MAPS

NUMMER 37 • NAJAAR 2008

NEDERLANDSTALIGE TEX GEBRUIKERSGROEP

Voorzitter
Taco Hoekwater
ntg-president@ntg.nl

Secretaris
Willi Egger
ntg-secretary@ntg.nl

Penningmeester
Wybo Dekker
ntg-treasurer@ntg.nl

Bestuurslid
Karel Wesseling
k.h.wesseling@planet.nl

Postadres
Nederlandstalige TEX
Gebruikersgroep
Maasstraat 2
5836 BB Sambeek

Postgiro
1306238
t.n.v. NTG, Deil
BIC-code: PSTBNL21
IBAN-code: NL05PSTB0001306238

E-mail bestuur
ntg@ntg.nl

E-mail MAPS redactie
maps@ntg.nl

WWW
www.ntg.nl

De **Nederlandstalige TEX Gebruikersgroep (NTG)** is een vereniging die tot doel heeft de kennis en het gebruik van TEX te bevorderen. De NTG fungeert als een forum voor nieuwe ontwikkelingen met betrekking tot computergebaseerde documentopmaak in het algemeen en de ontwikkeling van 'TEX and friends' in het bijzonder. De doelstellingen probeert de NTG te realiseren door onder meer het uitwisselen van informatie, het organiseren van conferenties en symposia met betrekking tot TEX en daarmee verwante programmatuur.
De NTG biedt haar leden ondermeer:

- Tweemaal per jaar een NTG-bijeenkomst.
- Het NTG-tijdschrift MAPS.
- De 'TEX Live'-distributie op DVD/CDROM inclusief de complete CTAN softwarearchieven.
- Verschillende discussielijsten (mailing lists) over TEX-gerelateerde onderwerpen, zowel voor beginners als gevorderden, algemeen en specialistisch.
- De FTP server `ftp.ntg.nl` waarop vele honderden megabytes aan algemeen te gebruiken 'TEX-producten' staan.
- De WWW server `www.ntg.nl` waarop algemene informatie staat over de NTG, bijeenkomsten, publicaties en links naar andere TEX sites.
- Korting op (buitenlandse) TEX-conferenties en -cursussen en op het lidmaatschap van andere TEX-gebruikersgroepen.

Lid worden kan door overmaking van de verschuldigde contributie naar de NTG-giro (zie links); vermeld IBAN- zowel als SWIFT/BIC-code en selecteer shared cost. Daarnaast dient via `www.ntg.nl` een informatieformulier te worden ingevuld. Zonodig kan ook een papieren formulier bij het secretariaat worden opgevraagd.
De contributie bedraagt € 40; voor studenten geldt een tarief van € 20. Dit geeft alle lidmaatschapsvoordelen maar *geen stemrecht*. Een bewijs van inschrijving is vereist. Een gecombineerd NTG/TUG-lidmaatschap levert een korting van 10% op beide contributies op. De prijs in euro's wordt bepaald door de dollarkoers aan het begin van het jaar. De ongekorte TUG-contributie is momenteel $65.

MAPS bijdragen kunt u opsturen naar `maps@ntg.nl`, bij voorkeur in LaTEX- of ConTEXt formaat. Bijdragen op alle niveaus van expertise zijn welkom.

Productie. De Maps wordt gezet met behulp van een LaTEX class file en een ConTEXt module. Het pdf bestand voor de drukker wordt aangemaakt met behulp van pdftex 1.40.9 en luatex 0.30.2 draaiend onder Linux 2.6. De gebruikte fonts zijn Bitstream Charter, schreefloze en niet-proportionele fonts uit de Latin Modern collectie, en de Euler wiskunde fonts, alle vrij beschikbaar.

TEX is een door professor Donald E. Knuth ontwikkelde 'opmaaktaal' voor het letterzetten van documenten, een documentopmaaksysteem. Met TEX is het mogelijk om kwalitatief hoogstaand drukwerk te vervaardigen. Het is eveneens zeer geschikt voor formules in mathematische teksten.
Er is een aantal op TEX gebaseerde producten, waarmee ook de logische structuur van een document beschreven kan worden, met behoud van de letterzetmogelijkheden van TEX. Voorbeelden zijn LaTEX van Leslie Lamport, AMS-TEX van Michael Spivak, en ConTEXt van Hans Hagen.

Inhoudsopgave

Redactioneel

Soms ligt er meer dan genoeg kopij, maar vaker is het kantje boord of er genoeg is om zelfs maar een vloeipapier–dunne Maps mee te vullen. Dat laatste was deze keer het geval.

Als hoofdredakteur moet ik toegeven dat dat grotendeels mijn eigen schuld was. Artikelen komen niet uit zichzelf de redaktie mailbox binnen waaien, en ik had het porren van auteurs de afgelopen tijd wat laten versloffen. Dat er toch een redelijk gevulde Maps voor u ligt mag dan ook wel een klein wondertje heten.

Op 1 oktober, toen de productiefase officieel begon, was er welgeteld één artikel aangeleverd, en het leek er niet naar dat er meer zou komen. Dat artikel was 'CD and DVD Covers in ConTEXt' door *Hans van der Meer*. Een heel leuk artikel, maar met zeven pagina's vul je geen tijdschrift. En dat moest toch echt, want het was de bedoeling om Maps en TEX Live 2008 samen te verzenden.

Op zo'n moment slaat de paniek snel toe, en het is dan logisch om te kijken naar de enige oplossing die zowel snel als veilig is: het herdrukken van artikelen uit andere tijdschriften. Op die manier kwamen we aan twee artikelen die tevens aansloten bij presentaties op de afgelopen TUG bijeenkomst in Cork.

De eerste, 'Putting the Cork back in the bottle' door *Mojca Miklavec & Arthur Reutenauer* gaat over wat er zoal gedaan moet worden worden om de Unicode ondersteuning in programma's zoals XETEX en luaTEX te verbeteren.

Het tweede artikel uit de Cork proceedings is 'TEXworks: lowering the barrier to entry' door *Jonathan Kew*. Ten tijde van de conferentie was TEXworks nog grotendeels vapourware. Het artikel is daarom nogal kort en erg voorzichtig; sindsdien is de ontwikkeling van TEXworks al een heel eind verder.

Met die twee artikelen was die bron van artikelen opgedroogd. Een poging om wat van de sprekers van de afgelopen ConTEXt bijeenkomst in Bohinj te strikken voor een artikel leidde tot niets. De redder in de nood was natuurlijk *Hans Hagen* (die altijd wel iets heeft liggen dat afdrukbaar is). 'Dealing with xml in ConTEXt MkIV' is een voorlopige versie van een ConTEXt manual, maar interessant en kort genoeg voor plaatsing in de Maps.

Maar uiteindelijk was het de NTG najaarsbijeenkomst op 18 oktober jongstleden die de redding bracht: elke lezing op de bijeenkomst was goed voor een minstens één artikel. Ikzelf presenteerde en demonstreerde TEXworks, daarbij hoort uiteraard het artikel van Jonathan Kew dat ik hierboven al meldde.

Verder is daar een artikel dat aansluit bij *Hans Hagen*'s mini-workshop over het gebruik van Lua in TEX documenten: 'The TEX-Lua mix', een artikel vol met praktische voorbeelden.

Van de hand van *Willi Egger* zijn er zelfs twee artikelen: 'Printing labels with ConTEXt' en 'Printing envelopes with ConTEXt.'

Mijn tweede praatje was wel een 'eigen' productie, goed voor 'PDF genereren voor e-readers.'

Siep's presentatie is dan de enige die nog ontbreekt. En hoewel Siep geen tijd had om een origineel artikel te schrijven, wees ze mij wel op *Norbert Preining*'s 'TEX Live 2008 and the TEX Live Manager.'

Daarmee schoot het al aardig op met deze Maps, maar het werd nog wat meer. Ondertussen bleek *Hans Hagen*'s presentatie op de conferentie in Cork namelijk te scherp op de snede te zijn voor het productieproces van de TUGboat. Karl Berry vroeg ons of dat artikel in de Maps zou verschijnen, en hoewel we dat oorspronkelijk niet van plan waren, was dat een uitstekende hint.

Het leverde zelfs nog twee extra artikelen op. Het hoofd–artikel dat overeenkomt met de presentatie op de TUG conferentie is 'How to Convince Don and Hermann to use LuaTEX.' Maar de vormgeving van die presentatie was zelf ook interessant genoeg voor een eigen, kleurrijk, artikeltje: 'The Punk Module.' En over de conversie van het font van Metafont naar MetaPost bleek ook wel wat te vertellen: 'Punk from Metafont to MetaPost.'

En daarmee zijn er nog maar drie pagina's over die ik nog niet genoemd heb, alle drie door ondergetekende en alledrie aangemaakt op het allerlaatste moment. Deze 'Redactioneel' natuurlijk, en dan nog de aankondigingen voor de TUG en EuroTEX conferenties van volgend jaar.

EuroTEX volgend jaar wordt georganiseerd door de NTG, dus ik hoop velen van jullie te zien in Den Haag.

Veel leesplezier toegewenst,

Taco Hoekwater

TUG 2009

TUG 2009 will be held July 28-31, 2009, at the University of Notre Dame, in Notre Dame, Indiana, USA. The nearest airport is South Bend, Indiana (SBN). The location is about 90 miles northeast of Chicago.

The principal local organizer is Martha Kummerer, of the Notre Dame Journal of Formal Logic, which is sponsoring the conference. Email contact: `tug2009@tug.org`.

The TEX–Lua mix

Abstract
An introduction to the combination of TEX and the scripting language Lua.

Introduction

The idea of embedding Lua into TEX originates in some experiments with Lua embedded in the SciTE editor. You can add functionality to this editor by loading Lua scripts. This is accomplished by a library that gives access to the internals of the editing component.

The first integration of Lua in pdfTEX was relatively simple: from TEX one could call out to Lua and from Lua one could print to TEX. My first application was converting math written in a calculator syntax to TEX. Subsequent experiments dealt with MetaPost. At this point integration meant as little as: having some scripting language as an addition to the macro language. But, even in this early stage further possibilities were explored, for instance in manipulating the final output (i.e. the pdf code). The first versions of what by then was already called LuaTEX provided access to some internals, like counter and dimension registers and the dimensions of boxes.

Boosted by the Oriental TEX project, the team started exploring more fundamental possibilities: hooks in the input/output, tokenization, fonts and node lists. This was followed by opening up hyphenation, breaking lines into paragraphs and building ligatures. At that point we not only had access to some internals but also could influence the way TEX operates.

After that, an excursion was made to mplib, which fulfilled a long standing wish for a more natural integration of MetaPost into TEX. At that point we ended up with mixtures of TEX, Lua and MetaPost code.

As of mid-2008 we still need to open up more of TEX, like page building, math, alignments and the backend. Eventually LuaTEX will be nicely split up in components, rewritten in c, and we may even end up with Lua gluing together the components that make up the TEX engine. At that point the interoperation between TEX and Lua may be even richer than it is now.

In the next sections I will discuss some of the ideas behind LuaTEX and the relationship between Lua and TEX and how it presents itself to users. I will not discuss the interface itself, which consists of quite a number of functions (organized in pseudo-libraries) and the mechanisms used to access and replace internals (we call them callbacks).

TEX vs. Lua

TEX is a macro language. Everything boils down to either allowing stepwise expansion or explicitly preventing it. There are no real control features, like loops; tail recursion is a key concept. There are only a few accessible data structures, such as numbers, dimensions, glue, token lists and boxes. What happens inside TEX is controlled by variables, mostly hidden from view, and optimized within the constraints of 30 years ago.

The original idea behind TEX was that an author would write a specific collection of macros for each publication, but increasing popularity among non-programmers quickly resulted in distributed collections of macros, called macro packages. They started small but grew and grew and by now have become pretty large. In these packages there are macros dealing with fonts, structure, page layout, graphic inclusion, etc. There is also code dealing with user interfaces, process control, conversion and much of that code looks out of place: the lack of control features and string manipulation is solved by mimicking other languages, the unavailability of a float datatype is compensated by misusing dimension registers, and you can find provisions to force or inhibit expansion all over the place.

TEX is a powerful typographical programming language but lacks some of the handy features of scripting languages. Handy in the sense that you will need them when you want to go beyond the original purpose of the system. Lua is a powerful scripting language, but knows nothing of typesetting. To some extent it resembles the language that TEX was written in: Pascal. And, since Lua is meant for embedding and extending existing systems, it makes sense to bring Lua into TEX. How do they compare? Let's give some examples.

About the simplest example of using Lua in TEX is the following:

```
\directlua { tex.print(math.sqrt(10)) }
```

This kind of application is probably what most users will want and use, if they use Lua at all. However, we can go further than that.

Loops

In TEX a loop can be implemented as in the plain format (editorial line breaks, but with original comment):

```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body\let\next\iterate
  \else\let\next\relax\fi\next}
\let\repeat=\fi % this makes \loop..\if..\repeat
                % skippable
```

This is then used as:

```
\newcount \mycounter \mycounter=1
\loop
    ...
    \advance\mycounter 1
    \ifnum\mycounter < 11
\repeat
```

The definition shows a bit how TEX programming works. Of course such definitions can be wrapped in macros, like:

```
\forloop{1}{10}{1}{some action}
```

and this is what often happens in more complex macro packages. In order to use such control loops without side effects, the macro writer needs to take measures to permit, for instance, nested usage and avoid clashes between local variables (counters or macros) and user-defined ones. Above we used a counter in the condition, but in practice expressions will be more complex and this is not that trivial to implement.

The original definition of the iterator can be written a bit more efficiently:

```
\def\iterate
  {\body \expandafter\iterate \fi}
```

And indeed, in macro packages you will find many such expansion control primitives being used, which does not make reading macros easier.

Now, get me right, this does not make TEX less powerful, it's just that the language is focused on typesetting and not on general purpose programming, and in principle users can do without that: documents can be preprocessed using another language, and document specific styles can be used.

We have to keep in mind that TEX was written in a time when resources in terms of memory and cpu cycles were far less abundant than they are now. The 255 registers per class and (about) 3000 hash slots in original TEX were more than enough for typesetting a book, but in huge collections of macros they are not all that much. For that reason many macro packages use obscure names to hide their private registers from users and instead of allocating new ones with meaningful names, existing ones are shared. It is therefore not completely fair to compare TEX code with Lua code: in Lua we have plenty of memory and the only limitations are those imposed by modern computers.

In Lua, a loop looks like this:

```
for i=1,10 do
    ...
end
```

But while in the TEX example, the content directly ends up in the input stream, in Lua we need to do that explicitly, so in fact we will have:

```
for i=1,10 do
    tex.print("...")
end
```

And, in order to execute this code snippet, in LuaTEX we will do:

```
\directlua 0 {
    for i=1,10 do
        tex.print("...")
    end
}
```

So, eventually we will end up with more code than just Lua code, but still the loop itself looks quite readable and more complex loops are possible:

```
\directlua 0 {
    local t, n = { }, 0
    while true do
        local r = math.random(1,10)
        if not t[r] then
            t[r], n = true, n+1
            tex.print(r)
            if n == 10 then break end
        end
    end
}
```

This will typeset the numbers 1 to 10 in randomized order. Implementing a random number generator in pure TEX takes a fair amount of code and keeping track of already defined numbers in macros can be done with macros, but neither of these is very efficient.

Basic typesetting

I already stressed that TEX is a typographical programming language and as such some things in TEX are easier than in Lua, given some access to internals:

```
\setbox0=\hbox{x}\the\wd0
```

In Lua we can do this as follows:

```
\directlua 0 {
    local n = node.new('glyph')
    n.font = font.current()
    n.char = string.byte('x')
    tex.box[0] = node.hpack(n)
    tex.print(tex.wd[0]/65536 .. "pt")
}
```

One pitfall here is that TEX rounds off numbers differently than Lua. Both implementations can be wrapped in a macro resp. function:

```
\def\measured#1%
   {\setbox0=\hbox{#1}\the\wd0\relax}
```

Now we get:

```
\measured{x}
```

The same macro using Lua looks as follows:

```
\directlua 0 {
    function measure(chr)
        local n = node.new('glyph')
        n.font = font.current()
        n.char = string.byte(chr)
        tex.box[0] = node.hpack(n)
        tex.print(tex.wd[0]/65536 .. "pt")
    end
}
\def\measured#1{\directlua0{measure("#1")}}
```

In both cases, special tricks are needed if you want to pass for instance a # character to the TEX implementation, or a " to Lua; namely, using \# in the first case, and Lua's ``long strings" marked with double square brackets in the second.

This example is somewhat misleading. Imagine that we want to pass more than one character. The TEX variant is already suited for that, but the Lua function will now look like:

```
\directlua 0 {
    function measure(str)
        if str == "" then
            tex.print("0pt")
        else
            local head, tail = nil, nil
            for chr in str:gmatch(".") do
                local n = node.new('glyph')
                n.font = font.current()
                n.char = string.byte(chr)
                if not head then
                    head = n
                else
                    tail.next = n
                end
                tail = n
            end
            tex.box[0] = node.hpack(head)
            tex.print(tex.wd[0]/65536 .. "pt")
        end
    end
}
```

And still it's not okay, since TEX inserts kerns between characters (depending on the font) and glue between words, and doing all of this in Lua takes more code. So, it will be clear that although we will use Lua to implement advanced features, TEX itself still has quite a lot of work to do.

Typesetting stylistic variations

In the following examples we show code, but it is not of production quality. It just demonstrates a new way of dealing with text in TEX.

Occasionally a design demands that at some place the first character of each word should be uppercase, or that the first word of a paragraph should be in small caps, or that each first line of a paragraph has to be in dark blue. When using traditional TEX the user then has to fall back on parsing the data stream, and preferably you should then start such a sentence with a command that can pick up the text. For accentless languages like English this is quite doable but as soon as commands (for instance dealing with accents) enter the stream this process becomes quite hairy.

The next code shows how ConTEXt MkII defines the \Word and \Words macros that capitalize the first characters of a word or words. The spaces are really important here because they signal the end of a word.

```
\def\doWord#1%
  {\bgroup\the\everyuppercase\uppercase{#1}%
   \egroup}

\def\Word#1%
  {\doWord#1}
```

```
\def\doprocesswords#1 #2\od
  {\doifsomething{#1}{\processword{#1} % space!
  \doprocesswords#2 \od}}

\def\processwords#1%
  {\doprocesswords#1 \od\unskip}

\let\processword\relax

\def\Words
  {\let\processword\Word \processwords}
```

The code here is not that complex. We split off each word and feed it to a macro that picks up the first token (hopefully a character) which is then fed into the `\uppercase` primitive. This assumes that for each character a corresponding uppercase variant is defined using the `\uccode` primitive. Exceptions can be dealt with by assigning relevant code to the token register `\everyuppercase`. However, such macros are far from robust. What happens if the text is generated and not input as is? What happens with commands in the stream that do something with the following tokens?

A Lua-based solution could look as follows:

```
\def\Words#1{\directlua 0
for s in unicode.utf8.gmatch("#1", "([^ ])") do
   tex.sprint(string.upper(
                       s:sub(1,1)) .. s:sub(2))
 end
}
```

But there is no real advantage here, apart from the fact that less code is needed. We still operate on the input and therefore we need to look to a different kind of solution: operating on the node list.

```
function CapitalizeWords(head)
  local done = false
  local glyph = node.id("glyph")
  for start in node.traverse_id(glyph,head) do
    local prev, next = start.prev, start.next
    if prev and prev.id == kern
       and prev.subtype == 0 then
      prev = prev.prev
    end
    if next and next.id == kern
       and next.subtype == 0 then
      next = next.next
    end
    if (not prev or prev.id ~= glyph)
       and next and next.id == glyph then
      done = upper(start)
    end
  end
  return head, done
end
```

A node list is a forward-linked list. With a helper function in the `node` library we can loop over such lists. Instead of traversing we can use a regular while loop, but it is probably less efficient in this case. But how to apply this function to the relevant part of the input? In LuaTEX there are several callbacks that operate on the horizontal lists and we can use one of them to plug in this function. However, in that case the function is applied to probably more text than we want.

The solution for this is to assign attributes to the range of text which a function is intended to take care of. These attributes (there can be many) travel with the nodes. This is also a reason why such code normally is not written by end users, but by macro package writers: they need to provide the frameworks where you can plug in code. In ConTEXt we have several such mechanisms and therefore in MkIV this function looks (slightly simplified) as follows:

```
function cases.process(namespace,attribute,head)
  local done, actions = false, cases.actions
  for start in node.traverse_id(glyph,head) do
    local attr = has_attribute(start,attribute)
    if attr and attr > 0 then
      unset_attribute(start,attribute)
      local action = actions[attr]
      if action then
        local _, ok = action(start)
        done = done and ok
      end
    end
  end
  return head, done
end
```

Here we check attributes (these are set on the TEX side) and we have all kind of actions that can be applied, depending on the value of the attribute. Here the function that does the actual uppercasing is defined somewhere else. The `cases` table provides us a namespace; such namespaces need to be coordinated by macro package writers.

This approach means that the macro code looks completely different; in pseudo code:

```
\def\Words#1{{<setattribute><cases>
              <somevalue>#1}}
```

Or alternatively:

```
\def\StartWords {\begingroup
      <setattribute><cases><somevalue>}
```

```
\def\StopWords {\endgroup}
```

Because starting a paragraph with a group can have unwanted side effects (such as `\everypar` being expanded inside a group) a variant is:

```
\def\StartWords
    {<setattribute><cases><somevalue>}
\def\StopWords {<resetattribute><cases>}
```

So, what happens here is that the user sets an attribute using some high level command, and at some point during the transformation of the input into node lists, some action takes place. At that point commands, expansion and the like can no longer interfere.

In addition to some infrastructure, macro packages need to carry some knowledge, just as with the `\uccode` used in `\uppercase`. The `upper` function in the first example looks as follows:

```
local function upper(start)
  local data, char = characters.data, start.char
  if data[char] then
    local uc = data[char].uccode
    if uc and
       fonts.tfm.id[start.font].characters[uc]
    then
      start.char = uc
      return true
    end
  end
  return false
end
```

Such code is really macro package dependent: LuaTEX provides only the means, not the solutions. In ConTEXt we have collected information about characters in a `data` table in the `characters` namespace. There we have stored the uppercase codes (`uccode`). The `fonts` table, again ConTEXt specific, keeps track of all defined fonts and before we change the case, we make sure that this character is present in the font. Here `id` is the number by which LuaTEX keeps track of the used fonts. Each glyph node carries such a reference.

In this example, eventually we end up with more code than in TEX, but the solution is much more robust. Just imagine what would happen when in the TEX solution we would have:

```
\Words{\framed[offset=3pt]{hello world}}
```

It simply does not work. On the other hand, the Lua code never sees TEX commands, it only sees the two words represented by glyph nodes and separated by glue.

Of course, there is a danger when we start opening TEX's core features. Currently macro packages know what to expect, they know what TEX can and cannot do, and macro writers have exploited every corner of TEX, even the darkest ones. while the dirty tricks in The TEX-book had an educational purpose, those of users sometimes have obscene traits. If we just stick to the trickery introduced for parsing input, converting this into that, doing some calculations, and the like, it will be clear that Lua is more than welcome. It may hurt to throw away thousands of lines of impressive code and replace it by a few lines of Lua but that's the price the user pays for abusing TEX. Eventually ConTEXt MkIV will be a decent mix of Lua and TEX code, and hopefully the solutions programmed in those languages are as clean as possible.

Of course we can discuss until eternity whether Lua is the best choice. Taco, Hartmut and I are pretty confident that it is, and in the couple of years that we have been working on LuaTEX nothing has proved us wrong yet. One can fantasize about concepts, only to find out that they are impossible to implement or hard to agree on; we just go ahead using trial and error. We can talk over and over how opening up should be done, which is what the team does in a nicely closed and efficient loop, but at some points decisions have to be made. Nothing is perfect, neither is LuaTEX, but most users won't notice it as long as it extends TEX's life and makes usage more convenient.

Groups

Users of TEX and MetaPost will have noticed that both languages have their own grouping (scope) model. In TEX grouping is focused on content: by grouping the macro writer (or author) can limit the scope to a specific part of the text or have certain macros live within their own world.

```
.1. \bgroup .2. \egroup .1.
```

Everything done at 2 is local unless explicitly told otherwise. This means that users can write (and share) macros with a small chance of clashes. In MetaPost grouping is available too, but variables explicitly need to be saved.

```
.1. begingroup; save p; path p; .2. endgroup .1.
```

After using MetaPost for a while this feels quite natural because an enforced local scope demands multiple return values which is not part of the macro language. Actually, this is another fundamental difference between the languages: MetaPost has (a kind of) functions, which TEX lacks. In MetaPost you can write

```
draw origin
    for i=1 upto 10: ..(i,sin(i)) endfor;
```

but also:

```
draw some(0) for i=1 upto 10: ..some(i) endfor;
```

with

```
vardef some (expr i) =
    if i > 4 : i = i - 4 fi ;
    (i,sin(i))
enddef ;
```

The condition and assignment in no way interfere with the loop where this function is called, as long as some value is returned (a pair in this case).

In TEX things work differently. Take this:

```
\count0=1
\message{\advance\count0 by 1 \the\count0}
\the\count0
```

The terminal will show:

```
\advance \count 0 by 1 1
```

At the end the counter still has the value 1. There are quite a few situations like this, for instance when data such as a table of contents has to be written to a file. You cannot write macros where such calculations are done, hidden away, and only the result is seen.

The nice thing about the way Lua is presented to the user is that it permits the following:

```
\count0=1
\message{\directlua0{%
  tex.count[0] = tex.count[0] + 1}%
  \the\count0}
\the\count0
```

This will report 2 to the terminal and typeset a 2 in the document. Of course this does not solve everything, but it is a step forward. Also, compared to TEX and MetaPost, grouping is done differently: there is a `local` prefix that makes variables (and functions are variables too) local in modules, functions, conditions, loops, etc. The Lua code in this article contains such locals.

An example: XML

In practice most users will use a macro package and so, if a user sees TEX, he or she sees a user interface, not the code behind it. As such, they will also not encounter the code written in Lua that handles, for instance, fonts or node list manipulations. If a user sees Lua, it will most probably be in processing actual data. Therefore, in this section I will give an example of two ways to deal with xml: one more suitable for traditional TEX, and one inspired by Lua. It demonstrates how the availability of Lua can result in different solutions for the same problem.

MkII: stream-based processing

In ConTEXt MkII, the version that deals with pdfTEX and XETEX, we use a stream-based xml parser, written in TEX. Each < and & triggers a macro that then parses the tag and/or entity. This method is quite efficient in terms of memory but the associated code is not simple because it has to deal with attributes, namespaces and nesting.

The user interface is not that complex, but involves quite a few commands. Take for instance the following xml snippet:

```
<document>
    <section>
        <title>Whatever</title>
        <p>some text</p>
        <p>some more</p>
    </section>
</document>
```

When using ConTEXt commands, we can imagine the following definitions:

```
\defineXMLenvironment[document]
          {\starttext} {\stoptext}
\defineXMLargument   [title]
          {\section}
\defineXMLenvironment[p]
          {\ignorespaces}{\par}
```

When attributes have to be dealt with, for instance a reference to this section, things quickly start looking more complex. Also, users need to know what definitions to use in situations like this:

```
<table>
  <tr><td>first</td> ... <td>last</td></tr>
  <tr><td>left</td>  ... <td>right</td></tr>
</table>
```

Here we cannot be sure that a cell does not contain a nested table, which is why we need to define the mapping as follows:

```
\defineXMLnested[table]{\bTABLE} {\eTABLE}
\defineXMLnested[tr]   {\bTR}    {\eTR}
```

```
\defineXMLnested[td]    {\bTD}    {\eTD}
```

The \defineXMLnested macro is rather messy because it has to collect snippets and keep track of the nesting level, but users don't see that code, they just need to know when to use what macro. Once it works, it keeps working.

Unfortunately mappings from source to style are never that simple in real life. We usually need to collect, filter and relocate data. Of course this can be done before feeding the source to TEX, but MkII provides a few mechanisms for that too. For instance, to reverse the order you can do this:

```
<article>
    <title>Whatever</title>
    <author>Someone</author>
    <p>some text</p>
</article>

\defineXMLenvironment[article]
    {\defineXMLsave[author]}
    {\blank author: \XMLflush{author}}
```

This will save the content of the author element and flush it when the end tag article is seen. So, given previous definitions, we will get the title, some text and then the author. You may argue that instead we should use for instance xslt but even then a mapping is needed from the xml to TEX, and it's a matter of taste where the burden is put.

Because ConTEXt also wants to support standards like MathML, there are some more mechanisms but these are hidden from the user. And although these do a good job in most cases, the code associated with the solutions has never been satisfying.

Supporting xml this way is doable, and ConTEXt has used this method for many years in fairly complex situations. However, now that we have Lua available, it is possible to see if some things can be done more simply (or differently).

MkIV: tree-based processing

After some experimenting I decided to write a full blown xml parser in Lua, but contrary to the stream-based approach, this time the whole tree is loaded in memory. Although this uses more memory than a streaming solution, in practice the difference is not significant because often in MkII we also needed to store whole chunks.

Loading xml files in memory is very fast and once it is done we can have access to the elements in a way similar to xpath. We can selectively pipe data to TEX and manipulate content using TEX or Lua. In most cases this is faster than the stream-based method. An interesting fact is that we can do this without linking to existing xml libraries, and as a result we are pretty independent.

So how does this look from the perspective of the user? Say that we have the simple article definition stored in demo.xml.

```
<?xml version ='1.0'?>
<article>
    <title>Whatever</title>
    <author>Someone</author>
    <p>some text</p>
</article>
```

This time we associate so-called setups with the elements. Each element can have its own setup, and we can use expressions to assign them. Here we have just one such setup:

```
\startxmlsetups xml:document
   \xmlsetsetup{main}{article}{xml:article}
\stopxmlsetups
```

When loading the document it will automatically be associated with the tag main. The previous rule associates the setup xml:article with the article element in tree main. We register this setup so that it will be applied to the document after loading:

```
\xmlregistersetup{xml:document}
```

and the document itself is processed with (the empty braces are an optional setup argument):

```
\xmlprocessfile{main}{demo.xml}{}
```

The setup xml:article can look as follows:

```
\startxmlsetups xml:article
    \section{\xmltext{#1}{/title}}
    \xmlall{#1}{!(title|author)}
    \blank author: \xmltext{#1}{/author}
\stopxmlsetups
```

Here #1 refers to the current node in the xml tree, in this case the root element, article. The second argument of \xmltext and \xmlall is a path expression, comparable to xpath: /title means: the title element anchored to the current root (#1), and !(title|author) is the negation of (complement to) title or author. Such expressions can be more complex than the one above, for instance:

```
\xmlfirst{#1}{/one/(alpha|beta)/two/text()}
```

which returns the content of the first element that satisfies one of the paths (nested tree):

```
/one/alpha/two
/one/beta/two
```

There is a whole bunch of commands like `\xmltext` that filter content and pipe it into TEX. These are calling Lua functions. This article is no manual, so we will not discuss them here. However, it is important to realize that we have to associate setups (consider them free formatted macros) with at least one element in order to get started. Also, xml inclusions have to be dealt with before assigning the setups. These are simple one-line commands. You can also assign defaults to elements, which saves some work.

Because we can use Lua to access the tree and manipulate content, we can now implement parts of xml handling in Lua. An example of this is dealing with so-called Cals tables. This is done in approximately 150 lines of Lua code, loaded at runtime in a module. This time the association uses functions instead of setups and those functions will pipe data back to TEX. In the module you will find:

```
\startxmlsetups xml:cals:process
    \xmlsetfunction {\xmldocument} {cals:table}
                    {lxml.cals.table}
\stopxmlsetups

\xmlregistersetup{xml:cals:process}
\xmlregisterns{cals}{cals}
```

These commands tell MkIV that elements with a namespace specification that contains `cals` will be remapped to the internal namespace `cals` and the setup associates a function with this internal namespace.

By now it will be clear that from the perspective of the user Lua is hardly visible. Sure, he or she can deduce that deep down some magic takes place, especially when you run into more complex expressions like this (the `@` denotes an attribute):

```
\xmlsetsetup
  {main}
  {item[@type='mpctext' or @type='mrtext']}
  {questions:multiple:text}
```

Such expressions resemble xpath, but can go much further, just by adding more functions to the library.

```
item[position() > 2 and position() < 5
    and text() == 'ok']
item[position() > 2 and position() < 5
    and text() == upper('ok')]
item[@n=='03' or @n=='08']
item[number(@n)>2 and number(@n)<6]
item[find(text(),'ALSO')]
```

Just to give you an idea, in the module that implements the parser you will find definitions that match the function calls in the above expressions.

```
xml.functions.find   = string.find
xml.functions.upper  = string.upper
xml.functions.number = tonumber
```

So much for the different approaches. It's up to the user what method to use: stream-based MkII, tree-based MkIV, or a mixture.

TEX–Lua in conversation

The main reason for taking xml as an example of mixing TEX and Lua is in that it can be a bit mind-boggling if you start thinking of what happens behind the scenes. Say that we have

```
<?xml version ='1.0'?>
<article>
    <title>Whatever</title>
    <author>Someone</author>
    <p>some <b>bold</b> text</p>
</article>
```

and we use the setup shown before with `article`.

At some point, we are done with defining setups and load the document. The first thing that happens is that the list of manipulations is applied: file inclusions are processed first, setups and functions are assigned next, maybe some elements are deleted or added, etc. When that is done we serialize the tree to TEX, starting with the root element. When piping data to TEX we use the current catcode regime; linebreaks and spaces are honored as usual.

Each element can have a function (command) associated and when this is the case, control is given to that function. In our case the root element has such a command, one that will trigger a setup. And so, instead of piping content to TEX, a function is called that lets TEX expand the macro that deals with this setup.

However, that setup itself calls Lua code that filters the title and feeds it into the `\section` command, next it flushes everything except the title and author, which again involves calling Lua. Last it flushes the author. The nested sequence of events is as follows:

lua: Load the document and apply setups and the like.

lua: Serialize the `article` element, but since there is an associated setup, tell TEX to expand that one instead.
  tex: Execute the setup, first expand the `\section` macro, but its argument is a call to Lua.
    lua: Filter `title` from the subtree under `article`, print the content to TEX and return control to TEX.
  tex: Tell Lua to filter the paragraphs i.e. skip `title` and `author`; since the `b` element has no associated setup (or whatever) it is just serialized.
    lua: Filter the requested elements and return control to TEX.
  tex: Ask Lua to filter `author`.
    lua: Pipe `author`'s content to TEX.
  tex: We're done.
lua: We're done.

This is a very simple case. In my daily work I am dealing with rather extensive and complex educational documents where in one source there is text, math, graphics, all kind of fancy stuff, questions and answers in several categories and of different kinds, to be reshuffled or not, omitted or combined. So there we are talking about many more levels of TEX calling Lua and Lua piping to TEX, etc. To stay in TEX speak: we're dealing with one big ongoing nested expansion (because Lua calls expand), and you can imagine that this somewhat stresses TEX's input stack, but so far I have not encountered any problems.

Final remarks

Here I discuss several possible applications of Lua in TEX. I didn't mention yet that because LuaTEX contains a scripting engine plus some extra libraries, it can also be used purely for that. This means that support programs can now be written in Lua and that we need no longer depend on other scripting engines being present on the system. Consider this a bonus.

Usage in TEX can be categorized in four ways:

1. Users can use Lua for generating data, do all kind of data manipulations, maybe read data from file, etc. The only link with TEX is the print function.
2. Users can use information provided by TEX and use this when making decisions. An example is collecting data in boxes and use Lua to do calculations with the dimensions. Another example is a converter from MetaPost output to pdf literals. No real knowledge of TEX's internals is needed. The MkIV xml functionality discussed before demonstrates this: it's mostly data processing and piping to TEX. Other examples are dealing with buffers, defining character mappings, and handling error messages, verbatim ... the list is long.
3. Users can extend TEX's core functionality. An example is support for OpenType fonts: LuaTEX itself does not support this format directly, but provides ways to feed TEX with the relevant information. Support for OpenType features demands manipulating node lists. Knowledge of internals is a requirement. Advanced spacing and language specific features are made possible by node list manipulations and attributes. The alternative `\Words` macro is an example of this.
4. Users can replace existing TEX functionality. In MkIV there are numerous examples of this, for instance all file io is written in Lua, including reading from zip files and remote locations. Loading and defining fonts is also under Lua control. At some point MkIV will provide dedicated splitters for multicolumn typesetting and probably also better display spacing and display math splitting.

The boundaries between these categories are not set in stone. For instance, support for image inclusion and mplib in ConTEXt MkIV sits between categories 3 and 4. Categories 3 and 4, and probably also 2, are normally the domain of macro package writers and more advanced users who contribute to macro packages. Because a macro package has to provide some stability it is not a good idea to let users mess around with all those internals, due to potential interference. On the other hand, normally users operate on top of a kernel using some kind of api, and history has proved that macro packages are stable enough for this.

Sometime around 2010 the team expects LuaTEX to be feature complete and stable. By that time I can probably provide a more detailed categorization.

Hans Hagen
Pragma ADE
http://pragma-ade.com

Putting the Cork back in the bottle

Improving Unicode support in TEX

Abstract
Until recently, all of the hyphenation patterns available for different languages in TeX were using 8-bit font encodings, and were therefore not directly usable with UTF-8 TeX engines such as XeTeX and LuaTeX. When the former was included in TeX Live in 2007, Jonathan Kew, its author, devised a temporary way to use them with XeTeX as well as the "old" TeX engines. Last spring, we undertook to convert them to UTF-8, and make them usable with both sorts of TeX engines, thus staying backwardly compatible. The process uncovered a lot of idiosyncrasies in the pattern-loading mechanism for different languages, and we had to invent solutions to work around each of them.

Introduction

Hyphenation is one of the most prominent features of TEX, and since it is possible to adapt it to many languages and writing systems, it should come as no surprise that there were so many patterns created so quickly for so many languages in the relatively early days of TEX development. As a result, the files that are available often use old and dirty tricks, in order to be usable with very old versions of TEX. In particular, all of them used either 8-bit encodings or accent macros (`\'e`, `\v\{z\}`, etc.); Unicode did not yet exist when most of these files were written.

This was a problem when XeTEX was included in TEX Live in 2007, since it expects UTF-8 input by default. Jonathan Kew, the XeTEX author, devised a way of using the historical hyphenation patterns with both XeTEX and the older extensions of TEX: for each pattern file ⟨hyph⟩`.tex`, he wrote a file called `xu-`⟨hyph⟩`.tex` that detects if it is run with XeTEX or not; in the latter case, it simply inputs ⟨hyph⟩`.tex` directly, and otherwise, it takes actions to convert all the non-ASCII characters to UTF-8, and then inputs the pattern file.

To sum up, in TEX Live 2007, XeTEX used the original patterns as the basis, and converted them to UTF-8 on the fly.

In the ConTEXt world, on the other hand, the patterns had been converted to UTF-8 for a couple of years, and were converted back to 8-bit encodings by the macro package, depending on the font encoding.

In an attempt to go beyond that and to unify those approaches, we then decided to take over conversions for all the pattern files present in TEX Live at that time (May 2008), for inclusion in the 2008 TEX Live release.

The new architecture

The core idea is that after converting the patterns to UTF-8, the patterns are embedded in a structure that can make them loadable with both sorts of TEX engines, the ones with native UTF-8 support (XeTEX, LuaTEX) as well as the ones that support only 8-bit input.[1]

The strategy for doing so was the following: for each language ⟨lang⟩, the patterns are stored in a file called `hyph-`⟨lang⟩`.tex`. These files contain only the raw patterns, hyphenation exceptions, and comments. They are input by files called `loadhyph-`⟨lang⟩`.tex`. This is where engine detection happens, such as this code for Slovenian:

```
% Test whether we received one or two arguments
\def\testengine#1#2!{\def\secondarg{#2}}
% We are passed Tau (as in Taco or TEX,
% Tau-Epsilon-Chi), a 2-byte UTF-8 character
\testengine T!\relax
% Unicode-aware engines (such as XeTeX or LuaTeX)
% only see a single (2-byte) argument
\ifx\secondarg\empty
\message{UTF-8 Slovenian Hyphenation Patterns}
\else
\message{EC Slovenian Hyphenation Patterns}
\input conv-utf8-ec.tex
\fi
\input hyph-sl.tex
```

The only trick is to make TEX look at the Unicode character for the Greek capital Tau, in UTF-8 encoding: it uses two bytes, which are therefore read by 8-bit TEX engines as two different characters; thus the macro `\testengine` sees two arguments. UTF-8 engines, on the other hand, see a single character (Greek capital Tau), thus a single argument before the exclamation mark, and `\secondarg` is `\empty`.

If we're running a UTF-8 TEX engine, there is nothing to do but to input the file with the UTF-8 patterns; but if we're running an 8-bit engine, we have to convert

the UTF-8 byte sequences to a single byte in the appropriate encoding. For Slovenian, as for most European languages written in the Latin alphabet, it happens to be T1. This conversion is taken care of by a file named conv-utf8-ec.tex in our scheme. Let's show how it works with these three characters:[2]

- 'č' (UTF-8 ⟨0xc4, 0x8d⟩, T1 0xa3),
- 'š' (UTF-8 ⟨0xc5, 0xa1⟩, T1 0xb2),
- 'ž' (UTF-8 ⟨0xc5, 0xbe⟩, T1 0xba).

In order to convert the sequence ⟨0xc4, 0x8d⟩ to 0xa3, we make the byte 0xc4 active, and define it to output 0xa3 if its argument is 0x8d.[3] The other sequences work in the same way, and the extracted content of conv-utf8-ec.tex is thus:[4]

```
\catcode"C4=\active
\catcode"C5=\active
%
\def^^c4#1{%
\ifx#1^^8d^^a3\else % U+010D
\fi}
%
\def^^c5#1{%
\ifx#1^^a1^^b2\else % U+0161
\ifx#1^^be^^ba\else % U+017E
\fi\fi}
% ensure all chars above have valid lccode's:
\lccode"A3="A3 % U+010D
\lccode"B2="B2 % U+0161
\lccode"BA="BA % U+017E
```

As the last comment says, we also need to set nonzero \lccodes for the characters appearing in the pattern files, a task formerly carried out in the pattern file itself.

The information for converting from UTF-8 to the different font encodings has been retrieved from the encoding definition files for LaTeX and ConTeXt, and gathered in files called ⟨enc⟩.dat. The converter files are automatically generated with a Ruby script from that data.

The appendix shows table of the encodings we support.

Language tags: BCP 47 / RFC 4646

A word needs to be said about the language tags we used. As a corollary to the completely new naming scheme for the pattern files and the files surrounding them, we wanted to adopt a consistent naming policy for the languages, abandoning the original names completely, because they were problematic in some places. Indeed, they used ad hoc names which had been chosen by very different people over many years, without any attempt to be systematic; this has led to awkward situations; for example, the name ukhyphen.tex for the British English patterns: while "UK" is easily recognized as the abbreviation for "United Kingdom", it could also be the abbreviation for "Ukrainian" language, and unless one knows all the names of the pattern files by heart, it is not possible to determine what language is covered by that file from the name alone.

It was therefore clear that in order to name files that had to do with different *languages*, we had to use language codes, not country codes. But this was not sufficient either, as can be seen from the example of British English, since it's not a different language from English.

Upon investigation, it turned out that the only standard able to distinguish all the patterns we had was the IETF "Best Current Practice" recommendation 47 (BCP 47), which is published as RFC documents; currently, it's RFC 4646.[5] This addresses all the language variants we needed to tag:

- Languages with variants across countries or regions, like English.
- Languages written in different scripts, like Serbian (Latin and Cyrillic).
- Languages with different spelling conventions, like Modern Greek (which underwent a reform known as *monotonic* in 1982), and German (for which a reform is currently happening, started in 1996).

A list of all the languages with their tags can be found in appendix.

Dealing with the special cases

There were so many special cases that one might say that the generic case was the special one!

Pattern files designed for multiple encodings

The first problem we encountered was with patterns that tried to accommodate both the OT1 and the T1 encoding in the same file.

The first language for which this had been done was, historically, German, and the same scheme was subsequently adopted for French, Danish, and Latin. The idea is the following: in each of these languages, there are characters that are encoded at different positions in OT1 and in T1; for German, it is the sharp s 'ß'; for French, it is the character 'œ', etc. In order to deal with that, each pattern that happened to contain one of these characters was duplicated in the file, with intricate macros to ignore them selectively, depending on the font encoding used.

This would have been very awkward to reproduce in our architecture, if at all possible: it would have meant that each word such as, say, "cœur" in French would need to yield two different byte strings in 8-bit mode, for OT1 and T1 (c^^1bur and c^^f7ur, respectively). We therefore decided to put the duplicate patterns in a separate file called spechyph-⟨lang⟩-ot1.tex that is input only in legacy mode, after the main file hyph-⟨lang⟩.tex.

The patterns packaged in this fashion should therefore behave in the same way as the historical files, enabling a few breakpoints with non-ASCII characters in OT1 encoding. We would like to stress, though, that OT1 is definitely not the way to go for these languages. We only supported this behaviour for the sake of compatibility, but we doubt it is very useful: if one uses OT1 for German or French, one would indeed have a few patterns with 'ß' or 'œ', respectively, but many more patterns, with accented characters, would be missed. In order to take full advantage of the hyphenation patterns, one needs to use T1 fonts.

It has to be noted that in addition, we ended up not using the aforementioned approach in the case of German, because we wanted to account for the ongoing work to improve the German patterns; thus, we decided to use the new patterns with the UTF-8 engines, but not with the 8-bit engines, for compatibility reasons. In the latter case, we simply include the original pattern file in T1 directly, with no conversion whatsoever. For the three other languages, though (French, Danish and Latin), we used a spechyph-⟨lang⟩-ot1.tex file.

Multiple pattern sets for the same language

Another interesting issue was with Ukrainian and Russian, where different complications arose.

First, the pattern files were also devised for multiple encodings, but in a different manner: here, the encoding is selected by setting the control sequence \Encoding before the pattern file is loaded. Depending on the value of that macro, the appropriate conversion file is then input, that works in the same way as our conv-utf8-⟨enc⟩.tex files. There is of course a default value for \Encoding, which for both languages is T2A,[6] the most widespread font encoding for Russian and Ukrainian, and the one used in the pattern files; thus, no conversion is necessary if \Encoding is kept to its default value.

Then, both Russian and Ukrainian had several pattern files, with different authors and/or hyphenation rules (phonetic, etymological, etc.). Those were selected with a control sequence called \Pattern, by default as for Russian (by Aleksandr Lebedev), and mp for Ukrainian (by Maksym Polyakov).

Both those choices could, of course, be overridden only at format-building time, since the patterns are frozen at that moment.

Finally, they used a special trick, implemented in file hypht2.tex, to enable hyphenation inside words containing hyphens, similar to Bernd Raichle's hypht1.tex for T1 fonts.

Those three features had to be addressed in very different ways in our structure: while the first one was irrelevant in UTF-8 mode, it would have implied fundamental changes in our loadhyph-⟨lang⟩.tex files for 8-bit engines, since the implicit assumption that any language uses exactly one 8-bit encoding would no longer be met. The second feature was easier to handle, but still demanded additional features in our loadhyph-⟨lang⟩.tex files. Finally, the third feature, although certainly very interesting, seemed more fragile than what we felt was acceptable.

Upon deliberation, we then decided to not include those features in the UTF-8 patterns before TeX Live 2008 was out, but to still enable them in legacy mode, in order to ensure backward compatibility. And thanks to subsequent discussions with Vladimir Volovich, who devised the way the Russian patterns were packaged, and inspired the Ukrainian ones, we could include a list of hyphenated compound words which we put in files called exhyph-ru.tex and exhyph-uk.tex, respectively. The strategy we used is thus:

- In UTF-8 mode, input the UTF-8 patterns, then the ex- file.
- In legacy mode, simply input the original pattern file directly.

Therefore, the only feature missing, overall, in TeX Live 2008, is the ability to choose one's favorite patterns in UTF-8 mode: for each language, we only converted the default set of patterns to UTF-8. Setting \Pattern will thus have no effect in this case, but it will behave as before in 8-bit mode. Now that TeX Live 2008 has been released we intend to change that behaviour soon, and to enable the full range of features that the original pattern files had.

It should also be noted that in TeX Live 2007, Bulgarian used the same pattern-loading mechanism, but that there was actually only one possible encoding, and only one pattern file, so there was no real choice, and it was therefore straightforward to adapt the Bulgarian patterns to our new architecture.

TEX Live 2008

The result of our work has been put on CTAN under the package name hyph-utf8, and is the basis for hyphenation support in TEX Live 2008. We don't consider our work to be finished (see next section), and we welcome any discussion on our mailing-list (tex-hyphen@tug.org). We also have a home page at http://tug.org/tex-hyphen, to which readers are referred for more information.

The package has been released in the TDS layout, with the TEX files in `tex/generic/hyph-utf8` and subdirectories. The encoding data and Ruby scripts are available in source/generic/hyph-utf8. Some language-specific documentation has been put in `doc/generic/hyph-utf8`.

And now ...

There still are tasks we would like to carry out: the hypht1.tex / hypht2.tex behaviour has already been mentioned, and one of the authors has lots of ideas on how to improve Unicode support *yet more* in UTF-8 TEX engines.

We appeal to pattern authors to make contact with us in order to improve and enhance our package; many of them have already communicated with us, to our greatest pleasure, and we're confident that our effort will be understood by all the developers dealing with language-related problems.[7]

Among the immediate and practical problems is, in particular:

... for something completely different

Babel would need to be enhanced in order to enable different "variants" for at least two languages. One is Norwegian, for which two written forms exist, known as "bokmål" and "nynorsk" (ISO 639-1 `nb` and `nn`, respectively).[8] At the moment, Babel has only one "Norwegian" language. The second is Serbian, which can be written in both the Latin and the Cyrillic alphabets; these possible variants which are not yet taken into account in Babel.

Acknowledgements

First and foremost, we wish to thank wholeheartedly Karl Berry, who supported the project from the beginning and guided us with advice, as well as Hans Hagen, Taco Hoekwater and Jonathan Kew, for their technical help, and, finally, Norbert Preining, who went through the trouble of integrating the new package into TEX Live.

Notes

1. A note on vocabulary: in this article, we use the word "engine" or "TEX engine" for extensions to the program TEX, in contrast to macro packages. We then refer to (TEX) engines with native UTF-8 support as "UTF-8 engines", and to the others as "8-bit engines", or sometimes "legacy engines", borrowing from Unicode lingo.
2. The only non-ASCII characters in Slovenian.
3. The same method would work flawlessly if the sequence contained three or more bytes — although this case doesn't arise in our patterns — since the number of bytes in a UTF-8 sequence depends only on the value of the first byte.
4. Problems would happen if a `T1` byte had been made active in that process, but for reasons inherent to the history of TEX font encodings, as well as Unicode, this *is never the case for the characters used in the patterns*, a fact the authors consider a small miracle. The proof of this is much too long to be given in this footnote, and is left to the reader.
5. In the past, it has been RFC 1766, then RFC 3066, and is currently being rewritten, with the working title RFC 4646bis. RFC 4646 is available at ftp://ftp.rfc-editor.org/in-notes/rfc4646.txt, and the current working version of RFC 4646bis (draft 17) at http://www.ietf.org/internet-drafts/draft-ietf-ltru-4646bis-17.txt.
6. Actually `t2a`, lowercase.
7. The acknowledgement section, had it been as long as the authors would have wished it to be, would have more than doubled the size of this article.
8. The ISO standard also includes a code for "Norwegian", `no`, although this name is formally ambiguous.

Mojca Miklavec and Arthur Reutenauer
mojca.miklavec@gmail.com

List of supported languages

ar	Arabic
eu	Basque
bg	Bulgarian
zh-latn	Chinese Pinyin
cop	Coptic
hr	Croatian
cs	Czech
da	Danish
nl	Dutch
en-us	English, American
en-gb	English, British
eo	Esperanto
et	Estonian
fa	Farsi
fi	Finnish
fr	French
de-1996	German, "new" spelling
de-1901	German, "old" spelling
grc	Greek, Ancient
grc-x-ibycus	Greek, Ancient, Ibycus encoding
el-monoton	Greek, Monotonic
el-polyton	Greek, Polytonic
hu	Hungarian
is	Icelandic
id	Indonesian
ia	Interlingua
ga	Irish
it	Italian
la	Latin
mn-cyrl	Mongolian
mn-cyrl-x-2a	Mongolian (new patterns)
no	Norwegian
nb	Norwegian Bokmål
nn	Norwegian Nynorsk
pl	Polish
pt	Portuguese
ro	Romanian
ru	Russian
sr-cyrl	Serbian, Cyrillic script
sr-latn	Serbian, Latin script
sh-cyrl	Serbo-Croatian, Cyrillic script
sh-latn	Serbo-Croatian, Latin script
sl	Slovene
es	Spanish
sv	Swedish
tr	Turkish
uk	Ukrainian
hsb	Upper Sorbian
cy	Welsh

List of supported encodings

ConTeXt	LaTeX	Comments
ec	T1	"Cork" encoding
il2	latin2	ISO 8859-2
il3	latin3	ISO 8859-3
lmc	lmc	montex (Mongolian)
qx	qx	Polish
t2a	t2a	Cyrillic

PDF genereren voor e-readers

Abstract
NotuDoc is een commerciële internet applicatie die ConTEXt gebruikt voor het on-the-fly genereren van pdf documenten, onder andere voor de e-readers van iRex technologies. Dit artikel geeft een blik achter de schermen.

Inleiding

Dit artikel gaat over het genereren van PDF bestanden voor e-readers. Voordat we daarop dieper ingaan, eerst wat uitleg over de applicatie (NotuDoc) waar dit proces een onderdeel van is, en een korte introductie van de gebruikte e-readers.

Het genereren van een PDF document vanuit NotuDoc is slechts een relatief klein onderdeel van de applicatie, maar toch komt daar al wel het een en ander bij kijken.

NotuDoc

Het bedrijf NotuBiz (http://notubiz.nl) houdt zich bezig met alles wat er komt kijken bij het vastleggen en publiceren van (raads)vergaderingen via de moderne media. NotuBiz verzorgt bijvoorbeeld live streaming en ook digitale verslagen kunnen via NotuBiz op het internet beschikbaar gesteld worden. De klantenkring bestaat uit de lokale overheden, uiteraard met name in Nederland.

In de praktijk bleek dat de informatiestroom *voorafgaand* aan de vergaderingen vaak ook aanzienlijk verbeterd kon worden. NotuDoc is uit dit idee ontstaan: het is een internet applicatie die (al dan niet voorlopige) vergaderagenda's koppelt aan de bijbehorende vergaderstukken zoals commissierapporten, presentaties en offertes. Na deze koppeling wordt het resultaat beschikbaar gesteld aan de relevante partijen via het netwerk en/of via PDF bestandsexport.

Door alle benodigde vergaderstukken te combineren op één plaats wordt het makkelijker voor de deelnemers aan de vergadering om zich voor te bereiden. Bovendien staat na afloop van de vergadering nu alles al klaar voor officiële publicatie, wellicht na een koppeling met het verslag van de vergadering.

NotuDoc is een kant-en-klaar commercieel product, en heeft vrij uitgebreide configuratiemogelijkheden. Dat is met name van belang omdat de interface moet aansluiten bij de vormgeving van de website van de klant, maar ook een het gaat nog iets verder: niet alle klanten hebben dezelfde vergader-structuur en zeker niet allen hebben dezelfde opzet voor intern databeheer. Verschillende variaties daarvan worden standaard aangeboden als onderdeel van NotuDoc, ingrijpendere veranderingen (denk aan koppelingen met document management systemen) zijn mogelijk op offertebasis. NotuDoc is geschreven en wordt onderhouden door Elvenkind BV in samenwerking met NotuBiz, en maakt gebruik van Elvenkind's development framework dat geschreven is in perl 5.

E-readers

Op dit moment is NotuDoc voorbereid op het genereren van PDF documenten voor twee verschillende e-readers, beide ontwikkeld door iRex Technologies (http://www.irextechnologies.com), een spin–off van Philips. Naast deze twee apparaten (iLiad en DR1000) is het uiteraard ook mogelijk om PDFs te genereren voor printing of voor interactief gebruik op een computer.

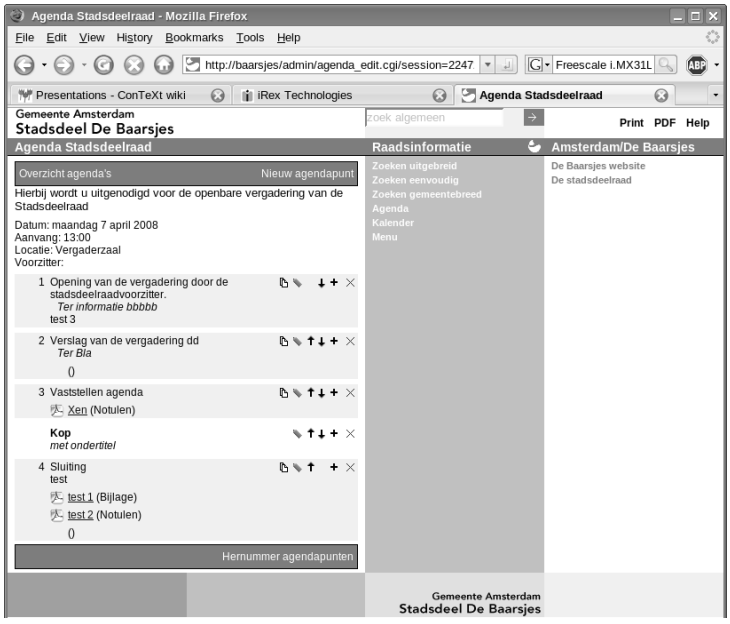


Figure 1. Hoofdscherm van de internet applicatie

Zowel de iLiad als de nieuwere DR1000 zijn gebaseerd op dezelfde basis-technologie. De iLiad bestaat nu al enkele jaren, en wordt onder andere gebruikt voor de digitale versie van het NRC Handelsblad. De DR1000 is een maand geleden gelanceerd. Zoals te verwachten is, is de DR1000 wat groter en sneller dan zijn voorganger, maar verder zijn er op de vormgeving na weinig technologische verschillen.

Beide apparaten zijn gebaseerd op zogenaamd `elektronisch papier', een technologie waarbij het getoonde zichtbaar blijft ook als het scherm *niet* vele malen per seconde ververst wordt.

Een belangrijk voordeel van deze technologie is dat er hierdoor veel minder stroom nodig is, waardoor de levensduur van de batterijen veel langer is dan bij de gewone TFT of LCD schermen. Een bijkomend voordeel is dat het scherm niet constant verlicht hoeft te worden, wat veel rustiger is voor het oog van de lezer.

Anderzijds zijn er natuurlijk nadelen aan elektronisch papier. De twee grootste daarvan zijn dat de reactiesnelheid van het scherm veel lager is dan bij gewone computerschermen en (het meest in het oog springend) dat de huidige versies alleen in staat zijn om grijstonen te tonen, geen kleur.

Beide apparaten gebruiken tevens de Wacom Penabled technologie (http://www.wacom.com/tabletpc/what_is_penabled.cfm) die het mogelijk maakt om rechtstreeks op het scherm aantekeningen en schetsen te maken, zodat je bijvoorbeeld correctie-aantekeningen in een PDF kunt maken. De bijgeleverde (windows) software is in staat zulke aantekeningen te combineren met de originele PDF, bijvoorbeeld voor verzending per email.

Beide apparaten bieden ondersteuning voor PDF, HTML, mobipocket, en enkele bitmap formaten. Alle door iRex gebruikte en ontwikkelde software is open source die werkt op een linux versie die speciaal bedoeld is voor consumentenapparaten. Verbinding maken met de PC voor het uploaden van bestanden gebeurt via USB of via een optioneel tcp/ip (wireless) netwerk. Het opslagmedium is een verwisselbaar SD (DR1000) of CF/MMC (iLiad) kaartje.

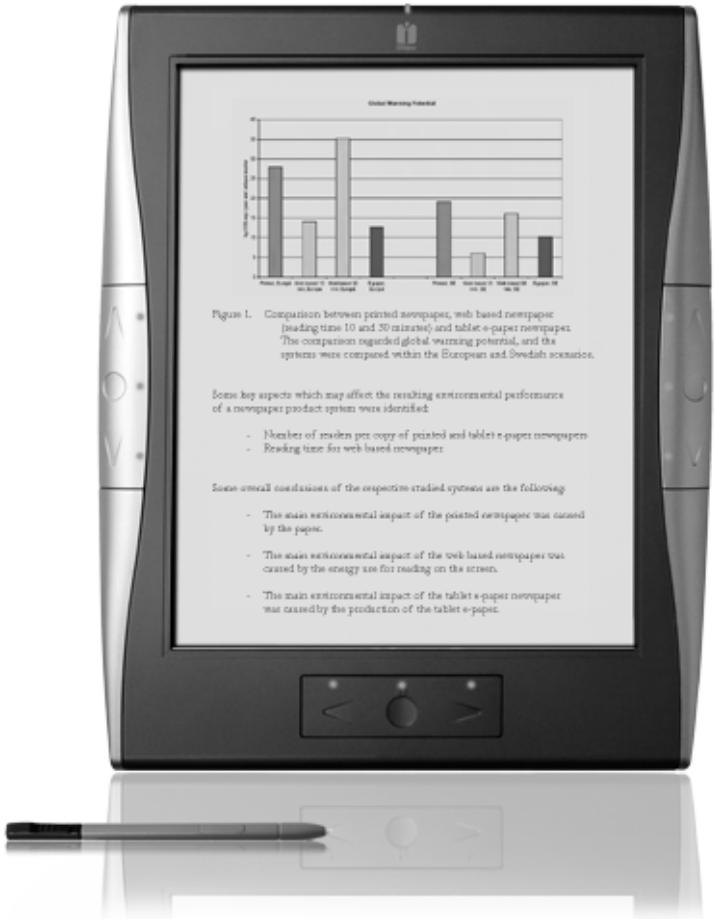


Figure 2. De iLiad (links) en DR1000 (rechts).

PDF generatie

De PDF generatie in NotuDoc wordt gedaan door een perl script dat volledig template-gestuurd is. Het gebruikt vrijwel identieke code voor het genereren van TEX als voor de generatie van de HTML pagina's, alleen de character escape functies en de bestandsnamen zijn specifiek voor TEX aangepast. Net als de website worden de PDF documenten ook runtime gegenereerd via een aanroep van `texexec`. De gebruikte distributie is ConTEXt minimals (http://minimals.contextgarden.net/).

ConTEXt templates

De applicatie heeft per klant een instelling opgeslagen voor de gewenste PDF uitvoer layout. Als voorbeeld neem ik de `iLiad', maar het kan ook iets anders zijn zoals `DR1000' of gewoon `A4'. Los van deze globale voorkeur is het mogelijk om per klant de opmaak te configureren zodat die aansluit bij de huisstijl.

De opmaak instellingen worden gedaan via ConTEXt macros en zijn gescheiden van de afhandeling vanuit de web applicatie. De applicatie draagt alleen zorg voor het omwerken van de database gegevens van de agenda naar een ConTEXt invoerbestand en het exporteren van de bijbehorende vergaderstukken naar PDF bestanden. Al het andere wordt gedaan door de ConTEXt macro files die door het perl script alleen maar gekopieerd worden.

`agenda-iliad.tex`

Dit is het hoofd bestand, en hierin vinden twee verschillende soorten vervangingen plaats.

In de listing hieronder zie je twee regels staan die lijken op de HTML syntax voor zogenaamde `server side includes'. Dat is geen toeval: zoals hierboven al vermeld werd gebruikt het systeem voor de generatie van de TEX bestanden dezelfde code als voor het aanmaken van de HTML pagina's.

De twee `#include` bestandjes worden ingelezen door het perl script en op die plek in de TEX uitvoer tussengevoegd. De exacte inhoud van deze bestanden wordt in de volgende paragraaf uitgelegd.

De tweede soort vervanging gebruikt de trefwoorden in hoofdletters en tussen # tekens. Die trefwoorden worden vervangen door de feitelijke inhoud (en metadata) van de agenda. Het trefwoord `#LIST#` is daarbij het belangrijkst omdat daarin zich effectief de hele inhoud van de agenda bevindt, die wordt namelijk recursief wordt opgebouwd.

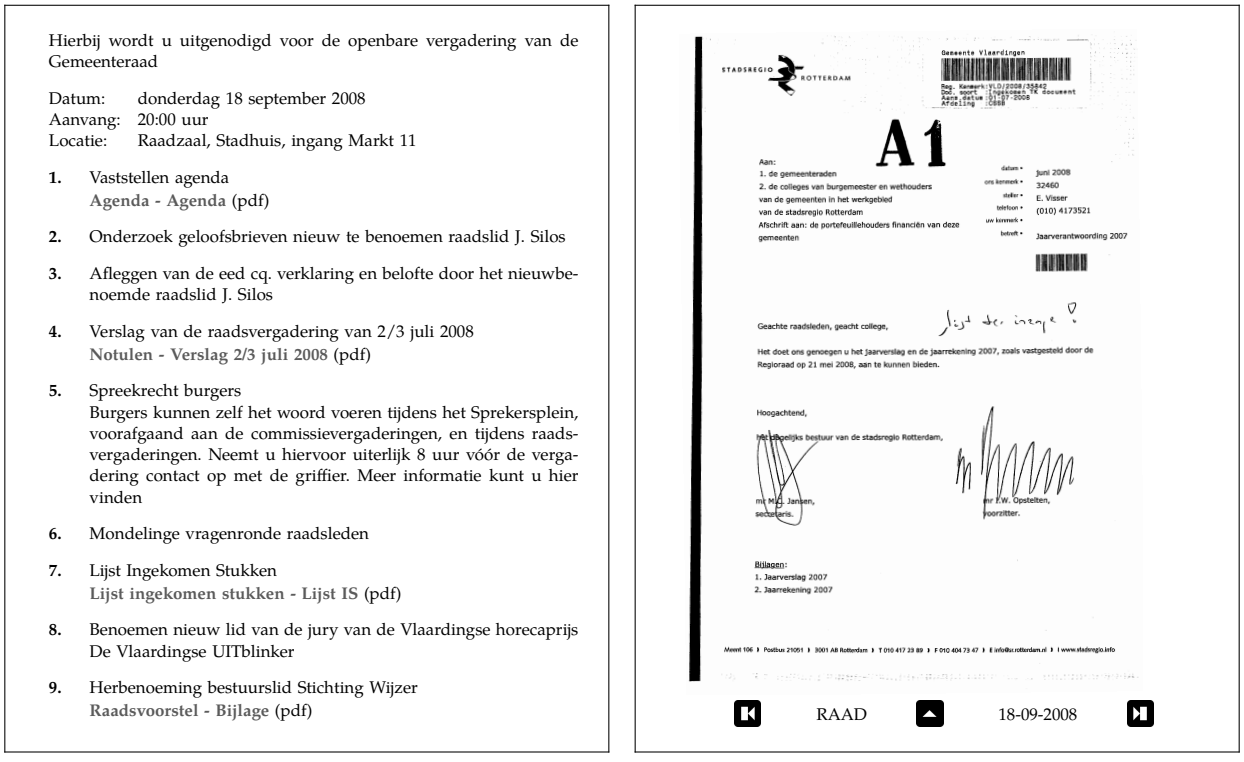
Een agenda bestaat uit meta-informatie zoals plaats en tijd, en een aantal agendapunten. Agendapunten kunnen eventueel gerangschikt zijn in categoriën, en optioneel kan er per agendapunt een aantal agendastukken zijn in verschillende bestandsformaten. Dit alles wordt aangestuurd door kleine template bestandjes die op diverse niveaus worden aangeroepen.

```
\unprotect
<!--#include src='agenda-macros-00.tex' -->
<!--#include src='agenda-macros-iliad.tex' -->
\protect

\starttext
\startagenda[Gremium={#GREMIUM#},
             Datum={#DATUM#},
             Datumkort={#DATUMKORT#},
             Categorie={#CATEGORIE#},
             Aanvang={#AANVANG#},
             Locatie={#LOCATIE#},
             Aanhef={#AANHEF#},
             Koptitel={#TITLE2#},
             Titel={#TITLE#}]

\startpunten
#LIST#
\stoppunten

\stopagenda
\stoptext
```

`header_line.tinc`

Dit template wordt gebruikt voor de tussenkopjes die behoren bij eventuele categoriën van vergaderpunten.

```
\startheaderline
  [Titel={#TEXT#},Pagina=#PAGE#,Aard={#AARD#}]
\startheaderbody
#BODY#
\stopheaderbody
\stopheaderline
```



Hierbij wordt u uitgenodigd voor de openbare vergadering van de Gemeenteraad

Datum: donderdag 18 september 2008
Aanvang: 20:00 uur
Locatie: Raadzaal, Stadhuis, ingang Markt 11

1. Vaststellen agenda
   Agenda - Agenda (pdf)
2. Onderzoek geloofsbrieven nieuw te benoemen raadslid J. Silos
3. Afleggen van de eed cq. verklaring en belofte door het nieuwbenoemde raadslid J. Silos
4. Verslag van de raadsvergadering van 2/3 juli 2008
   Notulen - Verslag 2/3 juli 2008 (pdf)
5. Spreekrecht burgers
   Burgers kunnen zelf het woord voeren tijdens het Sprekersplein, voorafgaand aan de commissievergaderingen, en tijdens raadsvergaderingen. Neemt u hiervoor uiterlijk 8 uur vóór de vergadering contact op met de griffier. Meer informatie kunt u hier vinden
6. Mondelinge vragenronde raadsleden
7. Lijst Ingekomen Stukken
   Lijst ingekomen stukken - Lijst IS (pdf)
8. Benoemen nieuw lid van de jury van de Vlaardingse horecaprijs De Vlaardingse UITblinker
9. Herbenoeming bestuurslid Stichting Wijzer
   Raadsvoorstel - Bijlage (pdf)

STADSREGIO ROTTERDAM

Gemeente Vlaardingen

A1

Aan:
1. de gemeenteraden
2. de colleges van burgemeester en wethouders van de gemeenten in het werkgebied van de stadsregio Rotterdam
Afschrift aan: de portefeuillehouders financiën van deze gemeenten

datum • juni 2008
ons kenmerk • 32460
steller • E. Visser
telefoon • (010) 4173521
uw kenmerk •
betreft • Jaarverantwoording 2007

Geachte raadsleden, geacht college,

Het doet ons genoegen u het jaarverslag en de jaarrekening 2007, zoals vastgesteld door de Regioraad op 21 mei 2008, aan te kunnen bieden.

Hoogachtend,

het dagelijks bestuur van de stadsregio Rotterdam,

mr M.C. Jansen, secretaris.

mr F.W. Opstelten, voorzitter.

Bijlagen:
1. Jaarverslag 2007
2. Jaarrekening 2007

Meent 106 ▸ Postbus 21051 ▸ 3001 AB Rotterdam ▸ T 010 417 33 89 ▸ F 010 404 73 47 ▸ E info@sr.rotterdam.nl ▸ I www.stadsregio.info

RAAD 18-09-2008

Figure 3. De eerste en één van de vervolgpagina's van een voor de iliad gegenereerde PDF

`puntnr_line.tinc`
Dit is het template voor elk van de aparte agendapunten. `#BODY#` bevat de verklarende tekst bij dit punt, `#DOCS#` bevat de lijst van bijbehorende stukken. Deze laatste is zelf weer een programmatisch opgebouwde lijst omdat er meer dan één vergaderstuk per agendapunt kan zijn.

```
\startpunt[Nummer={#NR#},Titel={#PUNT#},Aard={#AARD#}]
\startpuntbody
#BODY#
\stoppuntbody
\startpuntdocs
#DOCS#
\stoppuntdocs
\stoppunt
```

`puntdoc_line.tinc`
Dit is het eerste van drie mogelijke templates voor een vergaderstuk. Deze is voor vergaderstukken (d.w.z. geëxporteerde PDF bestanden) die zullen worden meegenomen als appendices in de te genereren PDF.

```
\agendadocument[#ICON#]{#LINK#}{#LABEL#}
```

`puntnodoc_line.tinc`
Deze template wordt gebruikt voor vergaderstukken die eigenlijk zouden moeten worden meegenomen als appendices (omdat het PDFs zijn), maar die op grond van configuratie parameters te groot zijn bevonden voor daadwerkelijk gebruik. Er wordt daarom een aparte macro gebruikt die een gepaste meldingstekst kan tonen.

```
\agendanodocument[#ICON#]{#LABEL#}
```

`puntdoc_line_noembed.tinc`
Dit is de derde mogelijkheid, deze is bedoeld voor non-PDF vergaderstukken zoals Microsoft Office bestanden en powerpoint presentaties. Omdat bestanden in die formaten niet kunnen worden getoond is er geen hyperlink mogelijk, en daarom is het trefwoord `#LINK#` in dit geval niet aanwezig.

```
\agendadocument[#ICON#]{}{#LABEL#}
```

ConTEXt macros

Zoals hierboven al werd vermeld zijn de gebruikte ConTEXt macros opgesplitst in twee verschillende bestanden.

Het eerste bestand heeft de naam `agenda-macros-00.tex`, en dit bestand wordt ongewijzigd gebruikt door alle klanten en alle PDF layouts. Het bevat een generieke implementatie van de macros die we eerder zagen in de template bestanden. Deze macros zorgen alleen voor de infrastructuur en doen zelf geen vormgeving. Voor de vormgeving zijn er aanroepen van `\directsetup`.

Typerend voor de inhoud van dit bestand zijn macro definities zoals deze:

```
\def\dostartagenda[#1]%
  {\getparameters
      [Agenda]
      [Gremium=,Datum=,Datumkort=,
       Categorie=,Aanvang=,locatie=,
       Aanhef=,Titel=,Koptitel=,
       Voorzitter=,
       #1]%
   \pagereference[firstpage]
   \directsetup{agenda:start}}

\def\stopagenda
  {\directsetup{agenda:stop}}
```

en deze:

```
\def\agendadocument[#1]#2#3%
  {\doifnotempty
     {#2}
     {\doglobal \appendtoks \addimage{#2}{#3}\to \everyendagenda }%
   \def\DocumentType{#1}%
   \def\DocumentFile{#2}%
   \def\DocumentBody{#3}%
   \pagereference[#2-referer]
   \directsetup{agenda:document}}
```

De macro `\addimage` is de interessantste macro in dit bestand. Hij krijgt als argument de naam van een geëxporteerd PDF bestand door, en zorgt ervoor dat zo'n PDF pagina voor pagina wordt ingelezen via `\externalfigure`. In een wat versimpelde vorm ziet die er als volgt uit:

```
\unexpanded\def\addimage#1#2{%
  \pagereference[#1]
  \xdef\previouspdf{\currentpdf}%
  \gdef\currentpdf{#1}%i
  \getfiguredimensions[#1.pdf]%
  \imgcount=\noffigurepages
  \dorecurse
```

```
      {\the\imgcount}
      {\externalfigure
          [#1.pdf]
          [page=\recurselevel,
           factor=max,
           size=cropbox]%
       \page}%
    \pagereference[#1-last]
}
```

In de appendices van de gegenereerde PDF (zie de figuur) is er een extra interactie–regel onderaan de pagina met daarop drie buttons die springen naar de eerste pagina van de huidige appendix, de eerste pagina van de volgende appendix, en naar de referentie naar deze appendix in de agenda zelf. Deze hyperlinks gebruiken de waarden van `\currentpdf` en `\previouspdf`.

De gevraagde `setups` en de algemene layout definities staan in `agenda-macros-iliad.tex`. Dit bestand kan specifiek gedefinieerd zijn voor een bepaalde klant, of er kan een generieke vorm gebruikt worden: er is een default bestand voor één voor elk van de voorgedefinieerde PDF layouts.

De PDF layouts voor de e-reader zijn verschillend van de layouts voor een PC scherm of printer, maar de meeste verschillen zijn voor de hand liggend. Uiteraard is er een eigen (kleiner) papierformaat. De ruimte op een e-reader is schaars, dus die moet zo goed mogelijk gebruikt worden, dus er worden heel kleine marges gedefinieerd. PDF object compressie wordt uitgeschakeld, omdat de hardware van de e-readers erg licht is in vergelijking met een PC. Het Kleuren–subsysteem van ConTEXt wordt aangezet, maar in grijswaarden. Et cetera.

Het meest ingrijpende verschil is dat de vergaderstukken die bij `normaal' gebruik aparte bestanden zouden blijven hier worden ingebed in het hoofdbestand. Dit maakt het overzetten van de bestanden naar de e-reader eenvoudiger, maar belangrijker is dat dit de gebruikersvriendelijkheid van het resultaat verbetert: Externe PDF hyperlinks worden danwel niet ondersteund (iLiad) of zijn erg langzaam (DR1000).

Een greep uit de inhoud van `agenda-macros-iliad.tex`:

```
\definepapersize [iliad] [width=124mm,height=152mm]

\setuppapersize [iliad] [iliad]

\enableregime[utf8]

\pdfminorversion = 4

\setuplayout[height=14.5cm,
             footer=12pt,
             footerdistance=6pt,
             width=11cm,
             topspace=12pt,
             header=0pt,
             backspace=24pt,
             leftmargin=12pt,
             rightmargin=12pt]

\setupcolors[state=start,conversion=yes,
             reduction=yes,rgb=no,cmyk=no]

\definecolor[papercolor][r=1,b=1,g=1]
```

```
...

\setupbackgrounds[page][state=repeat,
                       background=color,
                       backgroundcolor=papercolor]

...

\startsetups agenda:start
  \blank
  \setupfootertexts[\dofooteragenda]
  \setupfooter[state=high]
  \AgendaGremium
  \blank
  \starttabulate[|l|p|]
  \NC Datum:   \NC \ss\AgendaDatum\NC \NR
  \NC Aanvang: \NC \ss\AgendaAanvang\NC\NR
  \NC Locatie: \NC \ss\AgendaLocatie \NC\NR
  \stoptabulate
  \blank
\stopsetups

\startsetups agenda:stop
  \page
  \the\everyendagenda
  \everyendagenda={}
\stopsetups

\startsetups punten:start
  \startitemize[width=24pt]
\stopsetups

\startsetups punten:stop
  \stopitemize
\stopsetups

....

\startsetups agenda:nodocument
  {\DocumentBody {\tfx bestand te groot voor inclusie}\par }%
\stopsetups
```

Tenslotte

De generatie van PDF bestanden is een klein maar belangrijk onderdeel van NotuDoc. We hebben gekozen voor gebruik van TEX vanwege de hoge kwaliteit van de uitvoer en specifiek voor ConTEXt vanwege het gemak waarmee de vormgeving gescheiden kan worden van de gegevens.

Taco Hoekwater
Elvenkind BV
taco@elvenkind.com

Dealing with xml in ConTEXt MkIV

Introduction

This manual presents the MkIV way of dealing with xml. Although the traditional MkII streaming parser has a charming simplicity in its control, for complex documents the tree based MkIV method is more convenient. We expect that the old method will be used less and less and eventually it might become a module in MkIV.

The user interface is sort of experimental but most commands discussed here are in use already in styles that we make and therefore these commands will stay. Over time we will add more examples to this document.

If you are familiar with xml processing in MkII, then you will have noticed that the MkII commands have `XML` in their name. The MkIV commands have a lowercase `xml` in their names. That way there is no danger for a mixup.

You may wonder why we do these manipulations in TEX and not use xslt instead. The advantage of an integrated approach is that it simplifies usage. Think of not only processing the a document, but also using xml for managing resources in the same run. Also, an xslt approach is just as verbose (after all, you still need to produce TEX code) and probably less readable. In the case of MkIV the integrated approach is also faster and gives us the option to manipulate content at runtime using Lua.

This manual is dedicated to Taco Hoekwater, one of the first ConTEXt users, and also the first to use it for processing xml. Who could have thought at that time that we would have a more convenient way of dealing with those angle brackets.

Hans Hagen, Pragma ADE, August 2008

This is the first version of this manual. Some details of the implementation might change and this manual may contain errors.

Setting up a converter

from structure to setup

We use a very simple document structure for demonstrating how a converter is defined. In practice a mapping will be more complex, especially when we have a style with non standard titles and formatting.

```
<?xml version='1.0' standalone='yes?>

<document>
  <section>
    <title>Some title</title>
    <content>
      <p>a paragraph of text</p>
      <p>another paragraph of text</p>
    </content>
  </section>
</document>
```

Suppose this document is stored in the file `demo.xml`, then the following code can be used as starting point:

```
\startxmlsetups xml:demo:base
  \xmlsetsetup{demo}{*}{-}
  \xmlsetsetup{demo}{document|section|p}{xml:demo:*}
\stopxmlsetups

\xmlregisterdocumentsetup{demo}{xml:demo:base}

\startxmlsetups xml:demo:document
  \title{Contents}
  \placelist[chapter]
  \page
  \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:demo:section
  \section{\xmlfirst{#1}{/title}}
  \xmlfirst{#1}{/content}
\stopxmlsetups

\startxmlsetups xml:demo:p
  \xmlflush{#1}\endgraf
\stopxmlsetups

\xmlprocessfile{demo}{demo.xml}{}
```

Watch out! These are not just setups, but specific xml setups which get an argument passed (the `#1`). If for some reason your xml processing fails, it might be that you mistakenly have used a normal setup definition.

For the moment stop wondering what some (empty) arguments are doing here. Contrary to the style definitions this interface looks rather low level (with no optional arguments) and the main reason for this is that we want processing to be fast. So, the basic framework is:

```
\startxmlsetups xml:demo:base
  % associate setups with elements
\stopxmlsetups

\xmlregisterdocumentsetup{demo}{xml:demo:base}

% define setups for matches

\xmlprocessfile{demo}{demo.xml}{}
```

In this example we mostly just flush the content of an element and in the case of a section we flush explicit child elements. The `#1` in the example code represents the current element.

The line:

```
\xmlsetsetup{demo}{*}{-}
```

sets the default for each element to 'just ignore it'. A + would make the default to always flush the content. This means that at this point we only handle:

```
<section>
  <title>Some title</title>
```

```
  <content>
    <p>a paragraph of text</p>
  </content>
</section>
```

In the next section we will deal with the slightly more complex itemize and figure placement.

alternative solutions

Dealing with an itemize is rather simple (as long as we forget about attributes that control the behaviour):

```
<itemize>
  <item>first</item>
  <item>second</item>
</itemize>
```

First we need to add `itemize` to the setup assignment:

```
\xmlsetsetup{demo}{document|section|p|itemize}{xml:demo:*}
```

The setup can look like:

```
\startxmlsetups xml:demo:itemize
  \startitemize
    \xmlfilter{#1}{/item/command(xml:demo:itemize:item)}
  \stopitemize
\stopxmlsetups

\startxmlsetups xml:demo:itemize:item
  \startitem
    \xmlflush{#1}
  \stopitem
\stopxmlsetups
```

An alternative is to map item directly:

```
\xmlsetsetup{demo}{document|section|p|itemize|item}{xml:demo:*}
```

and use:

```
\startxmlsetups xml:demo:itemize
  \startitemize
    \xmlflush{#1}
  \stopitemize
\stopxmlsetups

\startxmlsetups xml:demo:item
  \startitem
    \xmlflush{#1}
  \stopitem
\stopxmlsetups
```

Sometimes a more local solution makes sense, especially when the `item` tag is used for other purposes as well.

This leaves us with dealing with the resources, like figures.

```
<resource type='figure'>
  <caption>A picture of a cow.</caption>
  <content><external file="cow.pdf"/></content>
</resource>
```

Here we can use a more restricted match:

```
\xmlsetsetup{demo}{resource[@type='figure']}{xml:demo:figure}
\xmlsetsetup{demo}{external}{xml:demo:*}
```

and the definitions:

```
\startxmlsetups xml:demo:figure
  \placefigure
    {\xmlfirst{#1}{/caption}}
    {\xmlfirst{#1}{/content}}
\stopxmlsetups

\startxmlsetups xml:demo:external
  \externalfigure[\xmlatt{#1}{file}]
\stopxmlsetups
```

At this point it is good to notice that `\xmlatt{#1}{file}` is passed as it is, a macro call. This means that when a macro like `\externalfigure` uses the first argument frequently without first storing its value, the lookup is done several times. A solution for this is:

```
\startxmlsetups xml:demo:external
  \expanded{\externalfigure[\xmlatt{#1}{file}]}
\stopxmlsetups
```

Because the lookup is rather fast, normally there is no need to bother about this too much.

An alternative definition for placement is the following:

```
\xmlsetsetup{demo}{resource}{xml:demo:resource}
```

with:

```
\startxmlsetups xml:demo:resource
  \placefloat
    [\xmlatt{#1}{type}]
    {\xmlfirst{#1}{/caption}}
    {\xmlfirst{#1}{/content}}
\stopxmlsetups
```

This way you can specify `table` as type too. Because you can define your own float types, more complex variants are also possible. In that case it makes sense to provide some default behaviour too:

```
\definefloat[figure-here][figures-here][figure]
\definefloat[figure-left][figures-left][figure]
\definefloat[table-here] [tables-here] [table]
\definefloat[table-left] [tables-left] [table]

\setupfloat[figure-here][default=here]
\setupfloat[figure-left][default=left]
\setupfloat[table-here] [default=here]
\setupfloat[table-left] [default=left]

\startxmlsetups xml:demo:resource
  \placefloat
    [\xmlattdef{#1}{type}{figure}-\xmlattdef{#1}{location}{here}]
    {\xmlfirst{#1}{/caption}}
```

```
    {\xmlfirst{#1}{/content}}
\stopxmlsetups
```

In this example we support two types and two locations. We default to a figure placed (when possible) at the current location.

Filtering content

TEX versus LUA

It will not come as a surprise that we can access xml files from TEX as well as from Lua. In fact there are two methods to deal with xml in Lua. First there are the low level xml functions in the `xml` namespace. On top of those functions there is a set of functions in the `lxml` namespace that deals with xml in a more TEXie way. Most of these have similar commands at the TEX end.

```
\startxmlsetups first:demo:one
  \xmlsetsetup{demo}{*}{-}
  \xmlfilter{demo}{artist/name[text()='Randy Newman']/../albums%
    /album[position()=3]/../command(first:demo:two)}
\stopxmlsetups

\startxmlsetups first:demo:two
  \blank \start \tt
    \xmldisplayverbatim{#1}
  \stop \blank
\stopxmlsetups

\xmlregistersetup{first:demo:one}

\xmlprocessfile{demo}{music-collection.xml}{}
```

This gives the following snippet of verbatim xml code. The indentation is conform the indentation in the whole xml file.[1]

```
    <name>Land Of Dreams</name>
      <tracks>
       <track length="248">Dixie Flyer</track>
       <track length="212">New Orleans Wins The War</track>
       <track length="218">Four Eyes</track>
       <track length="181">Falling In Love</track>
       <track length="187">Something Special</track>
       <track length="168">Bad News From Home</track>
       <track length="207">Roll With The Punches</track>
       <track length="209">Masterman And Baby J</track>
       <track length="134">Follow The Flag</track>
       <track length="246">I Want You To Hurt Like I Do</track>
       <track length="248">It's Money That Matters</track>
       <track length="156">Red Bandana</track>
      </tracks>
```

An alternative written in Lua looks as follows:

```
\blank \start \tt \startluacode
  local m = lxml.load("mine","music-collection.xml") -- m == lxml.id("mine")
  local p = "artist/name[text()='Randy Newman']/../albums/album[position()=4]/.."
  local r, d, k = xml.filter(m,p)
```

1. The xml file contains the collection stored on my slimserver instance.

```
  lxml.displayverbatim(d)
\stopluacode \stop \blank
```

This produces:

```
    <name>Bad Love</name>
      <tracks>
       <track length="340">My Country</track>
       <track length="295">Shame</track>
       <track length="205">I'm Dead (But I Don't Know It)</track>
       <track length="213">Every Time It Rains</track>
       <track length="206">The Great Nations of Europe</track>
       <track length="220">The One You Love</track>
       <track length="164">The World Isn't Fair</track>
       <track length="264">Big Hat, No Cattle</track>
       <track length="243">Better Off Dead</track>
       <track length="236">I Miss You</track>
       <track length="126">Going Home</track>
       <track length="180">I Want Everyone To Like Me</track>
      </tracks>
```

You can use both methods mixed but in practice we will use the TEX commands in regular styles and the mixture in modules, for instance in those dealing with MathML and cals tables.

a few details

In ConTEXt 'setups' are a rather common variant on macros. An example of a setup is:

```
\startsetup doc:print
  \setuppapersize[A4][A4]
\stopsetup

\startsetup doc:screen
  \setuppapersize[S6][S4]
\stopsetup
```

Later on we can say something like:

```
\doifmodeelse {paper} {
  \setup[doc:print]
} {
  \setup[doc:screen]
}
```

Another example is:

```
\startsetup[doc:header]
  \marking[chapter]
  \space
  --
  \space
  \pagenumber
\stopsetup
```

in combination with:

```
\setupheadertexts[\setup{doc:header}]
```

Here the advantage is that instead of ending up with an unreadable header definition, we use a nicely formatted setup. The advantage of a setup is that spaces are ignored.

The only difference between setups and xml setups is that the latter ones get an argument (`#1`) that reflects the current node in the xml tree.

Commands

nodes and lpaths

The amount of commands available for manipulating the xml file is rather large. Many of the commands cooperate with so called setups, a fancy name for a collection of macro calls either or not mixed with text.

Most of the commands are just shortcuts to Lua calls, which means that the real work is done by Lua. In fact, what happens is that we have a continuous transfer of control from TEX to Lua, where Lua prints back either data (like element content or attribute values) or just invokes a setup whereby it passes a reference to the node resolved conform the path expression. The invoked setup itself might return control to Lua again, etc.

This sounds complicated but examples will show what we mean here. First we present the whole repertoire of commands. Because users can read the source code, they might uncover more commands, but only the ones discussed here are official. The commands are grouped in categories.

In the following sections `node` means a reference to a node: a document `id` (string) or an argument to a setup (result from a lookup). An `lpath` is a fancy name for a path expression (as with xslt) but resolved by Lua. A `filter` is an action that is applied to the result of a lookup.

loading

`\xmlload {id} {filename}` loads the file `filename` and registers it under `id`

`\xmlloadbuffer {id} {buffer}` loads the buffer `buffer` and registers it under `id`

`\xmlloaddata {id} {string}` loads `string` and registers it under `id`

`\xmlinclude {node} {lpath} {attribute}` includes the file specified by `attribute` of the element located by `lpath` at node `node`

`\xmlprocessfile {id} {filename} {initial-xml-setup}` registers file `filename` as `id` and process the tree starting with `initial-xml-setup`

`\xmlprocessbuffer {id} {buffer} {initial-xml-setup}` registers buffer `buffer` as `id` and process the tree starting with `initial-xml-setup`

`\xmlprocessdata {id} {string} {initial-xml-setup}` registers `string` as `id` and process the tree starting with `initial-xml-setup`

The initial setup defaults to `xml:process` that is defined as follows:

```
\startsetups xml:process
  \xmlregistereddocumentsetups\xmldocument
  \xmlmain\xmldocument
\stopsetups
```

First we apply the setups associated with the document (including common setups) and then we flush the whole document. The macro `\xmldocument` expands to the current document id. There is also `\xmlself` which expands to the current node number (`#1` in setups).

`\xmlmain {id}` returns the whole documents

Normally such a flush will trigger a chain reaction of setups associated with the child elements.

flushing data

When we flush an element, the associated xml setups are expanded. The most straightforward way to flush an element is the following. Keep in mind that the returned values itself can trigger setups and therefore flushes.

`\xmlflush {node}` returns all nodes under `node`
You can restrict flushing by using commands that accept a specification.

`\xmltext {node} {lpath}` returns the text of the matching `lpath` under `node`

`\xmlall {node} {lpath}` returns all nodes under `node` that matches `lpath`

`\xmlfirst {node} {lpath}` returns the first node under `node` that matches `lpath`

`\xmllast {node} {lpath}` returns the last node under `node` that matches `lpath`

`\xmlfilter {node} {lpath/filter}` at a match of `lpath` a filter `filter` is applied and the result is returned

`\xmlsnippet {node} {n}` returns the n^{th} element under `node`

`\xmlindex {node} {lpath} {n}` returns the n^{th} match of `lpath` at node `node`; a negative number starts at the end

`\xmlconcat {node} {lpath} {text}` returns the sequence of nodes that match `lpath` at `node` whereby `text` is put between each match

`\xmlconcatrange {node} {lpath} {text} {n} {m}` returns the n^{th} upto m^{th} of nodes that match `lpath` at `node` whereby `text` is put between each match

`\xmlcommand {node} {lpath} {xml-setup-id}` apply the given setup to each match of `lpath` at node `node`

`\xmlstrip {node} {lpath}` remove leading and trailing spaces from nodes under `node` that match `lpath`

`\xmlstripped {node} {lpath}` remove leading and trailing spaces from nodes under `node` that match `lpath` and return the content afterwards

`\xmlstripnolines {node} {lpath}` remove leading and trailing spaces as well as collapse embedded spaces from nodes under `node` that match `lpath`

`\xmlstrippednolines {node} {lpath}` remove leading and trailing spaces as well as collapse embedded spaces from nodes under `node` that match `lpath` and return the content afterwards

`\xmlinlineverbatim {node} {lpath}` return the content of the `lpath` match as inline verbatim code, that is no further interpretation (expansion) takes place and spaces are honoured

`\xmldisplayverbatim {node} {lpath}` return the content of the `lpath` match as display verbatim code, that is no further interpretation (expansion) takes place and leading and trailing spaces and newlines are treated special

information

The following commands return strings. Normally these are used in tests.

`\xmlname {node}` returns the complete name (including namespace prefix) of the given `node`

`\xmlnamespace {node}` returns the namespace of the given `node`

`\xmltag {node}` returns the tag of the element, without namespace prefix

`\xmltags {node} {lpath}` returns a comma-separated list of tags of elements that match the `lpath`

`\xmlcount {node} {lpath}` returns the number of matches of `lpath` at node `node`

`\xmlnofelements {node}` returns the number of elements at node `node`

`\xmlatt {node} {name}` returns the value of attribute `name` or empty if no such attribute exists

`\xmlattdef {node} {name} {default}` returns the value of attribute `name` or `default` if no such attribute exists

`\xmlattribute {node} {lpath} {name}` finds a first match for `lpath` at `node` and returns the value of attribute `name` or empty if no such attribute exists

`\xmlattributedef {node} {lpath} {name} {default}` finds a first match for `lpath` at `node` and returns the value of attribute `name` or `default` if no such attribute exists

manipulation

You can use Lua code to manipulate the tree and it makes no sense to duplicate this in TEX. So, we only provide an interface to the most useful manipulators.

`\xmldelete {node} {lpath}` deletes all children of `node` that match `lpath`

integration

If you write a module that deals with xml, for instance processing cals tables, then you need ways to control specific behaviour. For instance, you might want to add a background to the table. Such directives are collected in xml files and can be loaded on demand.

`\xmlloaddirectives {filename}` loads ConTEXt directives from `filename` that will get interpreted when processing documents

A directives definition file looks as follows:

```
<?xml version="1.0" standalone="yes"?>

<directives>
  <directive attribute='id' value="100"
    setup="cdx:100"/>
  <directive attribute='id' value="101"
    setup="cdx:101"/>
  <directive attribute='cdx' value="colors" element="cals:table"
    setup="cdx:cals:table:colors"/>
  <directive attribute='cdx' value="vertical" element="cals:table"
    setup="cdx:cals:table:vertical"/>
  <directive attribute='cdx' value="noframe" element="cals:table"
    setup="cdx:cals:table:noframe"/>
  <directive attribute='cdx' value="*" element="cals:table"
    setup="cdx:cals:table:*"/>
</directives>
```

Examples of usage can be found in `x-cals.mkiv`. The directive is triggered by an attribute. Instead of `setup` you can specify `before` and `after`.

`\xmldirectives {node} {lpath}` apply the setups directive associated with the found nodes

`\xmldirectivesbefore {node} {lpath}` apply the before directives associated with the found nodes

`\xmldirectivesafter {node} {lpath}` apply the after directives associated with the found nodes

Normally a directive will be put in the xml file, for instance as:

```
<?context-mathml-directive minus reduction yes ?>
```

Here the `mathml` is the general class of directives and `minus` a subclass, in our case a specific element. You can also invoke such directives directly:

`\xmlcontextdirective {kind} {class} {key} {value}` execute the directive associated with `kind` and pass three arguments to it

This assumes that there is a command `xmlkinddirective` or in the MathML example `xmlmathmldirective` that does something useful.

setups

The basic building blocks of xml processing are setups. These are just collections of macros that are expanded. These setups get one argument passed (`#1`):

```
\startxmlsetups somedoc:somesetup
    \xmlflush{#1}
\stopxmlsetups
```

This argument is normally a number that internally refers to a specific node in the xml tree. The user should see it as an abstract entity and not depend on it being a number. Just think of it as 'the current node'. You can (and probably will) call such setups directly:

`\xmlsetup {name} {node}` expands setup `name` and pass `node` as argument

However, in most cases the setups are associated to specific elements, something that users of xslt might recognize as templates.

`\xmlsetfunction {name} {lpath} {function}` associates function Lua `function` to the elements in namespace `name` that match `lpath`

`\xmlsetsetup {name} {lpath} {setup}` associates setups (TEX code) `setup` to the elements in namespace `name` that match `lpath`

`\xmlprependsetup {setup}` pushes `setup` to the front of global list of setups to be applied

`\xmlappendsetup {setup}` pushes `setup` to the end of global list of setups to be applied

`\xmlbeforesetup {setup} {position}` inserts `setup` before setup `position` in the global list of setups to be applied

`\xmlaftersetup {setup} {position}` inserts `setup` after setup `position` in the global list of setups to be applied

`\xmlremovesetup {setup}` removes `setup` from the global list of setups to be applied

`\xmlprependdocumentsetup {id} {setup}` pushes `setup` to the front of `id` specific list of setups to be applied

`\xmlappenddocumentsetup {id} {setup}` pushes `setup` to the end of `id` specific list of setups to be applied

`\xmlbeforedocumentsetup {id} {setup} {position}` inserts `setup` before setup `position` in the `id` specific list of setups to be applied

`\xmlafterdocumentsetup {id} {setup} {position}` inserts `setup` after setup `position` in the `id` specific list of setups to be applied

`\xmlremovedocumentsetup {setup}` removes `setup` from the `id` specific list of setups to be applied

`\xmlresetdocumentsetups {id}` removes all setups from the `id` specific list of setups to be applied

`\xmlflushdocumentsetups {id}` applies all setups in tagged with `id`

`\xmlregisteredsetups` applies all global setups to the current document

`\xmlregistereddocumentsetups` applies all document specific setups to the current document

testing

The following test macros all take a `node` as first argument and an `lpath` as second:

`\xmldoif {node} {lpath} {yes}` expands to `yes` when `lpath` matches at node `node`

`\xmldoifnot {node} {lpath} {no}` expands to `no` when `lpath` does not match at node `node`

`\xmldoifelse {node} {lpath} {yes} {no}` expands to `yes` when `lpath` matches at node `node` and to `no` otherwise

`\xmldoiftext {node} {lpath} {yes}` expands to `yes` when the node matching `lpath` at node `node` has some content

`\xmldoifnottext {node} {lpath} {no}` expands to `do-if-fase` when the node matching `lpath` at node `node` has no content

`\xmldoifelsetext {node} {lpath} {yes} {no}` expands to `yes` when the node matching `lpath` at node `node` has content and to `no` otherwise

`\xmldoifelseempty {node} {lpath} {yes} {no}` expands to `yes` when the node matching `lpath` at node `node` is empty and to `no` otherwise

`\xmldoifelseselfempty {node} {lpath} {yes} {no}` expands to `yes` when the node matching `lpath` at node `node` is empty and to `no` otherwise

initialization

The general setup command (not to be confused with setups) that deals with the MkIV tree handler is `\setupxml`. There are currently only a few options.

When you set `method` to `mkiv`, the traditional handler will not kick in when xml code ends up in TEX. When we have replaced all usage of the MkII method in the core of ConTEXt, we might make this default.

When you set `default` to `text` elements with no setup assigned will end up as text. When set to `none` such elements will be hidden. When no value is set the outcome depends on the method: interpreted as xml in for `mkii` and text for method `mkiv`.

You can set `compress` to `yes` in which case comment is stripped from the tree when the file is read. When `entities` is set to `yes` (this is the default) entities are replaced.

`\xmlregisterns {internal} {public}` associates an internal namespace (like `mml`) with one given in the document as url (like mathml)

`\xmlremapname {node} {lpath} {new-namespace} {new-tag}` changes the namespace and tag of the matching elements

`\xmlremapnamespace {node} {lpath} {from} {to}` replaces all references to the given namespace to a new one

`\xmlchecknamespace {id} {lpath} {new}` sets the namespace of the matching elements unless a namespace is already set

`\xmldefaulttotext {id}` makes all elements that don't have a setup associated resolve to text

`\xmldefaulttonone {id}` hides all elements that don't have a setup associated

`\xmlutfize {id}` convert all entities to utf if possible

helpers

Often an attribute will determine the rendering and this may result in many tests. Especially when we have multiple attributes that control the output such tests can become rather extensive and redundant because one gets $n \times m$ or more such tests.

Therefore we have a convenient way to map attributes, for instance onto strings or commands.

`\xmlmapvalue {category} {name} {value}` associate a `value` with a `name` and `category`

`\xmlvalue {category} {name} {default}` expand the value `value` associated with a `category` and `name` and if not resolved, expand `default`

This is used as follows. We define a couple of mappings in the same category:

```
\xmlmapvalue{emph}{bold}  {\bf}
\xmlmapvalue{emph}{italic}{\it}
```

Assuming that we have associated the following setup with the `emph` element, we can say (with `#1` being the current element):

```
\startxmlsetups demo:emph
  \begingroup
    \xmlvalue{emph}{\xmlatt{#1}{type}}{}
  \endgroup
\stopxmlsetups
```

In this case we have no default. The type attribute triggers the actions, as in:

```
normal <emph type='bold'>bold</emph> normal
```

This mechanism is not really bound to elements and attributes so you can use this mechanism for other purposes as well.

synonyms

A few of the discussed commands have synonyms

```
\xmlmapval                  \xmlmapvalue
\xmlval                     \xmlvalue
\xmlregistersetup           \xmlappendsetup
\xmlregisterdocumentsetup   \xmlappenddocumentsetup
```

Expressions and filters

path expressions

In the previous sections we used `lpath` expressions, which are a variant on `xpath` expressions as in xslt but in this case more geared towards usage in TEX. These mechanisms will be extended when needed.

A path is a sequence of matches. A simple path expression is:

```
a/b/c/d
```

Here each `/` goes one level deeper. We can go backwards in a lookup with `..`:

```
a/b/../d
```

We can also combine lookups, as in:

```
a/(b|c)/d
```

A negated lookup is preceded by a `!`:

```
a/(b|c)/!d
```

A wildcard is specified with a `*`:

```
a/(b|c)/!d/e/*/f
```

In addition to these tag based lookups we can use attributes:

```
a/(b|c)/!d/e/*/f[@type=whatever]
```

An `@` as first character means that we are dealing with an attribute. Within the square brackets there can be boolean expressions:

```
a/(b|c)/!d/e/*/f[@type=whatever and @id>100]
```

You can use functions as in:

```
a/(b|c)/!d/e/*/f[something(text()) == "oeps"]
```

There are a couple of predefined functions:

`position`	number	the current index of the matched element
`index`	number	the current index upto the matched element
`text`	string	the textual representation of the matched element
`name`	string	the full name of the matched element: namespace and tag
`ns`	string	the namespace of the matched element
`tag`	string	the tag of the matched element
`attribute`	string	the value of the attribute with the given name of the matched element

You can pass your own functions too. Such functions are defined in the `xml.expressions` namespace. We have defined a few shortcuts:

```
xml.expressions.contains = string.find
xml.expressions.find     = string.find
xml.expressions.upper    = string.upper
xml.expressions.lower    = string.lower
xml.expressions.number   = tonumber
xml.expressions.boolean  = toboolean -- mkiv specific
```

You can also use normal Lua functions as long as you make sure that you pass the right arguments. There are a few predefined variables available inside such functions.

r	table	the root of the element
d	table	the roots data table
k	number	the current index into the roots data table
e	table	the element (d[k])
ns	string	the namespace (e.rn or e.ns)
tg	string	the tag (e.tg)
dt	table	the content (e.dt)
at	table	a hash containing the attributes (e.at)
id	number	the current elements index (not counting text)
ps	number	the current elements position (differs from id if mixed elements)
tx	string	the (first) text (dt[1])

The given expression between [] is converted to a Lua expression so you can use the usual ingredients:

```
== ~= <= >= < > not and or ()
```

In addition, = equals == and != is the same as ~=. If you mess up the expression, you quite likely get a Lua error message.

functions as filters

At the Lua end a whole lpath expression results in a (set of) node(s) with its environment, but that is hardly usable in TEX. Think of code like:

```
for r, d, k in xml.elements(xml.load('text.xml'),"title") do
  -- r = root of the title element
  -- d = data table
  -- k = index in data table
end
```

Here d[k] points to the title element and in this case all titles in the tree pass by. In practice this kind of code is encapsulated in function calls, like those returning elements one by one, or returning the first or last match. The result is then fed back into TEX, possibly after being altered by an associated setup. We've seen the wrappers to such functions already in a previous section.

In addition to the previously discussed expressions, one can add so called filters to the expression, for instance:

```
a/(b|c)/!d/e/text()
```

In a filter, the last part of the lpath expression is a function call. The previous example returns the text of each element e that results from matching the expression. Examples of functions are:

text	string	returns the content
name	string	returns the (either or not remapped) namespace
ns	string	returns gives the original namespace
tag	string	returns the elements name
count	number	returns the elements name

Not all such functions make sense in TEX, for instance because they return a data structure that is useless for TEX itself. Instead of using functions like first(), you can as well use \xmlfirst which might be more efficient.

attribute(name)	returns the attribute with the given name
command(name)	expands the setup with the given name for each found element
position(n)	processes the n^{th} instance of the found element
first()	processes the first instance of the found element
last()	processes the last instance of the found element

These filters are in fact Lua functions which means that if needed more of them can be added. Indeed this happens in some of the xml related MkIV modules, for instance in the MathML processor.

tables

If you want to know how the internal xml tables look you can print such a table:

```
print(table.serialize(e))
```

This produces for instance:

```
t={
 ["at"]={
  ["label"]="whatever",
 },
 ["dt"]={ "some text" },
 ["ns"]="",
 ["rn"]="",
 ["tg"]="demo",
}
```

The `rn` entry is the renamed namespace (when renaming is applied). If you see tags like `@pi@` this means that we don't have an element, but (in this case) a processing instruction.

`@rt@`	the root element
`@dd@`	document definition
`@cm@`	comment, like `<!-- whatever -->`
`@cd@`	so called `CDATA`
`@pi@`	processing instruction, like `<?whatever we want ?>`

There are many ways to deal with the content, but in the perspective of TEX only a few matter.

`xml.sprint(e)`	print the content to TEX and apply setups if needed
`xml.tprint(e)`	print the content to TEX (serialize elements verbose)
`xml.cprint(e)`	print the content to TEX (used for special content)

Keep in mind that anything low level that you uncover is not part of the official interface unless mentioned in this manual.

Hans Hagen
Pragma ADE, Hasselt

Printing labels with ConTEXt

Abstract
Sometimes one needs to print a single label which will be glued onto a package, a large envelope or for the identification of a box. In certain situations one wants to produce a series of identical labels or one needs to typeset whole databases of addresses. ConTEXt offers the possibility of using the XY-arranging procedure to print on each of the labels being present on a sheet. Here a possible approach is presented for labels of the size 105 × 42.3mm i.e. (14 labels on a A_4). It is shown how to print a single label but also how to get multiple copies of the same content and how to prepare sheets of labels containing the addresses of a database.

Keywords
Maps, Context, layer, label, XY-arrangement

Introduction

Like the address-printing on an envelope, as described in an other article in this MAPS, printing labels from a multi-label sheet is not easy in the beginning. The issue often is, that only a single label is needed. The goal is to be able to use all of the labels on a given sheet one after the other.

When creating a tool to use all the labels, the sheet must be sent through the (laser-)printer as many times as there are labels on the sheet. Due to the fact that the carrier-sheet of the labels is quite thin, it is necessary to use the labels from bottom up.

In the following article a possible setup is given. The solution makes use of the XY-arrangement and layers.

Those who want to prepare bulk mailings in The Netherlands should also think of using the KIX-code (klantenindexcode, customer index code). The Dutch mail service TNT provides a barcode-font. With this font it is possible to print this code directly on the label.

Guidelines as provided by TNT

The Dutch Mail Service TNT provides guidelines for making a label. This information is also applicable to the printing of addresses on envelopes (See the other article in this MAPS).

In general one uses three to a maximum of six lines per address. The last line consists of the ZIP-code and the place, which is always printed in uppercase characters. The before last line contains either the street name + house number and possibly an extension to the house number or the P.O. box + number. Before these two lines a 'to the attention of' line can be added. The other lines will contain other information concerning the address of the receiver.

In The Netherlands, one has the opportunity to add a so called customer index code KIX (klantenindexcode) in the form of a barcode. This code is unique for each address. The barcode is composed of the ZIP-code + house number (or P.O. box + number) + separator character 'X' + house number extension. The KIX-code must be printed at 10pt.

Print the KIX-code at the top of the address or as an additional line beneath the address. Always keep a minimal distance of 2mm but no more than 15mm from the last line of the address. KIX-codes should not be printed with matrix-printers due to the printout quality of these printers.

For sending mail to foreign countries the name of the receiver's country is marked down in uppercase characters as the last line of the address. Do not use KIX-codes for mail to a foreign country.

The guidelines advise you to use sans-serif fonts in the address whenever possible. Italic, script fonts, gothic fonts, matrix-characters, condensed and expanded fonts are advised against because they negatively influence the automatic read results. The character size should be no less than 7pt, nor should it exceed 17pt. It is advised to use uppercase characters throughout the address if the character size is less than 10pt.

There are even more guidelines on how to setup the alignment, the interline space and the use of spaces between words and underlining. If one lets TEX typeset the text, those aspects should be within the given rules.

Return address

In The Netherlands one can place the return-address on top of the receiver's address. It is important that this is a single line only and it must be separated from the receiver's address by a thick rule of at least 1.2mm. The white space between the rule and the first line of the receiver's address should be 5mm.

Installing the KIX-font for MKIV

Download the KIX-font (ttf format). Unpack the zip-file and copy the font into a font-directory, where you prepare a new foundry map e.g. "TNT" in the ttf-directory.

There are different options to make the new font known to ConTEXt. Along the method for other fonts one can write a type-script-file containing the following lines:

```
\starttypescript [sans] [kix]
   \definefontsynonym
     [KIX-Roman]
       [file:kixbrg][features=default]
\stoptypescript

\starttypescript [sans] [kix] [name]
   \definefontsynonym
     [Sans][KIX-Roman][features=default]
\stoptypescript

\starttypescript [KIX]
   \definetypeface
     [KIX][ss][sans][kix][default]
\stoptypescript
```

Save the type-script-file in the user directory of ConTEXt as `type-TNT-KIX.tex`. Run `luatools --generate`.

Now one can add the KIX-font in the preamble of the working file:

```
\usetypescriptfile[type-TNT-KIX]
\usetypescript[KIX]
```

For typing the actual KIX-code a small macro is defined as follows:

```
\def\KIX#1%
   {\switchtobodyfont[KIX,ss,10pt]#1}
```

Because the KIX-font does not have any different styles and one uses it invariably in a single way, there is also a shorter way to tell ConTEXt how to use this font.

First a font synonym is created, where a symbolic name is linked to the font-file-name.

```
\definefontsynonym
   [KIX][file:kixbrg][features=default]
```

Again we define a macro for typesetting the KIX-code

```
\def\KIX
  {\groupedcommand
     {\definedfont[KIX at 10pt]}{}}
```

The contents of the label

The contents will consist of two elements. First, at the top of the label, the return-address is typeset with a thick black rule underneath. For this purpose one can use a buffer.

```
\startbuffer[Returnaddress]
   \framedtext
      [frame=off,bottomframe=on,
       rulethickness=2pt,
       offset=5pt]%
      {Willi Egger,
       Maasstraat 2,
       5836 BB~~{\sc Sambeek}}
\stopbuffer
```

The receiver's address is also placed in a buffer.

```
\startbuffer[Receiver]
   \framedtext{%
      \startlines
         NTG-secretary
         Maasstraat 2
         5836 BB~~{\sc Sambeek}
      \stoplines}
\stopbuffer

\startbuffer[ReceiverKIX]
   \framedtext{%
      \startlines
         NTG-secretary
         Maasstraat 2
         5836 BB~~{\sc Sambeek}
         \KIX{5836BB2}
      \stoplines}
\stopbuffer

\startbuffer[ReceiverAbroad]
   \framedtext{%
      \startlines
         NTG-secretary
         Maasstraat 2
         5836 BB~~{\sc Sambeek}
         THE NETHERLANDS
      \stoplines}
\stopbuffer
```

Setting up the tool

The first thing one has to do is to get the precise dimensions of the labels. For this example a standard A_4-label-sheet carrying 14 labels arranged in two columns is used. The sheet has no print-margins, so the labels fill the whole sheet. The labels are 105mm wide and 42.3mm high.

Next we define a paper-size with the measured dimensions of the label.

```
\definepapersize
   [Label]
   [height=42.3mm,width=105mm]
```

Now we tell ConTEXt that we will place this new paper-size on A_4:

```
\setuppapersize[Label][A4,portrait]
```

Because we will place multiple labels i.e. pages on the A_4 we set up the paper/sheet as follows.

```
\setuppaper
  [topspace=0mm,
   backspace=0mm,
   dx=2mm,
   dy=0mm,
   nx=2,
   ny=7,
   margin=0,
   width=210mm,
   height=297mm]
```

Now we need to instruct ConTEXt how the label/page should be set up.

```
\setuplayout
   [topspace=4mm,
    backspace=5mm,
    margin=0mm,
    width=95mm,
    height=34mm,
    header=0mm,
    footer=0mm]
```

Finally we instruct ConTEXt to use XY-arranging according to the parameters set in the \setuppaper-block. We will get 2 columns with 7 labels each (14 labels in total).

```
\setuparranging[XY]
```

Because the content of the buffers is in \framedtext we setup the behaviour of \framedtext with

```
\setupframedtexts
  [width=\textwidth,frame=off,offset=5pt]
```

In the following steps we set up the content of the label. One way to do this is to define a layer with the dimensions of the label.

```
\definelayer
  [Label]
  [width=\paperwidth,height=\paperheight]
```

We define two variables. The first one indicates which of the labels on the sheet will be used. The second defines the total number of labels on the sheet.

```
\def\Uselabel{8}     % label(s) to be typeset
\def\Totallabels{14} % No. of labels per sheet
```

In order to get the address typeset on the correct label, we loop over the number of labels on the sheet. If the counter equals the number defined in \Uselabel, ConTEXt typesets the label, otherwise it typesets nothing and moves on to the next label.

```
\dostepwiserecurse
  {1}
  {\Totallabels}
  {1}%
  {\ifnum\recurselevel=\Uselabel
     {\setlayer
        [Label]
        [preset=lefttop,
         location={right,bottom},
         y=-5mm,x=-3.5mm]
        {\switchtobodyfont[8pt]%
         \getbuffer[Retunraddress]}
      \setlayer
        [Etiket]
        [preset=leftbottom,
         location={right,top},
         y=.9cm,x=-3mm]
        {\switchtobodyfont[10pt]%
         \getbuffer[Receiver]}}
  \else
     {\setlayer
        [Label]
        [preset=lefttop,
         location={right,bottom}]
        {\strut}}
  \fi
  \placelayer[Label]
  \page }
```

In the future, there may be a need to define other label-sizes. Once you have more label-sizes defined, it is easier to put static information into an environment-file. All different definitions are written into a mode-paragraph. – The file we actually work with will contain the list of modes that can be enabled, the call for the environment file and definitions of the label to be used as well as the total number of labels on a sheet. The receiver's address is put in a buffer. By means

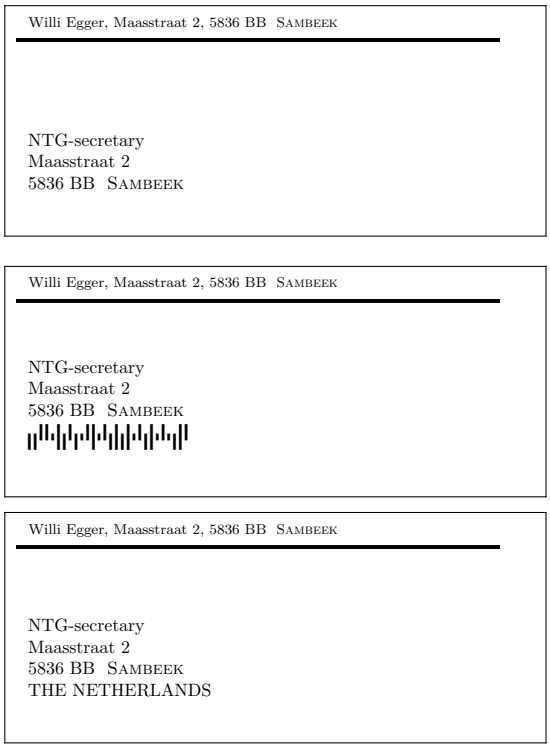


Figure 1. Results of the Examples of Different setups for Single Labels

of testing the active mode the respective loop-block is executed. So this file could look like this:

```
\enablemode[xxx]
\environment layout

\def\Uselabel{2}
\def\Totallabels{...}

\startbuffer[Receiver]
...
\stopbuffer

\doifmode{xxx}{%
   \dostepwiserecurse
      {1}{\Totallabels}{1}{ ... }}
```

Typesetting multiple labels with the same content

Sometimes one would like to typeset a series of labels with the same content. What one can do, is to define the labels, which should be skipped. Define the quantity of the labels needed. By means of looping over the number of labels one can get the desired result.

```
\def\Keepempty{4}
% Skip over the first 4 labels
\def\Quantity{10}

\dorecurse{\Keepempty}{%
  \setlayer
    [Label]
    [preset=righttop,
     location={left,bottom}]
    {\strut}
  \startstandardmakeup
     \placelayer[Label]
  \stopstandardmakeup}

\dorecurse{\Quantity}{%
  \setlayer
    [Label]
    [preset=righttop,
     location={left,bottom},
     x=8mm]
    {\switchtobodyfont[10pt]%
     \getbuffer[Labelcontent]}
 \startstandardmakeup
    \placelayer[Label]
 \stopstandardmakeup}
```

Typesetting series of labels with different content

If you have to deal with mailings, then one needs to be able to process e.g. a database with address-data. Provided that the database has the desired structure one can typeset the labels. For the labels used for the NTG-mailings, the following setup is used.

There is a layout file containing the static information. It holds a mode-paragraph defining the label-size, paper-setup, layout-setup and arranging-setup as described above.

For each member a buffer is prepared based on the members database. At the beginning of that file there is a definition of the total number of addresses contained in the database.

```
\def\Addresses{200}

...
\startbuffer[Adr52]
   \framedtext{%
      \startlines
         Dhr. W. Egger
         Maasstraat 2
         5836 BB~~{\sc Sambeek}
      \stoplines}
\stopbuffer
...
```

In the actual file, which is typeset the mode is enabled and the environment file is loaded. Then the database file is read in. Hereafter a loop over the total number of addresses is started.

```
\enablemode[ntg-Labels]
\environment layout
\input MAPS36

\dorecurse
  {\Addresses}
  {\setlayer
    [Label]
    [preset=leftbottom,
     location={right,top},
     y=-3mm,x=-3mm]
    {\switchtobodyfont[10pt]%
     \getbuffer[Adr\recurselevel]}
   \setlayer
    [Label]
    [preset=lefttop,
     location={right,bottom},
     y=.9cm,x=-3.5mm]
   {\switchtobodyfont[5pt]%
    \getbuffer[Retouraddress]}
  \startstandardmakeup
    \placelayer[Label]
  \stopstandardmakeup }
```

Conclusion

Label printing on a sheet containing several labels is a tricky job, due to the fact that the information must be placed very accurately. In many environments one can print labels, however it is often not possible to indicate which of the labels should be used. – With the described approach in ConTEXt it is possible to use all the labels on a sheet. The use of loops makes it possible to typeset either multiple copies of the same information or to typeset an address-database. The tool can easily be adapted to the purpose required.

Willi Egger
w.egger@boede.nl

Printing envelopes with ConTEXt
(example for using layers)

Abstract
Once in a while one has to prepare an envelope with printed address based on the guidelines provided by the Dutch mail service TNT. This short communication shows a way to achieve this with ConTEXt. The article shows the solution for the DL-type of envelope. From there, it is a small step to define other envelopes with different dimensions.

Keywords
Maps, Context, layer, envelope

Introduction

Sometimes one needs envelopes, which carry a printed address and also the address of the sender. In Europe there are a series of well-known sizes of envelopes. In order to be able to print on these envelopes one needs to make preparations. Furthermore the mail services want to process the envelopes automatically. Hence they give guidelines, where to put the information. In the following I would like to present you my solution to printing envelopes.

Envelope Sizes

There are different series of envelopes derived from the DIN paper sizes. Commonly one uses the C-series, however there is also a B-series which is slightly bigger than the C-series.

C-series		B-series	
C6	114×162	B6	125×176
C5	162×229	B5	176×250
C4	229×324	B4	250×353

Another envelope is called DL which is very often used in business-environments. The width is 220mm and its height is 112mm. The information contained in this section comes from http://en.wikipedia.org/wiki/Envelope_size.

Mail Service Requirements

The mail service provides you with guidelines in order to set up envelopes, which can be machine-read and automatically sorted for distribution. The Dutch mail service TNT provides the following instruction for automatically processed mail. The data is based on a brochure downloadable from http://www.tntpost.nl/zakelijk/aanvragen-bestellen/index.aspx. The dimensions are valid for mail with dimensions less than 265 × 380 × 32mm and less than 3kg of weight. The main areas are indicated on the following figure. It is advised to place the receiver's address centered in the receiver's address area.

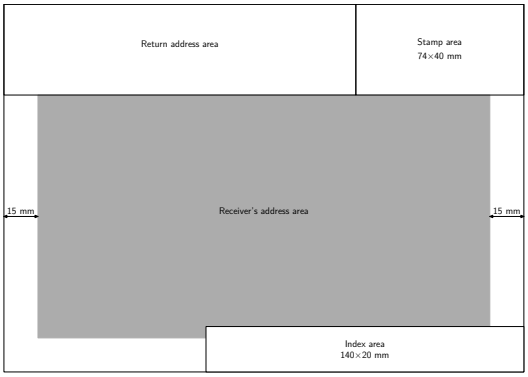


Figure 1. envelope C5 landscape

In the case of a portrait sized envelope the dimensions of the different areas remain the same, however the index area is rotated and adjusted to the lower left corner of the envelope.

C4-envelopes and larger envelopes have a different size and placement of the index area. The size is 100 × 30mm. In case of a landscape oriented envelope, this area is placed vertically (rotated) and down in the left corner. Otherwise the area is placed horizontally at the lower right corner.

For the postcard there is a different set of guidelines, though the basic dimensions of the areas do not differ from normal envelopes.

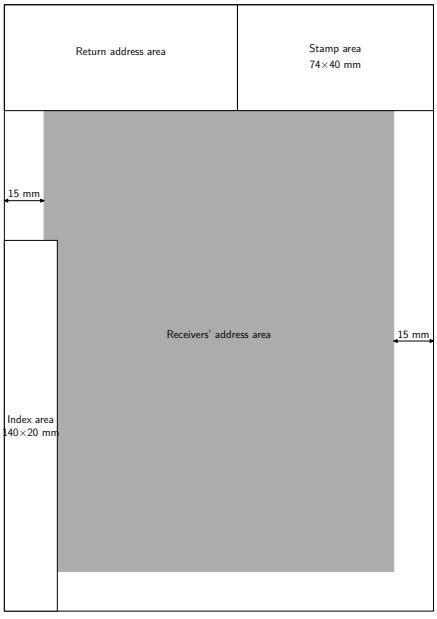


Figure 2. envelope C5 portrait

Remarkable is the size of the index area on a small postcard of 150 × 105mm. The receiver's address area must be at least 74mm wide, which is the width of the stamp area. The receiver's address area is separated from the text area by a vertical bar. The thickness of the this bar should be no less than 1.2mm.

Setting up the Envelope

First we need to define the size of the envelope. The most commonly used sizes are predefined in ConTEXt. – For this example we define a new size.

```
\definepapersize[DL][height=112mm,width=220mm]
\setuppapersize[DL][DL]
```

Later on we want to place blocks of information on this paper-size. In order to have the freedom to move these blocks to a specified position we define a layer, which covers the whole paper-size.

```
\definelayer
  [Envelope]
  [width=\paperwidth,height=\paperheight]
```

Basically we need to place two information-blocks, one containing the address of the recipient and the other containing the address of the sender. In the sender's information one will probably want to place a logo e.g. if the envelope is sent by the secretary of an association or a company. We put both information-blocks into buffers.

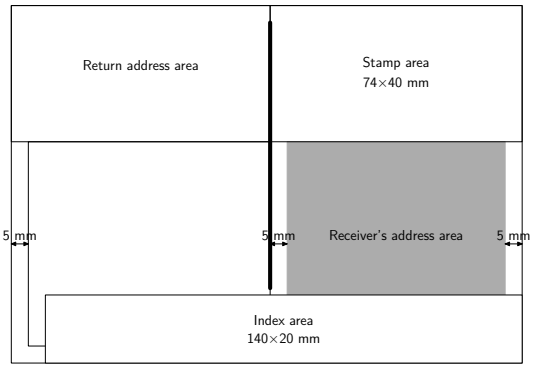


Figure 3. Postcard

```
\startbuffer[Sender]
   \startlines
      W. Egger
      Maassstraat 2
      5836 BB~~{\sc Sambeek}
   \stoplines
\stopbuffer

\startbuffer[Receiver]
   \startlines
      NTG-secretary
      Maasstraat 2
      5836 BB~~{\sc Sambeek}
   \stoplines
\stopbuffer
```

Now we can place the buffers on the already defined layer. The sender's address is placed near the upper left corner.

It is advised to put the receiver's address centered in the receiver's address area. Anyhow, it is always placed in such a way, that a minimum of 20mm or 30mm respectively white space remains for the index area at the bottom and that the address-block honors at least 15mm or 20mm respectively of white space as the right margin. For postcards the right margin is 5mm.

```
\setlayer
  [Envelope]
  [preset=rightbottom,
   location={left,top},y=2.7cm]
  {\framedtext[frame=off]
     {\getbuffer[Receiver]}}
\setlayer
  [Envelope]
  [preset=lefttop,
   location={right,bottom},x=1cm]
  {\framedtext[frame=off]
    {\getbuffer[Sender]}}
```

Now that the information has got its place, the only thing to do yet is typesetting the layer on the paper.

```
\starttext
   \placelayer[Envelope]
\stoptext
```

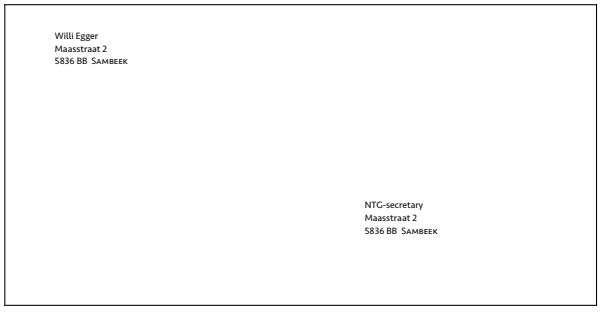


Figure 4. DL envelope

Handling other Envelope Sizes

For handling different envelope sizes in the same file, *modes* come in handy.

For each envelope size we prepare the setup in a mode-paragraph.

```
\startmode[DL-envelope]
   \definepapersize
     [DL][height=112mm,width=220mm]
   \setuppapersize
    [DL][DL]
   \definelayer
    [Envelope]
    [width=\paperwidth,height=\paperheight]
\stopmode
```

Another envelope definition might look as follows:

```
\startmode[C5-envelope]
   \setuppapersize
    [C5,landscape][C5,landscape]
   \definelayer
    [Envelope]
    [width=\paperwidth,height=\paperheight]
\stopmode
```

In the preamble you can enable the desired mode with

```
\enablemode[DL-envelope]
```

Of course the information placed on the layer will have different distances from the edges in each case. To solve this problem we set up the above mentioned `\setlayer`-block for each of the defined envelopes. In order to let ConTEXt know which of the definitions to use, we test on the enabled mode.

```
\doifmode[DL-envelope]{...\setlayer-block...}
```

In case of multiple definitions for different envelope sizes and different setups for the sender's address, it is worthwhile to put the static part of the information into a separate file. This file I usually call “layout.tex”. – In the file, where the actual typeset-information is stored, I have a list of all the modes which can be enabled. All but one of those rows are commented out.

```
\enablemode[DL-envelope]
% \enablemode[C5-envelope]
\environment layout
```

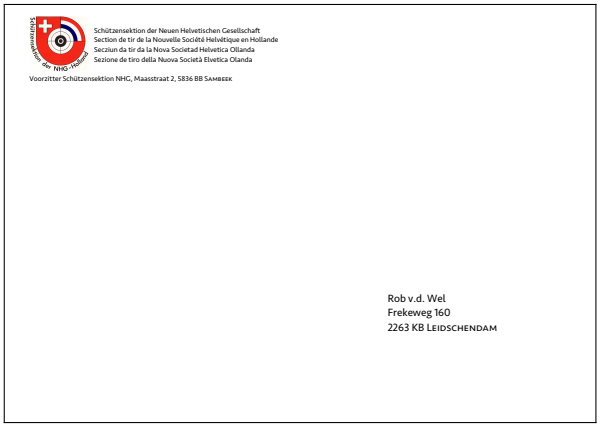


Figure 5. C5 envelope

Conclusion

Printing an envelope is not an easy task. Thanks to the flexible possibilities in ConTEXt with layers, this becomes an easy game. Defining the different sizes and layouts is always worth the invested time, because you will have perfect prints and most important always a consistent layout. Last but not least the mail service will appreciate a layout complying with their guidelines.

Willi Egger
w.egger@boede.nl

CD and DVD Covers in ConTEXt

Abstract
Production of CD and DVD covers in several variations using ConTEXt.

Keywords
CD, DVD, jewelcase

Introduction

In the fall of 2005 there appeared in the NTG MAPS an article by Dennis van Dok about his code for typesetting a jewelcase cover[1]. This article has inspired me to both adapt it to ConTEXt and elaborate on it, for which he kindly gave his permission. The result is the *hvdm-cas* module that can typeset covers for CDs, DVDs and jewelcase boxes. These covers can be customized in a great number of ways.

Typesetting covers

The setup of the parameters governing the production of covers is effected with macro `\setupcds[.1.]` The settable parameters will be presented gradually in the text and are summarized in table 1. One can print their values in the log by calling macro `\showcdcaseparameters`.

The sole macro for the production of covers is `\startcase..\stopcase`. It is used as follows:

```
\startcase[.1.][.2.]
  <contents of left page>
  \page
  <contents of right page>
\stopcase
```

The optional parameters in `[.1.]` are applied to both the left and right page of the cover. In addition the parameters in `[.2.]` enable one to override these in the right page. Those parameters are the same as for `\setupcds`. The `\page` separates the left and righthand page; its presence its mandatory, even where the right page has no content as is the case for the backside of the jewelcase.

Types of covers

There are four types of covers, three for CDs and one for DVDs. Selection of `[type=cd]` produces the top of figure 1. This one can be folded over and placed inside the front of a jewelcase. Another possibility is to fold the cover, put a CD or DVD into it and store that in one of those cheap transparent plastic sleeves. In the middle is the jewelcase cover with `[type=jewel]` having left and right extensions. At the bottom is a very thin cover made by `[type=slim]`, to be used with very small CD-cases. These are only about 3.8mm thick and have limited space for a title on the spine, which extends a bit to the left side.

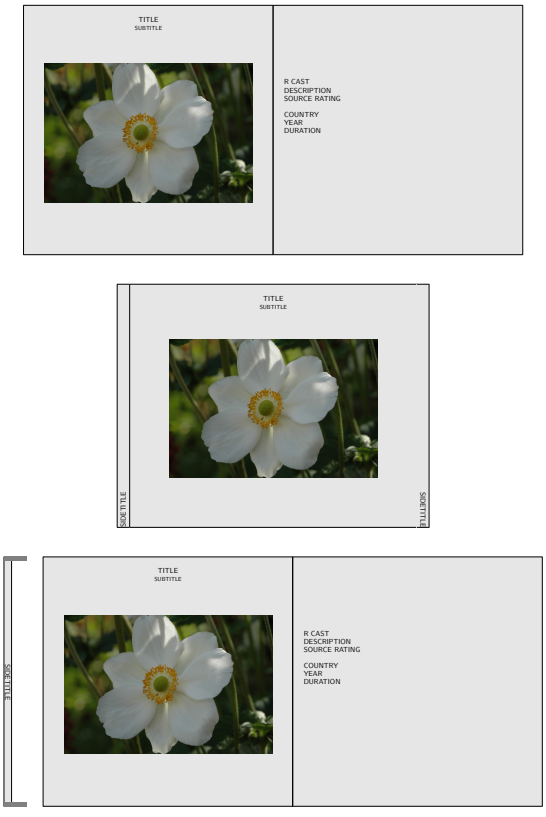


Figure 1. Cover types for CD

Covers for DVDs are produced with `[type=dvd]`. On the top of figure 2 stands the usual DVD case with a spine of 14mm (`[spine=big]`). Those with a lot of DVDs might prefer the smaller variant, having a spine of only 7mm (`[spine=small]`). One can store twice as many of the latter in the same space. The width

of the spine can also be chosen at will by specifying a dimension; for example `[spine=8mm]`. This also works for the CD-types.

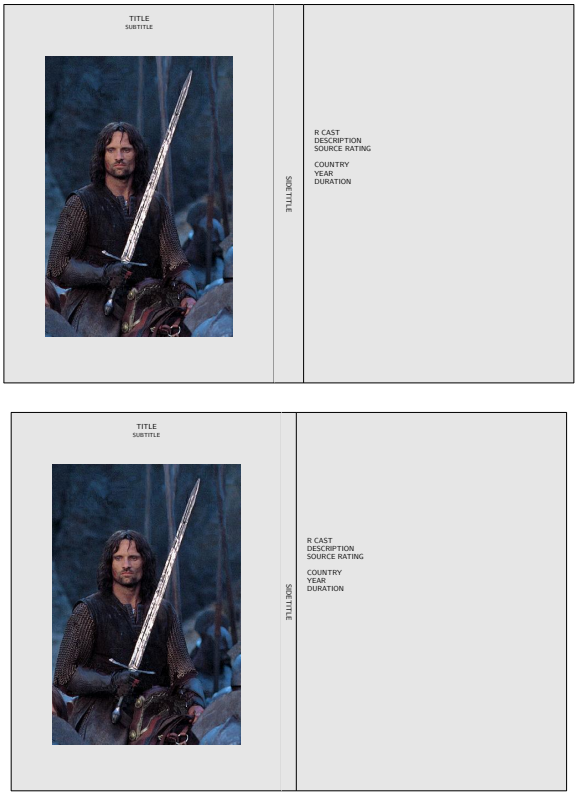


Figure 2. Cover types for DVD

Output states

The printable versions of CD and DVD covers (`[state=final]`) are rotated by 90 degrees in order to fit them on the paper as is shown in figure 3. The paper is layed out in portrait format and the cover centered on it. Note that `[cutmarks=color]` has been used here; it is taken straight from `\setuplayout` (values are `off`, `on`, `color`). The other state value `[state=draft]` will put the cover in landscape format in the same orientation as in figures 1 and 2. That way the text is easier to read, of course. Additionally the frame lines can be given another color in draft mode, preferably one that is well visible.

Printing also requires a setup of the papersize. This is effected by setting parameter `[output=page]`, which is the default. The other value is `[output=box]`. Choosing `box` suppresses the output and leaves the result available in a TEX box register. Calling `\getcdcase` copies the contents of this box into the running input stream. Putting that box in standard

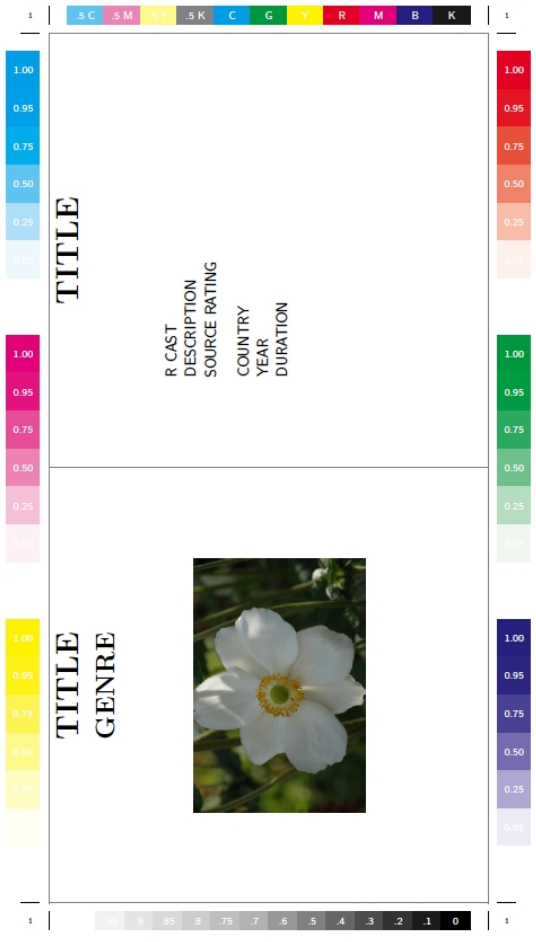


Figure 3. Final printable CD cover

ConTEXt macros like `scale` and `rotate` enables one to manipulate it further; in fact the illustrations in this article have been made that way.

Positioning of contents

Two parameters govern the size of the contents area on the coverpages. These are `offset` and `margin`, parameters that take a dimension as their value. With an offset of zero the contents is placed tightly within the enclosing frame, as can be seen in the top of figure 4. A positive offset shrinks the inner frame on all sides by that amount; its effect is shown in the left page of the bottom illustration. On the right side a positive `margin` has been added, narrowing the typing area and enlarging the margins left and right. As might be expected, a negative value will enlarge the typing space by diminishing the respective margin.[2]

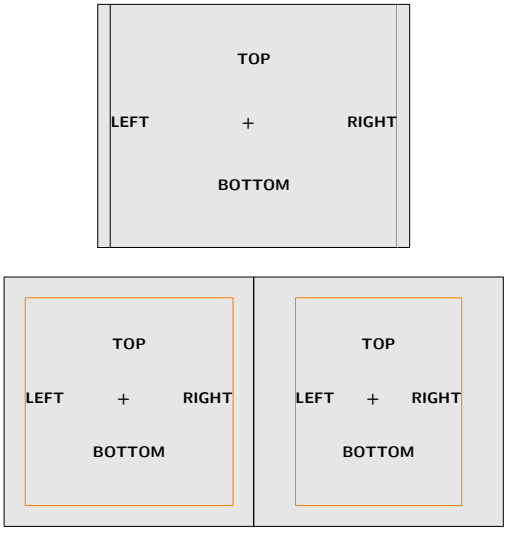


Figure 4. Offset and margin parameters

The contents can be positioned vertically with `[location=middle]` (the default), `top`, `bottom` or `none`. Figure 5 illustrates the middle and top placement. Likewise the sidetitle has its vertical position governed by `sidelocation`, having the same set of values; figure 5 shows the `top` option whereas figure 2 has the sidetitle in the middle.

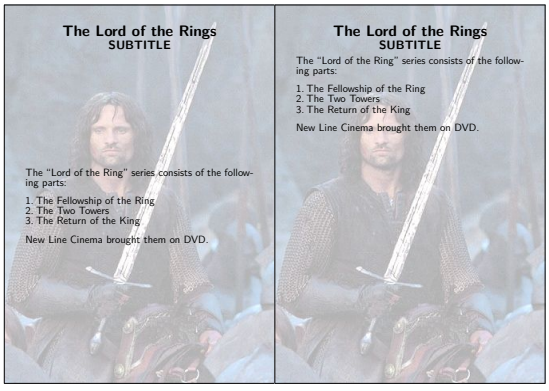


Figure 5. Vertical content positioning

Titles, pictures and overlays

Several macros facilitate the placement of titles, subtitle, pictures and overlays. When the value of `title` and/or `subtitle` parameters has been set, these are automatically placed at the top of the contents. By default then, they appear on both pages of the covers. By judicious application of `[title=<text>]` or `[title=]` on the first and/or second parameter of `\startcase[][]` one can let these appear and disappear at will.

Pictures may be placed with `\casepicture`. Its first (optional) argument governs the horizontal position. The permissable values are `[left]` and `[right]`, but if this parameter is left empty the picture will be centered. The second (optional) argument is transferred to the second parameter of the `\externalfigure[][]` with which the picture is placed. The third argument designates the picture, either a filename or a picture reference from `\useexternalfigure`. The pictures in figures 1 and 2 were placed that way.

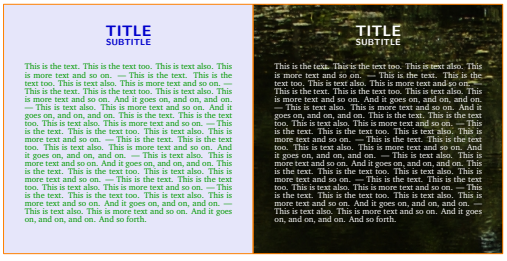


Figure 6. Title, picture, overlay, color

Coloring the background of frames in ConTEXt, according to its documentation, is governed by the `background` parameter, that can take on the values `screen`, `none`, `color`, `foreground` and `name`; `[background=color]` is the default in this module.[3] A specific color for the background is then set through `[backgroundcolor=color]`. By this the left page of figure 6 has received the value `lavender`. On the right side of that figure an overlay was set with `[overlay=picture]` in the second argument of `\startcase`. Note that this picture will fill the whole page, so that it will appear distorted if the dimensions differ from those of the cover page.

The style and color of the elements can be customized too. For example, on the left side in figure 6 the title and subtitle are blue, while the text is green. On the right side all colors are white in order to make them visible against the dark overlay. The parameter `style` is used for the text, which is the default for the other elements. The settable parameters are `[style=fontcommand]`, `titlestyle`, `subtitlestyle`, `sidetitlestyle`, `[framecolor=color]`, `foregroundcolor`, `titlecolor` and `sidetitlecolor`. In this figure the color of the frame was changed to orange. Use `[frame=off]` to remove the frame.

The background of the inner frame (made visible in figure 4) can be given a color different from the outer one with `[innerbackgroundcolor=color]`. Something similar applies to the placement of overlays. Set `[inneroverlay=on]` and an overlay fills the inner frame instead of the whole of the coverpage. These possibilities are illustrated in figure 7.

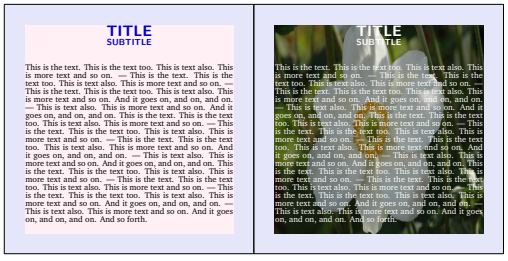


Figure 7. Background variations

Formatting content

Content can be formatted in three variations: `[format=none]`, `columns` en `packed`. The typesetting of a coverpage takes place within a `\framed` and the formatting of the content on the inside depends solely on the user input. With format `none` nothing special is done. In figure 6 this is the option chosen.

The second possibility is typesetting in columns. The ConTEXt construct used here is `\startsimplecolumns..\stopsimplecolumns`. The reason for not using the more complete `column` or `columnset` implementations is that these do not work inside `\framed`. The cover of figure 9 is typeset with the `columns` option while figure 10 has the `packed` option. In the latter the content space is filled as much as possible. This is especially useful when for example a large number of mp3's has been burned on the CD. With this option a great number of titles can be put on one cover page. For more information on the ins and outs of the packed format one is referred to the article by Dennis van Dok.

```
\track \artist{Bassano} Frais et Gaillard, 1591...(3:06).
\track \artist{Bovicelli} Io son ferito, 1594...(6:36).
\track \artist{De Cabez\'on} Un gay bergier, 1578...(2:38).
\track \artist{Bovicelli} Anchor che co'l partire, 1594...(3:54).
\track \artist{Ortiz} Recercada segonda, 1553...(3:05).
\track \artist{Ortiz} Recercada tercera, 1553...(2:33).
\track \artist{Coelho} Susanne un jour, 1620...(4:50).
\track \artist{Taeggio} Pulchra es, amica mea, 1620...(5:35).
\track \artist{Bassano} Susanne un jour, 1591...(3:33).
\track \artist{Bovicelli} Angelus ad pastores ait, 1594...(4:32).
\track \artist{De Cabez\'on} Pour un plaisir, 1578...(2:04).
\track \artist{Rogniono} Un gay bergier, 1592...(4:17).
\track \artist{Salaverde} Susanne un jour, 1638...(6:53).
\track \artist{Luzzaschi} Aura soave, 1601...(2:53).
\track \artist{Luzzaschi} O Primavera, 1601...(2:53).
\track \artist{Bassano} Anchor che co'l partire, 1591...(3:22).
\track \artist{Bassano} La bella netta ignuda, 1591...(5:13).
\track \artist{Bassano} Ricercata prima, 1585...(3:28).
\track \artist{Bassano} Tirsi morir volea, 1591...(5:36).
\track \artist{Bassano} Un gay bergier, 1591...(2:36).
\blanktrack
\track[nonumber] Total time...(79:54).
```

Figure 8. Input of column format example

Finally there are some macro's that facilitate the typesetting of music tracks. An example of their use is found figures 9 and 10. The first of these also illustrates the use of `casebefore` and `caseafter`. The title is separated from the body by a horizontal rule placed with `[casebefore=\hrule]`. The command given on the `casebefore` is executed between the typesetting of title-subtitle and the start of the contents. Likewise `[caseafter=\hrule]` is executed just behind the contents.

Figure 9. Example in column format

Figure 10. Example in packed format

Each music track is described by a `\track` macro. Its format is

```
\track[number] <description>...(duration).
```

The three dots ... separate the description from the duration. Within the description the `\artist{text}` macro switches the font to the `\artiststyle`. The parentheses around the duration can be substituted for something else through the parameters `timebefore`

and timeafter. When the duration between the parentheses is left empty, both the duration and the filling dots are omitted from the output.

Tracknumbers can be preceded and followed by commands/text through numberbefore and numberafter. Their number of digits is set with numbersize and a prefill with leading zeroes can be chosen with [zeroes=yes]. Font settings are provided for the track, tracknumber, artist and duration entries. Color settings are available for track, tracknumber and duration.

Tracks are numbered consecutively and in figure 9 their width is set by [numbersize=2]. The last entry, giving the total duration of the CD, had its tracknumber suppressed through value nonumber.

In order to fit things on a page, one can fiddle with the space between the lines, except for the packed format where other rules reign. Set for example [interlinespace=2ex] on the first cover page and reset it on the second one with interlinespace=. One can also set the standard values small, medium and big. The altered linespacing goes in effect just after the titles have been typeset, but before the execution of casebefore.

In figure 10 the same tracks are typeset, but now in packed format. The two series of tracks were delineated here with "\title Side A." and "\title Side B." All text between \title and the final dot is taken as a title and typeset between —'s. This option is useful when combining several mp3-compressed albums on one CD. The example also demonstrates the [zeroes=yes] option in order to make every tracknumber the same size with preceding zeroes.

The macro \blanktrack sets an empty track, allthough this will be invisible in the packed format. It is especially usefull as a filler to even out columns in the columns-format.[4]

Accessing variables

The standard values in the module can be redefined, of course. Their current values are given in table 2.

Inside a page one can access the internal dimensions. These are \dimens which receive their value during typesetting; thus these values have no meaning outside the page content. The following are available, their names speak for themselves; see figure 4 for an illustration of the frame parameters.

- \cdframewidth
- \cdframeheight
- \cdinnerframewidth
- \cdinnerframeheight
- \cdspinewidth
- \cdpagewidth

1. D. van Dok, *Jewel case listings for mp3 cdroms*, NTG-MAPS 33 (2005).
2. For clarity colored framelines on the innerframe were selected with [innerframe=on].
3. Do not forget to activate colors with \setupcolors[state=start].
4. Whereas \startcolumns can be made to advance to the next column with \column, \startsimplecolumns does not respond to it. That leaves us with evening out columns by adding blank tracks.

Hans van der Meer
H.vanderMeer@uva.nl

type	cd,slim,jewel,dvd	type of cover
spine	none,big,small,dimension	refined type
state	draft,final	form of output
output	page,box	print or boxregister
cutmarks	on,off,color	not for box
offset	0pt,dimension	offset of contents
margin	0pt,dimension	additional margin
location	middle,top,bottom,none	vertical position contents
sidelocation	middle,top,bottom,none	vertical position sidetitle
title	\empty,string	title text
subtitle	\empty,string	subtitle text
sidetitle	title,string	sidetitle text
format	none,columns,packed	format coverpage
overlay	none,picture	overlayed picture
inneroverlay	off,on	overlayed picture inner
casebefore	\empty,command	before content
caseafter	\empty,command	after content
n	2,number	number of columns
distance	5mm,dimension	column distance
numbersize	2,number	digits of track number
zeroes	no,yes	preceding zeroes
numberbefore	\empty,command	before tracknumber
numberafter	\empty,command	after tracknumber
timebefore	(,command	before duration
timeafter	),command	after duration
frame	on,off	show coverframe
framerule	.4pt,dimension	frame rulesize
innerframe	off,on	show inner coverframe
innerframerule	.4pt,dimension	innerframe rulesize
interlinespace	empty,dimension	change interlinespace
style	\ss,fontcommand	font of contents
titlestyle	style,fontcommand	font of title
subtitlestyle	style,fontcommand	font of subtitle
sidetitlestyle	style,fontcommand	font of sidetitle
trackstyle	style,fontcommand	font of track data
numberstyle	style,fontcommand	font of track data
timestyle	style,fontcommand	font of track length
artiststyle	style,fontcommand	font of artist data
foregroundcolor	black,color	color of page content
framecolor	cdgray,color	color frame
innerframecolor	cdgray,color	color innerframe
titlecolor	foregroundcolor,color	color titles
sidetitlecolor	foregroundcolor,color	color sidetitle
trackcolor	foregroundcolor,color	color track data
numbercolor	foregroundcolor,color	color track number
timecolor	foregroundcolor,color	color track length
background	color,none	background setting
backgroundcolor	white,color	frame background
innerbackgroundcolor	backgroundcolor,color	innerframe background
sidetitlebackgroundcolor	backgroundcolor,color	sidetitle background

Table 1. Parameters on `\startcase[.1.][.2.]`

`\caseheightcd`	120mm	cd: height
`\casewidthdvd`	120mm	cd: width
`\casesideslim`	3.8mm	slim cd: width spine
`\casesideslimspace`	15mm	slim cd: offset spine
`\snipruleheight`	2mm	slim cd: height cutout region
`\sniprulewidth`	11mm	slim cd: width cutout region
`\jewelcaseheight`	117mm	jewelcase: height
`\jewelcasewidth`	138mm	jewelcase: width
`\jewelcaseside`	6mm	jewelcase: sides
`\caseheightdvd`	182mm	dvd: height
`\casewidthdvd`	130mm	dvd: height
`\casesidebigdvd`	14mm	dvd: width big spine
`\casesidesmalldvd`	7mm	dvd: width small spine
`\grayrulecolor`	cdgray	final: color of greys
`\nonfinalgrayrulecolor`	darkred	draft: color of grays
`\grayrulesize`	.4pt	thickness gray rules

Table 2. Default definition values

Punk from Metafont to MetaPost

Abstract
To make Knuth's punk font usable with ConTEXt MKIV, it had to be converted from Metafont to MetaPost input. This article highlights the most important changes that had to be made in the conversion process.

Introduction

Donald Knuth's punk font is available from CTAN and in most TEX distributions, such as TEXLive. The TEXLive description has this to say about it:

> "A response to the assertion in a lecture that 'typography tends to lag behind other stylistic changes by about 10 years'. Knuth felt it was (in 1988) time to design a replacement for his designs of the 1970s, and came up with this font! The fonts are distributed as Metafont source. The package offers LaTEX support by Rohit Grover, from an original by Sebastian Rahtz, which is slightly odd in claiming that the fonts are T1-encoded. A (possibly) more rational support package is to be found in punk-latex."

Elsewhere in this Maps 37, you can read about the rather special characteristics of the punk font, and about the stepts needed to make it usable in the latest version of mplib–enabled ConTEXt. In an effort to reduce the overall noise level on these pages, the current article will not show you what the font looks like at all. There is enough of that in the two other articles.

As said already, the original font is based on Metafont. For use with mplib, we wanted a version that could be processed repeatedly by a single mplib instance. A bit of reorganisation was needed.

Punk in Metafont

The original distribution contains about a dozen Metafont input files. The content of the Metafont files is explained below, but we were only interested in the 10 point upright font, so we will ignore files like `punksl20.mf` (that generates a 20 point slanted version of the font).

punk10.mf

This is the parameter file for the 10 point font. It contains ten parameter assignments and then inputs the `punk.mf` file.

```
% 10-point PUNK font
designsize:=10pt#; font_identifier:="PUNK";
ht#:=7pt#;      % height of characters
u#:=1/4pt#;     % unit width
s#:=1.2pt#;     % extra sidebar
px#:=.6pt#;     % horizontal thickness of pen
py#:=.5pt#;     % vertical thickness of pen
dot#:=1.3pt#;   % diameter of dots
dev#:=.3pt#;    % standard deviation of punk
                % points
slant:=0;       % obliqueness
seed:=sqrt2;    % seed for random number
                % generator
input punk
bye
```

punk.mf

This is a typical Metafont macro file. It defines a few macros and sets up various drawing parameters for the characters.

```
% Font inspired by Gerard and Marjan Unger's
% lectures, Feb 1985
mode_setup;

randomseed:=seed;

define_pixels(u,dev);
define_blacker_pixels(px,py,dot);
define_whole_pixels(s);
xoffset:=s;
pickup pencircle xscaled px yscaled py;
punk_pen:=savepen;
pickup pencircle scaled dot; def_pen_path_;
path dot_pen_path;
dot_pen_path:=currentpen_path;
currenttransform:=identity slanted slant
                  yscaled aspect_ratio;

def beginpunkchar(expr c,n,h,v) =
  % code $c$; width is $n$ units
  hdev:=h*dev; vdev:=v*dev;
  % modify horizontal and
  %   vertical amounts of deviation
  beginchar(c,n*u#,ht#,0);
  italcorr ht#*slant;
```

```
  pickup punk_pen enddef;
  extra_endchar:=extra_endchar
      & "w:=w+2s;charwd:=charwd+2s#";

def ^ = transformed currenttransform enddef;

def makebox(text rule) =
 for y=0,h:
   rule((-s,y)^,(w-s,y)^); % horizontals
 endfor
 for x=-s,0,w-2s,w-s:
   rule((x,0)^,(x,h)^);  % verticals
 endfor
 enddef;
rulepen:=pensquare;

vardef pp expr z =
  z+(hdev*normaldeviate,vdev*normaldeviate)
enddef;

def pd expr z =   % {\bf drawdot}
  addto_currentpicture contour
    dot_pen_path shifted z.t_
        withpen penspeck
enddef;

input punkl % uppercase letters
input punkae % uppercase \AE, \OE, \O
input punkg % uppercase greek
input punkp % punctuation
input punkd % digits
input punka % accents

ht#:=.6ht#; dev:=.7dev;
input punksl % special lowercase
extra_beginchar:=extra_beginchar
  & "charcode:=charcode+32;";
input punkl % lowercase letters
extra_beginchar:=extra_beginchar
  & "charcode:=charcode-35;";
input punkae % lowercase \ae, \oe, \o

font_slant:=slant;
font_quad:=18u#+2s#;
font_normal_space:=9u#+2s#;
font_normal_stretch:=6u#;
font_normal_shrink:=4u#;
font_x_height:=ht#;
font_coding_scheme:=
  "TeX text without f-ligatures";
bye
```

Note that `punkl.mf` and `punkae.mf` are loaded twice, after some redefinitions have taken place. The combined effect of

```
ht#:=.6ht#; dev:=.7dev;
```

and

```
extra_beginchar:=extra_beginchar
 & "charcode:=charcode+32;";
```

is that the drawing routines for the uppercase characters (like 'P', with character code 80) are reused for the lowercase characters (like 'p', with character code 112). The heights and widths are diminished, and this makes punk a 'Caps and Small Caps' font.

punkl.mf, punkae.mf, punkg.mf, punkp.mf, punkd.mf, punka.mf, punksl.mf
These contain the drawing routines for the characters and a few `ligtable` commands for the standard tex ligatures like `--` and `''`. There is not that much to see, just a bunch of definitions like this:

```
beginpunkchar("P",13,1,2);
z1=pp(2u,0); z2=pp(2u,1.1h);
z3=pp(2u,.5h); z4=pp(w,.6[y3,y2]);
pd z1; pd z3;
draw z1--z2--z4--z3; % stem and bowl
endchar;
```

Punk in MetaPost

In the MetaPost version, we wanted to have only one file because that makes handling the font a bit easier. The file's name is `punkfont.mp`, and even though there is only one file now, the initial setup is much the same.

It begins with parameter settings, like this:

```
if unknown punk_font_loaded :

    if unknown scale_factor :
        scale_factor := 1 ;
    fi ;

    boolean punk_font_loaded ;

    punk_font_loaded := true ;
    warningcheck     := 0 ;
    designsize       := 10pt#;
    font_identifier  := "Punk Nova" ;

    ht#  :=  7pt# ; % height of characters
    u#   :=1/4pt# ; % unit width
    s#   := 0     ; % extra sidebar
    px#  := .6pt# ; % horizontal pen thickness
    py#  := .5pt# ; % vertical pen thickness
```

```
    dot# :=1.3pt# ; % diameter of dots
    dev# := .3pt# ; % standard deviation of
                    % punk points

  % seed          := sqrt2 ;
```

Most if the changes above should be self–explanatory. The only things worth noting are the test and setting of the `punk_font_loaded` boolean (this prevents errors when the file is being read multiple times) and the commented out definition of `seed`. That latter change is because we wanted the font to be truly random. Knuth's original only appears to be random. In fact it always has the exact same 'randomness'.

The next bit contains the assignments and macro definitions, much like `punk.mf`:

```
proofing     := 0 ;
pt           := .1pt ;
mag          := scale_factor * 10 ;
bp_per_pixel := bpppix_ * mag ;
```

MetaPost's `mfplain` doesn't have the `mode_setup` macro, so the important settings from that are given explicitly. The trickery with `scale_factor` and `pt` is just so the resulting figures will have a usable range (in PostScript big points).

Going on:

```
define_pixels(u,dev) ;
define_blacker_pixels(px,py,dot) ;
define_whole_pixels(s) ;
xoffset := s ;

pickup pencircle xscaled px yscaled py ;
punk_pen := savepen ;
pickup pencircle scaled dot ;
path dot_pen_path ;
dot_pen_path :=tensepath makepath currentpen;

defaultcolormodel := 1 ;

def beginpunkchar(expr c,n,h,v) =
  % code $c$; width is $n$ units
  hdev := h * dev ;
  % modify horizontal amounts of deviation
  vdev := v * dev ;
  % modify vertical amounts of deviation
  beginchar(c,n*u#,ht#,0) ;
  italcorr 0 ;
  pickup punk_pen
enddef ;
```

```
extra_endchar := extra_endchar
  & "w := w+2s ; charwd := charwd+2s# ;";

extra_endchar := extra_endchar
  & "setbounds currentpicture to (0,-d)"
  & "--(w*1.2,-d)--(w*1.2,ht#)--(0,ht#)"
  & "--cycle;";

def ^ = transformed currenttransform enddef ;

def makebox(text rule) =
    for y=0, h : % horizontals
        rule((-s,y)^,(w-s,y)^) ;
    endfor
    for x=-s, 0, w-2s, w-s : % verticals
        rule((x,0)^,(x,h)^) ;
    endfor
enddef ;

rulepen := pensquare ;

vardef pp expr z =
    z + (hdev * normaldeviate,
         vdev * normaldeviate)
enddef;

def pd expr z = % {\bf drawdot}
    addto currentpicture
       contour dot_pen_path
       shifted z.t_ withpen penspeck
enddef;
```

This is all pretty much the same as in Metafont. The trick with the multiple loading doesn't work because there is only the one file, but we did not want to manually adjust the drawing macros within de `beginpunkchar` commands. That is why the following definitions were added:

```
   def initialize_punk_upper =
       ht# := 7pt# ; dev# := .3pt# ;
   enddef ;
   def initialize_punk_lower =
       sht# := ht#; sdev := dev;
       ht# := .6ht# ; dev := .7dev ;
   enddef ;
   def revert_punk_lower =
       ht# := sht#; dev := sdev;
   enddef ;

fi ;
```

The `fi` ends the boolean test that was started at the top of the file, everything below this point can be safely re-interpreted.

The rest of the file consists of a few calls to these three macros and a whole bunch of character definitions. It starts like this:

```
initialize_punk_upper ;

beginpunkchar("A",13,1,2);
  z1=pp(1.5u,0); z2=(.5w,1.1h);
  z3=pp(w-1.5u,0);
  pd z1; pd z3;
  draw z1--z2--z3; % left and right diagonals
  z4=pp .3[z1,z2];
  z5=pp .3[z3,z2];
  pd z4; pd z5;
  draw z4--z5; % crossbar
endchar;
```

The MetaPost version of the font does not have any `ligtable` commands; the ligatures are automatically generated by ConTEXt. This is possible because the loaded font uses an (incomplete) Unicode encoding.

For example, we have:

```
beginpunkchar(8221,9,.3,.5);
  % '' quotedblright
  z1=pp(.5w-.5u,h); z2=pp(u,.6h);
  z3=pp(w-u,.95h); pd z1; pd z3;
  draw z1--z2--z3;  % stroke
endchar;
```

Incidentally, this is why the `warningcheck:=0;` above was needed. Without it, MetaPost would have complained about `Number too large`.

The last thing worth mentioning is that the original font was using 7-bit TEX roman encoding, which doesn't have a full ASCII set. We have added the missing definitions: underscore, caret, left brace, right brace, backslash, and the straight quote and double quote.

Taco Hoekwater & Hans Hagen

How to convince Don and Hermann to use LuaTEX

Odds are pretty low that Don Knuth will use LuaTEX for typesetting the next update of his opus magnum, and odds are even lower that Hermann Zapf will use mfub for Melior Nova. However, the next example of combining Metafont and TEX may draw their interest in this new variant: MetaTEX.

The font used here is called 'Punk' and is designed by Donald Knuth. There is a note in the file that says: 'Font inspired by Gerard and Marjan Unger's lectures, February 1985'. If you didn't notice it yet: punk is a random font.

You may wonder why we started looking into this masterpiece of font design. Well, there are a few reasons:

- We always liked this font, but after the rise of outline fonts it was not a natural candidate for using in documents. Fun is always a good motive.
- For many years we have been suggesting that special glyphs and/or aspects of typesetting could be realized by runtime generation of graphics, and we need this testbed for the Oriental TEX project: Idris needs stretchable inter-glyph connections.
- Taco likes using tricky MetaPost backgrounds for his presentations that demonstrate this programming language.
- Hartmut loves to tweak the backend and runtime font generation will demand some extensions to the font inclusion and literal handlers.
- Because Hans attends many TEX confedences together with Volker Schaa, he has promised him to avoid repeating talk and presentation layouts, and so a new presentation style was needed.

To this we can add an already mentioned motivation: convince Don and Hermann to use LuaTEX ... who knows. And, if that fails, maybe they can team up for an extensions to this font: more style variants, proper math and the full range of Unicode glyphs.

The punk font is written in Metafont and there are multiple sources. These are merged into one file which is to be processed using the mfplain format. Definitions of chadacters in this font look like:

```
beginpunkchar("A",18.1,12);
    z1 = pp(1.5u,0) ; z2 = (.5w,1.1h) ; z3 = pp(w-1.5u,0) ;
    pd z1 ; pd z3 ; draw z1 ~ z2 ~ z3 ;
    z4 = pp 3/2[z1,z2] ; z5 = pp 3/2[z3,z2] ;
    pd z4 ; pd z5 ; draw z4 ~ z5;
endchar ;
```

When TEX needs a font, i.e. when we have something like this:

```
\font\somefont=whatever at 12pt
```

In ConTEXt control is delegated to a font loader written in Lua that is hooked into TEX. This loader interprets the name and if needed filters the specification from it. Think of this:

```
\font\somefont=whatever*smallcaps at 16pt
```

This means: load font whatever and enable the smallcaps features. Howeved this mechanism is mostly geared towadds Type1 and OpenType ponts. But punk is neither: it's a Metafont, and we need to treat it as such. We will use LuaTEX's powerful virtual font technology because that way we can smuggle the proper shapes in the final file. And ... no bitmaps and no funny encoding.

In ConTEXt MkIV thede is a preliminary virtual font definition mechanism. There is no advanced TEX interface yet so we need to do it in Lua. Fortunately we do have access to this from the font mechanism:

```
\font\somefont=mypunk@punk at 20pt
```

This is a rather valid directive to create a font that internally will be called mypunk. For this the virtual font creation command punk will be used, and in a moment we will see what this triggers.

Of course, users will never see such low level definitions. They will use proper typescript, which set up a whole font system. For instance, in this document we use:

```
\switchtobodyfont[punk,12pt]
\baselineskip=14pt
```

Now, using punk in inself is not that much of a challenge, but how about using multiple instances of this font and then typeset the text chosing variants of a glyph at random. Of course this will have some trade-off in terms of runtime. In this document we use punk as the bodyfont and therefore it comes in sevedal sizes. On Hans's laptop generating the glyphs takes a while:

```
7500 glyphs, 12.887 seconds runtime, 581 glyphs/second
```

Fortunately MkIV provides a caching mechanism so once the fonts are generated, a next run will be more comfortable. This time we get reported:

```
0.187 seconds, 60 instances, 320.856 instances/second
```

which is not that bad for loading 60 files of 5 megabytes pdf literals each. The reason why the files are large is that although these glyphs look simple, in fact they are rather complex: each glyph at least one paths and several knots, and since a special pen is used, conversion results in a larger than normal description of a shape.

Since we use the standard converter from MetaPost to pdf, we can gain some generation time by using a dedicated converter for glyphs. Eventually the mplib library may even provide a proper charstring generator so we can construct real fonts at runtime.

So, how does this work behind the screens? Because we can use some of the mechanisms already present in ConTEXt it is not even that complex.

- The punk directive tells ConTEXt to create a virtual font. Such a font can be made out of real fonts; we use this for instance in the font feature combine, where we add virtually composed characters that are missing by combining characters present. However, here we have no real font.
- And so this virtual font is not built on top of an existing font, but spawns a mplib process that will build the font, unless it is present in the cache on disk. The shapes are converted to pdf literals and for each character a proper definition table is made.
- In total 10 such fonts are made, but only one is returned to the font callback that asked us to provide the font. The list of the alternatives is stored in the Lua table that represents the font and kept at the Lua end. So, for each size used, a unique set of 10 variants is generated.
- The randomizer operates on the node list. Instead of using a dedicated mechanism for this, we hijack one of the attribute values of the case swapper already present in MkIV. After that we can selectively turn on and off the randomizer.
- At some point TEX will hand over the node lists to ConTEXt. At that moment a lot of things can happen to the list, and one of them is a sequence of character handlers, of which the mentioned case handler is one. The handler sweeps over the node list and for each glyph node triggers a function that is bound to the attribute value.

- This function is rather trivial: it looks at the font id of the glyph, and resolves it to the font table. If that table has a list of alternatives, it will randomly choose one and assign it to the font attribute of the glyph. That's all.
- Eventually the backend routines will inject the PDF literals that were collected in the commands table of the virtual glyph.

It will not come as a surprise that our resulting file is larged than what we get when using traditional outline fonts or just one instance of punk. However, this is just an experiment, and eventually a proper font constructor will be provided, so that the glyph drawing is delegated to the font renderer. An intermediate optimization can be to use so called PDF xforms, but a properly runtime generated font is best because then we can search in the file too.

Because by now reading the punk font should go fluently we can now move on to the code. We already have a fonts namespace, which we now extend with a MetaPost sub namespace:

```
fonts.mp = fonts.mp or { }
```

We set a version number and define a cache on disk. When the number changes fonts stored in the cache will be regenerated when needed. The containers module provides the relevant function.

```
fonts.mp.version = 101
fonts.mp.cache = containers.define("fonts", "mp", fonts.mp.version, true)
```

We already have a metapost namespace, and within it we define a sub namespace:

```
metapost.characters = metapost.characters or { }
```

Now we're ready for the real action: we define a dedicated flusher that will be passed to the MetaPost converter. A next version of mplib will provide the tfm font information which gives better glyph dimensions, plus additional kerning information. All this code is defined in a closure (do ... end) which nicely hides the local variables.

```
local characters, descriptions = { }, { }
local factor, total, variants = 100, 0, 0
local l, n, w, h, d  = { }, 0, 0, 0, 0

local flusher = {
    startfigure = function(chrnum,llx,lly,urx,ury)
        l, n = { }, chrnum
        w, h, d = urx - llx, ury, -lly
        total = total + 1
    end,
    flushfigure = function(t)
        for i=1,#t do
            l[#l+1] = t[i]
        end
    end,
    stopfigure = function()
        local cd = characters.data[n]
        descriptions[n] = {
            unicode = n,
            name = cd and cd.adobename,
            width = w*100,
            height = h*100,
            depth = d*100,
        }
        characters[n] = {
            commands = {
                { "special", "pdf: " .. table.concat(l," ") },
            }
        }
    end
}
```

In the normal converter, the start and stop function do the packaging in a box. The flush function is called when literals need to be flushed. This threesome does as much as collecting glyph information in the list table. Intermediate literals are stored in the l table. Each glyph has a description and (in this case) one command that defines the virtual shape. The name is picked up from the character data table that is present in MkIV.

As told before we generate multiple instances per requested font and here is how it happens. We initialize the mfplain format and reset it afterwards. The funk definition file is adapted for multiple runs. Scaling happens here because later on the scaler has no knowledge about what is present in the commands. We use a few helpers for processing the MetaPost code and format the final font table in a way ConTEXt MkIV likes. Currently the parameters (font dimensions) are rather hard coded, but this will change when mplib can provide them.

```
FUNCTION
METAPOST.CHARACTERS.PROCESS (MPXFORMAT, NAME, INSTANCES, SCALEFACTOR)
    INPUT.STARTTIMING(METAPOST.CHARACTERS)
    SCALEFACTOR = SCALEFACTOR OR 1
    INSTANCES = INSTANCES OR 10
    LOCAL FONTNAME = FILE.STRIPSUFFIX(FILE.BASENAME(NAME))
    LOCAL HASH = FILE.ROBUSTNAME(STRING.FORMAT(
        "%S %04I %04H", FONTNAME, SCALEFACTOR*1000, INSTANCES))
    LOCAL LISTS = CONTAINERS.READ(FONTS.MP.CACHE(), HASH)
    IF NOT LISTS THEN
        INPUT.STARTTIMING(FLUSHER)
        LOCAL DATA = IO.LOADDATA(INPUT.FIND_FILE(NAME))
        METAPOST.RESET(MPXFORMAT)
        LISTS = { }
        FOR F=1,INSTANCES DO
            CHARACTERS, DESCRIPTIONS = { }
            METAPOST.PROCESS(
                MPXFORMAT,
                {
                    "RANDOMSEED := ", F*10 .. ";",
                    "SCALE_FACTOR := " .. SCALEFACTOR .. ";",
                    DATA
                },
                FALSE,
                FLUSHER
            )
            LISTS[#LISTS+1] = {
                DESIGNSIZE = 655360,
                NAME = STRING.FORMAT("%S-%03I",HASH,F),
                PARAMETERS = {
                    SLANT         =   0,
                    SPACE         =  333 * SCALEFACTOR,
                    SPACE_STRETCH =  166.5 * SCALEFACTOR,
                    SPACE_SHRINK  =  111 * SCALEFACTOR,
                    X_HEIGHT      =  431 * SCALEFACTOR,
                    QUAD          = 1000 * SCALEFACTOR,
                    EXTRA_SPACE   =   0
                },
                ["TYPE"] = "VIRTUAL",
                CHARACTERS = CHARACTERS,
                DESCRIPTIONS = DESCRIPTIONS,
            }
        END
        METAPOST.RESET(MPXFORMAT) -- SAVES MEMORY
        LISTS = CONTAINERS.WRITE(FONTS.MP.CACHE(), HASH, LISTS)
        INPUT.STOPTIMING(FLUSHER)
    END
    VARIANTS = VARIANTS + #LISTS
    INPUT.STOPTIMING(METAPOST.CHARACTERS)
    RETURN LISTS
END
```

We're not yet there. This was just a font generator that returns a list of fonts defined in a format liked by MkIV and not that far from what TEX wants back from us. Next we define the main definition function, the one that is called when the font is defined as virtual font. The special number -1000 tells the scaler to honour the designsize, which boils down to no scaling, but just copying to the final table that is passed to TEX. The define function returns an ID which we will use later.

The scaler uses the descriptions to add dimensions (and other data needed) in the characters table. This is something MkIV specific.

```
function fonts.vf.aux.combine.commands.metafont(g,v)
    local size = g.specification.size
    local data = metapost.characters.process(v[2],v[3],v[4],size/65536)
    local list, t = { }, { }
    for d=1,#data do
        t = data[d]
        t = fonts.tfm.scale(t, -1000)
        t.id = font.define(t)
        list[#list+1] = t.id
    end
    for k, v in pairs(t) do
        g[k] = v -- kind of replace, when not present, make nil
    end
    g.variants = list
end
```

We hook this into the ConTEXt font handler and from now on the Punk is recognized:

```
fonts.define.methods.install( "punk", { { "metafont", "mfplain", "punkfont.mf",
10 } } )
```

Now that we can define the font, we need to deal with the randomizer. This is optional fun. The mentioned case swappers are implemented in the cases namespace:

```
local fontdata = fonts.tfm.id

cases.actions[99] = function(current)
    local c = current.char
    local used = fontdata[current.font].variants
    if used then
        local p = math.random(1,#used)
        current.font = used[p]
        return current, true
    else
        return current, false
    end
end
```

This function is called in one of the passes over the node list. Thanks to this framework we don't need that much code. We didn't show two statistics functions. They are the reason why we keep track of the total number of glyphs defined.

This leaves us defining the interface, so here we go:

```
\def\startrandompunk{\begingroup\setcharactercasing[99]}
\def\stoprandompunk {\endgroup}
```

The set command just sets the attribute that we associated with casing (one of the many attributes). The number 99 is rather arbitrary.

If you follow the development of LuaTEX and MkIV (we do talks at conferences, keep track of the development history in mk.pdf, and report on the ConTEXt mailing list) you will have noticed that we often use somewhat extreme examples to explore and test the functionality and this is no exception. As usual it helped us to improve the code and extend our todo list. Can the previous code convince the grand wizards to start using LuaTEX? Probably not. Anyway, let's just hope that they will put the addition of funk math to their todo list. In the meantime we've already started adding missing characters:

{ ı \v } { ’ \" } { ı \" } { ’ \u } { ’ \v } { ı \~ }

Also, because we can be sure that Mojca Miklavec's first test will be if her favourite characters č, š and ž are supported, we made sure that we composed those accented characters as well. (This is accomplished by adding fonts.vf.aux.compose_characters(t) at an undisclosed location in the previous code...)

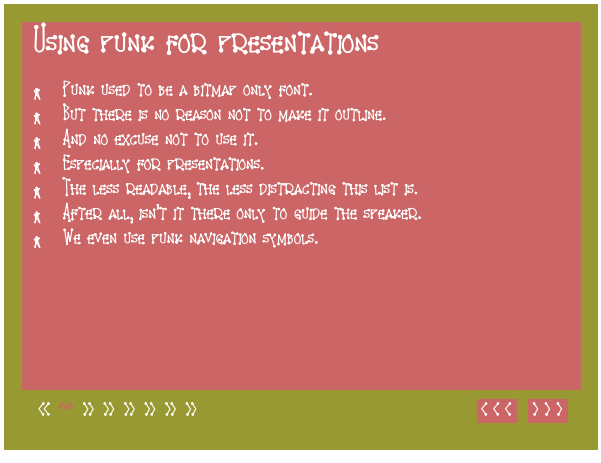
Hans Hagen
Pragma ADE

Taco Hoekwater
Elvenkind BV

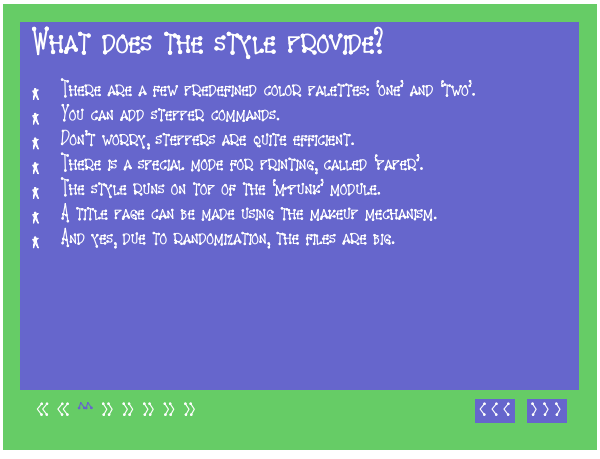
The Punk Module

As with most new tricks in ConTEXt, the punk module was first used for presentations. Such a presentation looks as follows:

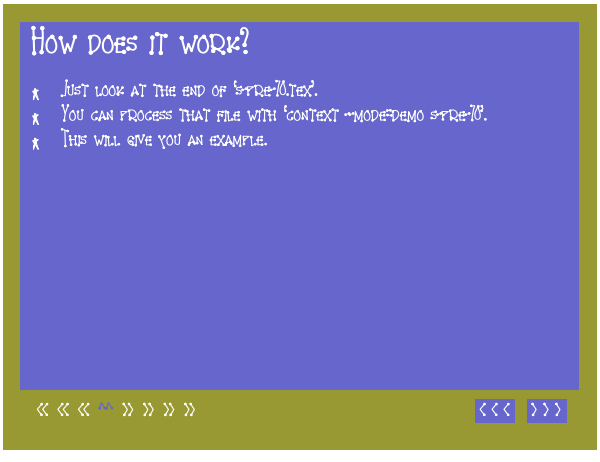
As usual we start with a title page, followed by (not too many) pages with text. As most recent presentation styles, this one works quite well with the stepper: by clicking on the page, more text (or whatever) shows up on the page by simply turning on layers.



If you don't like these colors, you can use another color palette, or define your own by simply copying and patching code from the style.



The presentation style uses the `m-punk` module, which contains code similar to the code discussed in the previous article.



The annotated style follows one the next pages.

```
% engine=luatex

%D \module
%D   [      file=s-pre-70,
%D        version=2008.04.15,
%D          title=\CONTEXT\ Style File,
%D       subtitle=Presentation Environment 70,
%D         author=Hans Hagen,
%D           date=\currentdate,
%D      copyright=PRAGMA / Hans Hagen]
%C
%C This module is part of the \CONTEXT\ macro||
%C package and is therefore copyrighted by
%C \PRAGMA. See mreadme.pdf for details.

\usemodule[punk] \usetypescript[punk]
\setupbodyfont[punk,20pt]

%D At the cost of more runtime and a larger
%D output file, we turn on randomization.
%D The instances are cached in the MkIV cache,
%D so successive runs use the same shapes.

\EnableRandomPunk

%D We use the regular screen size paper and
%D layout setup.

\setuppapersize
  [S6][S6]

\setuplayout
  [topspace=30pt,
   backspace=30pt,
   width=middle,
   height=fit,
   header=0pt,
   footer=0pt,
   bottomdistance=24pt,
   bottom=30pt,
   bottom=18pt,
   top=0pt]

\setupinterlinespace
  [top=height,
   line=1.25\bodyfontsize]

\setupcolors
  [state=start,
   textcolor=white]

\setupinteraction
  [state=start,
   %click=off,
   menu=on]

%D We predefine a few palets. Of course you can
%D define more.

\definecolor[punkblue]  [r=.4,b=.8,g=.4]
\definecolor[punkgreen] [r=.4,b=.4,g=.8]
\definecolor[punkred]   [r=.8,b=.4,g=.4]
\definecolor[punkyellow][r=.6,g=.6,b=.2]

\definepalet [punk-one]
   [textcolor=punkblue,pagecolor=punkgreen]
\definepalet [punk-two]
   [textcolor=punkred,pagecolor=punkyellow]
\definepalet [punk-three]
   [textcolor=punkblue,pagecolor=punkyellow]
\definepalet [punk-one-reverse]
   [textcolor=punkgreen,pagecolor=punkblue]
\definepalet [punk-two-reverse]
   [textcolor=punkyellow,pagecolor=punkred]
\definepalet [punk-three-reverse]
   [textcolor=punkyellow,pagecolor=punkblue]

\setuppalet[punk-one]

%D We use a few backgrounds. The hyperlink that
%D invokes the stepper is hooked into the text
%D background.

\definelayer
  [page]
  [width=\paperwidth,
   height=\paperheight]

\setupbackgrounds
  [page]
  [background={color,page},
   backgroundcolor=pagecolor,
   setups=pagestuff]

\setupbackgrounds
  [text]
  [background={color,invoke},
   backgroundoffset=12pt,
   backgroundcolor=textcolor]

%D We need different symbols for itemized
%D lists.

\definesymbol[1][\hbox{\lower1ex\hbox{*}}]
\definesymbol[2][\endash]
\definesymbol[3][\letterhash]
\definesymbol[3][>]

%D We don't want these reversed clicked areas
%D in Acrobat.
```

```
\setupinteraction
  [click=no]

%D We define a rather simple navigational panel
%D  at the bottom

\setupinteractionmenu
  [bottom]
  [color=white, % pagecolor,
   contrastcolor=white, % pagecolor,
   background=color,
   backgroundcolor=textcolor,
   frame=off,
   height=24pt,
   left=\hfill,
   middle=\hskip12pt]

\setupsubpagenumber
  [state=start]

\startinteractionmenu[bottom]
    \txt
        \interactionbar
          [alternative=d,
           symbol=yes,
           color=white,
           contrastcolor=textcolor]
    \\
    \hfilll
    \but [previouspage] < < < \\
    \but [nextpage] > > > \\
\stopinteractionmenu

%D Instead of the normal symbols we use more
%D punky ones.

\startsymbolset [punk]
    \definesymbol[previous]   [\string<\string<]
    \definesymbol[somewhere] [\string^\string^]
    \definesymbol[next]      [\string>\string>]
\stopsymbolset

\setupinteraction[symbolset=punk]

%D Because the font is rather large, we use
%D less whitespace.

\setuphead
  [chapter]
  [after={\blank[big]}]

%D Run this file with the command:
%D   \type {context --mode=demo s-pre-70}
%D in order to get an example.

\doifnotmode{demo} {\endinput}

\usemodule[pre-60] % use the stepper

\starttext

\title {Punk for dummies}

\dorecurse{10} {
  \title{Just a few dummy pages}
  \StartSteps \startitemize[packed]
  \startitemize
    \startitem bla \FlushStep \stopitem
    \startitem bla bla \FlushStep \stopitem
    \startitem bla bla bla \FlushStep \stopitem
  \stopitemize \StopSteps
}

\stoptext
```

Hans Hagen
Pragma ADE

TEXworks: lowering the barrier to entry

Abstract
A multi-platform competitor for TeXShop is described: TEXworks.

Keywords
editor, gui, interface, multi-platform, TeX front-end, TeXShop

Introduction

One of the most successful TEX interfaces in recent years has been Dick Koch's award-winning TeXShop on MacOSX. I believe a large part of its success has been due to its relative simplicity, which has invited new users to begin working with the system without baffling them with options or cluttering their screen with controls and buttons they don't understand. Experienced users may prefer environments such as iTeXMac, AUCTeX (or on other platforms, WinEDT, Kile, TeXmaker, or many others), with more advanced editing features and project management, but the simplicity of the TeXShop model has much to recommend it for the new or occasional user.

Besides the relatively "clean" interface, a second factor in TeXShop's success is probably the use of a PDF-centric workflow, with pdfTEX as the default typesetting engine. PDF is the de facto standard for fully-formatted pages; every user knows what a PDF file is and what they can do with it. Bypassing DVI reduces the apparent complexity of the overall process, and so reduces the "intimidation factor" for a newcomer. But TeXShop is built on MacOSX-specific technologies, and is available only to Mac users. There does not seem to be an equivalent tool available on other platforms; there are many TEX editors and environments, but none with this particular focus.

The TEXworks project is an effort to build a similar TEX environment ("front end") that will be available for all today's major desktop operating systems — in particular, MS Windows (XP and Vista), typical GNU/Linux distributions, and other X11-based systems, in addition to MacOSX.

To achieve this, TEXworks is based on cross-platform, open source tools and libraries. In particular, the Qt toolkit was chosen for the quality of its cross-platform user interface capabilities, with native "look and feel" for each platform being a realistic target. Qt also provides a rich application framework, facilitating the relatively rapid development of a usable product.

The standard TEXworks workflow will also be PDF-centric, using pdfTEX and XeTeX as typesetting engines and generating PDF documents as the default formatted output. Although it will still be possible to configure a processing path based on DVI, newcomers to the TEX world need not be concerned with DVI at all, but can generally treat TEX as a system that goes directly from marked-up text files to ready-to-use PDF documents.

TEXworks includes an integrated PDF viewer, based on the Poppler library, so there is no need to switch to an external program such as Acrobat, xpdf, etc., to view the typeset output. The integrated viewer also allows it to support source ↔ preview synchronization (e.g., control-click within the source text to locate the corresponding position in the PDF, and vice versa). This capability is based on the "SyncTEX" feature developed by Jérôme Laurens, now integrated into the XeTeX and pdfTEX engines in TEX Live 2008, MikTEX, and other current distributions.

Features for initial release

Figure 1 shows the current TEXworks prototype running on Windows Vista. While this is not a finished interface, it gives an impression of how the application will look. TEXworks version 0 will be an easy-to-install application offering:

1. Simple (non-intimidating — this is *not* emacs or vi!) GUI text editor with
   a. Unicode support using standard OpenType fonts
   b. multi-level undo/redo
   c. search & replace, with (optional) regular expressions as well as simple string match
   d. comment/uncomment lines, etc.
   e. TEX/LaTEX syntax coloring
   f. TEX-aware spell checker
   g. auto-completion for easy insertion of common commands
   h. templates to provide a starting point for common document types

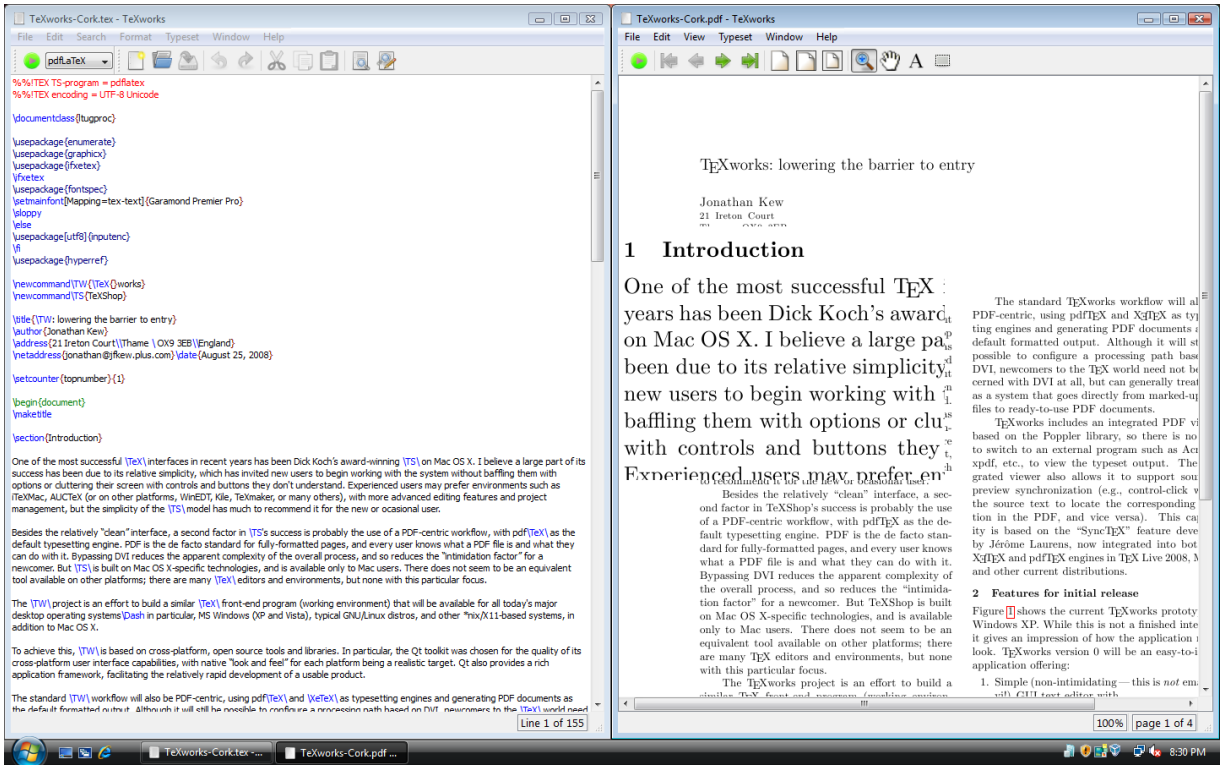


Figure 1. A recent TEXworks build running on Windows Vista: source and preview windows, with the TeXShop-style magnifying glass in use.

2. Ability to run TEX on the current document to generate PDF
   a. extensible set of TEX commands (with common commands such as pdftex, pdflatex, xelatex, context, etc. being preconfigured)
   b. also support running BibTeX, Makeindex, etc.
   c. any terminal output appears in a "console" panel of the document window; automatically hidden if no errors occur
   d. "root document" metadata so "Typeset" works from an `\included` file
3. Preview window to view the output
   a. anti-aliased PDF display
   b. automatically opens when TEX finishes
   c. auto-refresh when re-typesetting (stay at same page/view)
   d. TeXShop-like "magnifying glass" feature to examine detail in the preview
   e. one-click re-typesetting from either source or preview
   f. source ↔ preview synchronization based on Jérôme Laurens' SyncTEX technology

Current status

An early TEXworks prototype was demonstrated at the BachoTeX conference (April 2008). It became more widely available version (though still considered a prototype, not a finished release) when a version was posted in mid-July before the TUG 2008 conference. The current code is available as source (easy to build on typical GNU/Linux systems), and as precompiled binaries for Windows and MacOSX.

At this time, the application supports text editing and PDF viewer windows, and has the ability to run a typesetting job and refresh the output view, etc. There is not yet any documentation, and many potential "power user" features do not yet exist, but it is a usable tool in its current state. In addition to Windows (XP and Vista), it runs on MacOSX (see figure 2) and GNU/Linux systems (figure 3).

A few features remain to be implemented before a formal release of "version 0", including "single instance" behavior, and various options for window positioning; appropriate installer packages for MacOSX and Windows are also needed, to simplify setup.

More information may be found online via the TEXworks home page at http://texworks.org/.

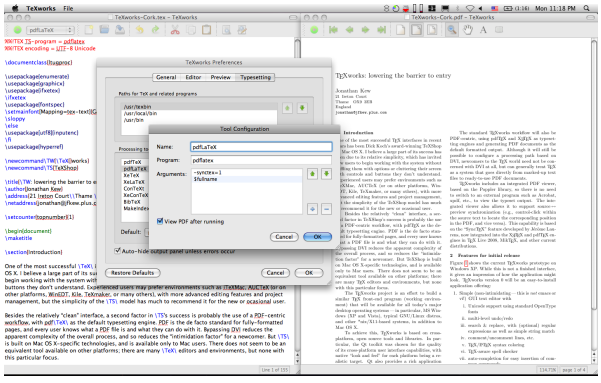


Figure 2. TEXworks running on MacOSX: using the Preferences dialog to configure a typesetting tool.

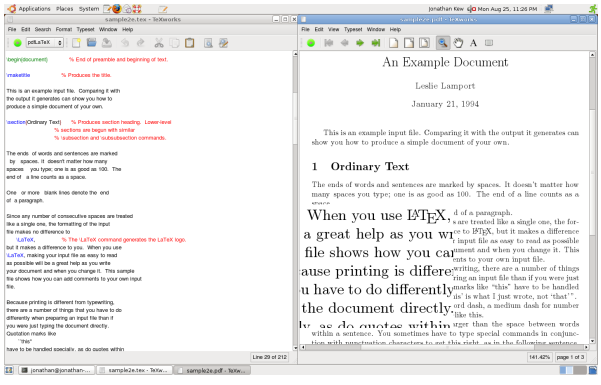


Figure 3. TEXworks running on a typical GNU/Linux system (Ubuntu).

Future plans

After the release of version 0, several major additional features are planned; some ideas high on the priority list include:

- ☐ intelligent handling of TEX errors
- ☐ assistance with graphics inclusion and format conversions
- ☐ text search and copy in the PDF preview
- ☐ support rich PDF features such as transitions, embedded media (sound, video), annotations, etc.
- ☐ customizable palettes of symbols, commands, etc.
- ☐ TEX documentation lookup/browser
- ☐ interaction with external editors and other tools
- ☐ additional support for navigating in the source, e.g., "folding" sections of text, recognizing document structure tags such as `\section`, etc.

I expect development priorities to be guided by user feedback as well as developer interest, once the initial version 0 release is available.

Invitation to participate

TEXworks is a free and open source software project, and all are welcome to participate and contribute to its development. This does not necessarily mean writing code; many other roles are equally important. Some possible ways to participate include:

- ☐ use the prototype for some real work, and give feedback on what's good, what's bad, what's broken
  - if there's a current binary download available for your platform, try that
  - get the code and try building it on your platform; provide bug reports (and fixes!) for whatever problems show up
- ☐ dig in to the code, and submit patches to implement your favorite missing features
- ☐ write on-line help, documentation and tutorials for newcomers to TEXworks and TEX
- ☐ review and enhance the command completion lists available for the integrated editor
- ☐ provide well-commented templates for various types of document
- ☐ design icons for the toolbars, etc.; TEXworks has some nice icons from Qt and the Tango project, but others are merely rough placeholders
- ☐ use the Qt Linguist tool to localize the user interface for your language
- ☐ package TEXworks appropriately for your favorite GNU/Linux or BSD distribution, or create an installer for Windows or MacOSX

There is a TEXworks mailing list for questions and discussions related to the project; for details, see: http://tug.org/mailman/listinfo/texworks/.

The TEXworks source itself is maintained in a Google Code project at http://code.google.com/p/texworks/. Resources available through this site include the Subversion source repository, precompiled binaries for Windows and MacOSX, and an issue tracker for bug reports and feature suggestions.

Thanks

The TEXworks project arose out of discussions at several recent TUG meetings, and has received generous support from TUG's TEX development fund and its contributors, and from UK-TUG. Special thanks to Karl Berry for his encouragement and support, and to Dick Koch for showing us the potential of a clean, simple TEX environment for the average user.

Jonathan Kew
jonathan@jfkew.plus.com

TEX Live 2008 and the TEX Live Manager*

Abstract
TeX Live 2008 has been released recently, and the DVDs are ready to go gold. This is the first release of TeX Live shipping the TeX Live Manager, tlmgr for short. Besides taking over some of the tasks from texconfig (which has never been available for Windows) it finally brings many new features to the TeX Live world, most importantly the option for dynamic updates.
This article will present the new TeX Live Installer, the TeX Live Manager, and at the end lists other changes in TeX Live 2008.

Important note

This article describes the status of the TEX Live Manager as it will be shipped around October 2008, and not the one on DVD. The version on the DVD works fine for local configuration tasks (which is why we felt it could be shipped), but is not sufficiently robust for reliable updates over the Internet. Users' first update will be to get the new tlmgr.

Introduction

After more than one year of development work TEX Live 2008 has been released with a complete new infrastructure [**?**]. At first these infrastructure changes were only relevant for the developers themselves, since it made life (a bit) easier and the system more consistent due to the elimination of duplicated information.

As a first user-visible change came the unification of the installer, so that all supported platforms now share the same installer. Furthermore, this installer has gotten a GUI which also is uniform across all platforms. On Unix systems the only prerequsites are a Perl installation, and for the GUI the installation of Perl/Tk. On Windows we ship a minimal Perl with the necessary modules.

The first part of this article will give an overview of the new installer.

The second user-visible change came from the addition of the TEX Live Manager, or tlmgr for short, to the list of programs. It manages an existing TEX Live installation, both packages and options. Besides performing many of the same actions as `texconfig` it has the ability to install additional packages, update and remove existing ones, make backups, search within and list all packages.

The new installer

The creation of a new TEX Live installer was necessitated by the new package infrastructure [**?**]. From a user's point of view the new installer has only one visual change, but there other significant changes. In particular:

- It is possible to install TEX Live from the Internet.

*Originally presented at the GuIT Conference 2008 in Pisa, and published in ArsTEXnica, issue 6

```
================> TeX Live installation procedure <================

==> Note: Letters/digits in <angle brackets> indicate menu items <==
==>       for commands or configurable options                   <==

 Proposed platform: Intel x86_64 with GNU/Linux

 <B> binary systems: 1 out of 15

 <S> Installation scheme (scheme-full)

 Customizing installation scheme:
   <C> standard collections
   <L> language collections
   83 collections out of 84, disk space required: 1426 MB

 <D> directories:
   TEXDIR (the main TeX directory):
     /usr/local/texlive/2008
   TEXMFLOCAL (directory for site-wide local files):
     /usr/local/texlive/texmf-local
   TEXMFSYSVAR (directory for variable and automatically generated data):
     /usr/local/texlive/2008/texmf-var
   TEXMFSYSCONFIG (directory for local config):
     /usr/local/texlive/2008/texmf-config
   TEXMFHOME (directory for user-specific files):
     ~/texmf

 <O> options:
   [ ] use letter size instead of A4 by default
   [X] create all format files
   [X] install macro/font doc tree
   [X] install macro/font source tree
   [ ] create symlinks in standard directories

 <V> set up for running from DVD

Other options:
=========================================================================
 <I> start installation to hard disk
 <H> help
 <Q> quit

Enter command:
```

Figure 1. Main menu of the text mode installer

- There is just one installer, which can run either in text mode, emulating the former install-tl.sh shell script, or in GUI mode (more or less emulating the TeX Live 2007 tlpmgui).
- The Windows installation is much closer to using the same implementation as Unix.

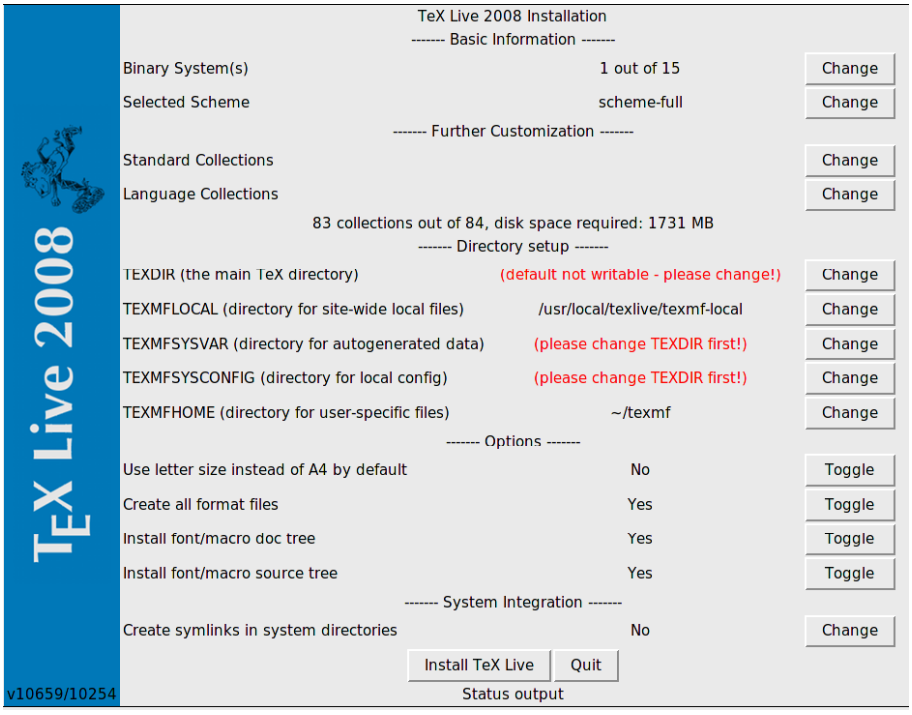


Figure 2. Main window of the GUI installer

Install TEX Live from the Internet

If you got (by the time you read this article) a TEX Live DVD you can just start the installer as usual. On Windows this will be by default the GUI installer (see below), on Unix the text mode installer.

We also ship an installation package [**?**] containing all the necessary files for an installation over the network. By default, normal installation will use a CTAN mirror, selected using the `http://mirror.ctan.org` service. (See `http://tug.org/ctan.html#sites`.)

Two installers for network downloads are provided. `install-tl-unx.tar.gz` supports Unix only. `install-tl.zip` additionally contains a small subset of Perl for Windows which is required to bootstrap the system. The latter works on all platforms supported by TEX Live. The sole reason for providing a separate package for Unix is its significantly smaller size.

In any case you can override the source from which you want to install with the command line option `-location`.

The text mode installer

If you have used TEX Live in recent years you will see no big changes in the text mode installer (see fig. 1); we tried to keep it as close as possible to the one used in former releases. One new option is at the bottom of the menu, namely *set up for running from DVD*. This is what we call *live* installation: it sets up a minimal writable environment on your computer, while all the input files and binaries remain on the DVD.

The GUI Installer

The GUI installer has nearly the same functionality as the text version; the option to set up for live installation is the only missing piece. It is written in Perl/Tk and thus should run on all platforms (on Unix Perl/Tk has to be installed).

The main window can be seen in fig. 2. It should remind you very much of the text mode installer. As with the text mode installer it allows you to change which binary systems should be installed (fig. 3), select the scheme to be installed (fig. 4), where a *scheme* is a pre-defined set of collections to be installed, and further specify in more detail which collections (a collection is a set of packages) and which language packages to install (fig. 5 and 6). You can select the installation directories for TEX Live and toggle some options, all in line with the installer of recent years and the text mode installer.

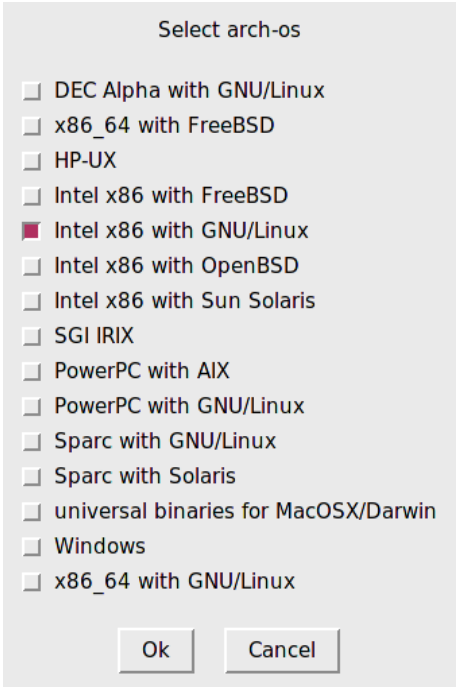


Figure 3. Binary system select window

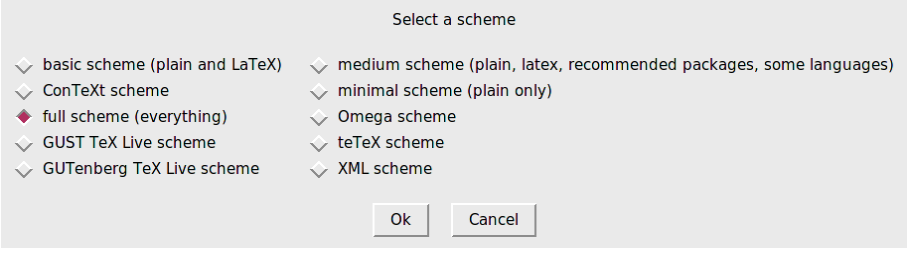


Figure 4. Scheme select window

During installation the main window's status line will indicate what is going on currently, and at the same time the program will print out to the terminal the same output as the normal installer.

For both the text mode installer and the GUI mode installer, a log file with more details is created in the installation directory as `install-tl.log`. (If you report installation problems, please send us this log file.)

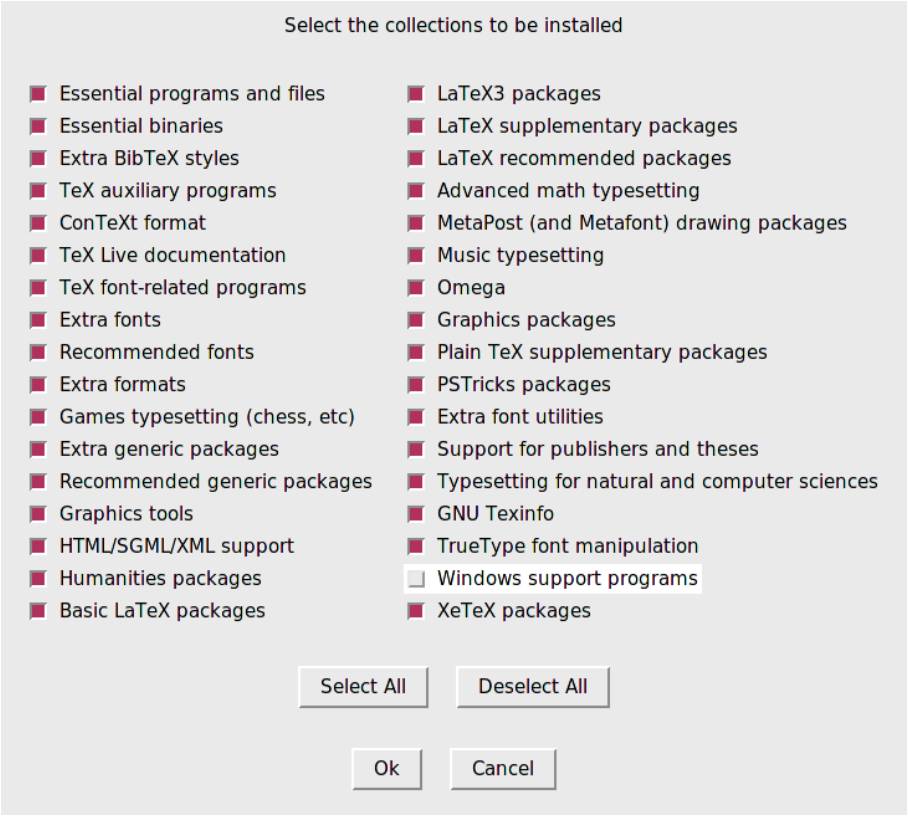


Figure 5. Collections select window

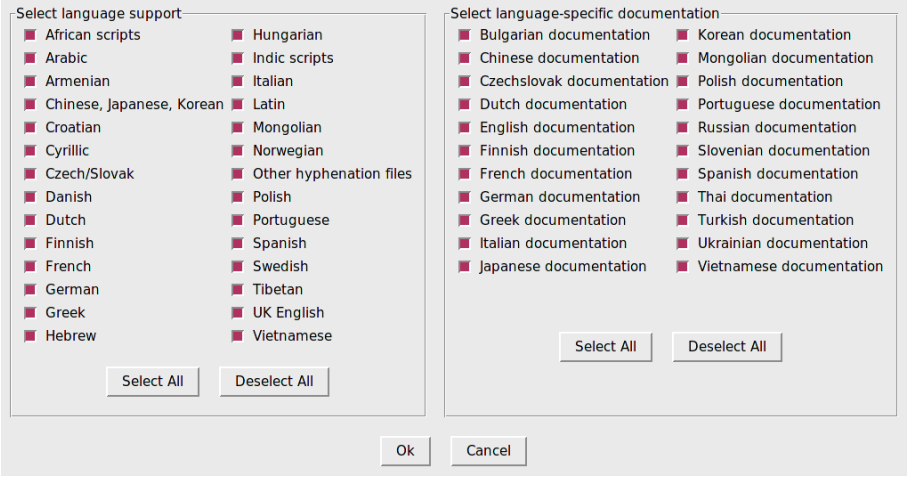


Figure 6. Language packs select window

Bringing Windows in line with Unix

TEX Live 2008 supports Windows 2000 and later. By dropping older Windows versions, there is much less need to treat Windows specially.

Under Windows 2000 and later, users have a real home directory, viz. `%USERPROFILE%`, usually `C:\Documents and Settings\`*username*.

This is now reflected in tilde expansion by Kpathsea: `~/texmf` is expanded to `%USERPROFILE%\texmf` under Windows and to `$HOME/texmf` under Unix.

It is also possible to differentiate between system settings and user settings. Happily, there is no longer any need to have a different set of `texmf` trees on Windows, or to leave out scripts such as `fmtutil-sys` and `updmap-sys`. We also have a single `texmf.cnf` used on all platforms.

The TEX Live Manager

TEX Live Manager provides a wealth of options and commands; we will explain them all here, some tersely, some in more detail. Of course we expect to add more features in the future.

The TEX Live Database

First of all, it is important to understand where all the information about installed packages and other options are saved. This is the TEX Live database, which normally can be found in `ROOT/tlpkg/texlive.tlpdb` (where `ROOT` is the destination folder you have choosen at installation time). It contains the list of all packages, all the files installed, and in addition to that collects configuration information like the default installation source, and the options you have set at installation time (e.g., whether you want A4 or letter-size paper by default).

Most of tlmgr's actions will load the local database, and many actions will also load a remote database: If you want to install a package, the TEX Live Manager will load the database of the specified installation server and checks whether this package exists.

Although we say *remote*, it is not necessarily a remote network location. If you install from DVD the default installation source will be the DVD, and tlmgr will load the database located on the DVD when needed.

General syntax of tlmgr

The general syntax of tlmgr is

```
tlmgr [opt]... action [opt]... [arg]...
```

The first set of options before the main `action` configure the general operation of tlmgr, while the second set of options are specific to the chosen `action`. We do not support mixing and reshuffling of all these options, partly for the sake of clarity, and also for programming reasons. The first set of options can contain:

- `--location` *loc* specifies the location from which packages should be installed or updated, overriding the location found in the installation's TEX Live Package Database (TLPDB).
- `--gui` starts the GUI version of tlmgr. The GUI does not support all the bells and whistles of the command-line program. It is in fact a separate program that calls the command-line version to do the actual work. The difference between this `--gui` option and the `gui` action (see below) is that given the option, tlmgr tries to open the GUI directly at the screen for the specified `action`.
- `--gui-lang` *ll* selects the language in which the GUI will appear. Normally the GUI tries to deduce your language from the environment (on Windows via the registry, on Unix via `LC_MESSAGES`). If that fails you can select a different language by giving this option a two-letter language code.

Furthermore, some standard options are supported: `--help` (also `-h`) for getting help, `-q` which supresses informational messages, and `-v` (verbose) for turning on

debugging options. With `--version` the script will show you the version of your TEX Live system and of itself.

The actions

There is a (permanently growing) list of actions, currently: `help`, `version`, `gui`, `install`, `update`, `backup`, `restore`, `remove`, `option`, `paper`, `arch`, `search`, `show`, `list`, `check`, `uninstall`, `generate`.

The general actions.

- `search` [*option*`...`] *what* Without any options, search in the list of locally installed packages for package names or descriptions matching `what`. If you give the option `--global` it also searches the remote database. This might differ in case you have not installed all of TEX Live, but only a part of it. Finally, if you specify `--file` then files are searched, and not package names.
- `show` *pkg*`...` gives you more detailed information on the listed packages. If all packages are installed locally, it does not consult the remote database.
- `list [collections|schemes]` With no argument, lists all packages available at the default install location, prefixing those already installed with "i ". With an argument lists only collections or schemes, as requested.
- `uninstall` This action will ask for confirmation, and then remove the entire installation. Don't do it or we will be sad. If you give the `--force` option, it does not even ask, but proceeds immediately with the removal.
- `check [files|collections|all]` Executes one (or all) check(s) on the consistency of the installation. For `files` it checks that all files listed in the local database are actually present, and lists those missing.The option `--use-svn` will use the `svn` command to check for the files.
- `gui` starts the GUI, as explained above at `--gui`.
- `version` is the same as `--version`.
- `help` is the same as `--help`.

The configuration actions.

- `option [show]` Shows all configuration settings currently saved in the local database. The `show` option is accepted and ignored.
- `option` *key* [*value*] Without the *value*, shows the current value of the configuration option *key*; with *value*, sets this configuration option. Currently accepted `keys` are `location` (default installation location), `formats` (create formats at installation time), `docfiles` (install documentation files), `srcfiles` (install source files). These are the options you have set at installation time and will be honoured at *later* install and upgrade actions. For example, changing the `docfiles` options from false to true will not install or remove the already present documentation files. But a subsequent update will install or remove them.
- `paper` *paper* Sets the default papersize; possible values are `a4` and `letter`.
- *program* `paper [help|`*paper*`]` This allows setting different paper sizes for the specified *program*: `xdvi`, `dvips`, `pdftex`, `dvipdfm`, `dvipdfmx`, `context`. Without any additional argument it reports the currently selected papersize. With `help`, it issues all the supported paper sizes for that program. And if you specify a paper size, it will be set as default papersize for the given program.

□ `generate` *what* This command generates one or more configuration files, as follows:

Giving `language.dat` for *what* generates the `language.dat` file which specifies the hyphenation patterns to be loaded for LaTeX-based formats. Giving `language.def` for *what* generates the `language.def` file which specifies hyphenation patterns to be loaded for `etex`-based formats. Specifying `language` for `what` generates both of these files.

Giving `fmtutil` for *what* generates the `fmtutil.cnf` file which contains the definitions of all formats available.

Giving `updmap` for *what* generates the `updmap.cfg` file which lists all the installed font map files.

For `fmtutil` and the language files, recreating is normal and both the installer and tlmgr routinely call that.

For `updmap`, however, neither the installer nor tlmgr use `generate`, because the result would be to disable all maps which have been manually installed via `updmap-sys --enable`, e.g., for proprietary or local fonts. Only the changes in the `--localcfg` file mentioned below are incorporated by `generate`.

On the other hand, if you only use the fonts and font packages within TeX Live, there is nothing wrong with using `generate updmap`. Indeed, we use it to generate the `updmap.cfg` file that is maintained in the live source repository.

If the files `language-local.dat`, `language-local.def`, `fmtutil-local.cnf`, or `updmap-local.cfg` are present under `TEXMFLOCAL` in the respective directories, their contents will be simply merged into the final files, with no error checking of any kind.

The package management actions.

□ `install` *pkg*... installs the packages given as argument. By default, installing a package also installs all of its dependencies. The following options are supported: `--no-depends` will not install dependent packages. There is also `--no-depends-at-all` which in addition disregards the tightly coupled packages architecture-specific executables; for example, `bin-bibtex` and `bin-bibtex.i386-linux`. That is something you should never use unless you are sure you know what you are doing. `--dry-run` fakes the installation without changing anything.

□ `update` *pkg*... updates the packages given as arguments. In addition, if the *pkg* is a collection, and the remote server has new packages in this collection, they will be installed, following the dependencies specified in the collection. Options:

`--list` Lists the packages which would be updated or newly installed, but does not do the update. It also lists the revision numbers of the local and the remote packages.

`--all` Update all out-of-date packages.

`--dry-run` Fake the updates without changing anything.

`--backupdir` *directory* Save a snapshot of the current package (as installed) in *directory*, before the package is updated. This way one can easily recover in case an update turned out as not working. See the `restore` action for details.

`--no-depends` Do not install normal dependencies.

`--no-depends-at-all` See `install` above for this option.

- `remove` *pkg*... removes the packages given as arguments. Removing a collection will remove all package dependencies (but not collection dependencies) in that collection, unless `--no-depends` is specified. However, when removing a package, dependencies are never removed.
  Options:

  `--no-depends` Do not remove dependent packages.

  `--no-depends-at-all` See `install` above for this option.

  `--force` By default, when removing a package or collection would invalidate a dependency of another collection/scheme, the package is not be removed and an error is given. With this option, the package will be removed unconditionally. Use with care.

  `--dry-run` Fake the removals without actually changing anything.

- `backup` *pkg*... makes a backup of the given packages, or all packages with `--all`, to the directory specified with `--backupdir` (must exist and be writable.
  The following options are supported:

  `--backupdir` *directory* The directory is a required argument and must specify an existing directory where backups are to be placed.

  `--all` Make a backup of all packages in the TEX Live installation. This will take quite a lot of space and time.

- `restore --backupdir` *dir* [*pkg* [*rev*]]
  If no `pkg` is given (and thus no `rev`), lists the available backup revisions for all packages.
  With `pkg` given but no `rev`, list all available backup revisions of `pkg`.
  With both `pkg` and `rev`, tries to restore the package from the specified backup.
  The option `--backupdir dir` is required, and must specify a directory with backups.
  The option `--dry-run` is also supported, as usual.

- `arch` *operation arg*...If *operation* is `list`, this lists the names of architectures (`i386-linux`, ...) available at the default install location.
  If *operation* is `add`, adds the executables for each of the following arguments (architecture names) to the installation.
  The option `--dry-run` is also supported, as usual.

Typical usage of tlmgr

Here we present some typical usage examples of the TEX Live Manager.

Installing a new collection. Suppose that you installed `scheme-medium` and then realize that the hyphenation patterns for some language you are using haven't been installed, say, Norwegian. First you fire up tlmgr to search for the support:

```
$ tlmgr search --global norwegian
collection-langnorwegian - Norwegian
hyphen-norwegian -
```

and then to install this collection:

```
$ tlmgr install collection-langnorwegian
install: collection-langnorwegian
install: hyphen-norwegian
regenerating language.dat
regenerating language.def
```

and then it continues to regenerate all the format files depending on either `language.dat` or `language.def`. (If the the `formats` option is changed to false in the local database, the format rebuilding will be skipped. The default is to do so, to keep them up to date without manual intervention.)

Searching for a package. You want to typeset an invitation in a special form, say in the shape of a heart. Your first try is

```
$ tlmgr search paragraph
```

but that yields no output. Maybe it's not installed? So try a global search:

```
$ tlmgr search -global paragraph
tlmgr: installation location /src/TeX/texlive-svn/Master
 bigfoot - Footnotes for critical editions
 edmargin - Multiple series of endnotes for critical editions
 footmisc - A range of footnote options
 genmpage - Generalization of LaTeX's minipages
 hanging - Hanging paragraphs
 ibycus-babel - Use the Ibycus 4 Greek font with Babel
 insbox - A TeX macro for inserting pictures/boxes into paragraphs
 layouts - Display various elements of a document's layout
 lettrine - Typeset dropped capitals
 lineno - Line numbers on paragraphs
 lipsum - Easy access to the Lorem Ipsum dummy text
 moresize - Allows font sizes up to 35.83pt
 ncctools - A collection of general packages for LaTeX
 paralist - Enumerate and itemize within paragraphs
 picinpar - Insert pictures into paragraphs
 plari - Typesetting stageplay scripts
 seqsplit - Split long sequences of characters in a neutral way
 shapepar - A macro to typeset paragraphs in specific shapes
 vwcol - Variable-width multiple text columns
```

and here we are, `shapepar` seems to be what's needed. So let us see what it is:

```
$ tlmgr show shapepar
tlmgr: installation location /src/TeX/texlive-svn/Master
Package:  shapepar
Category: Package
ShortDesc: A macro to typeset paragraphs in specific shapes.
LongDesc: \shapepar is a macro to typeset paragraphs in a
special ...
Installed: No
Collection:collection-latexextra
```

Ok, confirmed, now we can either install the respective collection using

```
$ tlmgr install collection-latexextra
```

which will install quite a lot of packages, or only that one single package in the hope that it does not depend on anything else:

```
$ tlmgr install shapepar
tlmgr: installation location /src/TeX/texlive-svn/Master
install: shapepar
running mktexlsr
...
```

These examples are about finding uninstalled packages. The default for TEX Live is a full installation, i.e., everything is installed that is available.

Updating your installation. After the initial installation you want to get the latest and greatest of everything, but first you want to see what that means:

```
$ tlmgr update --list
tlmgr: installation location /mnt/cdrom
Cannot load TeX Live database from /mnt/cdrom at /home/norbert/tltest ⤸
    ⤹ /2008/bin/i386-linux/tlmgr line 1505, <TMP> line 1982.
```

Hmm, there seems to be an error, it tries to install from the DVD which you returned to your friend last week. Well, then you should switch to the network installation source; best to do it for all future sessions by saving it as default location. But what was that strange address again? Fortunately you can tell tlmgr to use CTAN and it will know what to do:

```
$ tlmgr option location ctan
tlmgr: setting default installation location to http://mirror.ctan.org ⤸
    ⤹ /systems/texlive/tlnet/2008
```

Fine. Now let us see what we can upgrade:

```
$ tlmgr update --list
shapepar: local: 10400, source: 10567
bin-texlive: local: 10693, source: 10750
pdftex: local: 10622, source: 10705
texlive.infra: local: 10685, source: 10748
```

Well, some things are there, so let us update all of them at once:

```
$ tlmgr update --all
update: shapepar (10400 -> 10567) ... done
update: bin-texlive (10693 -> 10750) ... done
update: pdftex (10622 -> 10705) ... done
update: texlive.infra (10685 -> 10748) ... done
running mktexlsr ...
```

Paper size configuration. You are moving to Japan and want letter as your default paper size; nothing easier:

```
$ tlmgr paper letter
```

will switch to letter for the most important programs, and at also recreate the formats.

The GUI for tlmgr

To make most Windows users and some Unix users happy we provide a front end for the TEX Live Manager written in Perl/Tk. It does not do the actual work, but leaves that for tlmgr. It also does not provide quite the full functionality of tlmgr, but almost all of it is there.

This program features several screens for different functionalities: installation, update, and removal of packages, removal of TEX Live as a whole, architecture support and configuration.

The GUI is started with either `tlmgr gui` or `tlmgr --gui action` where `action` is one of the actions given above. In the latter case it tries to open the respective screen of the GUI.

The install screen

The first window to be seen normally is the package installation screen (fig. 7).

At the top you see the current installation source, as given either on the command line of tlmgr, or in the absence of a command line argument as taken from

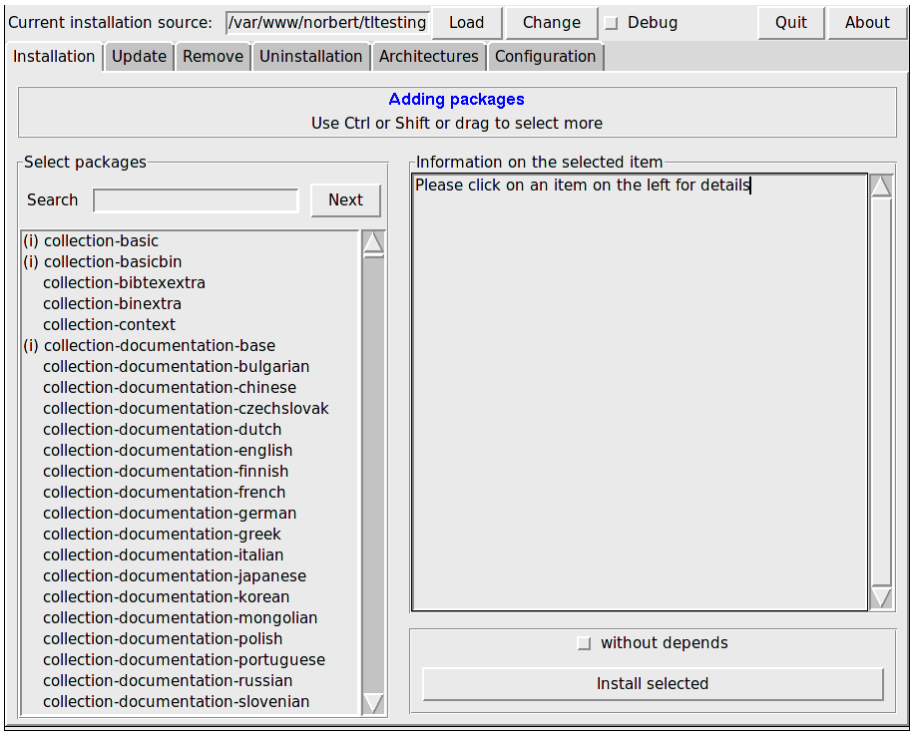


Figure 7. TEX Live Manager GUI install screen

the default option. It is *not* loaded automatically, you have to press the Load button, or the Change button to select a different installation source for this run only. Below you see the list of available packages on the left, first all the collections and schemes, then all the other packages in alphabetic order. You can search by entering a string into the search text field, which immediately jumps to the first entry. The button Next jumps to the next match. After selecting one package you can see its description in the right half of this screen. Below there is the action button for installing the selected package(s), and also a switch that allows you to install a package without those it depends on.

The update screen

The update screen is similar to the install screen, but only lists those packages which have an upgrade available on the installation location. The upper part of the right pane gives you information on the package, and in the action area below you see two buttons, one for updating only the selected packages, and one for updating all packages.

In fig. 8 you can see the update screen with updates available and the information for the selected package shown in the right part of the screen.

The remove screen

The remove screen is also similar to the install screen, with the list of all installed packages in the left part, the information window in the upper right part, and the action area with two toggles and the remove button in the lower right part; see fig. 9.

The two toggles correspond to the option `--force` and `--no-depends` of the tlmgr `remove` action, see above.

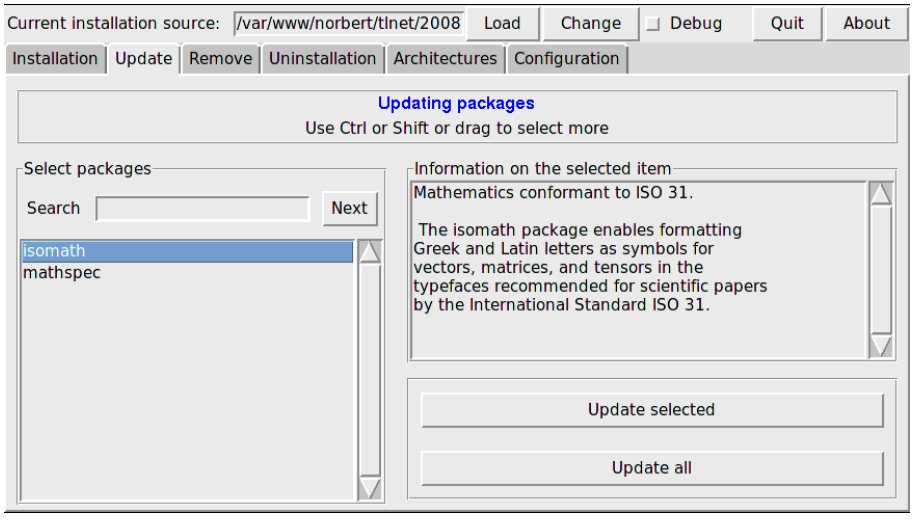


Figure 8. TEX Live Manager GUI update screen

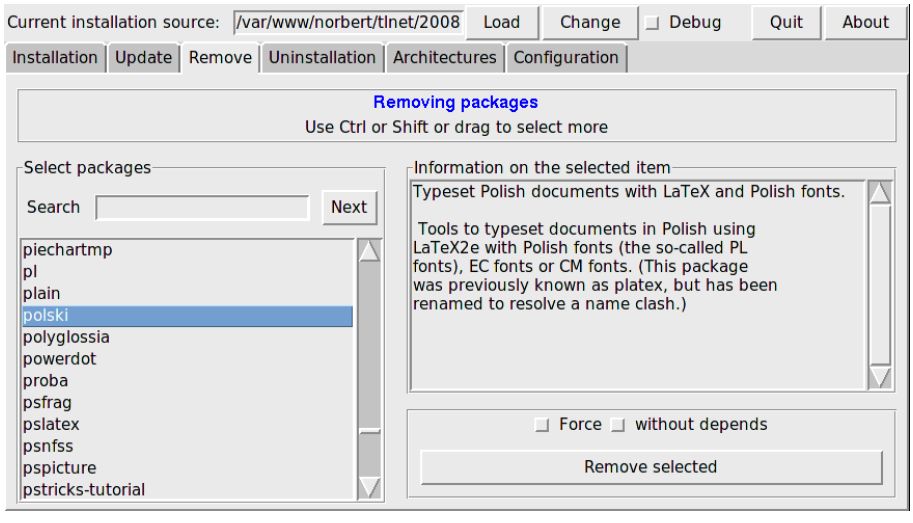


Figure 9. TEX Live Manager GUI remove screen

The uninstallation screen

This screen only sports one button which allows you to completely remove the TEX Live installation from your system. This button is not present on Windows systems, being replaced by an informational note that you should use the Add/Remove entry from the Control Panel.

The architectures screen

TEX Live allows you to install the binaries for several architecture-operating system combinations in case you want to distribute your installation via NFS or other means in an inhomogen local network, see fig. 10.

This screen lists the available architectures at the current installation source, and allows you to select new architectures to be installed by pressing the Apply changes button.

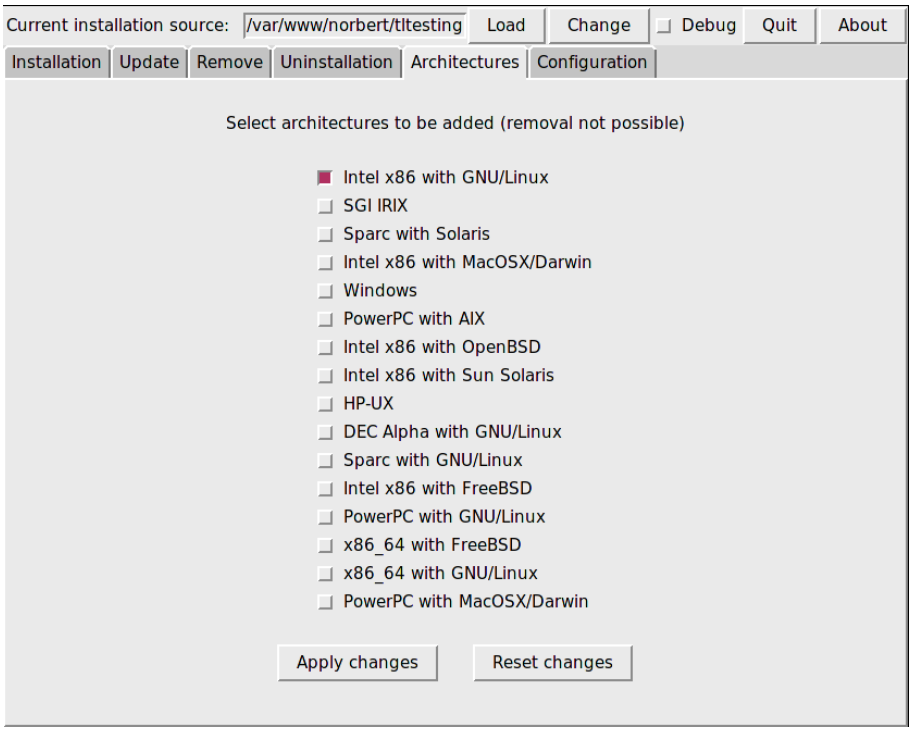


Figure 10. TEX Live Manager GUI architectures screen

Note that the *removal* of architectures is currently not supported, and that the whole screen is disabled on Windows systems since Windows does not support normal symbolic links.

The config screen

This screen allows the user to comfortably examine and set the various options of the TEX Live installation, see figure 11.

In the upper part you can change the defaults for the installation source, whether formats should be created (and updated) by default, and whether macro/font documentation and source files should be installed.

In the lower left part you can set the letter for all the programs to either `A4` or `letter`, or for each program individually. In the latter case you can choose from a wide range of paper formats depending on the programs support.

In the lower right part there are some convenience buttons for updating the `ls-R` databases, the outline font list (`updmap-sys`) and rebuilding all formats.

Execution of the commands

As mentioned above, this GUI is only a front end and leaves the actual work to tlmgr itself. So every action you do (installation, removal, etc.) will pop up a window where the output of the tlmgr process is shown.

On Unix systems that output will be shown immediately. Windows lacks good support for forking in Perl/Tk, and thus you have to wait until the whole process has terminated before the output appears. That can take quite some time, so please be patient.

We are working on merging the tlmgr and its GUI into one program so that the output would become more immediate in all cases.

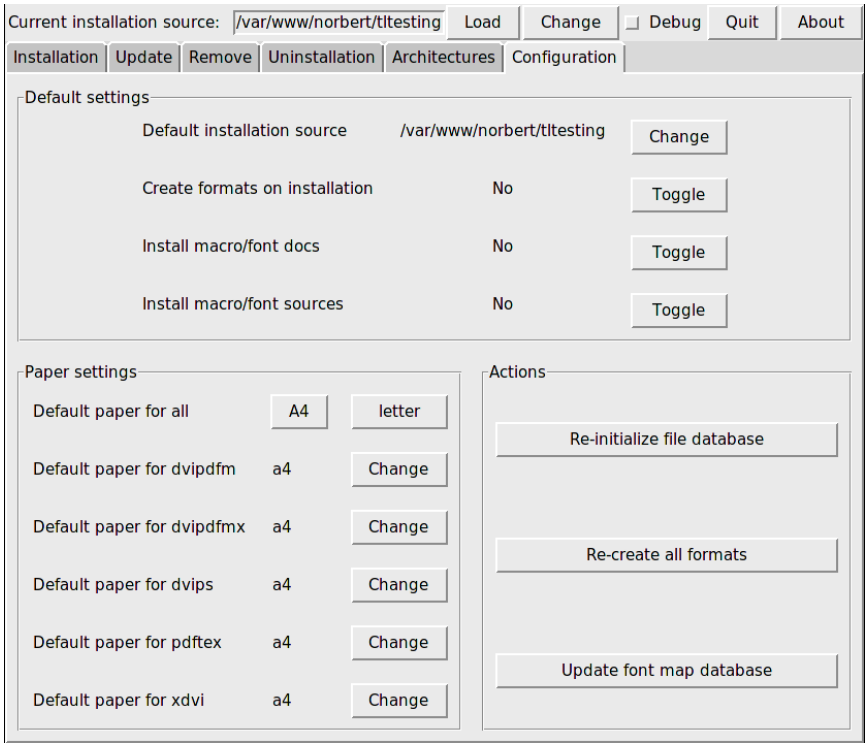


Figure 11. TEX Live Manager GUI config screen

What else is there?

Besides reworking the whole infrastructure, which is only user-visible in the new installer and the TEX Live Manager, as with every year, all the programs and packages have been updated. We currently ship around 1400 normal packages, e.g., LaTEX and font packages, and around 300 other packages, mostly documenation and a few packages which are TEX Live internal.

The new player in the game this year is the new engine LuaTEX (http://luatex.org); besides a new level of flexibility in typesetting, this provides an excellent scripting language for use both inside and outside of TEX documents.

Windows-specific features

To be complete, a TEX Live installation needs support packages that are not commonly found on a Windows machine. TEX Live provides the missing pieces:

Perl and Ghostscript. Because of the importance of Perl and Ghostscript, TEX Live includes 'hidden' copies of these programs. TEX Live programs that need them know where to find them, but they don't betray their presence through environment variables or registry settings. They aren't full-scale distributions, and shouldn't interfere with any system installations of Perl or Ghostscript.

Command-line tools. A number of Windows ports of common Unix command-line programs are installed along with the usual TEX Live binaries. These include gzip, chktex, jpeg2ps, unzip, wget and the command-line utilities from the xpdf suite. (The xpdf viewer itself is not available for Windows, but the Sumatra PDF viewer is based on it: `http://blog.kowalczyk.info/software/sumatrapdf`.)

`fc-cache` helps XeTEX to handle fonts more efficiently.

PS_View. Also installed is PS_View, a new PostScript viewer that is free software; see fig. 12. It also supports viewing of PDF files and is extremely fast. Please contact us with any suggestions, this program is in active development.

dviout This is a DVI previewer which is shipped only in the support directory of the DVD, but you will get it if you use the network update procedure. See fig. 13 for a screenshot.

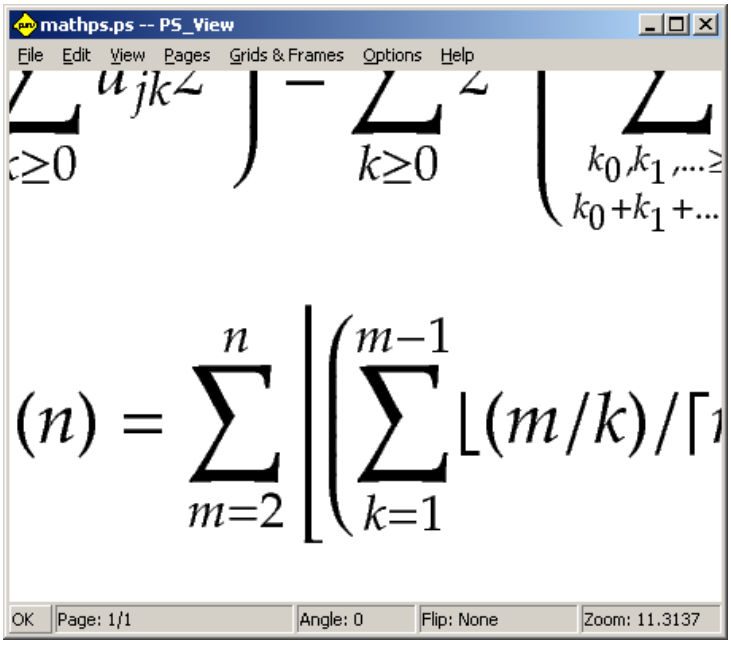


Figure 12. PS_View allows very high magnification, and renders PDF, too

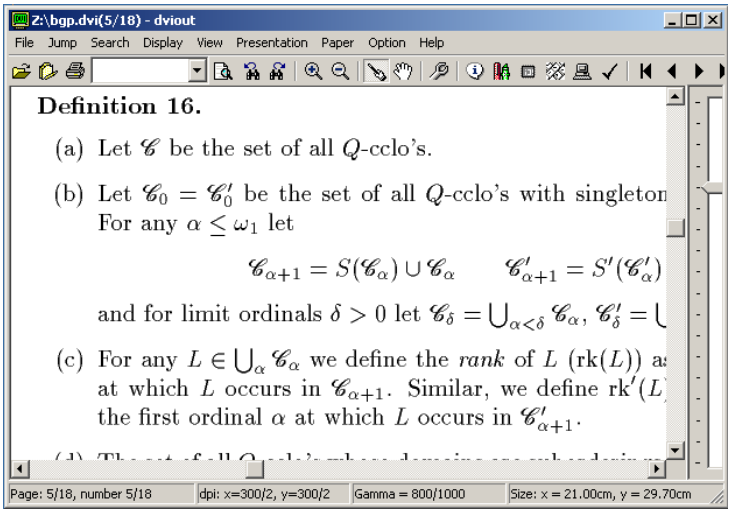


Figure 13. DVIout on Windows

Final remarks and other resources

The TEX Live Manager is very much work in progress, and its GUI even more. We are adding new functionality frequently, and improving existing functionality to make it more robust. If you see any anomalies don't hesitate to contact us at texlive@tug.org, and we will try to improve it further.

As with most volunteer projects the group of core programmers is quite small. Most of tlmgr and its GUI has been programmed by the author with some minor contributions from others. Anyone being more or less able to program Perl is heartily invited to join forces and help us, there are long lists of TODOs for the TEX Live Manager, let alone for all of TEX Live.

If you are searching more information on TEX Live your starting place should be http://tug.org/texlive/ and the documentation page `http://tug.org/texlive/doc.html`.

The list of people to thank is too long to be included here, please see the online TEX Live documentation, Chapter 9 (Acknowledgments), for the ever growing list. Of course one name has to be mentioned and that is Karl Berry who with great enthusiasm and perpetual support (and a sometimes critical voice if I was too fast in implementing something!) prepared the TEX Live 2008 release.

Norbert Preining
preining@logic.at

EuroTEX 2009

3rd ConTEXt Meeting

The Dutch TEX Language User Group and the ConTEXt task force are pleased to invite you to the combined EuroTEX 2009 conference and third international ConTEXt meeting.

24 – 28 August 2009, The Hague

Call for Papers

As usual, proposals for presentations and workshops are welcomed on just about any topic of interest to TEX users, but the conference focus will be on

Educational uses of TEX

such as manuals, courseware and college presentations, so we especially welcome proposals on subjects in those fields.

The language of the conference is English. Please send abstracts and proposals in plain text or TEX format to the conference committee at `eurotex@ntg.nl`.

Registration

The conference is made possible by the Netherlands Defence Academy (NLDA) that graciously invited us to their facilities, including the on-site hotel.

http://www.ntg.nl/EuroTeX2009/

Participants who complete registration before February 1, 2009 will benefit from a special early bird discount.