

## Tagged PDF in ConTeXt

Hans Hagen

### 1 Introduction

Occasionally users asked me if ConTeXt can produce tagged PDF and the answer to that has been: I'll implement it when I need it. However, users tell me that publishers more and more demand tagged PDF files, although one might wonder what for, except maybe for accessibility. Another reason for not having spent too much time on it before is that the specification was not that inviting.

At any rate, when I saw Ross Moore<sup>1</sup> presenting tagged math at TUG 2010, I decided to look up the spec once more and see if I could get into the mood to implement tagging. Before I started it was already clear that there were a couple of boundary conditions:

- Tagging should not put a burden on the user but users should be able to tag themselves.
- Tagging should not slow down a run too much; this is no big deal as one can postpone tagging till the last run.
- Tagging should in no way interfere with typesetting, so no funny nodes should be injected.
- Tagging should not make the code look worse, neither the document source, nor the low level ConTeXt code.

And of course implementing it should not take more than a few days' work, certainly not in an exceptionally hot summer.

You can 'google' for one of Ross's documents (like `DML_002-2009-1_12.pdf`) to see how a document source looks at his end using a special version of pdfTeX. However, the version on my machine didn't support the shown primitives, so I could not see what was happening under the hood. Unfortunately it is quite hard to find a properly tagged document so we have only the reference manual as starting point. As the pdfTeX approach didn't look that pleasing anyway, I just started from scratch.

Tags can help Acrobat Reader when reading out the text aloud. But you cannot browse the structure in the no-cost version of Acrobat and as not all users have the professional version of Acrobat, the fact that a document has structure can go unnoticed. Add to that the fact that the overhead in terms of bytes is quite large as many more objects are generated, and you will understand why this feature is not enabled by default.

<sup>1</sup> He is often exploring the boundaries of PDF, Unicode and evolving techniques related to math publishing so you'd best not miss his presentations when you are around.



Figure 1: A tag list in Acrobat.

### 2 Implementation

So, what does tagging boil down to? We can best look at how tagged information is shown in Acrobat. Figure 1 shows the content tree that has been added (automatically) to a document while figure 2 shows a different view.

In order to get that far, we have to do the following:

- Carry information with (typeset) text.
- Analyse this information when shipping out pages.
- Add a structure tree to the page.
- Add relevant information to the document.

That first activity is rather independent of the other three and we can use that information for other purposes as well, like identifying where we are in the document. We carry the information around using attributes. The last three activities took a bit of

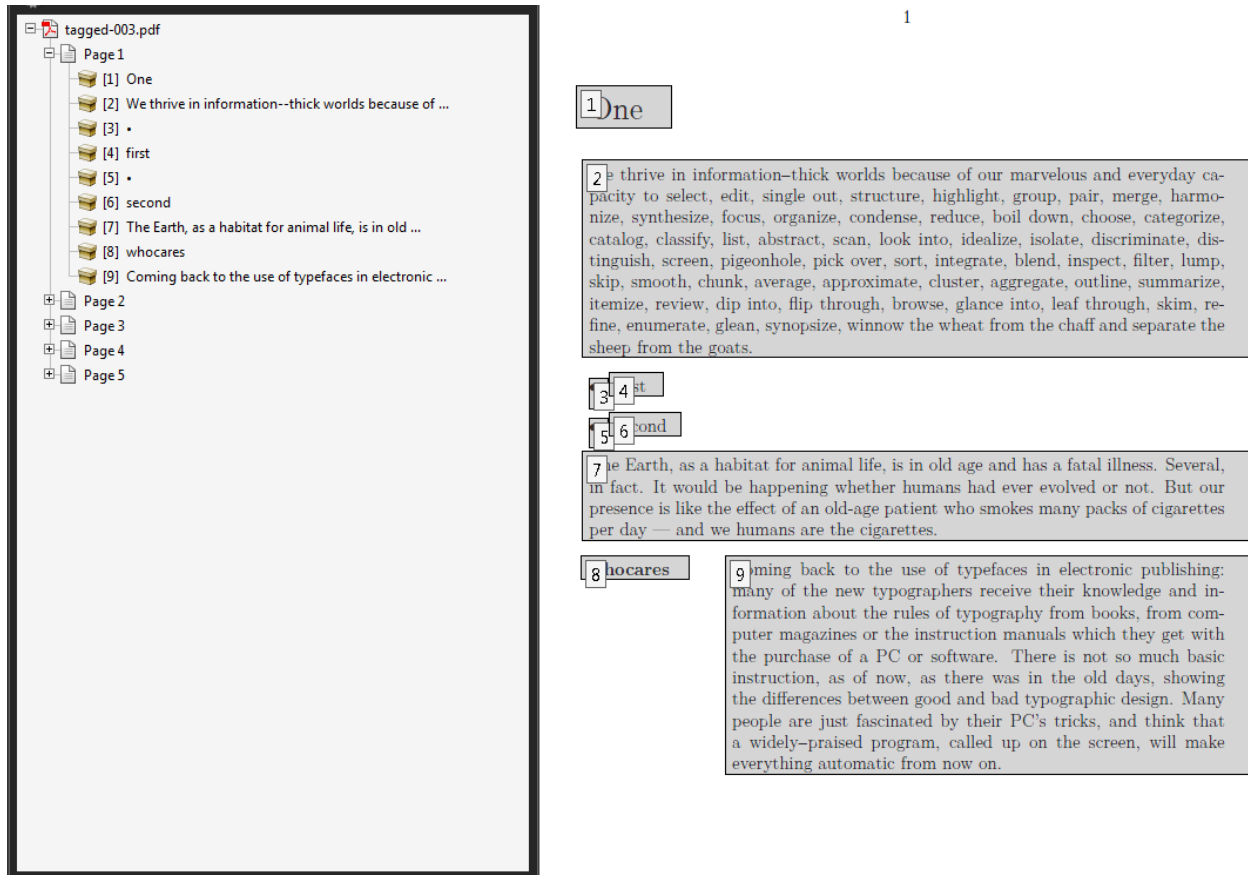


Figure 2: Acrobat showing the tag order.

experimenting mostly using the “Example of Logical Structure” from the PDF standard 32000-1:2008.

This resulted in a tagging framework that uses explicit tags, meaning the user is responsible for the tagging:

```
\setupstructure[state=start,method=none]
\starttext
\startelement[document]
\startelement[chapter]
\startelement[p] \input davis \stopelement\par
\stopelement

\startelement[chapter]
\startelement[p] \input zapf \stopelement\par
\startelement[whatever]
\startelement[p] \input tufte \stopelement\par
\startelement[p] \input knuth \stopelement\par
\stopelement
\stopelement
...
\stopelement
\stoptext
```

Since this is not much fun, we also provide an automated variant. In the previous example we

explicitly turned off automated tagging by setting method to none. By default it has the value auto.

```
\setupstructure[state=start]
% default is method=auto

\definedescription[whatever]
\starttext
\startfrontmatter
\startchapter[title=One]
\startparagraph \input tufte \stopparagraph
\startitemize
\startitem first \stopitem
\startitem second \stopitem
\stopitemize
\startparagraph \input ward \stopparagraph
\startwhatever {Hermann Zapf} \input zapf
\stopwhatever
\stopchapter
\stopfrontmatter

\startbodymatter
...
```

If you use commands like `\chapter` you will not get the desired results. Of course these can be

supported but there is no real reason for it, as in MkIV we advise using the `start-stop` variant.

It will be clear that this kind of automated tagging brings with it a couple of extra commands deep down in ConTeXt and there (of course) we use symbolic names for tags, so that one can overload the built-in mapping.

```
\setuptaglabeltext[en][document=text]
```

As with other features inspired by viewer functionality, the implementation of tagging is independent of the backend. For instance, we can tag a document and access the tagging information at the TeX end. The backend driver code maps tags to relevant PDF constructs. First of all, we just map the tags used at the ConTeXt end onto themselves. But, as validators expect certain names, we use the PDF rolemap feature to map them to (less interesting) names. The next list shows just a few of the currently used internal names, with the PDF ones between parentheses.

```
construct (Span), delimited (Quote),
delimitedblock (BlockQuote), description (Div),
...
tabulaterow (TR), verbatim (Code),
verbatimblock (Code), verbatimline (Code).
```

So, the internal ones show up in the tag trees as shown in the examples but applications might use the rolemap which normally has less detail.

Because we keep track of where we are, we can also use that information for making decisions.

```
\doifinelementelse{structure:section}
{yes} {no}
\doifinelementelse{structure:chapter}
{yes} {no}
\doifinelementelse{division:*-structure:chapter}
{yes} {no}
\doifinelementelse{division:*-structure:*}
{yes} {no}
```

As shown, you can use `*` as a wildcard. The elements are separated by `-`. If you don't know what tags are used, you can always enable the tag related tracker:

```
\enabletrackers[structure.tags]
```

This tracker reports the identified element chains to the console and log.

### 3 Special care

Of course there are a few complications. First of all the tagging model sort of contradicts the concept of a nicely typeset document where structure and outcome are not always related. Most TeX users are aware of the fact that TeX does not have space characters and does a great job on kerning and hyphenation. The tagging machinery on the other hand

uses a rather dumb model of strings separated by spaces.<sup>2</sup> But we can trick TeX into providing the right information to the backend so that words get nicely separated. The non-optimized function that does this looks as follows:

```
function injectspaces(head)
local p
for n in node.traverse(head) do
local id = n.id
if id == node.id("glue") then
if p and p.id == node.id("glyph") then
local g = node.copy(p)
local s = node.copy(n.spec)
g.char, n.spec = 32, s
p.next, g.prev = g, p
g.next, n.prev = n, g
s.width = s.width - g.width
end
elseif id == node.id("hlist")
or id == node.id("vlist") then
injectspaces(n.list,attribute)
end
p = n
end
end
```

Here we squeeze in a space (given that it is in the font which it normally is when you use ConTeXt) and make a compensation in the glue. Given that your page sits in box 255, you can do this just before shipping the page out:

```
injectspaces(tex.box[255].list)
```

Then there are the so-called suspects: things on the page that are not related to structure at all. One is supposed to tag these specially so that the built-in reading equipment is not confused. So far we could get around them simply because they don't get tagged at all and therefore are not seen anyway. This might well be enough of a precaution.

Of course we need to deal with mathematics. Fortunately the presentation MathML model is rather close to TeX and so we can map onto that. After all we don't need to care too much about back-mapping here. The currently present code is rather experimental and might get extended or thrown out in favour of inline MathML. Figure 3 demonstrates that a first approach does not even look that bad. In future versions we might deal with table-like math constructs, like matrices.

This is a typical case where more energy has to be spent on driving the voice of Acrobat but I will do that when we find a good reason.

<sup>2</sup> The search engine on the other hand is rather clever on recognizing words.





Tags

- <document>
  - <structure> chapter
    - <structurenumber>
    - <structuretitle>
    - <structurecontent>
      - <paragraph>
        - Let's see what a user defined command does:
        - <construct> notabene
          - whow
          - !
        - <float> figure
          - <floatcontent>
            - a simple graphic
          - <floatcaption>
            - Figure 1.1
          - <floattag>
            - test
        - <float> figure
        - <float> figure
          - <floatcontent>
            - <image>
              - PathPathPath
          - <floatcaption>
          - <mpgraphic>
            - Path
          - <paragraph>
            - Yet another paragraph.

# 1 chapter

Let's see what a user defined command does: **whow!**

a simple graphic

Figure 1.1 test

a simple graphic

Figure 1.2 test

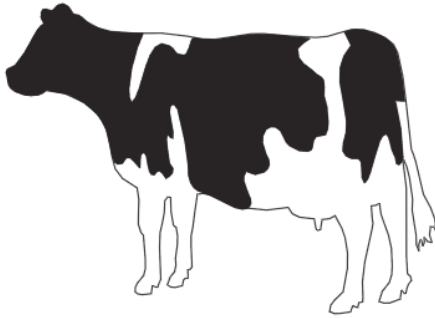


Figure 1.3 test

Yet another paragraph.

Figure 8: Float tags end up in the text stream. Watch the user defined construct.

sheep from the

- first<sup>1</sup>
- second

The Earth, as a  
in fact. It would  
presence is like  
per day — and

**whocares**

<sup>1</sup> test

Content Order Tags

Tags

- <document>
  - <division> frontpart
    - <structure> chapter
      - <structuretitle>
      - <structurecontent>
        - <paragraph>
          - We thrive in information--thick worlds because of ...
        - <itemgroup> itemize
          - <item>
            - <itemtag>
            - <itemcontent>
              - first1
            - <description> footnote
              - <descriptiontag>
                - 1
              - <descriptioncontent>
                - test
            - <item>
        - <paragraph>
        - <description> whatever

Figure 9: Footnotes are shown at the place in the input (flow).