in figure 6.11. Of course this approach is not that user friendly but it just serves as example. In ConTEXt we can follow a different route. First we define a new font feature. It is probably clear that we need some code elsewhere that does something useful with this information, but I will nos show this as it is rather ConTEXt dependent.

```
\definefontfeature
  [vertical]
  [vertical={%
    orientation=3,%
    down=.1,%
    right=.1,%
    ranges={%
      cjkcompatibility,%
      cjkcompatibilityforms,%
      cjkcompatibilityideographs,%
```

```
      cjkcompatibilityideographssupplement,%
      cjkradicalssupplement,%
    % cjkstrokes,%
      cjksymbolsandpunctuation,%
      cjkunifiedideographs,%
      cjkunifiedideographsextensiona,%
      cjkunifiedideographsextensionb,%
      cjkunifiedideographsextensionc,%
      cjkunifiedideographsextensiond,%
      cjkunifiedideographsextensione,%
      cjkunifiedideographsextensionf,%
    }%
  }]
```

We apply this feature to a font:

```
\definefont
  [NotoCJKvertical]
  [NotoSansCJKtc-Regular*default,vertical @ 24pt]

\setbox\MyVerticalBox\hbox\bgroup
    \NotoCJKvertical
    \startscript[hangul]%
    \dorecurse{20}{ \stripe{test #1} }%
    \unskip
    \stopscript
\egroup

\ruledvbox orientation 1 to \textwidth \bgroup
    \setupinterlinespace[40pt]
    \hsize .95\textheight
    \unhbox\MyVerticalBox
    \vfill
\egroup
```

The result is shown in figure 6.12. Again this approach is not that user friendly but it already is a bit easier.

通用规范汉字表 test 1 通用规范汉字表 test 2 通用规范汉字表 test 3
通用规范汉字表 test 4 通用规范汉字表 test 5 通用规范汉字表 test 6
通用规范汉字表 test 7 通用规范汉字表 test 8 通用规范汉字表 test 9
通用规范汉字表 test 10 通用规范汉字表 test 11 通用规范汉字表
test 12 通用规范汉字表 test 13 通用规范汉字表 test 14 通用规范汉
字表 test 15 通用规范汉字表 test 16 通用规范汉字表 test 17 通用规
范汉字表 test 18 通用规范汉字表 test 19 通用规范汉字表 test 20

Figure 6.11 Some vertical magic using manipulations.

通用规范汉字表 test 1 通用规范汉字表 test 2 通用规范汉字表 test 3 通用规范汉字表 test 4 通用规范汉字表 test 5 通用规范汉字表 test 6 通用规范汉字表 test 7 通用规范汉字表 test 8 通用规范汉字表 test 9 通用规范汉字表 test 10 通用规范汉字表 test 11 通用规范汉字表 test 12 通用规范汉字表 test 13 通用规范汉字表 test 14 通用规范汉字表 test 15 通用规范汉字表 test 16 通用规范汉字表 test 17 通用规范汉字表 test 18 通用规范汉字表 test 19 通用规范汉字表 test 20

Figure 6.12 Some vertical magic using fonts.

7 Performance

7.1 Introduction

Those who've read the other documents describing the development of LuaTEX, know that performance is always on my radar. A decent performance is a must for a useable workflow, especially because typesetting is a multi-pass process.[26] One page reference changing from two digits to three digits can influence whatever follows and we're not only talking of a different page break, even a change in line breaks can have consequences. The core engine cannot be made much faster. When the (single core) run has the whole cpu available not much can be gained. But multiple processes are run at the same time, the cache has to be shared and misses can become an issue. So, efficiency of code is still important. Occasionally a (tiny) improvement can be made, but only the accumulation of such improvements can make a dent. The feeling is that over time LuaTEX has not become slower but we keep an eye on possible other improvements. The memory footprint is also something to keep an eye on.[27]

The more we delegate to Lua, the less we can benefit from for instance cpu improvements: in that case the Lua virtual machine is the bottleneck. And there is not much we can do about that. This also means that when we delegate more to Lua we sacrifice performance. Sometimes things can be done more efficient in Lua, but those are often tasks that are not performed frequently. That said, I'm convinced most of the ConTEXt code is quite efficient and not much can be gained.

The biggest change in LuaMetaTEX is the backend. We gain some efficiency in terms of speed, performance and output in some cases, while in other cases we loose a bit. On the average the small performance hit is bearable. Because ConTEXt users don't complain about performance I think that I have some slack here.

7.2 An example

There are a few places where LuaTEX looks ahead to check something and goes back when the condition is not met. Take these:

---

[26] I'm often baffled by reports of (non-ConTEXt) LuaTEX users about the performance of LuaTEX. It seems easier to blame an engine than ones own macros or setup and most of those tests make no sense anyway. Believe it or not, but if performance of ConTEXt MkIV was much worse than MkII (using pdfTEX or XETEX) it would have backfired and the project would never have taken of. Just think of this: would Hans really use LuaTEX and continue with development if it were that slow?

[27] Of course this is all becoming less relevant now that having e.g. a browser open in the background will set you back with a constant 5–10% cpu load and slowly accumulating gigabyte memory usage. That actually was something I had to keep in mind when running LuaMetaTEX benchmarks.

```
\hbox   {...}
\hbox   to 10cm {...}
\hrule  width 10cm height 10cm \relax
\dimen0 =10cm
\dimen0 10cm
\mydimen 10cm
\toks0  {...}
\toks0  \toks2
```

Spaces and sometimes `\relax` after the trigger (`\hbox`, `\dimen`, etc.) are skipped and in some case there can be an optional `=` sign. So, there are quite some cases where there is first a check for an optional equal which itself can be preceded by optional spaces. When there is no equal sign the last seen token is pushed back into the scanned which effectively means that a temporary token is allocated, and a one token list is pushed on the input stack. Then scanning goes on. The same can happen with the open brace in case of a token list assignment: it gets pushed back and the content scanned checks it again. In the case of keywords something similar takes place, because here LuaTEX checks explicitly for e.g. type width, and when it is not found again it pushes back consumed tokens and checks for the `width`. In the case of the specifiers of the box we don't need to check at all when we have an opening brace. In the follow up, when the `orientation` keyword was added, and the `dir` and `bdir` were replaced by `direction` a little bit more was optimized.

In LuaTEX this code comes from pdfTEX which takes if from TEX, but in both cases some code side effects occur from the transition from Pascal to C. But, in LuaTEX we stick to the C, so we can try to get rid of these artifacts. During the last years, especially when additional keywords were introduces (for instance for attributes) already some optimization took place. In the follow up again some optimizations were applied, for instance quite often we can combine the check for an equal sign with skipping the spaces.

The gain is not spectacular but as all small bits add up eventually it is measurable in a complex run. What definitely is true, is that we avoid some memory access which in turn might pay back when multiple runs happen in parallel.

Of course one can argue that such optimizations are to be avoided but as long as they don't obscure the code, it's okay. After all, just as one optimizes for instance a compression algorithm or search routine, there is no reason not to mildly optimize some of the critical code in LuaTEX. And in ConTEXt we have plenty of opportunities to check if that works out well. At some point some might be retrofit into LuaTEX 1.2 (or later).[28]

[28] But it makes less sense now that there are variants popping up that might depend on the stable base.

8 Cleanup

8.1 Introduction

Original TEX is a literate program, which means that code and documentation are mixed. This mix, called a web, is split into a source file and a TEX file and both parts are processed independently into a program (binary) and a typeset document. The evolution of TEX went through stages but in the end a Pascal web file was the result. This fact has lead to the more or less standard web2c compilation infrastructure which is the basis for TEXLive.

So, TEX is a woven program and this is also true for the starting point of LuaTEX: pdfTEX. But, because we wanted to open up the internals, and because Lua is written in C, already in an early stage Taco decided to start from the C translated from Pascal. A permanent conversion was achieved using additional scripts and the original documentation stayed in the source. The one large file was split into more logical smaller parts and combined with snippets from Aleph.

After we released version 1.0 I went through the documentation parts of the code and normalized that a bit. The at that moment still sort of simple web files became regular C files, and the idea was (and is) that at some point it should be possible to process the documentation (using ConTEXt).

Over time the C code evolved and functions ended up in places that at that made most sense at that moment. After the previously described stripping process, I decided to go through the files and see if a bit of reshuffling made sense, mostly because that would make documenting easier. (I'm not literate enough to turn it into a proper literate program.) It was also a good moment to get rid of unused code (not that much) and unused macros (some more than expected). It also made sense to change a few names (for instance to avoid potential future clashes with `lua_` core functions). However, all this takes quite some careful checking and compilation runs, so I expect that after this first cleanup, for quite some time stepwise improvements can happen (especially in adding comments).[29] [30]

One of the things that I keep in mind when doing this, is that we use Lua. This component compiles on most relevant platforms and as such we can assume that LuaMetaTEX also

[29] This is and will be an ongoing effort. It probably doesn't show, but getting the code base in the state it is in now, took quite some time. It probably won't take away complaints and nagging but I've decided no longer to pay attention to those on the sideline.

[30] In the end not much pdfTEX and Aleph code is present in LuaMetaTEX, but these were useful intermediate steps. No matter how lean LuaMetaTEX becomes, I have a weak spot for pdfTEX as it always served us well and without it TEX would be less present today.

should (and can be) made a bit less dependent on old mechanisms that are used in stock LuaTEX. For instance, we don't come from Pascal any longer but there are traces of that transition still present. We also don't use specific operating system features, and those that we use are also used in Lua. And, as we try to share code we can also delegate some (more) to Lua. For instance file related code is not dependent on other components in the TEX infrastructure, but maybe at some point the runtime loadable kpse library can kick in. So, basically the idea is to sort of go bare bone first and later see how with the help of Lua we can get bring some back. For the record: this is not needed for ConTEXt as it already has this interface to tds.[31]

8.2 Motivation

The LuaTEX project started as an experiment of adding Lua to pdfTEX, which was done by Hartmut and in order to avoid confusion we named it LuaTEX. When we figured out that there this had possibilities we decided to go further and Taco took the challenge to rework the code base. Part of that work was sponsored by Idris' Oriental TEX project. I have fond memory of the intensive and rapid development cycles: online discussions, binaries going my directions, experimental ConTEXt code going the other way. When we had reached a sort of stable state but at some point, read: usage in ConTEXt had become crucial, a steady further development started, where Taco redid MetaPost into mplib, funded by user groups. At some point Luigi took over from Taco the task of integration of components (also into TEXLive), introduced LuaJIT into the binary, conducted the (again partially funded) swiglib project, followed by support for ffi. A while later I myself started messing around in the code base directly and continued extending the engine and Lua interfaces.

I could work on this because I have quite some freedom at the place where I work. We use (part of) ConTEXt for some projects and especially in dealing with xml we could benefit from LuaTEX. It must be said that (long running) projects like these never pay off (on the contrary, they cost a lot in terms of money and energy) so it's quite safe to conclude that LuaTEX development is to a large extend a (many man years) work of love for the subject. I guess that no sane company will do (permit) such a thing. It is also for that reason that I keep spending time on it, and as a simplification of the code base was always one of my dreams, this is what I spend my time on now. After all, LuaTEX is just juggling bytes and as it is written in C, and has no graphical user interface or complex dependencies, it should be possible to have a relative simple setup in terms of code files and compilation. Of course this is also made possible by the fact that I can use Lua. It's also why I decided to “Just do it”, and then “Let's see where I end up”. No matter how it turns out, it makes a good vehicle for further development and years of fun.

[31] This has been removed from my agenda.

8.3 Files

After a decade of adding and moving around code it's about time to reorganize the code a bit, but we do so without deviating too much from the original setup. For instance we started out with a small number of Lua interface macros and these were collected in a few files, and defined in one `h` file, but it made sense to have header files alongside the libraries that implement helpers. This is a rather tedious job but with music videos or video casts on a second screen it is bearable.

When I reached a state where we only needed the LuaTEX files plus the minimal set of libraries I tried to get rid of directories in the source tree that were placeholders, but with `automake` files, like those for pdfTEX and XETEX. After a couple of attempts I gave up on that because the build setup is rather hard coded for checking them. Also, there were some (puzzling) dependencies in the configuring on Omega files as well as some dvi related tools. So, that bit is for later to sort out.[32]

8.4 Command line arguments

As we need to set up a backend and deal with font loading in Lua, we can as well delegate some of the command line handling to Lua as well. Therefore, only the a limited set of options is dealt with: those that determine the startup and Lua behavior. In principle we can even get rid of all and always use a startup script but for now it makes sense to not deviate too much from a regular TEX run.

At the time of this writing some code is still in place that is a candidate for removal. For instance, using the `&` to define a format file has long be replaced by `--fmt`. There are sentimental reasons for keeping it but at the same time we need to realize that shells use these special characters too. A for me unknown (or forgotten) feature of prefixing a jobname with a `*` will be removed as it makes no sense. There is some MS Windows specific last resort code that probably will go too, unless I can figure out why it is needed in the first place.[33]

Now left with a very simple set of command line options it also makes sense to use a simple option analyzer, so that was a next step as it rid us of a dependency and produces less code.

So, the option parser has now been replaced by a simple variant that is more in tune with what will happen when you deal with options in Lua: no magic. One problem is

[32] Of course later the decision was made to forget about using `autotools` and go for an as simple as possible `cmake` solution.

[33] Intercepting these symbols has been dropped in favor of the command line flags.

that TEX's first input file is moved from the command line to the input buffer and a an interactive session is emulated. As mentioned before, there is some extra `&`, `*` and `\\` parsing involved. One can wonder if this still makes sense in a situation where one has to specify a format and Lua file (using `--fmt` and `--ini`) so that might as well be redone a bit some day.[34]

8.5 Platforms

When going through the code I noticed conditional sections for long obsolete platforms: `amiga`, `dos` and `djgpp`, `os/2`, `aix`, `solaris`, etc. Also, with 64 bit becoming the standard, it makes sense to assume that users will use a modern 64 platform (intel or arm combined with MS Windows or some popular Unix variant). We don't need large and complex code management for obscure platforms and architectures simply because we want to proof that LuaMetaTEX runs everywhere. With respect to MS Windows we use a cross compiler (`mingw`) as reference but native compilation should be no big deal eventually. We can cross that bridge when we have a simplified compilation set up. Right now it doesn't make sense to waste time on a native Microsoft compilation as it would also pollute the code with conditional sections. We'll see what happens when I'm bored.[35]

8.6 Stubs

A ConTEXt run is managed by mtxrun in combination with a specific script

```
mtxrun --script context
```

On windows, we use a stub because using a `cmd` file create an indirectness that is not seen as executable and therefore in other command files needs to be called in a special way to guarantee continuation. So, there we have a small binary:

```
mtxrun.exe ...
```

that will call:

```
luatex --luaonly mtxrun.lua ...
```

And when the stub has a different name than `mtxrun`, say:

[34] In the end only these explicit command line options were supported.

[35] In the meantime no effort is made to let the source compile otherwise than with the cross compiler. Best is to keep the code as clean as possible with respect to conditional code sections. So don't bother me with patches.

```
context.exe ...
```

it effectively becomes:

```
luatex --luaonly mtxrun.lua --script context ...
```

Because the stripped down version assumes some kind of initializations anyway a small extension made it possible to use LuaMetaTEX as stub too. So, when we rename `luametatex.exe` to `mtxrun.exe` (on Unix we don't use a suffix) it will start up as Lua interpreter when it finds a script with the name `mtxrun.lua` in the same path. When we rename it to `context.exe` it will search for `context.lua` and all that that script has to do is this:

```
arg[0] = "mtxrun"

table.insert(arg,1,"mtx-context")
table.insert(arg,1,"--script")

dofile(os.selfpath .. "/" .. "mtxrun.lua")
```

So, it basically becomes a call to `mtxrun`, but we stay in LuaMetaTEX. Because we want an isolated run this will launch LuaMetaTEX again with the right command line arguments. This sounds inefficient but because we have a small binary this is no real issue, and as that run is isolated, it cannot influence the caller. The overhead is really small: on my somewhat older laptop it's .2 seconds, but we had that management overhead already for decades, so no one bothers about it. On all platforms using symbolic links works ok too.

8.7 Global variables

There are quite a bit global variables and function in the code base, but in the process of opening up I got rid of some. The cleanup turned some more into locals which saved executable bytes (keep in mind that we also use the engine as Lua interpreter so, the smaller, the more friendly).[36] This is work in progress.

8.8 Memory usage

By going over all the code a couple of times, I was able to decrease the amount of used memory a bit as well as avoid some memory allocations. This has no consequences for performance but is nicer when multiple runs at the same time (e.g. on virtual machines)

[36] Later the global variables were collected in so called C structs.

have to compete for resources.[37]

8.9 METAPOST

The current code base doesn't have that many files. We can imagine that, when Lua can be compiled on a platform, that compiling LuaMetaTEX is also no that complicated. However, the rather complex build infrastructure demonstrates the opposite. One of the complications is that mplib is codes in cweb and that needs some juggling to get C. The process has quite some dependencies. There are some upstream patches needed, but for now occasionally checking with the upstream sources used for compiling mplib in LuaTEX works okay.[38]

As LuaMetaTEX is also used for experiments we use a copy of the Lua library interface. That way we don't interfere with the stable LuaTEX situation. When we play with extensions, we can always decide to backport them, once they are found useful and in good working order. But, as that interface was just C this was trivial.

8.10 Files

In a relative late stage I decided to cleanup some of the filename handling. First I got rid of the `area`, `name` and `ext` decomposition and optional recomposition. In the original engine that goes through the string pool and although there is some recovery in the end, with many files and fonts being used, the pool can get exhausted. For instance when you have hundreds of thousands of `\font \foo = bar` kind of definitions, each definition wipes out the previous entry in the hash, but its font name is kept in the string pool. I got rid of that side effect by reusing strings but in the end decided to avoid the pool altogether. It was then a small step to also do that for other filenames. In the process I also decided that it made no sense to keep the code around that reads a filename from the console: we now just quit. Restarting the program with a proper filename is no big deal today. I might do some more cleanup there. In the end we can best use a callback for handling input from the console.

[37] I will probably have to spend some more time on this in order to reach a state that I'm satisfied with.

[38] Later I decided to cleanup the mplib code: unused font related code was removed, the PostScript backend was untangled, the translation from cweb to C got done by a Lua script, aspects like error reporting and io were redone, and in the end some new extensions were added. Some of that might trickle back to th original, as long as it doesn't harm compatibility; after all MetaPost (the program) is standardized and considered functionally stable.

9 Rejected

9.1 Introduction

During the development of LuaTEX some extensions were considered but rejected after some experiments. I already forgot about some that were tried the last decade. I will not discuss what has been added already to LuaTEX.

9.2 Conditionals

The LuaTEX manual describes a few conditional primitives that were added. One thing I played with was a native definer, think of `\idef` but in the end rejected it, because in practice it was seldom needed. Another useful one would be `\ifnothing` but the current implementation of `\ifx` is already pretty efficient so there is nothing to gain here. Another rejected one is `\ifxcase` which takes a token and compares that with a sequence, like

```
\ifxcase\foo\alpha
\or\beta
\or\gamma
\else
\fi
```

As this was never available, in ConTEXt already different strategies were followed so I could only find a few places where this could make code more readable. But who knows, I might change my mind when I split the code base and can adapt code accordingly although it doesn't make much sense for the more high level modules because it would only affect a few lines and maintaining duplicate files is no fun.[39]

9.3 Dimensions

A primitive that returns the height plus depth would make sense (hd) but one can easily define one and the gain can be neglected. So, for now this has been rejected. Also, one can use the token scanners to implement that kind of primitives but of course that then does have a penalty in terms of performance.[40]

[39] But playing with extensions that make for better code *is* fun.

[40] Okay, in the end I decided to just add a primitive for this, but only as part of a larger set of box related primitives.

9.4 The something

I played a bit with intercepting `\the` so that we could define commands that also respond to this expander. It didn't work out well because full expansion happens, even with protected macros:

```
\protected\def\foo{...] \the\foo
```

We just have to accept this and it's no big deal.

9.5 Primitives

Occasionally I'm wondering if we should have a way to flag primitives and macros as being frozen but in the end it might not pay off. At some point I decided that at least the `\primitive` and `\ifprimitive` could go away as they are not really working as expected. It's better to have nothing than something bad. Also, we can easily clone the whole set of primitives in a new namespace with Lua if we want.[41]

[41] But . . . in the end we got something else back.

10 Whatsits

Whatsits provide the natural extension mechanism for TEX. In pdfTEX there are plenty such whatsits, for instance for pdf annotations. In LuaTEX this mechanism was reorganized so that the code was better isolated. In the first versions of LuaMetaTEX only a handful was left. Stepwise some were removed and in the end we could stick to only one general whatsit because one can implement the few needed to be compatible with TEX.

We started out with this set of whatsits:

open	open a file for writing (delayed)
write	write to an open file (or terminal otherwise)
close	close an opened file
special	write some literal pdf code to the output file
user	store and retrieve data in a node
latelua	execute code delayed (in the backend)
literal	write some literal pdf code to the output file, controlled by a mode
save	push the transformation state
restore	pop the transformation state
matrix	apply a transformation (`rx sx sy ry`)
savepos	register a position to be queried afterwards (`x y`)

The `\openout`, `\write` and `\closeout` primitives relate to the first three but they can be prefixed with `\immediate` in which case they don't end up as whatsits but are applied directly. The `special` is actually meant for dvi while the `\(pdf)literal` is for pdf output. The first four are available in regular TEX.

The last four are dealt with exclusively in the backend and by removing the backend they basically became no-ops. I kept them for a while but in the end decided to kick them out. Instead a generic whatsit was introduced that could be used as signal with the same function. That simple whatsit only has a subtype (and of course optionally attributes). And, as ConTEXt has its own backend, we can intercept them as we like. The saving in code is not spectacular but keeping it around (basically doing nothing) neither. The impact on ConTEXt was not that large because for instance saving positions is done differently and transformations are encapsulated in a few helpers that could easily be adapted.

From there it was a small step to also remove the literal whatsit, so then we had five whatsits left, plus the generic one. I then entered sentimental mode: should we keep the first four or not. Of course we want to be TEX compatible but we can remove the code and provide a compatible replacement using macros and our own simple whatsit nodes. That keeps all the housekeeping at the Lua end, simplifies the C, and we're still TEX.

Of course, once we remove these and only have the delayed Lua whatsit and user whatsits left, we can as well replace these too. In LuaTEX user nodes are actually not dealt with in the backend. One can create them at the Lua end and query them in callbacks. The TEX machinery just ignores them, like any whatsit. In retrospect they could have been first class nodes, but making them whatsits was wise because that way they can be ignored consistently when needed.

So, in the end all we need is a simple whatsit. As I removed the subtypes stepwise there was an intermediate mix of code to recognize simple whatsits from core whatsits but that distinction went away. Doing this kind of refactoring is best done stepwise because that way I can compile some large documents and see if things break. As a consequence again some code could be simplified as we basically no longer have extensions. Of course at the ConTEXt end the removed primitives had to be added but that didn't take much effort. The binary shrunk some 30K but (a small amount of) Lua code was added to provide a compatible functionality (not that we use it).

11 Feedback

11.1 Introduction

As LuaTEX 1.10 is basically frozen in terms of functionality not much can or will be added. But it made sense to some of the (small) improvements that were made in LuaMetaTEX got feedback to LuaTEX (or will be at some point). Because we are also experimenting, there can be a delay.[42]

Of course the question is “Should we feedback (retrofit) at all?”. I'm still not sure about it. There should be a good reason to do it because it can harm stability of the original. At some point ConTEXt can default to the follow up in which case testing the original becomes more difficult for specific features. I never ran into (useful) demands for extensions so retrofit can have a low priority.

Another factor is that when distributions start adding stuff to stock LuaTEX on top of what is our default (after all isn't that what happens with open source projects), it makes not much sense to look back and retrofit new functionality, because there is not much change that we will use such a variant ourselves and we could introduce errors in the process. Providing bloatware is not our objective.

Related to this is the question if we should always go into LMTX mode and I'm no longer sure if we shouldn't do that. We can use plain TEX with the regular LuaTEX backend and just forget about some generic framework. The danger of it backfiring is just too large. It is a waste of time and will keep us back.

One reason for a dual mode is that it made possible some timings in order to determine bottlenecks. I did some rough tests and that is enough to get the picture. Take this document:

```
\starttext
    \dorecurse
        {1000}
        {\samplefile{sapolsky} {\bf\samplefile{sapolsky}}\par}
\stoptext
```

Using regular LuaTEX this takes on an Intel i7-3840 mobile processor about 9.3 seconds while LuaMetaTEX needs 11.2 seconds, so we loose time. This is because we have only text so the native backend wins on piping out the page stream. On my domotica fitlet

[42] Later chapters mention a few more possible extensions.

with an low power AMD A10 processor running linux the runtime goes from 25.4 seconds to 27.8 seconds, so again a slow down.

But this is not a typical document! Add a bit more spice and the numbers reverse. For processing the LuaTEX manual stock LuaTEX takes 12.6 seconds on the Intel and LuaMetaTEX needs 12.4 seconds. On the AMD runtime goes from 35.1 seconds down to 32.8 seconds. So here we win some.

These are rough timings and a few weeks later we go these timings on the Intel:[43]

engine	backend	runtime	LuaJIT vm
LuaTEX 1.10	normal	12.4	9.9
LuaTEX 1.10	lmtx	12.7	9.8
LuaMetaTEX 2.00	lmtx	12.2	9.3

Because we have more Lua code active, we pay a price with LMTX but not on LuaMetaTEX (as of now, later we will see a performance bump). The gain when using the LuaJIT virtual machine is more noticeable. And, there is probably some more to gain. In case you wonder why this matters: think of the low power AMD processor. When we have to replace computers we can consider using low power ones, with weaker processors, less memory, and less cache. For the record: I use cross compiled MingW binaries on windows (they are quite a bit faster than native windows binaries). And the binaries are less than 3MB (small files and less resources are nice when running on remote file systems).

This all indicates that we have no real reason to stick to a mixed approach: if we want we can just switch to always LMTX and never look back.

11.2 Expressions

When writing an article that involved using a `\numexpr` it struck me that we should have a proper integer division. Just compare these:

```
\the\numexpr 13/2\relax
```

and

```
\scratchcounter13 \divide\scratchcounter 2 \the\scratchcounter
```

This gives **7** and **6**. We now also have:

```
\the\numexpr 13:2\relax
```

[43] On the more modern gaming laptop of a nephew we measured half these numbers.

which gives **6**. I considered using a double slash (as in Lua) but using a colon is easier. Of course those who make that an active character are probably toast. This is an easy patch but it's hard to predict possible side effects outside ConTEXt.

11.3 Looking ahead

Sometimes you want to look ahead and act upon the presence of a specific character. Implementing that in pure TEX primitives is no big deal, but especially when you want to ignore leading spaces it leads to rather verbose code when tracing is enabled. Out of curiosity I played with a primitive that can help us out. Although there is also a performance gain, in practice one will not notice that unless such a feature is used millions of times, but in that case the gain is noise compared to the rest of the run.

```
\def\foo{\futureexpand/\fooyes\foonop}
\def\fooyes/#1/#2{[#1:#2]}
\def\foonop    #1{(#1)}

\foo/yes/{one}\quad
\foo     {two}
```

We either expand `\fooyes` or `\foonop`, depending on the presence of a `/` after `\foo`. So, the result is:

```
[yes:one]  (two)
```

The next examples demonstrates two variants. The second one doesn't inject spaces back into the stream.

```
\def\f<{\futureexpand/\y\n}
\def\y/#1/{#1}
\def\n    {}

(\f</yes/>)\quad
(\f<     >)
```

Watch the space in the `\n` case.

```
(yes>)  ( >)
```

```
\def\f<{\futureexpandis/\y\n}
\def\y/#1/{#1}
\def\n    {}

(\f</yes/>)\quad
```

```
(\f<     >)
```

This time the space is not injected (`is` is short for ignore spaces).

```
(yes>)  (>)
```

I will probably use this one in ConTEXt, but as said, not for performance reasons but because it reduces code and therefore tracing.[44]

11.4 Checking numbers an dimensions

The ConTEXt user interface often uses parameters that take keywords as well as a number or dimension. In practice it has never been an issue to check for that but there are a few cases where we'd like to be a bit more flexible.

```
\doifelsenumber{123999999999999999}YN
\doifelsenumber               {123}YN
\doifelsenumber                 {A}YN
\doifelsenumber          {\char123}YN
\doifelsenumber          {\toks123}YN
\doifelsenumber{123\scratchcounter}YN

\doifelsedimension{123999999999999999pt}YN
\doifelsedimension               {123pt}YN
\doifelsedimension                   {A}YN
\doifelsedimension            {\char123}YN
\doifelsedimension            {\toks123}YN
\doifelsedimension    {123\scratchdimen}YN
```

This typesets:

N Y N N N Y
N Y N N N Y

especially the `123\scratch...` checking is tricky. For that reason we now have two new built-in checkers. Again, performance is not one of the arguments, because these checks are not much faster than what we have already, they are just a bit more robust for weird cases. A possible use of the primitives is:

```
\ifdimen123\or
```

[44] In the ConTEXt code base there are several places where less code takes precedence over efficiency. But in all cases extensive tests were done to see if it made a dent in practical performance.

```
  yes
\else % or \or
  no
\fi
```

and:

```
\ifnumber123\or
  yes
\else % or \or
  no
\fi
```

When a valid number or dimension is gobbled, the value pushed in the branches is 1, and when an error is encountered the value 2 is pushed. Deep down we have just an `\ifcase` and by not using the value zero we nicely skip the invalid code. It might look a bit weird but we need a sentinel for the number (and the `\or` serves as such, without introducing strange new constructs. We'll see if we keep it (as testing must prove its usefulness).

11.5 Comparing tokens

The following code compares (for instance) two strings:

```
\def\thisorthat#1#2%
  {\edef\one{#1}
   \edef\two{#2}
   \ifx\one\two
     this%
   \else
     that%
   \fi}

\thisorthat{foo}{bar}
```

but this looks a bit cleaner (in a trace):

```
\def\thisorthat#1#2%
  {\iftok{#1}{#2}%
     this%
   \else
     that%
```

```
    \fi}

\thisorthat{foo}{bar}
```

It's not that much faster (unless one uses it a real lot) as similar things have to happen to get the test to work. But the nice things of this checker is that it works with token registers and macros too. But in order use it in relevant places in ConTEXt I would have to adapt quite some code. This could actually be a reason for a MkIV freeze and LMTX code base (as with MkII). The question is: does it pay off?

12 LUA

12.1 Move to 5.4

Another experiment concerned testing Lua 5.4 which looks like a minor update in terms of new functionality but has some consequences. By now the old module model is even more deprecated and compatibility mode no longer makes much sense. As a consequence we now need to adapt the way libraries are loaded (and we use global ones) and a few other low level calls had to be adapted. This is no real issue and once that was done, I found out that the bit32 module was even more obsolete so I decided to get rid of it. We already have a bit32 replacement in ConTEXt so I had to enable that. As ConTEXt doesn't need compatibility mode it was no problem to drop that too.

The biggest changes in 5.4 are under the hood: some optimized byte code and a new generational garbage collector. I did a few runs and a 12.4 seconds run on the manual now dropped to around 12.1 and given that we spend (probably) more than half the time in Lua that means some 5% gain in performance. This is still more than the 9.6 seconds that LuaJIT needs but it looks like every Lua release gains a bit and I'm pretty sure that there is more to gain.[45]

An interesting experiment was to disable the automatic string to number conversion when a number is expected but a string is needed. So far I only had to adapt two lines of code in the in the meantime considerable amount of Lua code that comes with ConTEXt.

12.2 No more LUAJIT

One thing I had to consider was the future of LuaJIT. This project is sort of stalled and will not follow Lua development. Now, to some extend we can deal with this but with the faster Lua 5.4 around the corner, the limitations of LuaJIT with respect to loading large tables, as well as the fact that we need a patched hash function to get an advantage over regular Lua anyway, it makes sense to drop it in LuaMetaTEX. After discussing this with Alan, who crunched numbers in order to make impressive graphics with MetaPost, we came to the conclusion that we should not overestimate the benefits. There is still a gain but removing the need to support both could also makes it possible to improve existing code (although one should not expect too much from that; it's more a matter of convenience for me). Also, for as long as have LuajitTEX that is still an option when one has to squeeze out every second.

A valid question is if ditching LuaJIT will harm users. The answer to this depends on the

[45] In the meantime there are experiments in 5.4 with `<const>` directives which might have advantages.

kind of documents that you process. Given decent programming, you can gain quite a bit of runtime, but on the average the difference is not that large. There is for instance always the overhead of callbacks and crossing the so called C boundary that has an impact.

12.3 Performance

At the time of writing this Thomas Schmitz was wondering if there was a significant difference in runtime between the table mechanisms and especially natural tables and extreme tables. Some test demonstrated that extreme tables were best for his case. That case concerned generating about 400 pages of tables from xml files, including some juggling of data in Lua. The bottleneck in that document can be roughly simulated with the following test. We assume one pass over the table but in practice there are upto four, but only the last one has frames. So, the test concerns 80.000 (400 pages with 40 rows of 5 columns) calls to `\framed`.

```
1                                  \hpack{\framed                {oeps}}
2                                  \hpack{\framed[frame=off]     {oeps}}
3   \setupframed[frame=off]        \hpack{\framed                {oeps}}
4                                  \hpack{\framed[frame=on]      {oeps}}
5   \setupframed[frame=on]         \hpack{\framed                {oeps}}
6                                  \hpack{\framed[frame=closed]{oeps}}
7   \setupframed[frame=closed]     \hpack{\framed                {oeps}}
```

sample	luatex & mkiv	luajittex & mkiv	luametatex & lmtx
1	17.3	16.8	13.5
2	17.8	17.2	14.0
3	17.3	16.8	13.3
4	17.9	17.4	13.7
5	17.4	17.1	13.3
6	17.4	16.8	12.9
7	16.4	16.0	12.6

Even if we add the usual .1 second interval around these values it will be clear that we gain enough not to worry about the loss of LuaJIT, also because the gain is not in the Lua part only. A nice consequence of this is that when we replace the cpu's in a server with low power ones that perform 25% less, we can compensate that by using LMTX.[46]

[46] There's still room for improvement, because mid July 2019 we're at 12.9, 13.2, 12.9, 13.5, 13.0, 12.5 and 12.2 seconds or less. But don't expect too many miracles.

When wrapping this up, the LuaTEX manual processed with LMTX took slightly less than 11.9 seconds, compared to a normal run of 12.6 seconds, so we're gaining some there too. And just after I wrote this we went down to 11.7 seconds by (as experiment) changing the Lua virtual machine dispatcher, so there is still some to gain. In the energy saving fitlet with small amd processor processing the manual with stock LuaTEX takes about 37 seconds, but 33.5 with LMTX so here also we're not off worse.

12.4 Modules

Right from the start LuaTEX had some extra libraries linked in: `md5` (for hashing), `lfs` (for accessing file properties), `slunicode` (for basic utf handling), `gzip` and `zlib` (for zipping files and streams), `zip` (for accessing zip files) and `socket` (for communicating other than with files).

In LuaMetaTEX the not so useful `slunicode` library was removed pretty early but the others stayed around. The more backend specific `img` and `pdf` libraries went away too, as did the (already not used) `fontloader` library. The `kpse` library is also gone, as we do those things in Lua. The `epdf` library was kept. A couple of libraries were added, like `sha2`, `basexx`, and `flate`, plus a few handy helper libraries that are still experimental and therefore not mentioned here.

The `flate` library is also an experiment but will replace the `gzip` and `zlib` libraries. Currently these use `libz` but `libdeflate` will be the low level replacement once it support streams and is already used for `flate`. The `md5` library has been redone using utility code `pplib`, as `sha2` does. The type basexx library also falls back on utility code form `pplib` (that code is actually independent).

The `lfs` code has been replaced by a variant that omits features not common to the platforms and with a iterator that permits much faster directory scans and has a few more helpers. It is not compatible but we kept the name because of legacy usage. I might strip the socket code to what is actually used, but on the other hand: don't touch what works well. The original code doesn't change that much anyway.

13 Compilation

Compiling LuaTEX is possible because after all it's what I do on my machine. The LuaTEX source tree is part of a larger infrastructure: TEXLive. Managing that one is work for specialists and the current build system is the work of experts over a quite long period of time. When you only compile LuaTEX it goes unnoticed that there are many dependencies, some of which are actually unrelated to LuaTEX itself but are a side effect of the complexity of the build structure.

When going from LuaTEX to LuaMetaTEX many dependencies were removed and I eventually ended up with a simpler setup. The source tree went down to less than 30 MB and zipped to around 4 MB. That makes it possible to consider adding the code to the regular ConTEXt distribution.

One reason for doing that is that one keeps the current version of the engine packaged with the current version of ConTEXt. But a more important one is that it fulfils a demand. Some time ago we were asked by some teachers participating in a (basically free) math method for technical education what guarantees there are that the tools used are available forever. Now, even with LuaMetaTEX one has to set up a compiler but it is much easier than installing the whole TEXLive infrastructure for that. A third reason is that it gives me a comfortable feeling that I myself can compile it anywhere as can ConTEXt users who want to do that.

The source tree traditionally has libs in a separate directory (lua, luajit, zlib and zziplib). However, it is more practical to have them alongside our normal source. These are relative small collections of files that never change so there is no reason not to do it.[47]

Another assumption we're going to make is that we use 64 bit binaries. There is no need to support obsolete platforms either. As a start we make sure it compiles on the platforms used by ConTEXt users. Basically we make a kind of utility. For now I can compile the Windows 32 bit binaries that my colleague needs in half a minute anyway, but in the long run we will settle for 64 bits.

I spent about a week figuring out why the compilation is so complex (by selectively removing components). At some point compilation on os-x stopped working. When the minimum was reached I decided to abandon the automake tool chain and see if `cmake` could be used (after all, Mojca challenged that). In retrospect I should have done that sooner because in a day I could get all relevant platforms working. Flattening the source

[47] If I ever decide to add more libraries, only the minimal interfaces needed will be provided, but at this moment there are no such plans.

tree was a next step and so there is no way back now. What baffled me (and Alan, who at some point joined in testing os-x) is the speed of compilation. My pretty old laptop needed about half a minute to get the job done and even on a RaspberryPi with only a flash card just a few minutes were needed. At that point, as we could remove more make related files, the compressed 11 MB archive (`tar.xz`) shrunk to just over 2 MB. Interesting is that compiling mplib takes most time, and when one compiles in parallel (on more cores) that one finishes last.

For the record: I do all this on a laptop running MS Windows 10 using the Linux subsystem. When that came around, Luigi made me a working setup for cross compilation but in the meantime with GCC 8.2 all works out of the box. I edit the files at the MS Windows end (using SciTE), compile at the linux end, and test everything on MS Windows. It is a pretty convenient setup.

When compilation got faster it became also more convenient to do some more code reshuffling. This time I decided to pack the global variables into structures, more or less organized the way the header files were organized. It gives a bit more verbosity but also has the side effects that (at least in principle) the cpu cache can perform better because neighboring variables are often cached as part of the deal. Now it might be imagination, but in the process I did notice that mid March processing the manual went down to below 11.7 seconds while before it stayed around 12.1 seconds. Of course this is not that relevant currently, but I might make a difference on less capable processors (as in a low power setup). It anyway didn't hurt.

In the meantime some of the constants used in the program got prefixes or suffixes to make them more unique and for instance the use of `normal` as equivalent for zero was made a bit more distinctive as we now have more subtypes. That is: all the subtypes were collected in enumerations instead of C defines. Back to the basics.

End of 2020 I noticed that the binary had grown a bit relative to the mid 2020 versions. This surprised me because some improvements actually made them smaller, something you notice when you compile a couple of times when doing these things. I also noticed that the platforms on the compile farm had quite a bit of variation. In most cases we're still below my 3MB threshold, but when for instance cross compiled binaries become a few hundred MB larger one can get puzzled. In the LuaMetaFun manual I have this comment at the top:

```
------------------------   ------------------------   ------------------------
2019-12-17  32bit  64bit   2020-01-10  32bit  64bit   2020-11-30  32bit  64bit
------------------------   ------------------------   ------------------------
freebsd     2270k  2662k   freebsd     2186k  2558k   freebsd     2108k  2436k
openbsd6.6  2569k  2824k   openbsd6.6  2472k  2722k   openbsd6.8  2411k  2782k
linux-armhf 2134k          linux-armhf 2063k          linux-armhf 2138k  2860k
```

```
linux       2927k  2728k   linux       2804k  2613k   linux   (?) 3314k  2762k
                                                       linux-musl  2532k  2686k
osx                2821k   osx                2732k   osx                2711k
ms mingw    2562k  2555k   ms mingw    2481k  2471k   ms mingw    2754k  2760k
                                                       ms intel           2448k
                                                       ms arm             3894k
                                                       ms clang           2159k
------------------------   ------------------------   ------------------------
```

So why the differences? One possible answer is that the cross compiler now uses gcc9 instead of gcc8. It is quite likely that inlining code is done more aggressively (at least one can find remarks of that kind on the Internet). An interesting exception in this overview is the linux 32 bit version. The native Windows binary is smaller than the MingW binary but the clang variant is still smaller. For the native compilation we always enabled link time optimization, which makes compiling a bit slower but similar to regular compilation in WLS but when for the other compilers we turn on link time optimization the linker takes quite some time. I just turn it off when testing code because it's no fun to wait these additional minutes with gcc. Given that the native windows binary by now runs nearly as fast as the cross compiled ones, it is an indication that the native Windows compiler is quite okay. The numbers also show (for Windows) that using clang is not yet an option: the binaries are smaller but also much slower and compilation (without link time optimization) also takes much longer. But we'll see how that evolves: the compile farm generates them all.

So, what effects does link time optimization has? The (current) cross compiled binary is is some 60KB smaller and performs a little better. Some tests show some 3 percent gain but I'm pretty sure users won't notice that on a normal run. So, when we forget to enable it when we release new binaries, it's no big deal.

Another end 2020 adventure was generating arm binaries for os-x and Windows. This seems to work out well. The os-x binaries were tested, but we don't have the proper hardware in the compile farm, so for now users have to use Intel binaries on that hardware. Compiling the LuaMetaTEX manual on a 2020 M1 is a little more that twice as fast than on my 2013 i7 laptop running Windows. A native arm binary is about three times faster, which is what one expects from a more modern (also a bit performance hyped) chipset. On a RaspberryPi with 4MB ram, an external ssd on usb3, running Ubuntu 20, the manual compiles three times slower than on my laptop. So, when we limit conclusions to LuaMetaTEX it looks like arm is catching up: these modern chipsets (from Apple and Microsoft, although the later was not yet tested) with plenty of cache, lots of fast memory, fast graphics and speedy disks are six times faster than a cheap media oriented arm chipset. Being a single core consumer, LuaMetaTEX benefits more from faster cores than from more cores. But, unless I have these machines on my desk these

rough estimates have to do.

14 Stubs

14.1 Bare bone

The most barebone way to process a ConTEXt file is something like:

```
luametatex
    --fmt="<cache path to>/luametatex/cont-en"
    --lua="<cache path to>/luametatex/cont-en.lui"
    --jobname="article"
    "cont-yes.mkiv"
```

We pas extra options, like:

```
    --c:autopdf
    --c:currentrun=1
    --c:fulljobname="./article.tex"
    --c:input="./article.tex"
    --c:kindofrun=1
    --c:maxnofruns=9
    --c:texmfbinpath="c:/data/develop/tex-context/tex/texmf-win64/bin"
```

but for what we are going to discuss here it doesn't really matter. The main point is that we use a Lua startup file. That one has a minimal amount of code so that the format can be loaded as we like it. For instance we need to start up with initial memory settings.

The file `cont-yes` sets up the way processing content happens. This can be the `jobname` file but also something different. It is enough to know that this startup is quite controlled.

I will explore a different approach to format loading but for now this is how it goes. After al, we need to be compatible with LuaTEX and normal MkIV runs, at least for now.

14.2 Management (some history)

In ConTEXt we always had a script: `texexec`, originally a Modula2 program, later a Perl script, then a Ruby script but now we have `mtxrun`, a Lua script. All take care of making sure that the file is processed enough times to get the cross references, tables of contents, indexes, multi-pass data stable. It also makes it possible to avoid using these special binaries (or links) that trick the engine into thinking it is bound to a format: we never had `pdfcontext` or `luacontext`, just one `context`. Actually, because we have multiple user interfaces, we would have needed many stubs instead. Getting this ap-

proach accepted was not easy but in the meantime I've seen management scripts for other packages being mentioned occasionally.

The same is true for scripts: for a long time ConTEXt came with quite some scripts but when an average TEX distribution started growing, including many other scripts, we abandoned this approach and stuck to one management script that also launched auxiliary scripts. That way we could be sure that there were no clashes in names. If you look at a full TEX installation you see many stubs to scripts and more keep coming. How that can work out well without unexpected side effects (name clashes) is not entirely clear to me, as a modern computer can have large bin paths. Just imagine that all large programs (or ecosystems) would introduce hundreds of new 'binaries'.

Anyway, in the end a ConTEXt installation using MkIV only needs `mtxrun` and as bonus `context`. The above call is triggered by:

```
mtxrun --autogenerate --script context --autopdf article.tex
```

from the editor. Here we create formats when none is found, and start or activate the pdf viewer afterwards, so more minimal is:

```
mtxrun --script context article.tex
```

Normally there is also a `context` stub so this also works:

```
context article.tex
```

14.3 The launch process (more history)

In MkII, when we use pdfTEX, the actual launch of these script is somewhat complex and a bit different per platform. But, on all platforms kpse does the lookup of the script. Already long ago I found out that this startup overhead could amount to seconds on a complete TEXLive installation (imagine running over a network) which is why eventually we came up with the minimals. The reason is that the file databases have to be loaded: first for looking up, then for the stub that also needs that information and finally by the actual program. There were no ssd's then.

The first hurdle we took was to combine the lookup and the runner. Of course this is sort of out of our control because an installer can decide to still use a lookup approach but at least on MS Windows this was achieved quite easy. Sort of:

```
texexex -> [lookup] -->
    texexec.pl -> [lookup] ->
        pdftex + formats ->
```

```
[lookup] -> processing
```

The first lookup can be avoided by some fast relative lookup, but for more complex management the second one is always there. Over time this mechanism became more sophisticated, for instance we use caching, could work over sockets using a kpse server, etc.

When LuaTEX came around, it was already decided early that it also would serve as script engine for the ConTEXt runner, this time `mtxrun`. The way this works differs per platform. On Windows there is a small binary, say `runner.exe`. It gets two copies: `mtxrun.exe` and `context.exe`. If you find more copies on your system, something might be wrong with your installation.

```
mtxrun.exe  -> loads mtxrun.lua in same path
context.exe -> idem but runs with --script=context
```

The `mtxrun.lua` script will load its file database which is very efficient and fast. It will then load the given script and execute it. In the case of `context.exe` the `mtx-context.lua` script is loaded, which lives in the normal place in the TEX tree (alongside other scripts).

So, a minimal amount of programs and scripts is then:

```
texmf-win64/bin/luatex.exe
texmf-win64/bin/mtxrun.exe
texmf-win64/bin/mtxrun.lua
texmf-win64/bin/context.exe
```

with (we also need to font manager):

```
texmf-context/scripts/context/lua/mtx-context.lua
texmf-context/scripts/context/lua/mtx-fonts.lua
```

But . . . there is a catch here: LuaTEX has to be started in script mode in order to process `mtxrun`. So, in fact we see this in distributions.

```
texmf-win64/bin/luatex.exe
texmf-win64/bin/texlua.exe
texmf-win64/bin/mtxrun.exe
texmf-win64/bin/mtxrun.lua
texmf-win64/bin/context.exe
```

The `texlua` program is just a copy of `luatex` that by its name knows that is is supposed to run scripts and not process TEX files. The setup can be different using dynamic li-

braries (more files but a shared engine part) but the principles are the same. Nowadays the stub doesn't need the `texlua.exe` binary any more, so this is the real setup:

```
texmf-win64/bin/luatex.exe       large program
texmf-win64/bin/mtxrun.exe       small program
texmf-win64/bin/mtxrun.lua       large lua file
texmf-win64/bin/context.exe      small program
```

Just for the record: we cannot really use batch files here because we need to know the original command, and when run from a script that is normally not known. It works to some extend but for instance when started indirectly from an editor it can fail, depending on how that editor is calling programs. Therefore the stub is the most robust method.

On a Unix system the situation differs:

```
texmf-linux-64/bin/luatex        large program
texmf-linux-64/bin/texlua        symlink to luatex
texmf-linux-64/bin/mtxrun        large lua file
texmf-linux-64/bin/context       shell script that starts mtxrun
```

Here `mtxrun.lua` is renamed to `mtxrun` with a shebang line that triggers loading by `texlua` which is a symlink to `luatex` because shebang lines don't support the `--texlua` argument. As on windows, this is not really pretty.

14.4 The LMTX way (the present)

Now when we move to LMTX we need to make sure that the method that we choose is acceptable for distributions but also nicely consistent over platforms. We only have one binary `luametatex` with all messy logic removed and no second face like `metaluatex`. When it is copied to another instance (or linked) it will load the script with its own name when it finds one. So on Windows we now have:

```
texmf-win64/bin/luametatex.exe   medium program
texmf-win64/bin/mtxrun.exe       copy (or link) of luametatex
texmf-win64/bin/mtxrun.lua       large lua file
texmf-win64/bin/context.exe      copy (or link) of luametatex
texmf-win64/bin/context.lua      small lua file
```

and in Unix:

```
texmf-linux-64/bin/luametatex    mediumprogram
texmf-linux-64/bin/mtxrun        copy (or link) of luametatex
```

```
texmf-linux-64/bin/mtxrun.lua    large lua file
texmf-linux-64/bin/context       copy (or link) of luametatex
texmf-linux-64/bin/context.lua   small lua file
```

So, `luametatex[.exe]`, `mtxrun[.exe]` and `context[.exe]` are all the same. On both platforms there is `mtxrun.lua` (with suffix) and on both we also use the same runner approach. The `context.lua` script is really small and just sets the script command line argument before loading `mtxrun.lua` from the same path. In the case of copied binaries: keep in mind that the three copies together are not (much) larger than the `luatex` and `texlua` pair (especially when you take additional libraries into account).

The disadvantage of using copies is that one can forget to copy with an update, but the fact that one can use them might be easier for installers. It's up to those who create the installers.

One complication is that the `mtxrun.lua` script has to deal with the old and the new setup. But, when we release we will assume that one used either LuaTEX or LuaMetaTEX, not some mix. As `mtxrun` and `context` know what got it started they will then trigger the right engine, unless one passes `--engine=luatex`. In that case the LuaMetaTEX launcher will trigger a LuaTEX run. But a mixed installation is unlikely to happen.

14.5 Why not ...

Technically we could use one call for both the runner and TEX processor but when multiple runs are needed this would demand an internal engine reset as well as macro package reset while keeping some (multi-pass) data around. A way in-between could be to spawn the next run. In the end the gain would be minimal (we have now .2 seconds overhead per total run, which can trigger multiple passes, due to the management script, to basically we can neglect it. (Triggering the viewer takes more time.)

15 METAPOST

15.1 Introduction

Relatively late in the followup I started wondering about what to do with mplib. Alan Braslau is working on the `luapost` module and we discuss handy extensions written in Lua and MetaPost code but who knows what more is needed. Some ideas were put on delay but it looked like a good moment to pick up on them. One problem is that when we play with the mplib code itself in LuaMetaTEX, the question is how to keep in sync with the official library. In this chapter I'll discuss both: keeping up with the official code, and keeping ahead with ideas.

15.2 The code base

The mplib code is written in cweb and lives in files with the suffix `w`. These files need to be converted to `c` and `h` files, something that is done with the `ctangle` program. To avoid that dependency I just took the C files from LuaTEX, but I had to apply a few patches (to get rid of dependencies). Now, it is a fact that MetaPost doesn't really develop fast and in principle a diff could identify the changes easily. So, why shouldn't I also start experimenting with mplib itself in the follow up? It's easy to merge future changes (in both directions).

The first thing I wrote was a `w-to-c` script. This was not that hard given that I already had written lexers. After a first prototype worked out well, I redid the code a bit (so that in the future I can also implement support for change files for instance). A complication was that I found out that the regular cweb converter messes around a bit with the code. So, I had to write another script to mimmick that to the level that I could compare the results. For example, spaces are removed before and after operators and all leading space gets removed too. When I got the same output I could get rid of that code and output what I want. For instance I'd like to keep the spacing the same because compilers can warn about some issues, like missing `;` and misleading indentation in simple `if` and `while` constructs where braces are omitted.[48] One can argue that this is not important, but if not, then why enable warnings at all. I had to fix half a dozen places in the `w` file to make the compiler happy, so the price was small.

Once I had a more or less instantaneous conversion[49] I got the same feeling as with the rest of the code: experimenting became convenient due to the fast edit-compile cycle.

[48] This is no problem in for instance Pascal where we always have a `begin` and `end`.

[49] Conversion of the `w` files involved took just over half a second at that time, currently it takes just over a quarter of a second, on a relatively old machine that is.

So, with al this covered I could do what I always had wanted to do: remove traces of the backends (including the full PostScript one), because they are actually to be plugins, and also get rid of internal font handling, which is bound to Type1 (rendering) and small size tfm (generating). With respect to that export: I wonder if anyone used that these days because even the Gust font project always had their own tool chain alongside MetaPost. I could also void the hacks needed to trick the library in not being dependent of `png.h` and `zlib.h` headers, for which I had to use dummies.[50]

It took a few days scripting the converter (most time went into getting identical output in order to check the converter which was later dropped), a few days stripping unused code, another day cleaning up the remaining code and then I could start playing with some new extensions. The binary has shrunk with 200KB and the whole LuaMetaTEX code base in compressed `tar.xz` format is now below 1.8MB while before it was above 2MB. Not that it matters much, but it was an nice side effect.[51]

What new extensions would show up was still open. Because Alan and I play with scanners it made sense to look into that. Error handling and logging has also been on my radar for a while. In the process some more code might be dropped, but actually the current version is still useable as library for a stand alone program, given that one reconstructs the PostScript driver from the dropped code (not that much work). Some configuration options are missing then but that could be provided as extensions (after all we can have change files.) On the other hand, wrapping code in ConTEXt, like:

```
\starttext
\startMPpage
    ........
\stopMPpage
\startMPpage
    ........
\stopMPpage
\stoptext
```

will give a pdf file that can be converted to all kinds of formats, and the advantage is that one has full font support. There is already a script in the distribution that does this anyway.

15.3 Communication

The first experiment concerns a change in the interfacing between the MetaPost and Lua

[50] The converter can load a file with patches to be applied but by now there are no patches.

[51] Size matters as we want to code to end up in the ConTEXt distribution. It might grow a bit as side effect of adding some more features to mplib.

end. In the original library all file io is handled by the library itself. The filenames can be resolved via a callback. Once an instance is initialized, snippets of code are passed to the instance via the `execute` call. Log, terminal and error information is collected and returned as part of the return value (a table). This means that reporting back to the user has a delay: it can be shown *after* all code in the buffer has been processed. The code given as argument to `execute` is passed to the engine as (fake) terminal input, which nicely fits in the concept of interactive input, which already is part of the MetaPost concept.

In our follow up variant all file io goes via Lua. This means that we have a bit more control over matters. In ConTEXt we now can use the usual file handling code. One defines an `open_file` callback that returns a table with possible methods `close`, `reader` and `writer`, as in similar LuaTEX callbacks. A special file, with the name `terminal` is used for terminal communication. Now, when the `execute` command is handled, the string that gets passed ends up in the terminal, so the file handler has to deal with it: the string gets written to the handle, and the handle has to return it as lines on request. In ConTEXt we directly feed the to be executed code into the terminal cache.

It's all experimental and subject to changes but as we keep ConTEXt LMTX and LuaMetaTEX in sync, this is no problem. Users will not use these low level interfaces directly. It might take a few years to settle on this.

The reports that come from the MetaPost engine are now passed on to the `run_logger` callback. That one gets a target and a string passed. Where the original library can output stuff twice, once for the log and once for the console, in the new situation it gets output once, with the target being terminal, log file or both. The nice thing about this callback is that there is no delay: the messages come as the code is processed.

We combine this logging with the new `halt_on_error` flag, which makes the engine abort after one error. This mechanism will be improved as we go. The interaction option `silent` hides some of the less useful messages.

The overall efficiency of the library doesn't suffer from these changes, and in some cases it can perform even better. Anyhow, the user experience is much better with synchronous reports.

Although not strictly related to io, we already has extended the library with the option to support utf-8, which is handy for special symbols, as for instance used in the `luapost` library.

15.4 Scanning

Another extension is more fundamental in the sense that it can affect the way users see MetaFun: extending the user interface. It is again an example of why is having an

independent code base has benefits: we can do such experiments for a long time, before we decide that (and how) it can end up in the parent (of course the same is true for the mentioned io features). I will not discuss these features here. For now it is enough to know that it gets applied in ConTEXt and will provide a convenient additional interface. Once it is stable I'll wrap it up in writing.

16 TEX

16.1 Prefixes

The fact that we merged ε-TEX, a bit of pdfTEX and some of Aleph into LuaTEX, already makes it a non-standard TEX engine. In LuaMetaTEX we go a bit further. Completely outsourcing the backend has the side effect that some (extension related) primitives have to be implemented explicitly. The fact that Lua is integrated has consequences for, for instance, initialization. Defaulting to utf-8 input makes it different too. And delegating many font matters to Lua also doesn't make it behave like good old TEX.

Here I discuss another difference. One can argue that this definitely makes it less TEX, but in practice this is not that problematic. We're talking prefixes here. Traditional TEX has only prefixes:

1. `\global`: when used, it will make the next definition a global one. The `\globaldefs` parameter can be used to force global or local definitions.

2. `\long`: when applied, this will make a macro bark on a `\par` (or its equivalent) when grabbing an argument. In LuaTEX this check can be disabled.[52]

3. `\outer`: when applied the macro can only be used at the outer level.

Multiple prefixes can be given and their effects accumulate. The ε-TEX extension adds another one:

4. `\protected`: this will make a macro unexpandable inside an `\edef`, an `\xdef` or token list serialization.

In ConTEXt we never use(d) `\outer` and I can't even think of a useful application in a large macro package. in MkII most interface macros are defined as `\long`, and because in MkIV we block the complaints, we don't need this prefix either. On the other hand, many macros are defined `\protected`.[53]

When you look at the implementation, `\long` and `\outer` are properties of the so called command code: we have normal, long, outer and long outer macros, and each has a unique command code. For some reason `\protected` is not implemented with command codes, which would have doubled the number to eight, but as special token in-

[52] In a similar fashion barking about a `\par` in math mode can be disabled. Such warnings made much sense when a TEX run took much time and was triggered and traced on relative slow output devices.

[53] Or in ConTEXt speak, they are defined as `\unexpanded`, because we already had `\protected` as well as `\unexpanded` before these were introduces as primitives.

jected in front of the macro preamble. Using a command code would have made more sense as there is no real speed penalty in that, while the special token indicating is a macro (body) is protected now has to be intercepted in some cases.

Anyhow, already for a while I wondered if I should drop `\long` and `\outer` (making them no-ops). I also had on my agenda to promote `\protected` to a normal command code. And, already for a long time I wanted to play with a new prefix:[54]

5. `\frozen`: this will protect a macro (for now only a macro) against redefinition, which provides a bit of protection for a user.

Promoting `\protected` brings the set of call commands from four to eight, and a `\frozen` property would bump it to sixteen. This is still okay, but in some places it would involve mode testing. However, dropping `\long` and `\outer` would not only keep the set small (just four) but also rid it of some tests. There is no performance penalty either (even a bit of gain in case of many protected macros as we no longer need to skip the special signal token) and it even saves some memory (but not that much).

As a bonus there are a few more conditionals: `\ifprotected`, `\iffrozen`, and, very experimental, `\ifusercmd`, which can be used to check if something is user defined (often not a primitive). These probably only make sense for diagnostic purposes.

In the end, the implementation was not that hard. In the process I also removed the `\suppress...` parameters so `\par` no longer plays havoc. If this new prefix `\frozen` stays of will affect more definitions, we'll see.

16.2 Conditionals

Another domain where there have been some extensions is conditions. In a previous chapter I mentioned `\iftok` already. As this is not a manual I will not go into details about other new conditionals. For instance we have a few that can be used to check for valid dimensions and numbers. This can lead to a bit cleaner code, although for instance in ConTEXt we always used support macros for this. We seldom needed more than we had but when interfacing with MetaPost it helps a little.

Another, maybe interesting one is `\ifcondition` which when TEX is in jump over branches mode is seen as a valid `\if<cmd>` token but when it comes to expansion the following macro determines a true or false state. A second nice experiment is `\orelse` which is to be followed by a valid `\if<cmd>` token and makes for less nesting which sometimes looks nicer and also has some advantages.

[54] This is a typical example of a feature that I like playing with, before deciding if it will stay (as such).

I might wrap up these and other extensions in articles once they are considered stable and useful. But first I'll test them in real situation, which in practice means that ConTEXt users will test them, probably without noticing.

17 Retrospect

At some point in a new development, and LuaMetaTEX feels like that, there comes a moment when you need to make a decision. In this case the question is if we need to make hybrid MkIV and LMTX files or do the same as with the transition from MkII to MkIV: use two variants. For TEX files a conditional section has only overhead in the format generation as skipped code doesn't end up in the format. With conditional Lua code it's different: the ignored section is still present in byte code. But even for TEX code a conditional section is not entirely invisible: encountered control sequences are still creating (bogus) hash entries. So the question is: do we go lean and mean and do we omit historic non-LMTX code?

A comparison with the transition from MkII is actually relevant. For instance right from the start ConTEXt had an abstract backend layer, and support for engines and output formats was loaded on demand. There was never any specific code in the core. With MkIV we changed the model but there is still some abstraction.

In MkII we also had to deal with encodings and that has consequences for font handling, language support and input encodings. In MkIV all that changed: internal all is utf, as is normally the input (but we can still use encodings), and fonts are always mapped to Unicode.

Anyhow, much that made sense for MkII was no longer relevant for MkIV: code could be dropped. But some mechanisms were reimplemented using Lua: code was added. The user interface stayed the same but in MkIV uses a conceptually different approach deep down. Therefore the code base was split in MkII and MkIV files but this transition was made stepwise.

So should the same happen with LMTX? There is not that much that needs to be added to MkIV in terms of functionality. In the end, for the TEX code the differences are not that substantial, so there we can consider loading different files. The files involved are rather stable so there is not much danger of functionality between MkIV and LMTX getting out of sync. The same is true for the Lua files, although synchronization is probably more an issue there.

Another option is to always assume that LuaMetaTEX is used. For testing regular LuaTEX (patches) we can just use a 2019 stable ConTEXt. But in order for users to benefit from developments we then expect them all to move on to LMTX. Using a frozen 2019 version with upcoming LuaTEX is no big deal as we've done the same with MkII and that worked out okay.

When we started with ConTEXt development in the previous century we were doing pretty weird things. I remember getting comments that what we did made no sense

because it was not what TEX was meant for and some even suggested that it disrupted the picture. Highly structured input, a clear separation (and abstraction) of front and backend, inheritance and user defined styling, integrated support for xml, embedded MetaPost, advanced interactive documents, handling of fonts en encodings, the list is long. Occasionally some of the things that came with ConTEXt were ridiculed, like the fact that a script was used to manage the (multiple) run(s), but in the end, look at how many script are around now. Some even wondered why we used TEX at all because TEX was meant for typesetting math. And who needs xml let alone MathML? Or interactive pdf features? Much in ConTEXt and its management got smoother over time and the LuaMetaTEX engine fits nicely into this evolution. It's hard to keep the cutting edge but at least we have the instruments.

During BachoTEX 2019 (end of April, beginning of May) this project was presented the first time outside the ConTEXt community. During that meeting Mojca Miklavec, one of the driving forces behind ConTEXt, upgraded the compile farm that already was used to compile (intermediate versions of) LuaTEX and TEXLive to also compile `pplib` (handy for development) and LuaMetaTEX. This permits us to fine-tune the `cmake` setup which is still work in progress. And, also further improvements take place in the code base itself.

One of the properties of open source is that one can build upon an existing code base, so when at BachoTEX Arthur announced that he was going to make a merge of XETEX (which he maintains) and LuaTEX no one was surprised. But it could be a strong argument for a rather strict code freeze: spin-offs need stability. I've been told that there are now several projects where more libraries (like Harfbuzz) get integrated. Those cases don't influence the parent but here stability of the original also is expected, unless of course additional features go in these engines, which itself creates instability, but that's another matter. One could actually argue that the arrival of variants defeats the argument that stability is important: if a macro package uses new features, it needs to adapt, and naturally (temporary) issues might show up. Such are the dynamics of todays software development. History in general shows that not that much is persistent (or even accumulative) and programs are probably the least, so maybe the whole stability aspect has lost its relevance.[55] Of course LuaMetaTEX is also a follow up, but one of the ideas behind it was that I could use it as platform for (independent) experiments that could result in code being put into LuaTEX. Also, the changes have a limited impact: only ConTEXt will be affected.[56]

It is not feasible to make ConTEXt work with all kind of engines that in practice are not

[55] In a similar way as that the argument “Publishers want this or that, so we as TEX community need to provide it.” is no longer that relevant because publishing is now more a business model than vocation.

[56] So maybe, in the end, stability boils down to “The engine behaves the same and the ConTEXt that comes with it exploits its features as good as possible”.

used by its users. For instance, after XeTeX showed up it went through several iterations or font rendering, so we never really spent time on the low level features that it provided (there was no demand anyway). One cannot simply claim that one method is better than another that replaces it and expect constant adaptation (probably for the sake of a few potential users). There simply is no 'best' engine and no 'perfect' solution. Another aspect is that when we would adapt ConTeXt to LuaTeX variants the dependencies on specific functionality that itself depends on the outside world is kind of unavoidable. Especially languages and fonts are fluid and for the average user there is not that much difference in that department. Should we really complicate matters for a few (potential) users? In ConTeXt support like that is added on demand, driven by specific needs of users who use TeX for a reason and are willing to test.

There's enough huge and complex software around that demonstrates what happens when programs are extended, keep growing, their code base becoming more complex. Such a process doesn't really fit in my ideas about for TeX. We positioned 1.10 as long term stable, with the option to add a few handy things in the long run. For sure there are niches to fill and it is a fact that the TeX community can deal with variants of engines: just look at the different cjk engines around, with prefixes like p, up, ep, etc. But the question is, where does that put further LuaTeX development? And, more important, what consequences does it have for the ConTeXt code base?

The reason I mention this is that I had in mind to eventually backport features that work out well in LuaMetaTeX. I also mentioned that in order to support stock LuaTeX it made no sense to split the ConTeXt code base. After all, a few conditional sections could deal with the difference between LuaTeX and LuaMetaTeX: some differences could be temporary anyway. But, given recent developments it actually made sense to split the code base: why spent time on backporting when the engine user base is spread over different spinoffs. I can better just assume ConTeXt to exclusively use LuaMetaTeX and that other macro packages use (one or more) LuaTeX variants. I can then keep the generic code up to date and maybe occasionally add some proven stable features. It is also no big deal to keep the minimum subset needed for (plain) font handling compatible, assuming LuaTeX compatibility, as in the end that engine is the benchmark, especially when I strip it a bit from features not needed outside ConTeXt.

Thoughts like this show how fragile plans and predictions are: within a year one has to adapt ideas and assumptions. But it also proves that LuaMetaTeX was a good choice for ConTeXt, especially because it is bound to ConTeXt development, which keep the users independent and isolated from developments that don't mind that much the (side) effects on ConTeXt.

Around the ConTeXt meeting (or maybe a bit later) we hope to have the new installation infrastructure stable too (currently it is also experimental). By that time it will also be clear how we will proceed with the LMTX project. In the meantime I have decided so put

LuaMetaTEX specific files alongside the MkIV files, simply because I always need to be able run stock LuaTEX. In order to show the close relationship these files are flagged as MkXL, so we bump from 'Mark Four' to 'Mark Fourty'. The suffixes `mkiv`, `mkvi` and `mpiv` get company from `mkxl`, `mklx` and `mpxl`. Depending on backporting features, files can come and go. I'm not yet sure about the Lua files but the `lmt` suffix is already reserved for future use.[57] All this is also driven by (user) demand.

Consider this (and these thoughts) a snapshot. There will be the usual reports on experiments and developments. And in due time there will also be a manual for LuaMetaTEX.[58] And yes, at some point I have to make up my mind with respect to backporting features that have proven to be useful.

[57] This is because Lua 5.4 introduces some new syntax elements and where we can get away with the difference between 5.2 (LuajitTEX) and 5.3 (LuaTEX) such a syntax change is more drastic.

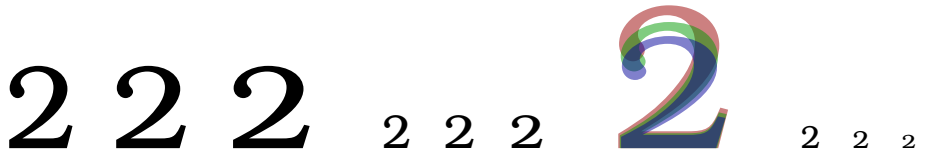
[58] In fact it already lives on my machine but I'm not in ready yet for the usual complaints about manuals, so I'm not in that much of a hurry.

18 Scaled fonts

18.1 History

The infrastructure for fonts makes up a large part of the code of any TEX macro package. We have to go back in time to understand why. When TEX showed up, fonts were collections of bitmaps and measures. There were at most 256 glyphs in a font and in order to do its job, TEX needed to know (and still needs to know) the width, height and depth of glyphs. If you want ligatures it also needs to know how to construct them from the input and when you want kerning there has to be additional information about what neighboring glyphs need a kern in between. Math is yet another subtask that demands extra information, like chains of glyphs that grow in size and if needed even recipes of how to construct large shapes from smaller ones.

Fonts come in sizes. Latin Modern and the original Computer Modern, for instance, have quite a few variants where the shapes are adapted to the size. This means that when you need a 9pt regular shape alongside a 12pt one, two fonts have to be loaded. This is quite visible in math where we have three related sizes: text, script and scriptscript, grouped in so called families. When we scale the digit 2 to the same height you will notice that the text, script and scriptscript sizes look different (the last three are unscaled):



Plenty has been written (in various documents that come with ConTEXt) about how this all works together and how it impacts the design of the system, so here I just give a short summary of what a font system has to deal with.

- In a bodyfont setup different sizes (9pt, 10pt, 12pt) can have their own specific set of fonts. This can result in quite a number of definitions that relate to the style, like regular, bold, italic, bold italic, slanted, bold slanted, etc. When possible loading the fonts is delayed. In ConTEXt often the number of fonts that are actually loaded is not that large.

- Some font designs have different shapes per bodyfont size. A minor complication is that when one is missing some heuristic best-match choice might be needed. Okay, in practice only Latin Modern falls into this category for ConTEXt. Maybe OpenType variable fonts can be seen this way, but, although we supported that right from the start, I haven't noticed much interest in the TEX community.

- Within a bodyfont size we distinguish size variants. We can go smaller (x and xx), for instance when we use sub- and superscripts in text, or we can go larger, for instance in titles (a, b, c, d, ...). Fortunately most of the loading of these can be delayed too.

- When instances are not available, scaling can be used, as happens for instance with 11pt in Computer Modern. Actually, this is why in ConTEXt we default to 12pt, because the scaled versions didn't look as nice as the others (keep in mind that we started in the age of bitmaps).

- Special features, such as smallcaps or oldstyle numerals, can demand their own definitions. More loading and automatic definitions can be triggered by sizes needed in, e.g., scripts and titles.

- A document can have a mixed setup, that is: using different font designs within one document, so some kind of namespace subsystem is needed.

- In an eight-bit font world, we not only have text fonts but also collections of symbols, and even in math there are additional symbol collections. In OpenType symbols end up in text fonts, but there we have tons of emojis and color fonts. All has to be dealt with in an integrated way. And we're not even talking of virtual fonts, (runtime) MetaPost generated fonts, and so on.

- In traditional eight-bit engines, hyphenation depends on a font's encoding, which can require loading a font multiple times in different encodings. This depends on the language mix used. A side point is that defining a European encoding covering most Latin languages was not that hard, especially when one keeps in mind that many eight-bit encodings waste slots on seldom used symbols, but by that time OpenType and Unicode input started to dominate.

- In the more modern OpenType fonts combinations of features can demand additional instances: one can think of language/script combinations, substitutions in base mode, special effects like emboldening, color fonts, etc.

- Math is complicated by the fact that in traditional TEX, alphabets come from different fonts, which is why we have many so-called families; a font can have several alphabets which means that some mapping can be needed. Operating on the size, shape, encoding and style axes puts some demands on the font system. Add to this the (often) partial (due to lack of fonts) bold support and it gets even more complicated. In OpenType all the alphabets come from one font.

- There is additional math auto-definition and loading code for the sizes used in text scripts and titles.

All this has resulted in a pretty complex subsystem. Although going OpenType (and emulated OpenType with Type1 fonts as we do in MkIV) removes some complications, like encodings, it also adds complexity because of the many possible font features, either dependent or not on script and language. Text as well as math got simpler in the TEX code, though that was traded for quite a bit of Lua code to deal with new features.

So, in order to let the font subsystem not impact performance too much, let alone extensive memory usage, the ConTEXt font subsystem is rather optimized. The biggest burden comes from fonts that have a dynamic (adaptive) definition because then we need to do quite a bit of testing per font switch, but even that has always been rather fast.

18.2 Reality

In MkIV and therefore also in LuaMetaTEX (LMTX) more font magic happens. The initial node lists that make up a box or paragraph can get manipulated in several ways and often fonts are involved. The font features (smallcaps, oldstyle, alternates, etc.) can be defined as static (part of the definition) or as dynamic (resolved on the spot at the cost of some overhead). Characters can be remapped, fonts can be replaced. The math subsystem in MkIV was different right from the start: we use a limited number of families (regular, bold, l2r and r2l), and stay abstract till the moment we need to deal with the specific alphabets. But still, in MkIV, we have the families with three fonts.

In the LuaMetaTEX manual we show some math magic for different fonts. As a side effect, we set up half a dozen bodyfont collections: Lucida, Pagella, Latin Modern, Dejavu, the math standard Cambria, etc. Even with delayed and shared font loading, we end up with 158 instances but quite a few of them are math fonts, at least six per bodyfont size: regular and bold (emboldened) text, script and scriptscript. Of course most are just copies with different scaling that reuse already loaded resources. In the final pdf we have 21 subsetted fonts.

If we look at the math fonts that we use today, there is however quite some overlap. It starts with a text font. From that, script and scriptscript variants are derived, but often these variants use many text size related shapes too. Some shapes get alternatives (from the `ssty` feature), and the whole clone gets scaled. But, much of the logic of, for instance, extensibles is the same.

A similar situation happens with large cjk fonts: there are hardly any advanced features involved there, so any size is basically a copy with scaled dimensions, and these fonts can be truly huge!

When we talk about features, in many cases in ConTEXt you don't define them as part of the font. For instance small caps can best be triggered by using a dynamic feature: applied to a specific stretch of text. In fact, often features like superiors of fractions

only work well on characters that fit the bill and produce weird side effects otherwise (a matter of design completeness). When the font handler does its work there are actually four cases: no features get applied (something that happens with, for instance, most monospaced fonts); base mode is used (which means that the TEX machinery takes care of constructing ligatures and injecting kerns); and node mode (where Lua handles the features). The fourth case is a special case of node mode where a different feature set is applied.[59] At the cost of some extra overhead (for each node mode run) dynamic features are quite powerful and save quite a lot of memory and definitions.[60] The overhead comes from much more testing regarding the font we deal with because suddenly the same font can demand different treatments, depending on what dynamic features are active.[61]

Although the font handling is responsible for much of the time spent in Lua, it is still reasonable given what has to be done. Because we have an extensible system, it's often the extensions that takes additional runtime. Flexibility comes at a price.

18.3 Progress

At some point I started playing with realtime glyph scaling. Here realtime means that it doesn't depend on the font definition. To get an idea, here is an example (all examples are additionally scaled for TugBoat):

```
test {\glyphxscale 2500 test} test
```

test **test** test

The glyphs in the current font get scaled horizontally without the need for an extra font instance. Now, this kind of trickery puts some constraints on the font handling, as is demonstrated in the next example. We use Latin Modern because that font has all these ligatures:

```
\definedfont[lmroman10-regular*default]%
e{\glyphxscale 2500 ff}icient
ef{\glyphxscale 2500 f}icient
ef{\glyphxscale 2500 fi}cient
e{\glyphxscale 2500 ffi}cient
```

e**ff**icient ef**f**icient ef**fi**cient e**ffi**cient

[59] We also have so-called plug mode where an external renderer can do the work but that one is only around due to some experiments during Idris Hamid's font development.

[60] The generic font handler that is derived from the ConTEXt one doesn't implement this, so it runs a little faster.

[61] Originally this model was introduced for a dynamic paragraph optimization subsystem for Arabic but in practice no one uses it because there are no suitable fonts.

In order to deal with this kind of scaling, we now operate not only on the font (id) and dynamic feature axes, but also on the scales, of which we have three variants: glyph scale, glyph xscale and glyph yscale. There is actually also a state dimension but we omit that for now (think of flagging glyphs as initial or final). This brings the number of axes to six. It is important to stress that in these examples the same font instance is used!

Just for the record: several approaches to switching fonts are possible but for now we stick to a simple font id switch plus glyph scale settings at the TEX end. A variant would be to introduce a new mechanism where id's and scales go together but for now I see no real gain in that.

18.4 Math

Given what has been discussed in the previous sections, a logical question would be "Can we apply scaling to math?" and the answer is "Yes, we can!". We can even go a bit further and that is partly due to some other properties of the engine.

From pdfTEX the LuaTEX engines inherited character protrusion and glyph expansions, aka hz. However, where in pdfTEX copies of the font are made that carry the expanded dimensions, in LuaTEX at some point this was replaced by an expansion field in the glyph and kern nodes. So, instead of changing the font id of expanded glyphs, the same id is used but with the applied expansion factor set in the glyph. A side effect was that in places where dimensions are needed, we call functions that calculate the expanded widths on request (as these can change during linebreak calculations) in combination with accessing font dimensions directly. This level of abstraction is even more present in LuaMetaTEX. This means that we have an uniform interface to fonts and as a side effect scaling need be dealt with in only a few places in the code.

Now, in math we have a few more complications. First of all, we have three sizes to consider and we also have lots of parameters that depend on the size. But, as I wanted to be able to apply scaling to math, the whole machinery was also abstracted in a way that, at the cost of some extra overhead, made it easier to work with scaled glyph properties. This means that we can stick to loading only one bodyfont size of math (note that each math family has three sizes, where the script and script sizes can have different, fine tuned, shapes) and just scale that on demand.

Once all that was in place it was a logical next step to see if we could stick to just a single instance. Because in LuaMetaTEX we try to load fonts efficiently we store only the minimally needed information at the TEX end. A font with no math therefore has less data per glyph. Again, this brings some abstraction that helped to implement the one instance mechanism. A math glyph has optional lists of increasing sizes and vertical or horizontal extensibles. So what got added was an optional chain of smaller sizes. If a character

has three different glyphs for the three sizes, the text glyph has a pointer to the script glyph which in turn has a pointer to the scriptscript glyph. This means that when the math engine needs a specific character at a given size (text, script, scriptscript) we just follow that chain.

In an OpenType math font the script and scriptscript sizes are specified as percentages of the text size. When the dimensions of a glyph are needed, we just scale on the fly. Again this adds some overhead but I'm pretty sure that no user will notice.

So, to summarize: if we need a character at scriptscript size, we access the text size glyph, check for a pointer to a script size, go there, and again check for a smaller size. We use only what fits the bill. And, when we need dimensions we just scale. In order to scale we need the relative size, so we need to set that up when we load the font. Because in ConTEXt we also can assemble a virtual OpenType font from Type1 fonts, it was actually that (old) compatibility feature, the one that implements Type1 based on OpenType math, that took the most time to adapt, not so much because it is complicated but because in LMTX we have to bypass some advanced loading mechanisms. Because we can scale in two dimensions the many (font-related) math parameters also need to be dealt with accordingly.

The end result is that for math we now only need to define two fonts per bodyfont setup: regular and bold at the natural scale (normally 10pt) and we share these for all sizes. As a result of this and what we describe in the next section, the 158 instances for the LuaMetaTEX manual can be reduced to 30.

18.5 Text

Sharing instances in text mode is relatively simple, although we do have to keep in mind that scaling is an extra axis when dealing with font features: two neighboring glyphs with the same font id and dynamics but with different scales are effectively from different fonts.

Another complication is that when we use font fallbacks (read: take missing glyphs from another font) we no longer have a dedicated instance but use a shared one. This in itself is not a problem but we do need to handle specified relative scales. This was not that hard to patch in ConTEXt LMTX.

We can enforce aggressive font sharing with:

```
\enableexperiments[fonts.compact]
```

After that we often use fewer instances. Just to give an idea, on the LuaMetaTEX manual we get these stats:

```
290 pages, 10.8 sec, 292M lua, 99M tex, 158 instances
290 pages,  9.5 sec, 149M lua, 35M tex,  30 instances
```

So, we win on all fronts when we use this glyph scaling mechanism. The magic primitive that deals with this is named `\glyphscale`; it accepts a number, where `1200` and `1.2` both mean scaling to 20% more than normal. But it's best not to use this primitive directly.

A specific scaled font can be defined using the `\definefont` command. In LMTX a regular scaler can be followed by two scale factors. The next example demonstrates this (as can be seen, the `yoffset` affects the baseline):

```
\definefont[FooA][Serif*default @ 12pt 1800 500]
\definefont[FooB][Serif*default @ 12pt 0.85 0.4]
\definefont[FooC][Serif*default @ 12pt]

\definetweakedfont[runwider] [xscale=1.5]
\definetweakedfont[runtaller][yscale=2.5,xscale=.8,yoffset=-.2ex]

{\FooA test test \runwider test test \runtaller test test}\par
{\FooB test test \runwider test test \runtaller test test}\par
{\FooC test test \runwider test test \runtaller test test}\par
```

We also use the new `\definetweakedfont` command here. This example not only shows the two scales but also introduces the offset.

test test test test test test

test test test test test test

test test test test test test

In compact mode this is one font. Here is another example:

```
\definetweakedfont[squeezed][xscale=0.9]

\startlines
$a = b^2 + \sqrt{c}$
{\squeezed $a = b^2 + \sqrt{c}$}
\stoplines
```

$a = b^2 + \sqrt{c}$
$a = b^2 + \sqrt{c}$

Watch this:

```
\startcombination[3*1]
    {\bTABLE
        \bTR \bTD foo \eTD \bTD[style=\squeezed] $x = 1$ \eTD \eTR
        \bTR \bTD oof \eTD \bTD[style=\squeezed] $x = 2$ \eTD \eTR
     \eTABLE}
    {local}
    {\bTABLE[style=\squeezed]
        \bTR \bTD $x = 1$ \eTD \bTD  $x = 3$ \eTD \eTR
        \bTR \bTD $x = 2$ \eTD \bTD  $x = 4$ \eTD \eTR
     \eTABLE}
    {global}
    {\bTABLE[style=\squeezed\squeezed]
        \bTR \bTD $x = 1$ \eTD \bTD  $x = 3$ \eTD \eTR
        \bTR \bTD $x = 2$ \eTD \bTD  $x = 4$ \eTD \eTR
     \eTABLE}
    {multiple}
\stopcombination
```

foo	$x = 1$
oof	$x = 2$

local

$x = 1$	$x = 3$
$x = 2$	$x = 4$

global

$x = 1$	$x = 3$
$x = 2$	$x = 4$

multiple

An additional style parameter is also honored:

```
\definetweakedfont[MyLargerFontA][scale=2000,style=bold]
test {\MyLargerFontA test} test
```

This gives:

test **test** test

Just for the record: the Latin Modern fonts, when set up to use design sizes, will still use the specific size-related files.

18.6 Hackery

You can use negative scale values, as is demonstrated in the following code:

```
\bTABLE[align=middle]
  \bTR
    \bTD a{\glyphxscale  1000 \glyphyscale  1000 bc}d \eTD
    \bTD a{\glyphxscale  1000 \glyphyscale -1000 bc}d \eTD
```

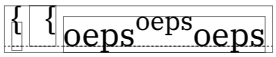
```
    \bTD a{\glyphxscale -1000 \glyphyscale -1000 bc}d \eTD
    \bTD a{\glyphxscale -1000 \glyphyscale  1000 bc}d \eTD
  \eTR
  \bTR
    \bTD \tttf +1000 +1000 \eTD
    \bTD \tttf +1000 -1000 \eTD
    \bTD \tttf -1000 -1000 \eTD
    \bTD \tttf -1000 +1000 \eTD
  \eTR
\eTABLE
```

gives:

abcd	a pc d	da cq	dd
+1000 +1000	+1000 -1000	-1000 -1000	-1000 +1000

Glyphs can have offsets and these are used for implementing OpenType features. However, they are also available on the TEX side. Take this example where we use the new `\glyph` primitive (a variant of `\char` that takes keywords):

```
\ruledhbox{
    \ruledhbox{\glyph yoffset 1ex options 0 123} % left curly brace
    \ruledhbox{\glyph xoffset .5em yoffset 1ex options "C0 123}
    \ruledhbox{oeps{\glyphyoffset 1ex \glyphxscale 800
                    \glyphyscale\glyphxscale oeps}oeps}
}
```



This example demonstrates that the `\glyph` primitive takes quite a few keywords: `xoffset`, `yoffset`, `xscale`, `yscale`, `left`, `right`, `raise`, `options`, `font` and `id` where the last two take a font identifier or font id (a positive number). For this article it's enough to know that the option indicates that glyph dimension should include the offset. In a moment we will see an alternative that doesn't need that.

```
\samplefile{jojomayer}
{\glyphyoffset .8ex
 \glyphxscale 700 \glyphyscale\glyphxscale
 \samplefile{jojomayer}}
{\glyphyscale\numexpr3*\glyphxscale/2\relax
 \samplefile{jojomayer}}
{\glyphyoffset -.2ex
```

```
 \glyphxscale 500 \glyphyscale\glyphxscale
 \samplefile{jojomayer}}
\samplefile{jojomayer}
```

To quote Jojo Mayer:

> If we surrender the thing that separates us from machines, we will be replaced by machines. The more advanced machines will be, the more human we will have to become. If we surrender the thing that separates us from machines, we will be replaced by machines. The more advanced machines will be, the more human we will have to become. If we surrender the thing that separates us from machines, we will be replaced by machines. The more advanced machines will be, the more human we will have to become. If we surrender the thing that separates us from machines, we will be replaced by machines. The more advanced machines will be, the more human we will have to become. If we surrender the thing that separates us from machines, we will be replaced by machines. The more advanced machines will be, the more human we will have to become.

Keep in mind that this can interfere badly with font feature processing which also used offsets. It might often work out okay vertically, but less well horizontally.

The scales, as mentioned, works with pseudo-scales but that is sometimes a bit cumbersome. This is why a special `\numericscale` primitive has been introduced.

```
1200 : \the\numericscale1200
1.20 : \the\numericscale1.200
```

Both these lines produce the same integer:

```
1200 : 1200
1.20 : 1200
```

You can do strange things with these primitives but keep in mind that you can also waste the defaults.

```
\def\UnKernedTeX
  {T%
   {\glyph xoffset -.2ex yoffset -.4ex `E}%
   {\glyph xoffset -.4ex options "60 `X}}

We use \UnKernedTeX\ and {\bf \UnKernedTeX} and {\bs \UnKernedTeX}:
the slanted version could use some more left shifting of the E.
```

This gives the TEX logos but of course we normally use the more official definitions instead.

We use TEX and TEX and *TEX*: the slanted version could use some more left shifting of the E.

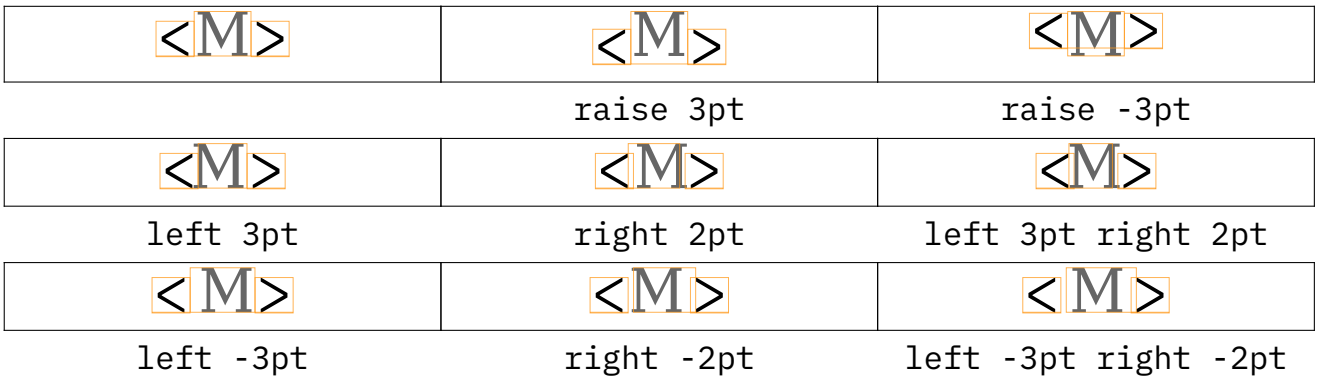
Because offsets are (also) used for handling font features like mark and cursive placement as well as special inter-character positioning, the above is suboptimal. Here is a better alternative:

```
\def\UnKernedTeX
  {T\glyph left .2ex raise -.4ex `E\glyph left .2ex `X\relax}
```

The result is the same:

We use TEX and TEX and *TEX*: the slanted version could use some more left shifting of the E.

But anyway: don't overdo it. We have dealt with such cases for decades without these fancy new features. The next example shows margins in action:



Here is another way of looking at it:

```
\glyphscale 4000
\vl\glyph                          `M\vl\quad
\vl\glyph raise  .2em              `M\vl\quad
\vl\glyph left   .3em              `M\vl\quad
\vl\glyph              right  .2em`M\vl\quad
\vl\glyph left  -.2em right -.2em`M\vl\quad
\vl\glyph raise -.2em right  .4em`M\vl
```

The raise as well as left and right margins are taken into account when calculating the dimensions of a glyph.

18.7 Implementation

Discussing the implementation in the engine makes no sense here, also because details might change. However, it is good to know that many properties travel with the glyph nodes, for instance the scales, margins, offsets, language, script and state properties, control over kerning, ligaturing, expansion and protrusion, etc. The dimensions (width, height and depth) are not stored in the glyph node but calculated from the font, scales and optionally the offsets and expansion factor. One problem is that the more clever (and nice) solutions we cook up, the more it might impact performance. So, I will delay some experiments till I have a more powerful machine.

One reason for *not* storing the dimensions in a glyph node is that we often copy those nodes or change character fields in the font handler and we definitely don't want the wrong dimensions there. At that moment, offsets and margin fields don't reflect features yet, so copying them is no big deal because at that moment these are still zero. However, dimensions are rather character bound so every time a character is set, we also would have to set the dimensions. Even worse, when we can set them, the question arises if they were already set explicitly. So, this is a can of worms we're not going to open: the basic width, height and depth of the glyph as specified in the font is used and combined with actual dimensions (likely already scaled according the glyph scales) in offset and margin fields.

Now, I have to admit that especially playing with using margins to glyphs instead of font kerns is more of an experiment to see what the consequences are than a necessity, but what would be the joy of TEX without such experiments? And as usual, in ConTEXt these will become options in the font handler that one can enable, or not.

19 Memory

19.1 Introduction

19.2 LUA

When you initialize Lua a proper memory allocator has to be provided. The allocator gets an old size and new size passed. When both are zero the allocator can `free` the blob, when the new size exceeds the old size the blob has to be `realloc`'s, and otherwise an initial `malloc` happens. When used with ConTEXt, LuaMetaTEX will do lots of calls to the allocator and often an initial allocation is followed by a reallocation, for instance because tables start out small but immediately grows a while after.

It is for this reason that early 2021 I decided to look into alternative allocators. I can of course code one myself, but where a LuaTEX run is a one time event, often with growing memory usage due to all kind of accumulating resources, using the engine as stand alone interpreter needs a more sophisticated approach than just keeping a bunch of bucket pools alive: when the script engine runs for months or even years memory should be returned to the operating system occasionally. We don't want the same side effects that html browsers have: during the day you need to restart them occasionally because they use up quite a bit of your computers memory (often for no real reason, so it probably has to do with keeping memory in store instead of returning it and/or it can be a side effect of a scattered pool ... who knows).

Instead of reinventing that wheel I ended up with testing Daan Leijen's `mimalloc` implementation: a not bloated, not too low level, reasonable sized library. Some simple experiments learned that it does make a difference in performance. The experiment was done with the native Microsoft compiler (msvc). One reason for that is that till that moment I preferred the cross compiled MingW versions (for cross compiling I use the linux subsystem that comes with MS Windows). Although native binaries compile faster and are smaller, the cross compiled ones perform somewhat better (often some 5%). Interesting is that making the format file is always much faster with a native binary, probably because the console output is supported better. When the alternative memory allocator is plugged into Lua suddenly the native version outperforms the cross compiled one (also by some 5%). The overall gain on a native binary for compiling the LuaMetaTEX manual is between 5 and 10% which was reason enough to continue this experiment. As a first step the native compiled version will default to it, later other platforms might follow.

19.3 TEX

Memory allocation in TEX has always been done by the engine itself. At startup a couple of big chunks are allocated and from that smaller blobs are taken. The largest chunks are for nodes, tokens and the table of equivalents (including the hash where control sequences are mapped onto registers and macros (lists of tokens). Smaller chunks are used for nesting states, after group restoration stacks, in- and output levels, etc. In modern engines the sizes of the chunks can be configured, some only at format generation time. In LuaMetaTEX we are more dynamic and after an initial (minimal) chunk allocation, when needed more memory will be allocated on demand, in steps, until a configured size is reached. That size has an upper limit (which if needed can be enlarged at compilation time). A side effect is that we (need to) do some more checking.

Node memory is special in the sense that nodes are basically offsets in a large array where each node has a number of slots after that offset. This is rather efficient in terms of performance and memory. New nodes (of any size) are taken from the node chunk and never returned. When freed they are appended to a list per size and that list serves as pool before new nodes get taken from the chunk. Variable size chunks are done differently, if only because we use them plenty in ConTEXt and they can lead to (excessive and) fragmented memory usage otherwise.

Tokens all have the same size so here there is only one list of free tokens. Because tokens and (most) nodes make it into linked lists those lists of free nodes and tokens are rather natural. And it's also fast. It all means that TEX itself does hardly any real memory allocation: only a few dozen large chunks. An exception is the string pool, where contrary to traditional TEX engines, the LuaTEX (and LuaMetaTEX) engines allocate strings using `malloc`. Those strings (used for control sequences) are never freed. In other cases where strings are used, like in for instance `\csname` construction, temporary strings are used. The same is true for some file related operations. None of these are real demanding in terms of excessive allocation and freeing. Also, in places that matter LuaMetaTEX is already quite optimized so using a different allocator gives no gain here.

Technically we could allocate nodes by using `malloc` but there are a few places in the engine that makes this hard. It can be done but then we need to make some conceptual changes (with regards to the way inserts are dealt with) and the question is if we gain much by breaking away from tradition. I guess there it will actually hurt performance if we change this. Another variant is where we allocate nodes of the same size from different pools but this doesn't bring us any gain either. A stringer argument is that changing the current (and historic) memory management of nodes will complicate the code.

A bit of an exception is the flow of information between Lua and TEX. There we do quite some allocation but it depends on how much a macro package demands of that.

19.4 METAPOST

When the MetaPost library was written, Taco changed the memory allocation to be more dynamic. One reason for this is that the number models (scaled, double, decimal, binary) have their own demands. For some objects (like numbers) the implementation uses a pool so it sits between the way TEX works and Lua when the standard allocator is used. This means that although quite some allocation is demanded, often the pool can serve the requests. (We might use a few more pools in the future.)

In LuaMetaTEX the memory related code has been reorganized a little so that (again as experiment) the `mimalloc` manager can be used. The performance gain is not as impressive as with Lua, but we'll see how that evolves when more demand poses more stress.

19.5 The verdict

In LuaMetaTEX version 2.09.4 and later the native MS Windows binaries now use the alternative `mimalloc` allocator. The gain is most noticeable for Lua and a little for TEX and MetaPost. The test suite with 2550 files runs in 1200 seconds which is quite an improvement over the MingW cross compiled binary that needs 1350 seconds. We do occasionally test a binary compiled with clang but that one is much slower than both others (compilation also takes much more time) but that might improve over time. Because of these results, it is likely that I'll also check out the other platforms, once the MS Windows binaries have proven to be stable (those are the once I use anyway).

20 Expressions

20.1 Introduction

Do we need bitwise expressions? Actually the answer is “no, although not until recently”. In ConTEXt MkII and MkIV we just use integer addition because we only need to enable things but in LMTX we want to control de detailed modes that some mechanisms in the engine provides and in order to not have tons of parameters these use bit sets. We manipulate these with the bitwise macros that actually are efficient Lua function calls. But, as with some other extensions in LuaMetaTEX, one way to prevent tracing clutter is to have a few handy primitives. So let's see what we got.

I haven't checked all operators and combinations yet!

20.2 Exploration

Already early in the LuaMetaTEX development (2019) the expression parser was extended with an integer division operator `:` that we actually use in LMTX, and soon after that I added basic bitwise operators but these were never activated but kept as comment because I didn't want to impact the scanner (even if we can afford to loose some performance because the scanner has been optimized). But in the process of cleaning up ‘todo’ comments in the source code I eventually arrived at expressions again.

The colon already makes the scanner incompatible because `\numexpr 1+2:` expects a number (which means that we cannot port back) and more operators only make that less likely. In ConTEXt I nearly always use `\relax` as terminator unless we're sure that lookahead is no issue.[62]

When going over the code in 2021, mostly because I wanted to get rid of some commented experiments, I decided that the extension should not go into the normal scanner but that a dedicated, simple and integer only scanner made more sense, so during a rainy summer weekend I started playing with that. It eventually became a bit more than initially intended, although the amount of code is rather minimal. The performance was

[62] In the ε-TEX expression parser, the normal `/` rounds the result. Both the `*` and `/` operator have a dedicated code path that assures no loss of accuracy. The `:` operator just divides like Lua's `//` which is an integer division operator. There are subtle differences between the division variants which can be noticeable when you go round trip. That is actually the main reason why this was one of the first things added to LuaMetaTEX as I wanted to get rid of some few scaled point rounding issues. The ε-TEX expression parser is somewhat complicated because it can deal with a mix of integers, dimensions and even glue, but always brings the result back to its main operating model. Because we adopted some of these ε-TEX rather early in ConTEXt lookahead pitfalls are taken care of already.

about twice that of the already available bitwise macros but operator precedence was not provided (apart from the multiplication and division operators). The final implementation was different, not that much faster on simple bitwise operations but could do more complex things in one go. Performance was not a real reason to provide this anyway because we're talking microseconds, it's more about less code and better readability.

The initial primitive command was `\bitexpr` and it supported nesting with parenthesis as the other expressions do. Because there are many operators, also verbose ones, the non-optional `\relax` token finishes parsing. But soon we moved on to two dedicated primitives.

20.3 Operators

The set of operators that we have to support is the following. Most have alternatives so that we can get around catcode issues.

add	+	
subtract	-	
multiply	*	
divide	/ :	
mod	%	mod
band	&	band
bxor	^	bxor
bor	| v	bor
and	&&	and
or	||	or
setbit	<undecided>	bset
resetbit	<undecided>	breset
left	<<	
right	>>	
less	<	
lessequal	<=	
equal	= ==	
moreequal	>=	
more	>	
unequal	<> != ~=	
not	! ~	not

I considered using `++` and type – as the `bset` and `bunset` shortcuts but that leads to issues because in TEX `-+-++--10` is a valid number and one never knows what sequence (without spaces) gets fed into an expression.

Originally I'd added some Unicode characters but for some reason support of logical operators is suboptimal so I removed that feature. Because these special characters are multi-byte utf sequences they are not that much better than verbose words anyway.

20.4 Integers and dimensions

When I was playing a bit with this feature, I wondered if we could mix in some dimensions. It was actually not that hard to add this: only explicit (verbose) dimensions had to be intercepted because dimen registers and such are seen as integers by the integer scanner. Once we're able do handle that, a next step was to make sure that `2 * 10pt` was permitted, something that the ε-TEX `\dimexpr` primitives can't handle. So, a variant of the dimen parser has to be used that makes the unit optional: `\dimexpression` and `\numexpression` were born.

The resulting parsers worked quite well but were about twice as slow as the normal expression scanners but that is no surprise because they do more. For instance we are case insensitive and need to handle letter and other (and in a few cases alignment and superscript) catcodes too. However, with a slightly tuned integer parser, also possible because the sentinel `\relax` makes parsing more predictable, and a dedicated unit scanner, in the end both the integer and dimension parser were performing well. It's not like we run them millions of times in a document.

Here is an example that results in 0xF8421:

```
\scratchcounter = \numexpression
    "00000 bor "00001 bor "00020 bor "00400 bor "08000 bor "F0000
\relax
```

And this gives 0xEFEFE:

```
\scratchcounter = \numexpression
    "FFFFF bxor "10101
\relax
```

We can give numerous example but you get the picture. In the above table you can see that some operators have equivalents. The reason for this is that a macro package can change catcodes and some characters have special meanings. So, the scanner is rather tolerant.

And this gives 'nop nop':

```
\scratchcounterone = 10
\scratchcountertwo = 20
```

```
\ifcase \numexpression
    (\scratchcounterone > 5) && (\scratchcountertwo > 5)
\relax yes\else nop\fi
%
\space
%
\scratchcounterone = 2
\scratchcountertwo = 4
\ifcase \numexpression
    (\scratchcounterone > 5) and (\scratchcountertwo > 5)
\relax nop\else yes\fi
```

The normal expansion rules apply, so one can use macros and other symbolic numbers. The only difference in handling dimensions is that we don't support `true` units but these are obsolete in LuaMetaTEX anyway.

In the end I decided to also add an extra conditional so that we can say:

```
\ifexpression (\scratchcounterone > 5) and (\scratchcountertwo > 5)\relax
    nop
\else
    yes
\fi
```

which looks more natural. Actually, this is an nowadays alias because we have two variants:

```
\ifnumexpression ... \relax ... \else ... \fi
\ifdimexpression ... \relax ... \else ... \fi
```

where the later is equivalent to

```
\ifboolean\dimexpression ... \relax ... \else ... \fi
```

20.5 Tracing

When `\tracingexpressions` is set to one or higher the intermediate 'reverse polish notation' stack that is used for the calculation is shown, for instance:

```
4:8: {numexpression rpn: 2 5 > 4 5 > and}
```

When you want the output on your console, you need to say:

```
\tracingexpressions 1
\tracingonline      1
```

The fact that we process the expression in two phases makes it possible to provide this kind of tracing.

20.6 Performance

The following table shows the results of 100.000 evaluations (per line) so you'll notice that there is a difference. But keep in mind that the new variant can so more, so it might pay off when we have cases that otherwise demand multiple traditional expressions.

```
\dimexpr 4pt*2 + 6pt\relax              : 0.049
\dimexpression 4pt*2 + 6pt\relax        : 0.047
\dimexpression 2*4pt + 6pt\relax        : 0.047

\numexpr 4 * 2 + 6\relax                : 0.038
\numexpression 2 * 4 + 6\relax          : 0.034

\numexpr 4*2+6\relax                    : 0.034
\numexpression 2*4+6\relax              : 0.038

\numexpr (1+2)*(3+4)\relax              : 0.045
\numexpression (1+2)*(3+4)\relax        : 0.060

\numexpr (1 + 2) * (3 + 4) \relax       : 0.050
\numexpression (1 + 2) * (3 + 4) \relax : 0.078
```

As usual I'll probably find some way to improve performance a bit but that might than also concern the traditional one. When we compare them, the new numeric scanner suffers from more options while the new dimension parser gain on the units. Also, keep in mind than the LuaMetaTEX normal parsers are already somewhat faster than the ones in LuaTEX. The numbers above are calculated when this document is rendered, so they may change over time and per run. The two engines compare as follows (mid 2021):

	LuaTEX	LuaMetaTEX
`\dimexpr 4pt*2 + 6pt\relax`	0.073	0.045
`\numexpr 4 * 2 + 6\relax`	0.034	0.028
`\numexpr 4*2+6\relax`	0.035	0.032
`\numexpr (1+2)*(3+4)\relax`	0.050	0.047
`\numexpr (1 + 2) * (3 + 4) \relax`	0.052	0.048

Of course tests like these are dubious because often cpu cache will keep the current code

accessible, but who knows.

It will probably take a while before I will use this in the source code because first I need to make sure that all works as expected and while doing that I might adapt some of this. But the basic framework is there.

21 The format file

It is interesting when someone compares macro package and used parameters like the size of a format file, the output of \tracingall, or startup time to make some point. The point I want to make here is that unless you know exactly what goes on in a run that involves a real document, which can itself involve multiple runs, such a comparison is rather pointless. For sure I do benchmark but I can only draw conclusions of what I (can) know (about). Yes, benchmarking your own work makes sense but doing that in comparisons to what you consider comparable variants assumes knowledge of more than your own work and objectives.

For instance, when you load few fonts, typeset one page and don't do anything that demands any processing or multiple runs, you basically don't measure anything. More interesting are the differences between 10 or 500 pages, a few font calls or tens of thousands, no color of extensive usage of color and other properties, interfacing, including inheritance of document constructs, etc. And even then, when comparing macro packages, it is kind of tricky to deduce much from what you observe. You really need to know what is going on inside and also how that relates to for instance adaptive font scaling. You can have a fast start up but if a users needs one tikz picture, loading that package alone will make you forget the initial startup time. You always pay a price for advanced features and integration! And we didn't even talk about the operating system caching files, running on a network share, sharing processors among virtual machines, etc.

Pointless comparing is also true for looking at the log file when enabling \tracingall. When a macro package loads stuff at startup you can be sure that the log file is larger. When a font or language is loaded the first time, or maybe when math is set up there can be plenty of lines dumped. Advanced analysis of conditions and trial runs come at a price too. And eventually, when a box is shown the configured depth and breadth really matter, and it might also be that the engine provides much more (verbose) detail. So, a comparison is again pointless. It can also backfire. Over the decades of developing ConTEXt I have heard people working on systems make claims like “We prefer not to . . . ” or “It is better to it this way . . . ” or (often about operating system) “It is bad that . . . ” just to see years later the same being done in the presumed better alternative. I can have a good laugh about that: do this and don't do that backfiring.

That brings us to the format file. When you make a ConTEXt format with the English user interface, with interfacing being a feature that itself introduces overhead, the LuaTEX engine will show this at the end:

```
Beginning to dump on file cont-en.fmt
 (format=cont-en 2021.6.9)
48605 strings using 784307 bytes
```

```
1050637 memory locations dumped; current usage is 414&523763
44974 multiletter control sequences
\font\nullfont=nullfont
0 preloaded fonts
```

The file itself is quite large: 11,129,903 bytes. However, it is actually much larger because the format file is compressed! The real size is 19.399.216. Not taking that into account when comparing the size of format files is kind of bad because compression directly relates to what resources a format use and how usage is distributed over the available memory blobs. The LuaTEX engine does some optimizations and saves the data sparse but the more holes you create, the worse it gets. For instance, the large character vectors are compartmentalized in order to handle Unicode efficiently so the used memory relates to what you define: do you set up all catcodes or just a subset. Maybe you delay some initialization to after the format is loaded, in which case a smaller format file gets compensated by more memory usage and initializaton time afterwards. Maybe your temporary macros create holes in the token array. The memory that is configured in the configuration files also matter. Some memory blobs are saved at their configured size, others dismiss the top part that is not used when saving the format but allocate the lot when the format is loaded. That means that memory usage in for instance LuaTEX can be much larger than a format file suggests. Keep in mind that a format file is basically a memory dump.

Now, how does LuaMetaTEX compare to LuaTEX. Again we will look at the size of the format file, but you need to keep in mind that for various reasons the LMTX macros are somewhat more efficient than the MkIV ones, in the meantime some new mechanism were added, which adds more TEX and Lua code, but I still expect (at least for now) a smaller format file. However when we create the format we see this (reformatted):

```
Dumping format 'cont-en.fmt 2021.6.9' in file 'cont-en.fmt':
tokenlist compacted from 489733 to 488204 entries,
1437 potentially aliased lua call/value entries,
max string length 69, 16 fingerprint
+ 16 engine + 28 preamble
+ 836326 stringpool
+ 10655 nodes + 3905660 tokens
+ 705300 equivalents
+ 23072 math codes + 493024 text codes
+ 38132 primitives + 497352 hashtable
+ 4 fonts + 10272 math + 1008 language + 180 insert
+ 10305643 bytecodes
+ 12 housekeeping = 16826700 total.
```

This looks quite different from the LuaTEX output. Here we report more detail: for each blob we mention the number of bytes used. The final result is a file that takes 16.826.700 bytes. That number should be compared with the 19.399.216 for LuaTEX. So, we need less indeed. But, when we compress the LuaMetaTEX format we get this: 5,913,932 which is much less than the 11,129,903 compressed size that the LuaTEX engine makes of it. One reason for using level 3 zip compression compression in LuaTEX is that (definitely when we started) it loads faster. It adds to creating the dump but doesn't really influence loading, although that depends a bit on the compiler used. It is not easy to see from these numbers what goes on, but when you consider the fact that we mostly store 32 bit numbers it will also be clear that many can be zero or have two or three zero bytes. There's a lot of repetition involved!

So let's look at some of these numbers. The mentioning of token list compaction relates to getting rid of holes in memory. Each token takes 8 bytes, 4 for the token identifier, internally called a cmd and chr, and 4 for a value like an integer or dimension value, or a glue pointer, or a pointer to a next token, etc. In our case compaction doesn't save that much.

The mentioning of potentially aliased Lua call/value entries is more a warning. Because the Lua engine starts fresh each run, you cannot store its ‘pointers’ and because hashes are randomized this means that you need to delay initialization to startup time, definitely for function tokens.

Strings in TEX can be pretty long but in practice they aren't. In ConTEXt the maximum string length is 69. This makes it possible to use one byte for registering the string length instead of four which saves quite a bit. Of course one large string will spoil this game.

The fingerprint, engine, preamble and later housekeeping bytes can be neglected but the string pool not. These are the bytes that make up the strings. The bytes are stored in format but when loaded become dynamically allocated. The LuaTEX engine and its successor don't really have a pool.

Now comes a confusing number. There are not tens of thousands of nodes allocated. A node is just a pointer into a large array so actually node references are just indices. Their size varies from 2 slots to 25; the largest are par nodes, while shape nodes are allocated dynamically. So what gets reported are the number of bytes that nodes take. each node slot takes 8 bytes, so a glyph node of 12 bytes takes 96 bytes, while a glue spec node (think skip registers) takes 5 slots or 40 bytes. These are amounts of memory that were not realistic when TEX was written. For the record: in LuaTEX glue spec nodes are not shared, so we have many more.

The majority of TEX related dump data is for tokens, and here we need 3905660 which means 488K tokens (each reported value also includes some overhead). The memory

used for the table of equivalents makes for some 88K of them. This table relates to macros (their names and content). Keep in mind that (math) character references are also macros.

The next sections that get loaded are math and text codes. These are the mentioned compartimized character properties. The number of math codes is not that large (because we delay much of math) but the text codes are plenty, think of lc, uc, sf, hj, catcodes, etc. Compared to LuaTEX we have more categories but use less space because we have an more granular storage model. Optimizing that bit really payed off, also because we have more vectors.

The way primitives and macro names get resolved is pretty much the same in all engines but by using the fact that we operate in 32 bit I could actually get rid of some parallel tables that handle saving and restore. Some optimizations relate to the fact that the register ranges are part of the game so basically we have some holes in there when they are not used. I guess this is why ε-TEX uses a sparse model for the registers above 255. What also saved a lot is that we don't need to store font names, because these are available in another way; even in LuaTEX that takes a large, basically useless, chunk. The memory that a macro without parameters consumes is 8 bytes smaller and in ConTEXt we have lots of these. We don't really store fonts, so that section is small, but we do store the math parameters, and there is not much we can save there. We also have more such parameters in LuaMetaTEX so there we might actually use more storage. The information related to languages is also minimal because patterns and exceptions are loaded at runtime. A new category (compared to LuaTEX) is inserts because in LuaMetaTEX we can use an alternative (not register based) variant. As you can see from the 180 bytes used, indeed ConTEXt is using that variant.

That leaves a large block of more than 10 million bytes that relates to Lua byte code. A large part of that is the huge Lua character table that ConTEXt uses. The implementation of font handling also takes quite a bit and we're not even talking of all the auxiliary Lua modules, xml processing, etc. When ConTEXt would load that on demand, which is nearly always, the format file would be much smaller but one would pay for it later. Loading the (some 600) Lua byte code chunks takes of course some time as does initialization but not much.

All that said, the reason why we have a large format file can be understood well if one considers what goes in there. The ConTEXt format files for pdfTEX and XETEX are 3.3 and 4.7 MB each which is smaller but not that much when you consider the fact that there is no Lua code stored and that there are less character tables and an ε-TEX register model used. But a format file is not the whole story. Runtime memory usage also comes at a price.

The current memory settings of ConTEXt are as follows; these values get reported when a format has been generated and can be queried at runtime an any moment:

	max	min	set	stp
string	2097152	150000	500000	100000
pool	100000000	10000000	20000000	1000000
hash	2097152	150000	250000	100000
lookup	2097152	150000	250000	100000
node	50000000	1000000	5000000	500000
token	10000000	1000000	10000000	250000
buffer	100000000	1000000	10000000	1000000
input	100000	10000	100000	10000
file	2000	500	2000	250
nest	10000	1000	10000	1000
parameter	100000	20000	100000	10000
save	500000	100000	500000	10000
font	100000	250	250	250
language	10000	250	250	250
mark	10000	50	50	50
insert	500	10	10	10

The maxima is what can be used at most. Apart from the magic number 2097152 all these maxima can be bumped at compile time but if you need more, you might wonder of your approach to rendering makes sense. The minima are what always gets allocated, and again these are hard coded defaults. The size can be configured and is normally the same as the minima but we use larger values in ConTEXt. The step is how much an initial memory blob will grow when more is needed than is currently available. The last four entries show that we don't start out with many fonts (especially when we use the ConTEXt compact font model not that many are needed) and because ConTEXt implements marks in a different way we actually don't need them. We do use the new insert properties storage model and for now the set sizes are enough for what we need.

In practice a LuaMetaTEX run uses less memory than a LuaTEX one, not only because memory allocation is more dynamic, but also because of other optimizations. When the compact font model is used (something ConTEXt) even less memory is needed. Even this claim should be made with care. Whenever I discuss the use of resources one needs to limit the conclusions to ConTEXt. I can't speak for other macro packages simply because I don't know the internals and the design decisions made and their impact on the statistics. As a teaser I show the impact of some definitions:

```
\chardef     \MyFooA1234
\Umathchardef\MyFooB"1 "0 "1234
\Umathcode   1 2 3 4
\def         \MyFooC{ABC}
\def         \MyFooD#1{A#1C}
```

```
\def         \MyFooE{\directlua{print("some lua")}}
```

The stringpool grows because we store the names (here they are of equal length). Only symbolic definitions bump the hashtable and equivalents. And with definitions that have text inside the number of bytes taken by tokens grows fast because every character in that linked list takes 8 bytes, 4 for the character with its catcode state and 4 for the link to the next token.

	stringpool	**tokens**	**equivalents**	**hashtable**	**total**
	836408	3906124	705316	497396	16828987
`\chardef`	836415	3906116	705324	497408	16829006
`\Umathchardef`	836422	3906116	705324	497420	16829025
`\Umathcode`	836422	3906124	705324	497420	16829033
`\def` (no arg)	836429	3906148	705332	497428	16829080
`\def` (arg)	836436	3906196	705340	497440	16829155
`\def` (text)	836443	3906372	705348	497452	16829358

So, every time a user wants some feature (some extra checking, a warning, color or font support for some element) that results in a trivial extension to the core, it can bump the size fo the format file more than you think. Of course when it leads to some overhaul sharing code can actually make the format shrink too. I hope it is clear now that there really is not much to deduce from the bare numbers. Just try to imagine what:

```
\definefilesynonym
  [type-imp-newcomputermodern-book.mkiv]
  [type-imp-newcomputermodern.mkiv]
```

adds to the format. Convenience has a price.