ConTEXt: Updating the code base

Hans Hagen

1 Introduction

After much experimenting with new code in MkIV a new stage in ConTEXt development was entered in the last quarter of 2011. This was triggered by several more or less independent developments. I will discuss some of them here since they are a nice illustration of how ConTEXt evolves.

2 Interfacing

Wolfgang Schuster, Aditya Mahajan and I were experimenting with an abstraction layer for module writers. In fact this layer itself was a variant of some new mechanisms used in the MkIV structure related code. That code was among the first to be adapted as it is accompanied by much Lua code and has been performing rather well for some years now.

In ConTEXt most of the user interface is rather similar and module writers are supposed to follow the same route as the core of ConTEXt. For those who have looked in the source the following code might look familiar:

```
\unexpanded\def\mysetupcommand
  {\dosingleempty\domysetupcommand}

\def\domysetupcommand[#1]%
  {..........
   \getparameters[\??my][#1]%
   ..........}
```

This implements the command `\mysetupcommand` that is used as follows:

```
\mysetupcommand[color=red,style=bold,...]
```

The above definition uses three rather low-level interfacing commands. The `\unexpanded` makes sure that the command does not expand in unexpected ways in cases where expansion is less desirable. (Aside: The ConTEXt `\unexpanded` prefix has a long history and originally resulted in the indirect definition of a macro. That way the macro could be part of testing (expanded) equivalence. When ε-TEX functionality showed up we could use `\protected` but we stuck to the name `\unexpanded`. So, currently ConTEXt's `\unexpanded` is equivalent to ε-TEX's `\protected`. Furthermore, in ConTEXt `\expanded` is not the same as the ε-TEX primitive. In order to use the primitives you need to use their `\normal...` synonyms.) The `\dosingleempty` makes sure that one argument gets seen by injecting a dummy when needed. At some point the `\getparameters` command will store the values of keys in a namespace that is determined by `\??my`. The namespace used here is actually one of the internal namespaces which can be deduced from the double question marks. Module namespaces have four question marks.

There is some magic involved in storing the values. For instance, keys are translated from the interface language into the internal language which happens to be English. This translation is needed because a new command is generated:

```
\def\@@mycolor{red}
\def\@@mystyle{bold}
```

and such a command can be used internally because in so-called unprotected mode `@?!` are valid in names. The Dutch equivalent is:

```
\mijnsetupcommando[kleur=rood,letter=vet]
```

and here the `kleur` has to be converted into `color` before the macro is constructed. Of course values themselves can stay as they are as long as checking them uses the internal symbolic names that have the language specific meaning.

```
\c!style{color}
\k!style{kleur}
\v!bold {vet}
```

Internally assignments are done with the `\c!` variant, translation of the key is done using the `\k!` alternative and values are prefixed by `\v!`.

It will be clear that for the English user interface no translation is needed and as a result that interface is somewhat faster. There we only need

```
\c!style{color}
\v!bold {bold}
```

Users never see these prefixed versions, unless they want to define an internationalized style, in which case the form

```
\mysetupcommand[\c!style=\v!bold]
```

has to be used, as it will adapt itself to the user interface. This leaves the `\??my` that in fact expands to `\@@my`. This is the namespace prefix.

Is this the whole story? Of course it isn't, as in ConTEXt we often have a generic instance from which we can clone specific alternatives; in practice, the `\@@mycolor` variant is used in a few cases only. In that case a setup command can look like:

```
\mysetupcommand[myinstance][style=bold]
```

And access to the parameters is done with:

```
\getvalue{\??my myinstance\c!color}
```

So far the description holds for MkII as well as MkIV, but in MkIV we are moving to a variant of this. At the cost of a bit more runtime and helper macros, we can get cleaner low-level code. The magic word here is `commandhandler`. At some point the new MkIV code started using an extra abstraction layer, but the code needed looked rather repetitive despite

subtle differences. Then Wolfgang suggested that we should wrap part of that functionality in a definition macro that could be used to define module setup and definition code in one go, thereby providing a level of abstraction that hides some nasty details. The main reason why code could look cleaner is that the experimental core code provided a nicer inheritance model for derived instances and Wolfgang's letter module uses that extensively. After doing some performance tests with the code we decided that indeed such an initializer made sense. Of course, after that we played with it, some more tricks were added, and eventually I decided to replace the similar code in the core as well, that is: use the installer instead of defining helpers locally.

So, how does one install a new setup mechanism? We stick to the core code and leave modules aside for the moment.

```
\definesystemvariable{my}
\installcommandhandler \??my {whatever} \??my
```

After this command we have available some new helper commands of which only a few are mentioned here (after all, this mechanism is still somewhat experimental):

```
\setupwhatever[key=value]
\setupwhatever[instance][key=value]
```

Now a value is fetched using a helper:

```
\namedwhateverparameter{instance}{key}
```

However, more interesting is this one:

```
\whateverparameter{key}
```

For this to work, we need to set the instance:

```
\def\currentwhatever{instance}
```

Such a current state macro already was used in many places, so it fits into the existing code quite well. In addition to `\setupwhatever` and friends, another command becomes available:

```
\definewhatever[instance]
\definewhatever[instance][key=value]
```

Again, this is not so much a revolution as we can define such a command easily with helpers, but it pairs nicely with the setup command. One of the goodies is that it provides the following feature for free:

```
\definewhatever[instance][otherinstance]
\definewhatever[instance][otherinstance][key=val]
```

In some cases this creates more overhead than needed because not all commands have instances. On the other hand, some commands that didn't have instances yet, now suddenly have them. For cases where this is not needed, we provide simple variants of commandhandlers.

Additional commands can be hooked into a setup or definition so that for instance the current situation can be updated or extra commands can be defined for this instance, such as `\start...` and `\stop...` commands.

It should be stressed that the installer itself is not that special in the sense that we could do without it, but it saves some coding. More important is that we no longer have the `@@` prefixed containers but use `\whateverparameter` commands instead. This is definitely slower than the direct macro, but as we often deal with instances, it's not that much slower than `\getvalue` and critical components are rather well speed-optimized anyway.

There is, however, a slowdown due to the way inheritance is implemented. That is how this started out: using a different (but mostly compatible) inheritance model. In the MkII approach (which is okay in itself) inheritance happens by letting values point to the parent value. In the new model we have a more dynamic chain. It saves us macros but can expand quite wildly depending on the depth of inheritance. For instance, in sectioning there can easily be five or more levels of inheritance. So, there we get slower processing. The same is true for `\framed` which is a rather critical command, but there it is nicely compensated by less copying. My personal impression is that due to the way ConTEXt is set up, the new mechanism is actually more efficient on an average job. Also, because many constructs also depend on the `\framed` command, that one can easily be part of the chain, which again speeds up a bit. In any case, the new mechanisms use much less hash space.

Some mechanisms still look too complex, especially when they hook into others. Multiple inheritance is not trivial to deal with, not only because the meaning of keys can clash, but also because supporting it would demand quite complex fully expandable resolvers. So for the moment we stay away from it. In case you wonder why we cannot delegate more to Lua: it's close to impossible to deal with TEX's grouping in efficient ways at the Lua end, and without grouping available TEX becomes less useful.

Back to the namespace. We already had a special one for modules but after many years of ConTEXt development, we started to run out of two character combinations and many of them had no relation to what name they spaced. As the code base is being overhauled anyway, it makes sense to also provide a new core namespace mechanism. Again, this is nothing revolutionary but it reads much more nicely.

```
\installcorenamespace {whatever}
\installcommandhandler
   \??whatever {whatever} \??whatever
```

This time deep down no `@@` is used, but rather something more obscure. In any case, no one will use

the meaning of the namespace variables, as all access to parameters happens indirectly. And of course there is no speed penalty involved; in fact, we are more efficient. One reason is that we often used the prefix as follows:

```
\setvalue{\??my:option:bla}{foo}
```

and now we just say:

```
\installcorenamespace {whateveroption}
\setvalue{\??whateveroption bla}{foo}
```

The commandhandler does such assignments slightly differently as it has to prevent clashes between instances and keywords. A nice example of such a clash is this:

```
\setvalue{\??whateveroption sectionnumber}{yes}
```

In sectioning we have instances named `section`, but we also have keys named `number` and `sectionnumber`. So, we end up with something like this:

```
\setvalue
  {\??whateveroption section:sectionnumber}{yes}
\setvalue
  {\??whateveroption section:number}{yes}
\setvalue{\??whateveroption :number}{yes}
```

When I decided to replace code similar to that generated by the installer a new rewrite stage was entered. Therefore one reason for explaining this here is that in the process of adapting the core code instabilities are introduced and as most users use the beta version of MkIV, some tolerance and flexibility is needed and it might help to know why something suddenly fails.

In itself using the commandhandler is not that problematic, but wherever I decide to use it, I also clean up the related code and that is where the typos creep in. Fortunately Wolfgang keeps an eye on the changes so problems that users report on the mailing lists are nailed down relatively fast. Anyway, the rewrite itself is triggered by another event but that one is discussed in the next section.

We don't backport (low-level) improvements and speedups to MkII, because for what we need TEX for, we consider pdfTEX and XETEX rather obsolete. Recent tests show that at the moment of this writing a LuaTEX MkIV run is often faster than a comparable pdfTEX MkII run (using UTF-8 and complex font setups). When compared to a XETEX MkII run, a LuaTEX MkIV run is often faster, but it's hard to compare, as we have advanced functionality in MkIV that is not (or differently) available in MkII.

3 Lexing

The editor that I use, called SciTE, has recently been extended with an extra external lexer module that makes more advanced syntax highlighting possible, using the Lua LPEG library. It is no secret that the user interface of ConTEXt is also determined by the way structure, definitions and setups can be highlighted in an editor.[1] When I changed to SciTE I made sure that we had proper highlighting there.

At Pragma one of the leading principles has always been: if the document source looks bad, mistakes are more easily made and the rendering will also be affected. Or phrased differently: if we cannot make the source look nice, the content is probably not structured that well either. The same is true for TEX source, although to a large extent there one must deal with the specific properties of the language.

So, syntax highlighting, or more impressively: lexing, has always been part of the development of ConTEXt and for instance the pretty printers of verbatim provide similar features. For a long time we assumed line-based lexing, mostly for reasons of speed. And surprisingly, that works out quite well with TEX. We used a simple color scheme suitable for everyday usage, with not too intrusive coloring. Of course we made sure that we had runtime spell checking integrated, and that the different user interfaces were served well.

But then came the LPEG lexer. Suddenly we could do much more advanced highlighting. Once I started playing with it, a new color scheme was set up and more sophisticated lexing was applied. Just to mention a few properties:

- We distinguish between several classes of macro names: primitives, helpers, interfacing, and user macros.
- In addition we highlight constant values and special registers differently.
- Conditional constructs can be recognized and are treated as in any regular language (keep in mind that users can define their own).
- Embedded MetaPost code is lexed independently using a lexer that knows the language's primitives, helpers, user macros, constants and of course specific syntax and drawing operators. Related commands at the TEX end (for defining and processing graphics) are also dealt with.
- Embedded Lua is lexed independently using a lexer that not only deals with the language but also knows a bit about how it is used in ConTEXt. Of course the macros that trigger Lua code are handled.
- Metastructure and metadata related macros are colored in a fashion similar to constants (after

[1] It all started with `wdt`, `texedit` and `texwork`, editors and environments written by myself in Modula2 and later in Perl Tk, but that was in a previous century.

all, in a document one will not see any constants, so there is no color clash).
- Some special and often invisible characters get a special background color so that we can see when there are for instance non-breakable spaces sitting there.
- Real-time spell checking is part of the deal and can optionally be turned on. There we distinguish between unknown words, known but potentially misspelled words, and known words.

Of course we also made lexers for MetaPost, Lua, XML, PDF and text documents so that we have a consistent look and feel.

When writing the new lexer code, and testing it on sources, I automatically started adapting the source to the new lexing where possible. Actually, as cleaning up code is somewhat boring, the new lexer is adding some fun to it. I'm not so sure if I would have started a similar overhaul so easily otherwise, especially because the rewrite now also includes speedup and cleanup. At least it helps to recognize less desirable left-overs of MkII code.

4 Hiding

It is interesting to notice that users seldom define commands that clash with low level commands. This is of course a side effect of the fact that one seldom needs to define a command, but nevertheless. Low-level commands were protected by prefixing them by one or more (combinations of) `do`, `re` and `no`'s. This habit is a direct effect of the early days of writing macros. For TEX it does not matter how long a name is, as internally it becomes a pointer anyway, but memory consumption of editors, loading time of a format, string space and similar factors determined the way one codes in TEX for quite a while. Nowadays there are hardly any limits and the stress that ConTEXt puts on the TEX engine is even less than in MkII as we delegate many tasks to Lua. Memory comes cheap, editors can deal with large amount of data (keep in mind that the larger the file gets, the more lexing power can be needed), and screens are wide enough not to lose part of long names in the edges.

Another development has been that in LuaTEX we have lots of registers so that we no longer have to share temporary variables and such. The rewrite is a good moment to get rid of that restriction.

This all means that at some point it was decided to start using longer command names internally and permit `_` in names. As I was never a fan of using `@` for this, underscore made sense. We have been discussing the use of colons, which is also nice, but has the disadvantage that colons are also used in the source, for instance to create a sub-namespace. When we have replaced all old namespaces, colons might show up in command names, so another renaming roundup can happen.

One reason for mentioning this is that users get to see these names as part of error messages. An example of a name is:

```
\page_layouts_this_or_that
```

The first part of the name is the category of macros and in most cases is the same as the first part of the filename. The second part is a namespace. The rest of the name can differ but we're approaching some consistency in this.

In addition we have prefixed names, where prefixes are used as consistently as possible:

`t_` token register
`d_` dimension register
`s_` skip register
`u_` muskip register
`c_` counter register, constant or conditional
`m_` (temporary) macro
`p_` (temporary) parameter expansion (value of key)
`f_` fractions

This is not that different from other prefixing in ConTEXt apart from the fact that from now on those variables (registers) are no longer accessible in a regular run. We might decide on another scheme but renaming can easily be scripted. In the process some of the old prefixes are being removed. The main reason for changing to this naming scheme is that it is more convenient to grep for them.

In the process most traditional `\if`s get replaced by 'conditionals'. The same is true for `\chardef`s that store states; these become 'constants'.

5 Status

We always try to keep the user interface constant, so most functionality and control stays stable. However, now that most users use MkIV, commands that no longer make sense are removed. An interesting observation is that some users report that low-level macros or registers are no longer accessible. Fortunately that is no big deal as we point them to the official ways to deal with matters. It is also a good opportunity for users to clean up accumulated hackery.

The systematic (file by file) cleanup started in the second half of 2011 and as of January 2012 one third of the core (TEX) modules have to be cleaned up and the planning is to get most of that done as soon as possible. However, some modules will be rewritten (or replaced) and that takes more time. In any case we hope that rather soon most of the code is stable enough that we can start working on new

mechanisms and features. Before that a cleanup of the Lua code is planned.

Although in many cases there are no fundamental changes in the user interface and functionality, I will wrap up some issues that are currently being dealt with. This is just a snapshot of what is happening currently and as a consequence it describes what users can run into due to newly introduced bugs.

The core modules of ConTEXt are loosely organized in groups. Over time there has been some reorganization and in MkIV some code has been moved into new categories. The alphabetical order does not reflect the loading order or dependency tree as categories are loaded intermixed. Therefore the order below is somewhat arbitrary and does not express importance. Each category has multiple files.

5.1 anch: anchoring and positioning

More than a decade ago we started experimenting with position tracking. The ability to store positional information and use that in a second pass permits for instance adding backgrounds. As this code interacts nicely with (runtime) MetaPost it has always been quite powerful and flexible on the one hand, but at the same time it was demanding in terms of runtime and resources. However, were it not for this feature, we would probably not be using TEX at all, as backgrounds and special relative positioning are needed in nearly all our projects.

In MkIV this mechanism had already been ported to a hybrid form, but recently much of the code has been overhauled and its MkII artifacts stripped. As a consequence the overhead in terms of memory probably has increased but the impact on runtime has been considerably reduced. It will probably take some time to become stable if only because the glue to MetaPost has changed. There are some new goodies, like backgrounds behind parshapes, something that probably no one uses and is always somewhat tricky but it was not too hard to support. Also, local background support has been improved which means that it's easier to get them in more column-based layouts, several table mechanisms, floats and such. This was always possible but is now more automatic and hopefully more intuitive.

5.2 attr: attributes

We use attributes (properties of nodes) a lot. The framework for this had been laid early in MkIV development, so not much has changed here. Of course the code gets cleaner and hopefully better as it is putting quite a load on the processing. Each new feature depending on attributes adds some extra overhead even if we make sure that mechanisms only kick in when they are used. This is due to the fact that attributes are linked lists and although unique lists are shared, they travel with each node. On the other hand, the cleanup (and de-MkII-ing) of code leads to better performance so on the average no user will notice this.

5.3 back: backend code generation

This category wraps backend issues in an abstract way that is similar to the special drivers in MkII. So far we have only three backends: PDF, XML, and XHTML. Such code is always in a state of maintenance, if only because backends evolve.

5.4 bibl: bibliographies

For a while now, bibliographies have not been an add-on but part of the core. There are two variants: traditional BIBTEX support derived from a module by Taco Hoekwater but using MkIV features (the module hooks into core code), and a variant that delegates most work to Lua by creating an in-memory XML tree that gets manipulated. At some point I will extend the second variant. Going the XML route also connects better with developments such as Jean-Michel Hufflen's MlBIBTEX.

5.5 blob: typesetting in Lua

Currently we only ship a few helpers but eventually this will become a framework for typesetting raw text in Lua. This might be handy for some projects that we have where the only input is XML, but I'm not that sure if it will produce nice results and if the code will look better. On the other hand, there are some cases where in a regular TEX run some basic typesetting in Lua might make sense. Of course I also need an occasional pet project so this might qualify as one.

5.6 buff: buffers and verbatim

Traditionally buffers and verbatim have always been relatives as they share code. The code was among the first to be adapted to LuaTEX. There is not that much to gain in adapting it further. Maybe I will provide more lexers for pretty-printing some day.

5.7 catc: catcodes

Catcodes are a rather TEX-specific feature and we have organized them in catcode regimes. The most important recent change has been that some of the characters with a special meaning in TEX (like ampersand, underscore, superscript, etc.) are no longer special except in cases that matter. This somewhat incompatible change surprisingly didn't lead to many problems. Some code that is specific for the MkII

XML processor has been removed as we no longer assume it is being used in MkIV.

5.8 char: characters

This important category deals with characters and their properties. Already from the beginning of MkIV character properties have been (re)organized in Lua tables and therefore much code deals with it. The code is rather stable but occasionally the tables are updated as they depend on developments in Unicode. In order to share as much data as possible and prevent duplicates there are several inheritance mechanisms in place but their overhead is negligible.

5.9 chem: chemistry

The external module that deals with typesetting chemistry was transformed into a MkIV core module some time ago. Not much has changed in this department but some enhancements are pending.

5.10 cldf: ConTEXt Lua documents

These modules are mostly Lua code and are the interface into ConTEXt as well as providing ways to code complete documents in Lua. This is one of those categories that is visited every now and then to be adapted to improvements in other core code or in LuaTEX. This is one of my favourite categories as it exposes most of ConTEXt at the Lua end which permits writing solutions in Lua while still using the full power of ConTEXt. A dedicated manual is on its way.

5.11 colo: colors and transparencies

This is rather old code, and apart from some cleanup not much has been changed here. Some macros that were seldom used have been removed. One issue that is still pending is a better interface to MetaPost as it has different color models and we have adapted code at that end. This has a rather low priority because in practice it is no real problem.

5.12 cont: runtime code

These modules contain code that is loaded at runtime, such as filename remapping, patches, etc. It does not make much sense to improve these.

5.13 core: all kinds of core code

Housekeeping is the main target of these modules. There are still some typesetting-related components here but these will move to other categories. This code is cleaned up when there is a need for it. Think of managing files, document project structure, module loading, environments, multipass data, etc.

5.14 data: file and data management

This category hosts only Lua code and hasn't been touched for a while. Here we deal with locating files, caching, accessing remote data, resources, environments, and the like.

5.15 enco: encodings

Because (font) encodings are gone, there is only one file in this category and that one deals with weird (composed or otherwise special) symbols. It also provides a few traditional TEX macros that users expect to be present, for instance to put accents over characters.

5.16 file: files

There is some overlap between this category and core modules. Loading files is always somewhat special in TEX as there is the TEX directory structure to deal with. Sometimes you want to use files in the so-called tree, but other times you don't. This category provides some management code for (selective) loading of document files, modules and resources. Most of the code works with accompanying Lua code and has not been touched for years, apart from some weeding and low-level renaming. The project structure code has mostly been moved to Lua and this mechanism is now more restrictive in the sense that one cannot misuse products and components in unpredictable ways. This change permits better automatic loading of cross references in related documents.

5.17 font: fonts

Without proper font support a macro package is rather useless. Of course we do support the popular font formats but nowadays that's mostly delegated to Lua code. What remains at the TEX end is code that loads and triggers a combination of fonts efficiently. Of course in the process text and math each need to get the proper amount of attention.

There is no longer shared code between MkII and MkIV. Both already had rather different low-level solutions, but recently with MkIV we went a step further. Of course it made sense to kick out commands that were only used for pdfTEX Type 1 and XETEX OpenType support but more important was the decision to change the way design sizes are supported.

In ConTEXt we have basic font definition and loading code and that hasn't conceptually changed much over the years. In addition to that we have so-called bodyfont environments and these have been made a bit more powerful in recent MkIV. Then there are typefaces, which are abstract combinations of fonts and defining them happens in typescripts.

This layered approach is rather flexible, and was greatly needed when we had all those font encodings (to be used in all kinds of combinations within one document). In MkIV, however, we already had fewer typescripts as font encodings are gone (also for Type 1 fonts). However, there remained a rather large blob of definition code dealing with Latin Modern; large because it comes in design sizes.

As we always fall back on Latin Modern, and because we don't preload fonts, there is some overhead involved in resolving design size related issues and definitions. But, it happens that this is the only font that ships with many files related to different design sizes. In practice no user will change the defaults. So, although the regular font mechanism still provides flexible ways to define font file combinations per bodyfont size, resolving to the right best matching size now happens automatically via a so-called Lua font goodie file which brings down the number of definitions considerably. The consequence is that ConTEXt starts up faster, not only in the case of Latin Modern being used, but also when other designs are in play. The main reason for this is that we don't have to parse those large typescripts anymore, as the presets were always part of the core set of typescripts. At the same time loading a specific predefined set has been automated and optimized. Of course on a run of 30 seconds this is not that noticeable, but it is on a 5 second run or when testing something in the editor that takes less than a second. It also makes a difference in automated workflows; for instance at Pragma we run unattended typesetting flows that need to run as fast as possible. Also, in virtual machines using network shares, the fewer files consulted the better.

Because math support was already based on OpenType, where ConTEXt turns Type 1 fonts into OpenType at runtime, nothing fundamental has changed here, apart from some speedups (at the cost of some extra memory). Where the overhead of math font switching in MkII is definitely a factor, in MkIV it is close to negligible, even if we mix regular, bold, and bidirectional math, which we have done for a while.

The low-level code has been simplified a bit further by making a better distinction between the larger sizes (`a` up to `d`) and smaller sizes (`x` and `xx`). These now operate independently of each other (i.e. one can now have a smaller relative `x` size of a larger one). This goes at the cost of more resources but it is worth the effort.

By splitting up the large basic font module into smaller ones, I hope that it can be maintained more easily although someone familiar with the older code will only recognize bits and pieces. This is partly due to the fact that font code is highly optimized.

5.18 grph: graphic (and widget) inclusion

Graphics inclusion is always work in progress as new formats have to be dealt with or users want additional conversions to be done. This code will be cleaned up later this year. The plug-in mechanisms will be extended (examples of existing plug-ins are automatic converters and barcode generation).

5.19 hand: special font handling

As we treat protrusion and hz as features of a font, there is not much left in this category apart from some fine-tuning. So, not much has happened here and eventually the left-overs in this category might be merged with the font modules.

5.20 java: JavaScript in PDF

This code already has been cleaned up a while ago, when moving to MkIV, but we occasionally need to check and patch due to issues with JavaScript engines in viewers.

5.21 lang: languages and labels

There is not much changed in this department, apart from additional labels. The way inheritance works in languages differs too much from other inheritance code so we keep what we have here. Label definitions have been moved to Lua tables from which labels at the TEX end are defined that can then be overloaded locally. Of course the basic interface has not changed as this is typically code that users will use in styles.

5.22 luat: housekeeping

This is mostly Lua code needed to get the basic components and libraries in place. While the `data` category implements the connection to the outside world, this category runs on top of that and feeds the TEX machinery. For instance conversion of MkVI files happens here. These files are seldom touched but might need an update some time (read: prune obsolete code).

5.23 lpdf: PDF backend

Here we implement all kinds of PDF backend features. Most are abstracted via the backend interface. So, for instance, colors are done with a high level command that goes via the backend interface to the `lpdf` code. In fact, there is more such code than in (for instance) the MkII special drivers, but readability comes at a price. This category is always work in progress as insights evolve and users demand more.

5.24 lxml: XML and lpath

As this category is used by some power users we cannot change too much here, apart from speedups and extensions. It's also the bit of code we use frequently at Pragma, and as we often have to deal with rather crappy XML I expect to move some more helpers into the code. The latest greatest trickery related to proper typesetting can be seen in the documents made by Thomas Schmitz. I wonder if I'd still have fun doing our projects if I hadn't, in an early stage of MkIV, written the XML parser and expression parser used for filtering.

5.25 math: mathematics

Math deserves its own category but compared to MkII there is much less code, thanks to Unicode. Since we support Type 1 as virtual OpenType nothing special is needed there (and eventually there will be proper fonts anyway). When rewriting code I try to stay away from hacks, which is sometimes possible by using Lua but it comes with a slight speed penalty. Much of the Unicode math-related font code is already rather old but occasionally we add new features. For instance, because OpenType has no italic correction we provide an alternative (mostly automated) solution.

On the agenda is more structural math encoding (maybe like openmath) but tagging is already part of the code so we get a reasonable export. Not that someone is waiting for it, but it's there for those who want it. Most math-related character properties are part of the character database which gets extended on demand. Of course we keep MathML up-to-date because we need it in a few projects.

We're not in a hurry here but this is something where Aditya and I have to redo some of the code that provides AMS-like math commands (but as we have them configurable some work is needed to keep compatibility). In the process it's interesting to run into probably never-used code, so we just remove those artifacts.

5.26 meta: metapost interfacing

This and the next category deal with MetaPost. This first category is quite old but already adapted to the new situation. Sometimes we add extra functionality but the last few years the situation has become rather stable with the exception of backgrounds, because these have been overhauled completely.

5.27 mlib: metapost library

Apart from some obscure macros that provide the interface between front- and backend this is mostly Lua code that controls the embedded MetaPost library. So, here we deal with extensions (color, shading, images, text, etc.) as well as runtime management because sometimes two runs are needed to get a graphic right. Some time ago, the MkII-like extension interface was dropped in favor of one more natural to the library and MetaPost 2. As this code is used on a daily basis it is quite well debugged and the performance is pretty good too.

5.28 mult: multi-lingual user interface

Even if most users use the English user interface, we keep the other ones around as they're part of the trademark. Commands, keys, constants, messages and the like are now managed with Lua tables. Also, some of the tricky remapping code has been stripped because the setup definitions files are dealt with. These are XML files that describe the user interface that get typeset and shipped with ConTEXt.

These files are being adapted. First of all the commandhandler code is defined here. As we use a new namespace model now, most of these namespaces are defined in the files where they are used. This is possible because they are more verbose so conflicts are less likely (also, some checking is done to prevent reuse). Originally the namespace prefixes were defined in this category but eventually all that code will be gone. This is a typical example where 15-year-old constraints are no longer an issue and better code can be used.

5.29 node: nodes

This is a somewhat strange category as all typeset material in TEX becomes nodes so this deals with everything. One reason for this category is that new functionality often starts here and is sometimes shared between several mechanisms. So, for the moment we keep this category. Think of special kerning, insert management, low-level referencing (layer between user code and backend code) and all kinds of rule and displacement features. Some of this functionality is described in previously published documents.

5.30 norm: normalize primitives

We used to initialize the primitives here (because LuaTEX starts out blank). But after moving that code this category only has one definition left and that one will go too. In MkII these files are still used (and actually generated by MkIV).

5.31 pack: wrapping content in packages

This is quite an important category as in ConTEXt lots of things get packed. The best example is

`\framed` and this macro has been maximally optimized, which is not that trivial since much can be configured. The code has been adapted to work well with the new commandhandler code and in future versions it might use the commandhandler directly. This is however not that trivial because hooking a setup of a command into `\framed` can conflict with the two commands using keys for different matters.

Layers are also in this category and they probably will be further optimized. Reimplementing reusable objects is on the horizon, but for that we need a more abstract Lua interface, so that will come first. This has a low priority because it all works well. This category also hosts some helpers for the page builder but the builder itself has a separate category.

5.32 page: pages and output routines

Here we have an old category: output routines (trying to make a page), page building, page imposition and shipout, single and multi column handling, very special page construction, line numbering, and of course setting up pages and layouts. All this code is being redone stepwise and stripped of old hacks. This is a cumbersome process as these are core components where side effects are sometimes hard to trace because mechanisms (and user demands) can interfere. Expect some changes for the good here.

5.33 phys: physics

As we have a category for chemistry it made sense to have one for physics and here is where the unit module's code ended up. So, from now on units are integrated into the core. We took the opportunity to rewrite most of it from scratch, providing a bit more control.

5.34 prop: properties

The best-known property in TEX is a font and color is a close second. Both have their own category of files. In MkII additional properties like backend layers and special rendering of text were supported in this category but in MkIV properties as a generic feature are gone and replaced by more specific implementations in the `attr` namespace. We do issue a warning when any of the old methods are used.

5.35 regi: input encodings

We still support input encoding regimes but hardly any TEX code is involved now. Only when users demand more functionality does this code get extended. For instant, recently a user wanted a conversion function for going from UTF-8 to an encoding that another program wanted to see.

5.36 scrn: interactivity and widgets

All modules in this category have been overhauled. On the one hand we lifted some constraints, for instance the delayed initialization of fields no longer makes sense as we have a more dynamic variable resolver now (which is somewhat slower but still acceptable). On the other hand some nice but hard to maintain features have been simplified (not that anyone will notice as they were rather special). The reason for this is that vaguely documented PDF features tend to change over time which does not help portability. Of course there have also been some extensions, and it is actually less hassle (but still no fun) to deal with such messy backend related code in Lua.

5.37 scrp: script-specific tweaks

These are script-specific Lua files that help with getting better results for scripts like CJK. Occasionally I look at them but how they evolve depends on usage. I have some very experimental files that are not in the distribution.

5.38 sort: sorting

As sorting is delegated to Lua there is not much TEX code here. The Lua code occasionally gets improved if only because users have demands. For instance, sorting Korean was an interesting exercise, as was dealing with multiple languages in one index. Because sorting can happen on a combination of Unicode, case, shape, components, etc. the sorting mechanism is one of the more complex subsystems.

5.39 spac: spacing

This important set of modules is responsible for vertical spacing, strut management, justification, grid snapping, and all else that relates to spacing and alignments. Already in an early stage vertical spacing was mostly delegated to Lua so there we're only talking of cleaning up now. Although ... I'm still not satisfied with the vertical spacing solution because it is somewhat demanding and an awkward mix of TEX and Lua which is mostly due to the fact that we cannot evaluate TEX code in Lua.

Horizontal spacing can be quite demanding when it comes down to configuration: think of a table with 1000 cells where each cell has to be set up (justification, tolerance, spacing, protrusion, etc.). Recently a more drastic optimization has been done which permits even more options but at the same time is much more efficient, although not in terms of memory.

Other code, for instance spread-related status information, special spacing characters, interline spacing and linewise typesetting all falls into this category

and there is probably room for improvement there. It's good to mention that in the process of the current cleanup hardly any Lua code gets touched, so that's another effort.

5.40 strc: structure

Big things happened here but mostly at the TEX end as the support code in Lua was already in place. In this category we collect all code that gets or can get numbered, moves around and provides visual structure. So, here we find itemize, descriptions, notes, sectioning, marks, block moves, etc. This means that the code here interacts with nearly all other mechanisms.

Itemization now uses the new inheritance code instead of its own specific mechanism but that is not a fundamental change. More important is that code has been moved around, stripped, and slightly extended. For instance, we had introduced proper `\startitem` and `\stopitem` commands which are somewhat conflicting with `\item` where a next instance ends a previous one. The code is still not nice, partly due to the number of options. The code is a bit more efficient now but functionally the same.

The sectioning code is under reconstruction as is the code that builds lists. The intention is to have a better pluggable model and so far it looks promising. As similar models will be used elsewhere we need to converge to an acceptable compromise. One thing is clear: users no longer need to deal with arguments but variables and no longer with macros but with setups. Of course providing backward compatibility is a bit of a pain here.

The code that deals with descriptions, enumerations and notes was already done in a MkIV way, which means that they run on top of lists as storage and use the generic numbering mechanism. However, they had their own inheritance support code and moving to the generic code was a good reason to look at them again. So, now we have a new hierarchy: constructs, descriptions, enumerations and notations where notations are hooked into the (foot)note mechanisms.

These mechanisms share the rendering code but operate independently (which was the main challenge). I did explore the possibility of combining the code with lists as there are some similarities but the usual rendering is too different as in the interface (think of enumerations with optional local titles, multiple notes that get broken over pages, etc.). However, as they are also stored in lists, users can treat them as such and reuse the information when needed (which for instance is just an alternative way to deal with end notes).

At some point math formula numbering (which runs on top of enumerations) might get its own construct base. Math will be revised when we consider the time to be ripe for it anyway.

The reference mechanism is largely untouched as it was already doing well, but better support has been added for automatic cross-document referencing. For instance it is now easier to process components that make up a product and still get the right numbering and cross referencing in such an instance.

Float numbering, placement and delaying can all differ per output routine (single column, multicolumn, columnset, etc.). Some of the management has moved to Lua but most is just a job for TEX. The better some support mechanisms become, the less code we need here.

Registers will get the same treatment as lists: even more user control than is already possible. Being a simple module this is a relatively easy task, something for a hot summer day. General numbering is already fine as are block moves so they come last. The XML export and PDF tagging is also controlled from this category.

5.41 supp: support code

Support modules are similar to system ones (discussed later) but on a slightly more abstract level. There are not that many left now so these might as well become system modules at some time. The most important one is the one dealing with boxes. The biggest change there is that we use more private registers. I'm still not sure what to do with the visual debugger code. The math-related code might move to the math category.

5.42 symb: symbols

The symbol mechanisms organizes special characters in groups. With Unicode-related fonts becoming more complete we hardly need this mechanism. However, it is still the abstraction used in converters (for instance footnote symbols and interactive elements). The code has been cleaned up a bit but generally stays as is.

5.43 syst: tex system level code

Here you find all kinds of low-level helpers. Most date from early times but have been improved stepwise. We tend to remove obscure helpers (unless someone complains loudly) and add new ones every now and then. Even if we would strip down ConTEXt to a minimum size, these modules would still be there. Of course the bootstrap code is also in this category: think of allocators, predefined constants and such.

5.44 tabl: tables

The oldest table mechanism was a quite seriously patched version of TABLE and finally the decision has been made to strip, replace and clean up that bit. So, we have less code, but more features, such as colored columns and more.

The (in-stream) tabulate code is mostly unchanged but has been optimized (again) as it is often used. The multipass approach stayed but is somewhat more efficient now.

The natural table code was originally meant for XML processing but is quite popular among users. The functionality and code is frozen but benefits from optimizations in other areas. The reason for the freeze is that it is pretty complex multipass code and we don't want to break anything.

As an experiment, a variant of natural tables was made. Natural tables have a powerful inheritance model where rows and cells (first, last, ...) can be set up as a group but that is rather costly in terms of runtime. The new table variant treats each column, row and cell as an instance of `\framed` where cells can be grouped arbitrarily. And, because that is somewhat extreme, these tables are called x-tables. As much of the logic has been implemented in Lua and as these tables use buffers (for storing the main body) one could imagine that there is some penalty involved in going between TEX and Lua several times, as we have a two, three or four pass mechanism. However, this mechanism is surprisingly fast compared to natural tables. The reason for writing it was not only speed, but also the fact that in a project we had tables of 50 pages with lots of spans and such that simply didn't fit into TEX's memory any more, took ages to process, and could also confuse the float splitter.

Line tables ... well, I will look into them when needed. They are nice in a special way, as they can split vertically and horizontally, but they are seldom used. (This table mechanism was written for a project where large quantities of statistical data had to be presented.)

5.45 task: lua tasks

Currently this is mostly a place where we collect all kinds of tasks that are delegated to Lua, often hooked into callbacks. No user sees this code.

5.46 toks: token lists

This category has some helpers that are handy for tracing or manuals but no sane user will ever use them, I expect. However, at some point I will clean up this old MkIV mess. This code might end up in a module outside the core.

5.47 trac: tracing

A lot of tracing is possible in the Lua code, which can be controlled from the TEX end using generic enable and disable commands. At the macro level we do have some tracing but this will be replaced by a similar mechanism. This means that many `\tracewhatevertrue` directives will go away and be replaced. This is of course introducing some incompatibility but normally users don't use this in styles.

5.48 type: typescripts

We already mentioned that typescripts relate to fonts. Traditionally this is a layer on top of font definitions and we keep it this way. In this category there are also the definitions of typefaces: combinations of fonts. As we split the larger into smaller ones, there are many more files now. This has the added benefit that we use less memory as typescripts are loaded only once and stored permanently.

5.49 typo: typesetting and typography

This category is rather large in MkIV as we move all code into here that somehow deals with special typesetting. Here we find all kinds of interesting new code that uses Lua solutions (slower but more robust). Much has been discussed in articles as they are nice examples and often these are rather stable.

The most important new kid on the block is margin data, which has been moved into this category. The new mechanism is somewhat more powerful but the code is also quite complex and still experimental. The functionality is roughly the same as in MkII and older MkIV, but there is now more advanced inheritance, a clear separation between placement and rendering, slightly more robust stacking, local anchoring (new). It was a nice challenge but took a bit more time than other reimplementations due to all kinds of possible interference. Also, it's not always easy to simulate TEX grouping in a script language. Even if much more code is involved, it looks like the new implementation is somewhat faster. I expect to clean up this code a couple of times.

On the agenda is not only further cleanup of all modules in this category, but also more advanced control over paragraph building. There is a parbuilder written in Lua on my machine for years already which we use for experiments and in the process a more LuaTEX-ish (and efficient) way of dealing with protrusion has been explored. But for this to become effective, some of the LuaTEX backend code has to be reorganized and Hartmut wants do that first. In fact, we can then backport the new approach to the built-in builder, which is not only faster but also more efficient in terms of memory usage.

5.50 unic: Unicode vectors and helpers

As Unicode support is now native all the MkII code (mostly vectors and converters) is gone. Only a few helpers remain and even these might go away. Consider this category obsolete and replaced by the `char` category.

5.51 util: utility functions

These are Lua files that are rather stable. Think of parsers, format generation, debugging, dimension helpers, etc. Like the data category, this one is loaded quite early.

5.52 Other TEX files

Currently there are the above categories which can be recognized by filename and prefix in macro names. But there are more files involved. For instance, user extensions can go into these categories as well but they need names starting with something like `xxxx-imp-` with `xxxx` being the category.

Then there are modules that can be recognized by their prefix: `m-` (basic module), `t-` (third party module), `x-` (XML-specific module), `u-` (user module), `p-` (private module). Some modules that Wolfgang and Aditya are working on might end up in the core distribution. In a similar fashion some seldom used core code might get moved to (auto-loaded) modules.

There are currently many modules that provide tracing for mechanisms (like font and math) and these need to be normalized into a consistent interface. Often such modules show up when we work on an aspect of ConTEXt or LuaTEX and at that moment integration is not high on the agenda.

5.53 MetaPost files

A rather fundamental change in MetaPost is that it no longer has a format (mem file). Maybe at some point it will read `.gz` files, but all code is loaded at runtime.

For this reason I decided to split the files for MkII and MkIV as having version specific code in a common set no longer makes much sense. This means that already for a while we have `.mpii` and `.mpiv` files with the latter category being more efficient because we delegate some backend-related issues to ConTEXt directly. I might split up the files for MkIV a bit more so that selective loading is easier. This gives a slight performance boost when working over a network connection.

5.54 Lua files

There are some generic helper modules, with names starting with `l-`. Then there are the `mtx-*` scripts for all kinds of management tasks with the most important one being `mtx-context` for managing a TEX run.

5.55 Generic files

This leaves the bunch of generic files that provides OpenType support to packages other than ConTEXt. Much time went into moving ConTEXt-specific code out of the way and providing a better abstract interface. This means that new ConTEXt code (we provide more font magic) will be less likely to interfere and integration is easier. Of course there is a penalty for ConTEXt but it is bearable. And yes, providing generic code takes quite a lot of time so I sometimes wonder why I did it in the first place, but currently the maintenance burden is rather low. Khaled Hosny is responsible for bridging this code to LATEX.

6 What next

Here ends this summary of the current state of ConTEXt. I expect to spend the rest of the year on further cleaning up. I'm close to halfway now. What I really like is that many users upgrade as soon as there is a new beta, and as in a rewrite typos creep in, I therefore often get a fast response.

Of course it helps a lot that Wolfgang Schuster, Aditya Mahajan, and Luigi Scarso know the code so well that patches show up on the list shortly after a problem gets reported. Also, for instance Thomas Schmitz uses the latest betas in academic book production, presentations, lecture notes and more, and so provides invaluable fast feedback. And of course Mojca Miklavec keeps all of it (and us) in sync. Such a drastic cleanup could not be done without their help. So let's end this status report with ... a big thank you to all those (unnamed) patient users and contributors.

◇ Hans Hagen
`http://pragma-ade.com`