The `tagpdf` package, v0.93

Ulrike Fischer

fischer@troubleshooting-tex.de

2022-01-13

This package is not meant for normal document production. It is mainly a tool to *research* tagging.

You need a very current LaTeX format. You need a very current L3 programming layer. You need the new LaTeX PDF management bundle.

This package is incomplete, experimental and quite probably contains bugs. At some time it will disappear when the code has been integrated into the LaTeX format.

This package can change in an incompatible way.

You need some knowledge about TeX, PDF and perhaps even lua to use it.

Issues, comments, suggestions should be added as issues to the github tracker:

`https://github.com/u-fischer/tagpdf`

Contents

1. Preface to version 0.93 ... 3

2. Preface to version 0.92 ... 3

3. Introduction ... 3
   - 3.1. Tagging and accessibility ... 4
   - 3.2. Engines and modes ... 4
   - 3.3. References and target PDF version ... 5
   - 3.4. Validation ... 5
   - 3.5. Examples wanted! ... 6

4. Changes ... 6
   - 4.1. Changes in 0.3 ... 6
   - 4.2. Changes in 0.5 ... 6
   - 4.3. Changes in 0.6 ... 7
   - 4.4. Changes in version 0.61 ... 7
   - 4.5. Changes in version 0.8 ... 7
   - 4.6. Changes in version 0.81 ... 8
   - 4.7. Changes in version 0.82 ... 8
   - 4.8. Changes in version 0.83 ... 8
   - 4.9. Changes in version 0.90 ... 8

4.10 Changes in version 0.92 9
4.11 Changes in version 0.93 9
4.12 Proof of concept: the tagging of the documentation itself 10

5. Setup 11
5.1. Modes and package options 11
5.2. Setup and activation 12

6. Tagging 13
6.1. Three tasks 14
6.2. Task 1: Marking the chunks: the mark-content-step 14
6.2.1. Generic mode versus lua mode in the mc-task 17
6.2.2. Commands to mark content and chunks 18
6.2.3. Luamode: global or not global – that is the question 20
6.2.4. Tips 21
6.2.5. Header and Footer 22
6.2.6. Links and other annotations 23
6.2.7. Math 25
6.2.8. Split paragraphs 25
6.2.9. Automatic tagging of paragraphs 26
6.3. Task 2: Marking the structure 26
6.3.1. Structure types 27
6.3.2. Sectioning 27
6.3.3. Commands to define the structure 27
6.3.4. Root structure 30
6.3.5. Attributes and attribute classes 30
6.4. Task 3: tree Management 31
6.5. A fully marked up document body 31
6.6. Lazy and automatic tagging 32
6.7. Adding tagging to commands 33

7. Alternative text, ActualText and text-to-speech software 33

8. Standard types and new tags 34

9. “Real” space glyphs 35

10. Structure destinations 36

11. Accessibility is not only tagging 37

12. Debugging 37

13. To-do 38

References 39

A. Some remarks about the PDF syntax 39

1. Preface to version 0.93

The main change in the package itself in this version is the support for structure destinations. This is a new type of destinations in PDF 2.0. For pdftex and luatex this requires new binaries. They will be included in texlive 2022, miktex already has the new pdftex, the new luatex will probably follow soon.

Beside this mostly some clean up and bug fixes has been done.

A more important step will be done in LaTeX itself in the next dev-release: The command `\DocumentMetadata` will be added to the format and will take over the role of `\DeclareDocumentMetadata` from `pdfmanagement-testphase` and additionally will also load the pdf management code. This will simplify the documents as it will no longer be needed to load a package.

2. Preface to version 0.92

In this version support for page breaks in pdftex has been added. As described in section 6.2.8, tagging markers must be added by *page*. That means that a paragraph that goes over two pages must get an end marker at the end on the first page and a new begin marker on the next page.

With lualatex that is rather easy to ensure, with pdflatex it requires quite sophisticated code. The method is described in Frank Mittelbach's talk at TUG 2021 "Taming the beast — Advances in paragraph tagging with pdfTeX and XeTeX" `https://youtu.be/SZHIeevyo3U?t=19551`. The new code requires a new version of the pdfmanagement-testphase package.

Please check also section 6.2.8 for possible pitfalls.

Also new in this version is the handling of header and footer: they will now be tagged as artifacts automatically. See section 6.2.5.

3. Introduction

Since many year the creation of accessible PDF-files with LaTeX which conform to the PDF/UA standard has been on the agenda of TeX-meetings. Many people agree that this is important and Ross Moore has done quite some work on it. There is also a TUG-mailing list and a webpage [5] dedicated to this theme.

But in my opinion missing are means to *experiment* with tagging and accessibility. Means to try out, how difficult it is to tag some structures, means to try out, how much tagging is really needed (standards and validators don't need to be right ...), means to test what else is needed so that a PDF works e.g. with a screen reader. Without such experiments it is imho quite difficult to get a feeling about what has to be done, which kernel changes are needed, how packages should be adapted.

This package tries to close this gap by offering *core* commands to tag a PDF[1].

My hope is that the knowledge gained by the use of this package will at the end allow to decide if and how code to do tagging should be part of the LaTeX kernel.

The package does not patch commands from other packages. It is also not an aim of the package to develop such patches. While at the end changes to various commands in many classes and packages will be needed to get tagged PDF files – and the examples accompanying the package try (or will try) to show various strategies – these changes should in my opinion be done by the class, package and document writers themselves using a sensible API provided by the kernel and not by some external package that adds patches everywhere and would need constant maintenance – one only need to look at packages like tex4ht or bidi or hyperref to see how difficult and sometimes fragile this is.

So this package deliberately concentrates on the basics – and this already quite a lot, there are much more details involved as I expected when I started.

I'm sure that it has bugs. Bugs reports, suggestions and comments can be added to the issue tracker on github. `https://github.com/u-fischer/tagpdf`.

Please also check the github site for new examples and improvements.

3.1. Tagging and accessibility

While the package is named `tagpdf` the goal is actually *accessible* PDF-files. Tagging is *one* requirement for accessibility but there are others. I will mention some later on in this documentation, and – if sensible – I will also try to add code, keys or tips for them.

So the name of the package is a bit wrong. As excuse I can only say that it is shorter and easier to pronounce.

3.2. Engines and modes

The package works currently with pdflatex and lualatex. First steps have been done to also enable support for xelatex and the latex-dvips-route; but this isn't yet much tested.

The package has two modes: the *generic mode* which should work in theory with every engine and the *lua mode* which works only with lualatex.

I implemented the generic mode first. Mostly because my tex skills are much better than my lua skills and I wanted to get the tex side right before starting to fight with attributes and node traversing.

While the generic mode is not bad and I spent quite some time to get it working I nevertheless think that the lua mode is the future and the only one that will be usable for larger documents. PDF is a page orientated format and so the ability of luatex to manipulate pages and nodes after the TeX-processing is really useful here. Also with luatex characters are normally already given as unicode.

[1] In case you don't know what this means: there will be some explanations later on.

The package uses quite a lot labels (in generic mode more than with luamode). At the begin it relied on the `zref` package, but switched now to a new experimental implementation for labels. The drawback of the new method is that they don't give yet good rerun messages if they have changed. I advise to use the `rerunfilecheck` package as a intermediate work-around.

3.3. References and target PDF version

My main reference for the first versions of this package was the free reference for PDF 1.7. [2] and so the package also targetted this version.

In 2018 PDF 2.0. has been released, and since 2020 all engines can set the version to 2.0. So the package will now target PDF 2.0. This doesn't mean that 2.0 will be required, but that the code and the options will be extended to support PDF 2.0. One example is the support for associated files, another the support for name spaces in version 0.82.

The package doesn't try to suppress all 2.0 features if an older PDF version is produced. It normally doesn't harm if a PDF contains keys unknown in its version and it makes the code faster and easier to maintain if there aren't too many tests and code pathes; so for example associated files will always be added. But tests could be added in case this leads to incompabilities.

It should be noted that some tools don't like PDF 2.0. PAC3 for example simply crashes, and pdftk will create a PDF 1.0 from it. This makes testing PDF 2.0 files a bit of a challenge.

3.4. Validation

PDF's created with the commands of this package must be validated:

- One must check that the PDF is *syntactically* correct. It is rather easy to create broken PDF: e.g. if a chunk is opened on one page but closed on the next page or if the document isn't compiled often enough.
- One must check how good the requirements of the PDF/UA standard are followed *formally*.
- One must check how good the accessibility is *practically*.

Syntax validation and formal standard validation can be done with preflight of the (non-free) adobe acrobat. It can also be done also with the free PDF Accessibility Checker (PAC 3) [7]. There is also the validator veraPDF [6]. A rather new and quite useful tool is "Next Generation PDF" [3], a browser application which converts a tagged PDF to html, allows to inspect its structure and also to edit the structure.

Practical validation is naturally the more complicated part. It needs screen reader, users which actually knows how to handle them, can test documents and can report where a PDF has real accessibility problems.

Preflight woes

Sadly validators can not be always trusted. As an example for an reason that I don't understand the adobe preflight don't like the list structure `L`. It is also possible that validators contradict: that the one says everything is okay, while the other complains.

3.5. Examples wanted!

To make the package usable examples are needed: examples that demonstrate how various structures can be tagged and which patches are needed, examples for the test suite, examples that demonstrates problems.

> Feedback, contribuations and corrections are welcome!

All examples should use the `\tagpdfsetup` key `uncompress` described in the next section so that uncompressed PDF are created and the internal objects and structures can be inspected and – hopefully soon – be compared by the l3build checks.

4. Changes

4.1. Changes in 0.3

In this version I improved the handling of alternative and actual text. See section 7. This change meant that the package relied on the module `l3str-convert`.

I no longer try to (pdf-)escape the tag names: it is a bit unclear how to do it at best with luatex. This will perhaps later change again.

4.2. Changes in 0.5

I added code to handle attributes and attribute classes, see section 6.3.5 and corrected a small number of code errors.

I added code to add "real" space glyphs to the PDF, see section 9.

4.3. Changes in 0.6

Breaking change! The attributes used in luamode to mark the MC-chunks are no longer set globally. I thought that global attribute would make it easier to tag, but it only leads to problem when e.g. header and footer are inserted. So from this version on the attributes are set locally and the effect of a `\tagmcbegin` ends with the current group. This means that in some cases more `\tagmcbegin` are needed and this affected some of the examples, e.g. the patching commands for sections with KOMA. On the other side it means that quite often one can omit the `\tagmcend` command.

4.4. Changes in version 0.61

- internal code adaptions to expl3 changes.
- dropped the compresslevel key – probably not needed.

4.5. Changes in version 0.8

- As a first step to include the code proper in the LaTeX kernel the module name has changed from `uftag` to `tag`. The commands starting with `\uftag` will stay valid for some time but then be deprecated.
- **Breaking change!** The argument of `newattribute` option should no longer add the dictionary bracket `<<..>>`, they are added by the code.
- **Breaking change!** The package now requires the new PDF management as provided for now by the package `pdfmanagement-testphase`. `pdfmanagement-testphase`, prepares the ground for better support for tagged PDF in LaTeX. It is part of a larger project to automatically generate tagged PDF https://www.latex-project.org/news/2020/11/30/tagged-pdf-FS-study/
- Support to add associated files to structures has been added with new keys `AF`, `AFinline` and `AFinline-o`.
- **Breaking change!** The support for other 8-bit input encodings has been removed. utf8 is now the required encoding.
- The keys `lang`, `ref` and `E` have been added for structures.
- The new hooks of LaTeX are used to tagged many paragraphs automatically. The small red numbers around paragraphs in the documentation show them in action. The main problem here is not to tag a paragraph, but to avoid to tag too many: paragraphs pop up in many places.

4.6. Changes in version 0.81

- 118 Hook code to tag links (URI and GoTo type) have been added. So normally they should simply work if tagging is activated. 118
- 119 Commands and keys to allow automatic paragraph tagging have been added. See section 6.2.9. As can be seen in this documentation the code works quite good already, but one should be aware that "paragraphs" can appear in many places and sometimes there are even more paragraph begin than ends. 119
- 120 A key to test if local or global setting of the mc-attributes in luamode is more sensible, see 6.2.3 for more details. 120
- 121 New commands to store and reset mc-tags. 121
- 122 PDF 2.0 namespaces are now supported. 122

4.7. Changes in version 0.82

123 A command `\tag_if_active:TF` to test if tagging is active has been added. This allow external packages to write conditional code. 123

124 The commands `\tag_struct_parent_int:` and `\tag_struct_insert_annot:nn` have been added. They allow to add annotations to the structure. 124

4.8. Changes in version 0.83

125 `\tag_finish_structure:` has been removed, it is no longer a public command. 125

4.9. Changes in version 0.90

- 126 Code has been cleaned up and better documented. 126
- 127 **More engines supported** The generic mode of `tagpdf` now works (theoretically, it is not much tested) with all engines supported by the pdfmanagement. So compilations with XeLaTeX or with dvips should work. But it should be noted that these engines and backends don't support the `interspaceword` option. With XeLaTeX it is perhaps possible implement something with `\XeTeXinterchartoks`, but for the dvips route I don't see an option (apart from lots of manual macros everywhere). 127
- Breaking change! 128 **MC-attributes are global again** In version 0.6 the attributes used in luamode to mark the MC-chunks were no longer set globally. This avoided a number of problems with header and footer and background material, but further tests showed that it makes it difficult to correctly mark things like links which have to interrupt the current marking code—the attributes couldn't easily escape groups added by users. See section 6.2.3 for more details. 128

- **key global-mc removed:** Due to the changes in the attribute keys this key is not longer needed.
- **key check-tags removed:** It doesn't fit. Checks are handled over the logging level.
- `\tagpdfget` has been removed, use the expl3 version if needed.
- The show commands `\showtagpdfmcdata`, `\showtagpdfattributes`, `\showtagstack` have been removed and replaced by a more flexible command `\ShowTagging`.
- The commands `\tagmcbegin` and `\tagmcend` no longer ignore following spaces or remove earlier one. While this is nice in some places, it also ate spaces in places where this wasn't expected. From now on both commands behave exactly like the expl3 versions.
- The lua-code to add real space glyphs has been separated from the tagging code. This means that `interwordspace` now works also if tagging is not active.
- The key `activate` has been added, it open the first structure, see below.

4.10. Changes in version 0.92

- support for page breaks in pdftex has been added, see section 6.2.8, This requires a new version of the pdfmanagement-testphase package.
- header and footer are tagged as artifacts automatically, see section 6.2.5.
- keys `alttext-o` and `actualtext-o` has been removed. `alttext` and `actualtext` will now expand once.

4.11. Changes in version 0.93

- Support for associated files in the root element (key `root-AF`) has been added. This allow e.g. to add a css-file which is be used if the PDF is converted to html.
- First steps have been done to adapt the package to planed changes in LaTeX: The command `\DocumentMetadata` will be added to the format and will take over the role of `\DeclareDocumentMetadata` from `pdfmanagement-testphase`.
- The package has now support for "structure destinations". This is a new type of destinations in PDF 2.0. For pdftex and luatex this requires new binaries. They will be included in texlive 2022, miktex already has the new pdftex, the new luatex will probably follow soon.
- The commands `\tagpdfifluatexT`, `\tagpdfifluatexTF` has been removed `\tagpdfifpdftexT`,

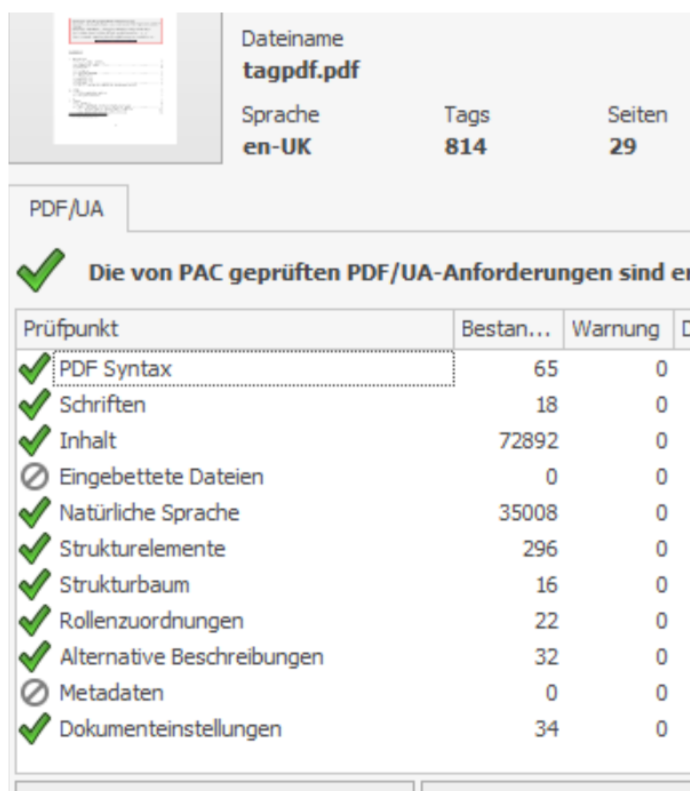
4.12. Proof of concept: the tagging of the documentation itself

143 Starting with version 0.6 the documentation itself has been tagged. The tagging wasn't (and isn't) in no way perfect. The validator from Adobe didn't complain, but PAX3 wanted alternative text for all links (no idea why) and so I put everywhere simple text like “link” and “ref”. The links to footnotes gave warnings, so I disabled them. I used types from the PDF version 1.7, mostly as I have no idea what should be used for code in 2.0. Margin notes were simply wrong … 143

144 The tagging has been improved and automated over time in sync with improvements and new features in the LaTeX kernel and the pdfmanagement code. 144

145 But even if the documentation passed the tests of the validators: as mentioned above passing a formal test doesn't mean that the content is really good and usable. I have a lot doubts that the code parts are really readable. The bibliography and the references must be improved. The user commands used for the tagging and also some of the patches used are still rather crude. So there is lot space for improvement. 145

> 146 Be aware that to create the tagged version a current lualatex-dev and a current version of the pdfmanagment-testphase package is needed. 146



5. Setup

The package requires the new PDF management. With a current version of `pdfmanagement-testphase` it can be loaded and activated like this:

```
\RequirePackage{pdfmanagement-testphase}
\DeclareDocumentMetadata
 {
  testphase = tagpdf, % load
  activate = tagging % activate and create the document structure
 }
\documentclass{article}
\begin{document}
some text
\end{document}
```

With LaTeX 2022-06-01 (or a current LaTeX-dev) the call will be simpler

```
\DocumentMetadata
 {
  testphase = tagpdf, % load + activate
 }
\documentclass{article}
\begin{document}
some text
\end{document}
```

Activation needed!

When the package is loaded it will – apart from loading more packages and defining a lot of things – not do much. You will have to activate it with `\tagpdfsetup` or as shown above in `\DeclareDocumentMetadata/\DocumentMetadata`.

Most commands do nothing if tagging is not activated, but in case a test is needed a command (with the usual p,T,F variants) is provided:

`\tag_if_active:TF`

The check is true only if *everything* is activated. In all other cases (also if tagging has been stopped locally) it will be false.

5.1. Modes and package options

The package has two different modes: The **generic mode** works (in theory, currently only tested with pdftex and luatex) probably with all engines, the **lua mode** only with luatex. The differences between both modes will be described later. The mode can be set with package options:

`luamode`

This is the default mode. It will use the generic mode if the document is processed with pdflatex and the lua mode with lualatex.

`genericmode`

This will force the generic mode for all engines.

5.2. Setup and activation

`\tagpdfsetup`{⟨*key-val-list*⟩}

This command setups the general behaviour of the package. The command should be normally used only in the preamble (for a few keys it could also make sense to change them in the document).

The key-val list understands the following keys:

activate-all Boolean, initially false. Activates everything, that's normally the sensible thing to do.

activate Like `activate-all`, *additionally* is opens at begin document a structure with `\tagstructbegin` and closes it at end document. The key accepts as value a tag name which is used as the tag of the structure. The default value is `Document`.

activate-mc Boolean, initially false. Activates the code related to marked content.

activate-struct Boolean, initially false. Activates the code related to structures. Should be used only if activate-mc has been used too.

no-struct-dest Starting with version 0.93 `tagpdf` will create automatically structure destinations (see section 10 if `hyperref` is used, if the engine supports it and if the pdf version is 2.0. With this key this can be suppressed.

activate-tree Boolean, initially false. Activates the code related to trees. Should be used only if the two other keys has been used too.

add-new-tag Allows to define new tag names, see section 8 for a description.

interwordspace Choice key, possible values are true/""on and false/off. The key activates/deactivates the insertion of space glyphs, see section 9. In the luamode it only works if at least activate-mc has been used.

log Choice key, possible values none, v, vv, vvv, all. Setups the log level. Changing the value affects currently mostly the luamode: "higher" values gives more messages in the log. The current levels and messages have been setup in a quite ad-hoc manner and will need improvement.

newattribute This key takes two arguments and declares an attribute. See 6.3.5.

luamode **show-spaces** Boolean. That's a debug option, it helps in lua mode to see where space glyph will be inserted if interwordspace is activated.

paratagging Boolean. This activate/deactivates the automatic tagging of paragraphs. It uses the `para/begin` and `para/end` hooks of the newest LaTeX version (2021-05-01). This is a first try to use this hooks, and the code is bound to change. Paragraphs can appear in many unexpected places and the code can easily break, so there is also an option to see where such paragraphs are: 173

paratagging-show Boolean. This activate/deactivates small red numbers in the places where the paratagging hook code is used. 174

tabsorder Choice key, possible values are row, column, structure, none. This decides if a `/Tabs` value is written to the dictionary of the page objects. Not really needed for tagging itself, but one of the things you probably need for accessibility checks. So I added it. Currently the tabsorder is the same for all pages. Perhaps this should be changed 175

luamode **tagunmarked** Boolean, initially true. When this boolean is true, the lua code will try to mark everything that has not been marked yet as an artifact. The benefit is that one doesn't have to mark up every deco rule oneself. The danger is that it perhaps marks things that shouldn't be marked – it hasn't been tested yet with complicated documents containing annotations etc. See also section 6.6 for a discussion about automatic tagging. 176

uncompress Sets both the PDF compresslevel and the PDF objcompresslevel to 0 and so allows to inspect the PDF. 177

6. Tagging

178 pdf is a page orientated graphic format. It simply puts ink and glyphs at various coordinates on a page. A simple stream of a page can look like this[2]: 178

```
stream
   BT
   /F27 14.3462 Tf                  %select font
   89.291 746.742 Td                %move point
   [(1)-574(Intro)-32(duction)]TJ   %print text
   /F24 10.9091 Tf                  %select font
    0 -24.35 Td                     %move point
    [(Let's)-331(start)]TJ          %print text
    205.635 -605.688 Td             %move point
    [(1)]TJ                         %print text
   ET
endstream
```

179 From this stream one can extract the characters and their placement on the page but not their semantic meaning (the first line is actually a section heading, the last the page number). And while in the example the order is correct there is actually no guaranty that the stream contains the text in the order it should be read. 179

[2]The appendix contains some remarks about the syntax of a PDF file

180 Tagging means to enrich the PDF with information about the *semantic* meaning and the *reading order*. (Tagging can do more, one can also store all sorts of layout information like font properties and indentation with tags. But as I already wrote this package concentrates on the part of tagging that is needed to improve accessibility.) 180

6.1. Three tasks

181 To tag a PDF three tasks must be carried out: 181

mc-task

1. 182 **The mark-content-task**: The document must add "labels" to the page stream which allows to identify and reference the various chunks of text and other content. This is the most difficult part of tagging – both for the document writer but also for the package code. At first there can be quite many chunks as every one is a leaf node of the structure and so often a rather small unit. At second the chunks must be defined page-wise – and this is not easy when you don't know where the page breaks are. Also in a standard document a lot text is created automatically, e.g. the toc, references, citations, list numbers etc and it is not always easy to mark them correctly. 182

struct-task

2. 183 **The structure-task**: The document must declare the structure. This means marking the start and end of semantically connected portions of the document (correctly nested as a tree). This too means some work for the document writer, but less than for the mc-task: at first quite often the mc-task and the structure-task can be combined, e.g. when you mark up a list number or a tabular cell or a section header; at second one doesn't have to worry about page breaks so quite often one can patch standard environments to declare the structure. On the other side a number of structures end in LaTeX only implicitly – e.g. an item ends at the next item, so getting the PDF structure right still means that additional mark up must be added. 183

tree-task

3. 184 **The tree management**: At last the structure must be written into the PDF. For every structure an object of type `StructElem` must be created and flushed with keys for the parents and the kids. A parenttree must be created to get a reference from the mc-chunks to the parent structure. A rolemap must be written. And a number of dictionary entries. All this is hopefully done automatically and correctly by the package 184

6.2. Task 1: Marking the chunks: the mark-content-step

188 To be able to refer to parts of the text in the structure, the text in the page stream must get "labels". In the PDF reference they are called "marked content". The three main variants needed here are: 188

Artifacts 189 They are marked with of a pair of keywords, `BMC` and `EMC` which surrounds the text. `BMC` has a single prefix argument, the fix tag name `/Artifact`. Artifacts should be used for irrelevant text and page content that should be ignored in the structure. Sadly it is often not possible to leave such text simply unmarked – the accessibility tests in Acrobat and other validators complain. 189

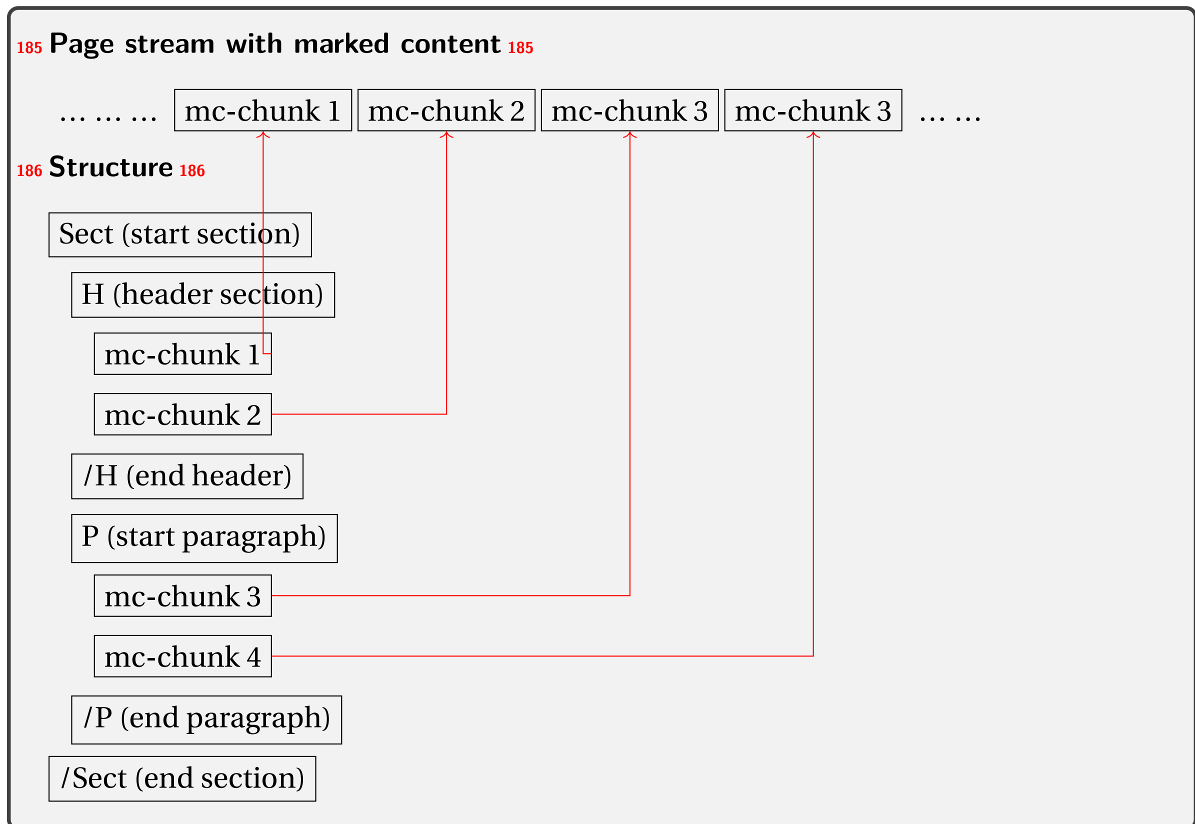


Figure 1: Schematical description of the relation between marked content in the page stream and the structure

```
/Artifact BMC
 text to be marked
/EMC
```

Artifacts with a type They are marked with of a pair of keywords, `BDC` and `EMC` which surrounds the text. `BDC` has two arguments: again the tag name `/Artifact` and a following dictionary which allows to specify the suppressed info. Text in header and footer can e.g. be declared as pagination like this:

```
/Artifact <</Type /Pagination>> BDC
 text to be marked
/EMC
```

Content Content is marked also with of a pair of keywords, `BDC` and `EMC`. The first argument of `BDC` is a tag name which describes the structural type of the text[3]Examples are `/P` (paragraph), `/H2` (heading), `/TD` (table cell). The reference mentions a number of standard types but it is possible to add more or to use different names.

[3]There is quite some redundancy in the specification here. The structural type is also set in the structure tree. One wonders if it isn't enough to use always `/SPAN` here.

In the second argument of BDC – in the property dictionary – more data can be stored. *Required* is an /MCID-key which takes an integer as a value:

```
/H1 <</MCID 3>> BDC
 text to be marked
/EMC
```

This integer is used to identify the chunk when building the structure tree. The chunks are numbered by page starting with 0. As the numbers are also used as an index in an array they shouldn't be "holes" in the numbering system (It is perhaps possible to handle a numbering scheme not starting by 0 and having holes, but it will enlarge the PDF as one would need dummy objects.).

It is possible to add more entries to the property dictionary, e.g. a title, alternative text or a local language setting.

The needed markers can be added with low level code e.g. like this (in pdftex syntax):

```
\pdfliteral page {/H1 <</MCID 3>> BDC}%
 text to be marked
\pdfliteral page {EMC}%
```

This sounds easy. But there are quite a number of traps, mostly with pdfLaTeX:

1. PDF is a page oriented format. And this means that the start BDC/BMC and the corresponding end EMC must be on the same page. So marking e.g. a section title like in the following example won't always work as the literal before the section could end on the previous page:

   ```
   \pdfliteral page {/H1 <</MCID 3>> BDC} %problem: possible pagebreak here
    \section{mysection}
   \pdfliteral page {EMC}%
   ```

   Using the literals *inside* the section argument is better, but then one has to take care that they don't wander into the header and the toc.

2. Literals are "whatsits" nodes and can change spacing, page and line breaking. The literal *behind* the section in the previous example could e.g. lead to a lonely section title at the end of the page.

3. The /MCID numbers must be unique on a page. So you can't use the literal in a saved box that you reuse in various places. This is e. g. a problem with longtable as it saves the table header and footer in a box.

4. The /MCID-chunks are leaf nodes in the structure tree, so they shouldn't be nested.

5. Often text in a document is created automatically or moved around: entries in the table of contents, index, bibliography and more. To mark these text chunks correctly one has to analyze the code creating such content to find suitable places to inject the literals.

6. The literals are inserted directly and not at shipout. This means that due to the asynchronous page breaking of TEX the MCID-number can be wrong even if the counter is reset at every page. This package uses in generic mode a label-ref-system to get around this problem. This sadly means that often at least three compilations are needed until everything has settled down.

   It can actually be worse: If the text is changed after the MCID-numbers have been assigned, and a new mc-chunk is inserted in the middle of the page, then all the numbers have to be recalculated and that requires again a number of compilations until it really settles down again. Internal references are especially problematic here, as the first compilation typically creates a non-link `??`, and only the second inserts the structure and the new mc. When the reference system in LaTeX will be extended, care will be taken to ensure that already the dummy text builds a chunk. Until then the advice is to first compile the document and resolve all cross-reference and to activate tagging only at the end.

7. There exist environments which process their content more than once – examples are `align` and `tabularx`. So one has to check for doublettes and holes in the counting system.

8. PDF is a page oriented format. And this means that the start and the end marker must be on the same page … *so what to do with normal paragraphs that split over pages??*. This question will be discussed in subsection 6.2.8.

6.2.1. Generic mode versus lua mode in the mc-task

While in generic mode the commands insert the literals directly and so have all the problems described above the lua mode works quite differently: The tagging commands don't insert literals but set some *attributes* which are attached to all the following nodes. When the page is shipped out some lua code is called which wanders through the shipout box and injects the literals at the places where the attributes changes.

This means that quite a number of problems mentioned above are not relevant for the lua mode:

1. Pagebreaks between start and end of the marker are *not* a problem. So you can mark a complete paragraph. If a pagebreak occur directly after an start marker or before an end marker this can lead to empty chunks in the PDF and so bloat up PDF a bit, but this is imho not really a problem (compared to the size increase by the rest of the tagging).

2. The commands don't insert literals directly and so affect line and page breaking much less.

3. The numbering of the MCID are done at shipout, so no label/ref system is needed.

4. The code can do some marking automatically. Currently everything that has not been marked up by the document is marked as artifact.

6.2.2. Commands to mark content and chunks

Generic mode only

213 In generic mode is vital that the end command is executed on the same page as the begin command. So think carefully how to place them. For strategies how to handle paragraphs that split over pages see subsection 6.2.8. 213

214 `\tagmcbegin{⟨key-val-list⟩}` 214

215 `\tag_mc_begin:n{⟨key-val-list⟩}` 215

216 These commands insert the begin of the marked content code in the PDF. They don't start a paragraph. *They don't start a group.* Such markers should not be nested. The command will warn you if this happens. 216

217 The key-val list understands the following keys: 217

tag 218 This is required, unless you use the artifact key. The value of the key is normally one of the standard type listed in section 8 (without a slash at the begin, this is added by the code). It is possible to setup new tags, see the same section. The value of the key is expanded, so it can be a command. The expansion is passed unchanged to the PDF, so it should with a starting slash give a valid PDF name (some ascii with numbers like `H4` is fine). 218

artifact 219 This will setup the marked content as an artifact. The key should be used for content that should be ignored. The key can take one of the values pagination, pagination/header, pagination/footer, layout, page, background and notype (this is the default). Text in the header and footer should normally be marked with artifact=pagination or pagination/header, pagination/footer but simply artifact (as it is now done automatically) should be ok too. 219

220 It is not quite clear if rules and other decorative graphical objects needs to be marked up as artifacts. Acrobat seems not to mind if not, but PAC 3 complained. 220

221 The validators complain if some text is not marked up, but it is not quite clear if this is a serious problem. 221

lua mode only

222 The lua mode will mark up everything unmarked as `artifact=notype`. You can suppress this behaviour by setting the tagpdfsetup key `tagunmarked` to false. See section 5.2. 222

stash 223 Normally marked content will be stored in the "current" structure. This may not be what you want. As an example you may perhaps want to put a marginnote behind or before the paragraph it is in the tex-code. With this boolean key the content is marked but not stored in the kid-key of the current structure. 223

label 224 This key sets a label by which you can call the marked content later in another structure (if it has been stashed with the previous key). Internally the label name will start with `tagpdf-`. 224

alttext 225 This key inserts an `/Alt` value in the property dictionary of the BDC operator. See section 7. The value is handled as verbatim string, commands are not expanded but the value will be expanded first once (so works like the key `alttext-o` in previous versions

which has been removed). That means that you can do something like in the following listing and it will insert `\frac{a}{b}` (hex encoded) in the PDF.

```
\newcommand\myalttext{\frac{a}{b}}
\tagmcbegin{tag=P,alttext=\myalttext}
```

actualtext This key inserts an `/ActualText` value in the property dictionary of the BDC operator. See section 7. The value is handled as verbatim string, commands are not expanded but the value will be expanded first once (so works like the key `actualtext-o` in previous versions which has been removed).

That means that you can do something like in the following listing and and it will insert `X` (hex encoded) in the PDF.

```
\newcommand\myactualtext{X}
\tagmcbegin{tag=Span,actualtext=\myactualtext}
```

According to the PDF reference, `/ActualText` should only be used on marked content sequence of type Span. This is not enforced by the code currently.

raw This key allows you to add more entries to the properties dictionary. The value must be correct, low-level PDF. E.g. `raw=/Alt (Hello)` will insert an alternative Text.

`\tagmcend`

`\tag_mc:end`

These commands insert the end code of the marked content. They don't end a group and it doesn't matter if they are in another group as the starting commands. In generic mode both commands check if there has been a begin marker and issue a warning if not. In luamode it is often possible to omit the command, as the effect of the begin command ends with a new `\tagmcbegin` anyway.

`\tagmcuse`

`\tag_mc_use:n`

These commands allow you to record a marked content that you stashed away into the current structure. Be aware that a marked content can be used only once – the command will warn you if you try to use it a second time.

`\tag_mc_end_push:`

`\tag_mc_begin_pop:n{`⟨*key-val-list*⟩`}`

If there is an open mc chunk, the first command ends it and pushes its tag on a stack. If there is no open chunk, it puts -1 on the stack (for debugging). The second command removes a value from the stack. If it is different from -1 it opens a tag with it. The command is mainly meant to be used inside hooks and command definitions so there is only an expl3 version. Perhaps other content of the mc-dictionary (for example the Lang) needs to be saved on the stacked too.

239 \tagmcifinTF{⟨*true code*⟩}{⟨*false code*⟩} 239

240 \tag_mc_if_in:TF{⟨*true code*⟩}{⟨*false code*⟩} 240

241 These commands check if a marked content is currently open and allows you to e.g. add the end marker if yes. 241

242 In *generic mode*, where marked content command shouldn't be nested, it works with a global boolean. 242

243 In *lua mode* it tests if the mc-attribute is currently unset. You can't test the nesting level with it! 243

244 \tag_get:n{⟨*key word*⟩} 244

245 This command give back some variables. Currently the only working key words are `mc_tag` and `struct_tag`. 245

6.2.3. Luamode: global or not global – that is the question

Luamode mode only

246 In luamode the mc-commands set and unset an attribute to mark the nodes. One can view such an attribute like a font change or a color: they affect all following chars and glue nodes until stopped. 246

247 From version 0.6 to 0.82 the attributes were set locally. This had the advantage that the attributes didn't spill over in area where they are not wanted like the header and footer or the background pictures. But it had the disadvantage that it was difficult for an inner structure to correctly interrupt the outer mc-chunk if it can't control the group level. For example this didn't work due to the grouping inserted by the user: 247

```
\tagstructbegin{tag=P}
\tagmcbegin{tag=P}
  Start paragraph
  {% user grouping
   \tag_mc_end_push:
   \tagstructbegin{tag=Em}
   \tagmcbegin{tag=Em}
   \emph{Emphasized test}
   \tagmcend
   \tagstructend
   \tag_mc_begin_pop:n{}
  }% user grouping
 Continuation of paragraph
\tagmcend
\tagstructend
```

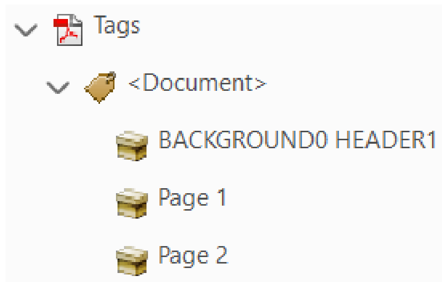
The reading order was then wrong, and the *emphasized text* moved in the structure at the end.

So starting with version 0.9 this has been reverted. The attribute is now global again. This solves the “interruption” problem, but has its price: Material inserted by the output routine must be properly guarded. For example

```
\RequirePackage{pdfmanagement-testphase}
\DeclareDocumentMetadata{uncompress}
\documentclass{article}
\usepackage{tagpdf}
\tagpdfsetup{activate,interwordspace=true}

\pagestyle{headings}
\begin{document}
\tagstructbegin{tag=Document}
\sectionmark{HEADER}
\AddToHook{shipout/background}{\put(5cm,-5cm){BACKGROUND}}
\tagmcbegin{tag=P}Page 1\newpage Page 2\tagmcend
\tagstructend
\end{document}
```

Here the header and the background code on the *first* page will be marked up as paragraph and added as chunk to the document structure. The header and the background code on the *second* page will be marked as artifact. The following figure shows how the tags looks like.



It is therefore from now on important to correctly markup such code. Header and footer typically should be artifacts. The LaTeX kernel hasn’t yet suitable hooks around header and footer to allow to automate this, but a first draft has been added with `pdfmanagement-testphase`. Starting with version 0.92 header and footer are marked as (simple) artifacts. If they contain code which needs a different markup it still must be added explicitly. With packages like `fancyhdr` or `scrlayer-scrpage` it is quite easy to add the needed code.

6.2.4. Tips

- Mark commands inside floats should work fine (but need perhaps some compilation rounds in generic mode).

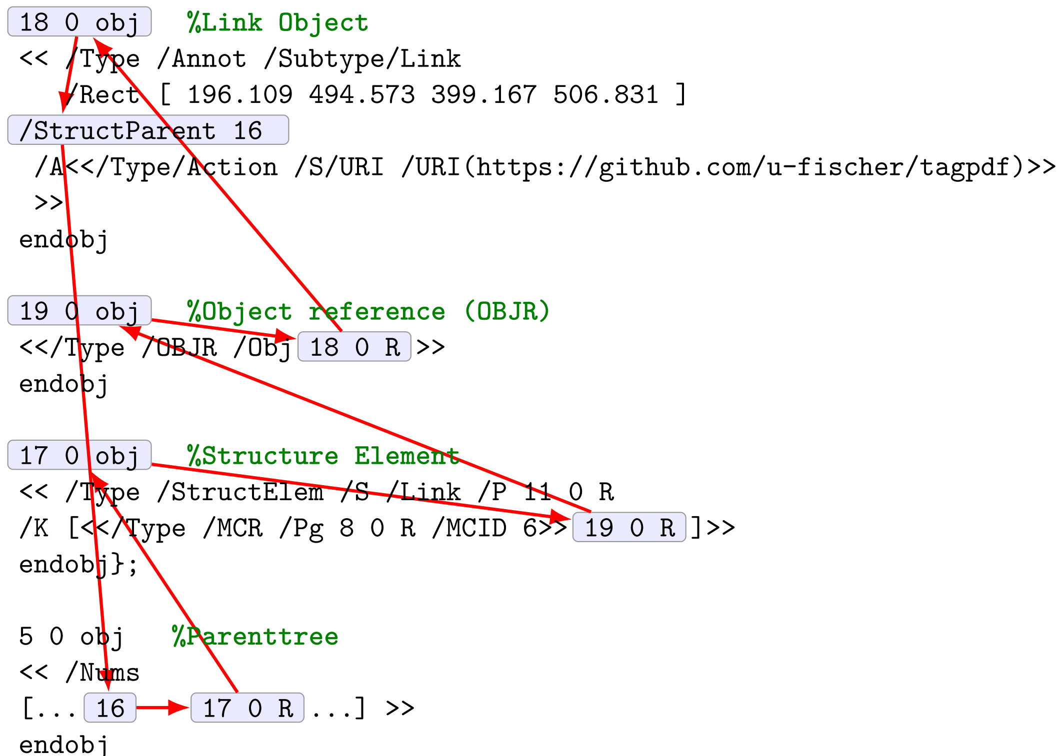


Figure 2: Structure needed for a link annotation

- In case you want to use it inside a `\savebox` (or some command that saves the text internally in a box): If the box is used directly, there is probably no problem. If the use is later, stash the marked content and add the needed `\tagmcuse` directly before oder after the box when you use it.
- Don't use a saved box with markers twice.
- If boxes are unboxed you will have to analyze the PDF to check if everything is ok.
- If you use complicated structures and commands (breakable boxes like the one from tcolorbox, multicol, many footnotes) you will have to check the PDF.

6.2.5. Header and Footer

Tagging header and footer is not trivial. At first on the technical side header and footer are typeset and attached to the page during the output routine and the exact timing is not really under control of the user. That means that when adding tagging there one has to be careful not to disturb the tagging of the main text—this is mostly important in luamode where the attributes are global and can easily spill over.

At second one has to decide about how to tag: in many cases header and footer can simply be ignored, they only contain information which are meant to visually guide the reader and so

are not relevant for the structure. This means that normally they should be tagged as artifacts. The PDF reference offers here a rather large number of options here to describe different versions of "ignore this". Typically the header and footer should get the type `Pagination` and this types has a number of subtypes like Header, Footer, PageNum. It is not yet known if any technology actually make use of this info. 258

259 But they can also contain meaningful content, for example an address. In such cases the content should be added to the structure (where?) but even if this address is repeated on every page at best only once. All this need some thoughts both from the users and the packages and code providing support for header and footers. 259

260 For now tagpdf added some first support for automatically tagging: Starting with version 0.92 header and footer are by default automatically marked up as (simple) artifacts. 260

261 With the key exclude-header-footer the behaviour can be changed: The value `false` disables the automatic tagging, the value `pagination` add additionally an `/Artifact` structure with the attribute `/Pagination`. 261

262 If some additional markup (or even a structure) is wanted, something like this should be used (here with the syntax of the `fancyhdr` package) to close the open mc-chunk and restart if after the content: 262

```
\ExplSyntaxOn
\cfoot{\leavevmode
  \tag_mc_end_push:
   \tagmcbegin{artifact=pagination/footer}
     \thepage
   \tagmcend
  \tag_mc_begin_pop:n{artifact}}
\ExplSyntaxOff
```

6.2.6. Links and other annotations

263 Annotations (like links or form field annotations) are objects associated with a geometric region of the page rather than with a particular object in its content stream. Any connection between a link or a form field and the text is based solely on visual appearance (the link text is in the same region, or there is empty space for the form field annotation) rather than on an explicitly specified association. 263

264 To connect such a annotation with the structure and so with surrounding or underlying text a specific structure has to be added, see 2: The annotation is added to a structure element as an object reference. It is not referenced directly but through an intermediate object of type OBJR. To the dictionary of the annotation a `/StructParent` entry must be added, the value is a number which is then used in the ParentTree to define a relationship between the annotation and the parent structure element. 264

265 To support this, `tagpdf` offers currently two commands 265

266 **`\tag_struct_parent_int:`** 266

267 This insert the current value of a global counter used to track such objects. It can be used to add the `/StructParent` value to the annotation dictionary. 267

268 `\tag_struct_insert_annot:nn`{⟨*object reference*⟩}{⟨*struct parent number*⟩} 268

269 This will insert the annotation described by the object reference into the current structure by creating the OBJR object. It will also add the necessary entry to the parent tree and increase the global counter referred to by `\tag_struct_parent_int:`. It does nothing if (structure) tagging is not activated. 269

270 Attention! As the second command increases the global counter at the end it changes the value given back by the first. That means that if nesting is involved care must be taken that the correct numbers is used. This should be easy to fulfil for most annotations, as there are boxes. There the second command should at best be used directly behind the annotation and it can make use of `\tag_struct_parent_int:`. For links nesting is theoretically possible, and it could be that future versions need more sophisticated handling here. 270

271 In environments which process their content twice like tabularx or align it would be the best to exclude the second command from the trial step, but this will need better support from these environments. 271

272 Typically using this commands is not often needed: Since version 0.81 `tagpdf` already handles (unnested) links, and form fields created with the `l3pdffield-testphase` package will be handle by this package. 272

273 The following listing shows low-level to create link where the two commands are used: 273

```
\pdfextension startlink
   attr
    {
    /StructParent \tag_struct_parent_int: %<----
    }
   user {
        /Subtype/Link
        /A
         <<
          /Type/Action
          /S/URI
          /URI(http://www.dante.de)
         >>
       }
   This is a link.
\pdfextension endlink
\tag_struct_insert_annot:xx {\pdfannot_link_ref_last:}{\tag_struct_parent_int:}
```

6.2.7. Math

Math is a problem. I have seen an example where *every single symbol* has been marked up with tags from MathML along with an `/ActualText` entry and an entry with alternate text which describes how to read the symbol. The PDF then looked like this

```
/mn <</MCID 6 /ActualText<FEFF0034>/Alt( : open bracket: four )>>BDC
...
/mn <</MCID 7 /ActualText<FEFF0033>/Alt( third s )>>BDC
...
/mo <</MCID 8 /ActualText<FEFF2062>/Alt( times )>>BDC
```

If this is really the way to go one would need some script to add the mark-up as doing it manually is too much work and would make the source unreadable – at least with pdflatex and the generic mode. In lua mode is it possible to hook into the `mlist_to_hlist` callback and add marker automatically. Some first implementation is done by Marcel Krüger in the luamml project.

But I'm not sure that this is the best way to do math. It looks rather odd that a document should have to tell a screen reader in such detail how to read an equation. It would be much more efficient, sensible and flexible if a complete representation of the equation in mathML could be stored in the PDF and the task how to read this aloud delegated to the screen reader. As PDF 2.0 introduced associated files it is probable that this will be the way to go but more investigations are needed here.

See also section 7 for some more remarks and tests.

6.2.8. Split paragraphs

Generic mode only

A problem in generic mode are paragraphs with page breaks. As already mentioned the end marker `EMC` must be added on the same page as the begin marker. But it is in pdflatex *very* difficult to inject something at the page break automatically. One can manipulate the shipout box to some extend in the output routine, but this is not easy and it gets even more difficult if inserts like footnotes and floats are involved: the end of the paragraph is then somewhere in the middle of the box.

So with pdflatex in generic mode one until now had to do the splitting manually.

The example `mc-manual-para-split` demonstrates how this can be done. The general idea was to use `\vadjust` in the right place:

```
\tagmcbegin{tag=P}
...
fringilla, ligula wisi commodo felis, ut adipiscing felis dui in
enim. Suspendisse malesuada ultrices ante.% page break
\vadjust{\tagmcend\pagebreak\tagmcbegin{tag=P}}
Pellentesque scelerisque
...
sit amet, lacus.\tagmcend
```

Starting with version 0.91 there is code which tries to resolve this problem. Basically it works like this: every mc-command issues a mark command (actually two slightly different). When the page is built in the output routine this mark commands are inspected and from them LaTeX can deduce if there is a mc-chunk which must be closed or reopened.

Please note

- The code requires the pdfmanagement-testphase version v0.95i.
- Typically you will need more compilations than previously, don't rely on the rerun messages, but if something looks wrong rerun.
- The code relies on that related `\tagmcbegin` and `\tagmcend` are in the same boxing level. If one is in a box (which hides the marks) and the other in the main galley, things will go wrong.

6.2.9. Automatic tagging of paragraphs

`\tagpdfparaOn`

`\tagpdfparaOff`

Another feature that emerged from the LaTeX tagged PDF project are hooks at the begin and end of paragraphs. `tagpdf` makes use of these hooks to tag paragraphs. This can be activated/deactivated (also locally) with options of `\tagpdfsetup` or with the two commands above. *This is very experimental and it requires a new LaTeX!*

The automatic tagging require that for every begin of a paragraph with the begin hook code there a corresponding end with the closing hook code. This can fail, e.g if a `vbox` doesn't correctly issue a `\par` at the end. If this happens the tagging structure can get every confused. It is therefore needed to check the structure carefully as currently no checks are implemented to check this automatically.

6.3. Task 2: Marking the structure

The structure is represented in the PDF with a number of objects of type `StructElem` which build a tree: each of this objects points back to its parent and normally has a number of kid elements, which are either again structure elements or – as leafs of the tree – the marked contents chunks marked up with the `tagmc`-commands. The root of the tree is the `StructTreeRoot`.

6.3.1. Structure types

The tree should reflect the *semantic* meaning of the text. That means that the text should be marked as section, list, table head, table cell and so on. A number of standard structure types is predefined, see section 8 but it is allowed to create more. If you want to use types of your own you must declare them. E.g. this declares two new types `TAB` and FIG and bases them on `P`:

```
\tagpdfsetup{
  add-new-tag = TAB/P,
  add-new-tag = FIG/P,
          }
```

6.3.2. Sectioning

The sectioning units can be structured in two ways: a flat, html-like and a more (in pdf/UA2 basically deprecated) xml-like version. The flat version creates a structure like this:

```
<H1>section header</H1>
<P> text</P>
<H2>subsection header</H2>
...
```

So here the headings are marked according their level with `H1`, `H2`, etc.

In the xml-like tree the complete text of a sectioning unit is surrounded with the `Sect` tag, and all headers with the tag `H`. Here the nesting defines the level of a sectioning heading.

```
<Sect>
 <H>section heading</H>
 <P> text</p>
 <Sect>
   <H>subsection heading</H>
   ...
 </Sect>
</Sect>
```

The flat version is more LaTeX-like and it is rather straightforward to patch `\chapter`, `\section` and so on to insert the appropriates `H`… start and end markers. The xml-like tree is more difficult to automate. If such a tree is wanted I would recommend to use – like the context format – explicit commands to start and end a sectioning unit.

6.3.3. Commands to define the structure

The following commands can be used to define the tree structure:

`\tagstructbegin`{⟨*key-val-list*⟩}

`\tag_struct_begin:n{`⟨*key-val-list*⟩`}`

These commands start a new structure. They don't start a group. They set all their values globally.

The key-val list understands the following keys:

tag This is required. The value of the key is normally one of the standard types listed in section 8. It is possible to setup new tags/types, see the same section. The value can also be of the form `type/NS`, where `NS` is the shorthand of a declared name space. Currently the names spaces `pdf`, `pdf2`, `mathml` and `user` are defined. This allows to use a different name space than the one connected by default to the tag. But normally this should not be needed.

stash Normally a new structure inserts itself as a kid into the currently active structure. This key prohibits this. The structure is nevertheless from now on "the current active structure" and parent for following marked content and structures.

label This key sets a label by which you can use the structure later in another structure. Internally the label name will start with `tagpdfstruct-`.

alttext This key inserts an `/Alt` value in the dictionary of structure object, see section 7. The value is handled as verbatim string and hex encoded. The value will be expanded first once (so works like the key `alttext-o` in previous versions which has been removed). That means that you can do something like this:

```
\newcommand\myalttext{\frac{a}{b}}
\tagstructbegin{tag=P,alttext=\myalttext}
```

and it will insert `\frac{a}{b}` (hex encoded) in the PDF. In case that the text begins with a command that should not be expanded protect it e.g. with a `\empty`.

actualtext This key inserts an `/ActualText` value in the dictionary of structure object, see section 7. The value is handled as verbatim string, The value will be expanded first once (so works like the key `alttext-o` in previous versions which has been removed). That means that you can do something like this:

```
\newcommand\myactualtext{X}
\tagstructbegin{tag=P,actualtext=\myactualtext}
```

and it will insert `X` (hex encoded) in the PDF. In case that the text begins with a command that should not be expanded protect it e.g. with a `\empty`

attribute This key takes as argument a comma list of attribute names (use braces to protect the commas from the external key-val parser) and allows to add one or more attribute dictionary entries in the structure object. As an example

```
\tagstructbegin{tag=TH,attribute= TH-row}
```

See also section 6.3.5.

attribute-class This key takes as argument a comma list of attribute names (use braces to protect the commas from the external key-val parser) and allows to add them as attribute classes to the structure object. As an example

```
\tagstructbegin{tag=TH,attribute-class= TH-row}
```

See also section 6.3.5.

title This key allows to set the dictionary entry `/Title` in the structure object. The value is handled as verbatim string and hex encoded. Commands are not expanded.

title-o This key allows to set the dictionary entry `/Title` in the structure object. The value is expanded once and then handled as verbatim string like the title key.

AF This key allows to reference an associated file in the structure element. The value should be the name of an object pointing to the `/Filespec` dictionary as expected by `\pdf_object_ref:n` from a current `l3kernel`. For example:

```
\group_begin:
\pdfdict_put:nnn {l_pdffile/Filespec} {AFRelationship}{/Supplement}
\pdffile_embed_file:nnn{example-input-file.tex}{}{tag/AFtest}
\group_end:
\tagstructbegin{tag=P,AF=tag/AFtest}
```

As shown, the wanted AFRelationship can be set by filling the dictionary with the value. The mime type is here detected automatically, but for unknown types it can be set too. See the `l3pdffile` documentation for details. Associated files are a concept new in PDF 2.0, but the code currently doesn't check the pdf version, it is your responsability to set it (this can be done with the `pdfversion` key in `\DeclareDocumentMetadata`).

root-AF This key allows to reference an associated file in the root structure element. Using the root can be e.g. useful to add a css-file. When converting the pdf to a html with e.g. ngpdf this css-file is then referenced in the head of the html.

AFinline This key allows to embed an associated file with inline content. The value is some text, which is embedded in the PDF as a text file with mime type text/plain.

```
\tagstructbegin{tag=P,AFinline=Some extra text}
```

AFinline-o This is like verb+AFinline+, but it expands the value once.

lang This key allows to set the language for a structure element. The value should be a bcp-identifier, e.g. `de-De`.

ref This key allows to add references to other structure elements, it adds the `/Ref` array to the structure. The value should be a comma separated list of structure labels set with the `label` key. e.g. `ref={label1,label2}`.

E This key sets the `/E` key, the expanded form of an abbreviation or an acronym (I couldn't think of a better name, so I sticked to E).

`\tagstructend`

323 `\tag_struct_end:` 323

324 These commands end a structure. They don't end a group and it doesn't matter if they are in another group as the starting commands. 324

325 `\tagstructuse{⟨label⟩}` 325

326 `\tag_struct_use:n{⟨label⟩}` 326

327 These commands insert a structure previously stashed away as kid into the currently active structure. A structure should be used only once, if the structure already has a parent you will get a warning. 327

6.3.4. Root structure

328 A document should have at least one structure which contains the whole document. A suitable tag is `Document` or `Article`. I'm considering to automatically inserting it. 328

6.3.5. Attributes and attribute classes

329 Structure Element can have so-called attributes. A single attribute is a dictionary(or a stream but this is currently not supported by the package as I don't know an use-case) with at least the required key `/O` (for "Owner" which describes the scope the attribute applies too. As an example here an attribute that can be attached to tabular header (type TH) and adds the info that the header is a column header: 329

```
<</O /Table /Scope /Column>>
```

330 One or more such attributes can be attached to a structure element. It is also possible to store such an attribute under a symbolic name in a so-called "ClassedMap" and then to attach references to such classes to a structure. 330

331 To use such attributes you must at first declare it in `\tagpdfsetup` with the key `newattribute`. This key takes two argument, a name and the content of the attribute. The name should be a sensible key name, the content a dictionary. 331

```
\tagpdfsetup
 {
  newattribute =
   {TH-col}{/O /Table /Scope /Column},
  newattribute =
   {TH-row}{/O /Table /Scope /Row},
  }
```

332 Attributes are only written to the PDF when used, so it is not a problem to predeclare a number of standard attributes. 332

333 It is your responsability that the content of the dictionary is valid PDF and that the values are sensible! 333

334 Attributes can then be used with the key attribute or attribute-class which both take a comma list of attribute names as argument: 334

```
\tagstructbegin{tag=TH,
 attribute-class= {TH-row,TH-col},
 attribute = {TH-row,TH-col},
 }
```

6.4. Task 3: tree Management

335 When all the document content has been correctly marked and the data for the trees has been collected they must be flushed to the PDF. This is done automatically (if the package has been activated) with an internal command in an end document hook. 335

336 `__tag_finish_structure:` 336

337 This will hopefully write all the needed objects and values to the PDF. (Beside the already mentioned `StructTreeRoot` and `StructElem` objects, additionally a so-called `ParentTree` is needed which records the parents of all the marked contents bits, a `Rolemap`, perhaps a `ClassMap` and object for the attributes, and a few more values and dictionaries). 337

6.5. A fully marked up document body

338 The following shows the marking needed for a section, a sentence and a list with two items. It is obvious that one wouldn't like to have to do this for real documents. If tagging should be usable, the commands must be hidden as much as possible inside suitable LaTeX commands and enviroments. 338

```
\begin{document}

\tagstructbegin{tag=Document}

 \tagstructbegin{tag=Sect}
  \tagstructbegin{tag=H}
   \tagmcbegin{tag=H} %avoid page break!
    \section{Section}
   \tagmcend
  \tagstructend
  \tagstructbegin{tag=P}
   \tagmcbegin{tag=P,raw=/Alt (x)}
    a paragraph\par x
```

```
   \tagmcend
  \tagstructend

 \tagstructbegin{tag=L} %List
  \tagstructbegin{tag=LI}
   \tagstructbegin{tag=Lbl}
    \tagmcbegin{tag=Lbl}
    1.
    \tagmcend
   \tagstructend
   \tagstructbegin{tag=LBody}
    \tagmcbegin{tag=P}
    List item body
    \tagmcend
   \tagstructend %lbody
  \tagstructend %Li

  \tagstructbegin{tag=LI}
   \tagstructbegin{tag=Lbl}
    \tagmcbegin{tag=Lbl}
    2.
    \tagmcend
   \tagstructend
   \tagstructbegin{tag=LBody}
    \tagmcbegin{tag=P}
    another List item body
    \tagmcend
   \tagstructend %lbody
  \tagstructend %Li
  \tagstructend %L

  \tagstructend %Sect
\tagstructend %Document
\end{document}
```

6.6. Lazy and automatic tagging

339 A number of features of PDF readers need a fully tagged PDF. As an example screen readers tend to ignore alternative text (see section 7) if the PDF is not fully tagged. Also reflowing a PDF only works for me (even if real space chars are in the PDF) if the PDF is fully tagged. 339

340 This means that even if you don't care about a proper structure you should try to add at least some minimal tagging. With pdflatex this is not easy due to the page break problem. But with lualatex you can use an `Document` structure and inside it rather large mc-chunks. This minimizes the needed work. 340

One could ask if in lua mode the code couldn't try to mark up unmarked parts e.g. as P-type chunks, like it marks them up as artifacts currently. Sadly this is not so easy, as it is quite difficult to reliably identify the structure and the place in the kids array where such chunks belongs too. I also don't think that it is really needed. It is not so difficult to define user macros which e.g. opens a structure and start an mc-chunk or which close an open mc-chunk before issuing the next `\tagmcbegin`.

6.7. Adding tagging to commands

As mentioned above the mc-markers should not be nested. Basically you write:

```
\tagmcbegin{..}some text ...\tagmcend
<optional structure commands>
\tagmcbegin{..}some other text\tagmcend
```

This is quite workable as long as you mark everything manually. But how to write commands, e.g for a tabular or a graphic, that do tagging automatically without breaking the flow and the structure?

7. Alternative text, ActualText and text-to-speech software

The PDF format allows to add alternative text through the /Alt and the /ActualText key. Both can be added either to the marked content in the page stream or to the object describing the structure.

The value of /ActualText (inserted by `tagpdf` with actualtext) is meant to replace single characters or rather small pieces of text. It can be used also without any tagging (e.g. with the package accsupp). If the PDF reader support this (adobe reader does, sumatra not) one can change with it how a piece of text is copied and pasted e.g. to split up a ligature.

/Alt (inserted by `tagpdf` with alttext) is a key to improve accessibility: with it one can add to a picture or something else an alternative text.

The file `ex-alt-actualtext.tex` shows some experiments I made with with both keys and text-to-speech software (the in-built of adobe and nvda). To sum them up:

- The keys have an impact on text-to-speech software only if the document is fully tagged.
- /ActualText should be at best used around short pieces of marked content.
- /Alt is used at best with a structure – this avoids problems with luatex where marked contents blocks can be split over pages.
- To some extend one can get a not so bad reading of math with the alternative text.

8. Standard types and new tags

The tags used to describe the type of a structure element can be rather freely chosen. PDF 1.7 and earlier only requires that in a tagged PDF all types should be either from a known set of standard types or are “role mapped” to such a standard type. Such a role mapping is a simple key-value in the RoleMap dictionary.

So instead of `H1` the type `section` could be used. The role mapping can then be declared with the `add-new-tag` key:

```
\tagpdfsetup{add-new-tag = section/H1}
```

In PDF 2.0 the situation is a bit more complicated. At first PDF 2.0 introduced *name spaces*. That means that a type can have more than one “meaning” depending on the name space it belongs to. `section (name space A)` and `section (name space B)` are two different types.

At second PDF 2.0 still requires that a tagged PDF maps all types to a standard type, but now there are three sets of standard types (The meanings of the PDF types can be looked up in the PDF-references [1, 4]):

1. The *standard structure namespace for PDF 1.7*, also called the *default standard structure namespace*. The public name of the namespace is `tag/NS/pdf`. This can be used to reference the namespace e.g. in attributes. These are the structure names from PDF 1.7: Document, Part, Sect, Div, Caption, Index, NonStruct, H, H1, H2, H3, H4, H5, H6, P, L, LI, Lbl, LBody, Table, TR, TH, TD, THead, TBody, TFoot, Span, Link, Annot, Figure, Formula, Form, Ruby, RB, RT, Warichu, WT, WP, Artifact, Art, BlockQuote, TOC, TOCI, Index, Private, Quote, Note, Reference, BibEntry, Code
2. The *standard structure namespace for PDF 2.0*. The public name of the namespace is `tag/NS/pdf2`. This can be used to reference the namespace e.g. in attributes. These are more or less same types as in PDF. The following types have been removed from this set: Art, BlockQuote, TOC, TOCI, Index, Private, Quote, Note, Reference, BibEntry, Code and the following are new:
DocumentFragment, Aside, H7, H8, H9, H10, Title, FENote, Sub, Em, Strong, Artifact
3. MathML 3.0 as an *other namespaces*. The public name of the namespace is `tag/NS/mathml`. This can be used to reference the namespace e.g. in attributes. There are nearly 200 types in this name space, so I refrain from listing them here.

To allow to this more complicated setup the syntax of the `add-new-tag` key has been extended. It now takes as argument a key-value list with the following keys. A normal document shouldn't need the extended syntax, the simple syntax `section/H1` should in most cases do the right thing.

tag This is the name of the new type as it should then be used in `\tagstructbegin`.

namespace This is the namespace of the new type. The value should be a shorthand of a namespace. The allowed values are currently `pdf`, `pdf2`, `mathml` and `user`. The default value (and recommended value for a new tag) is `user`. The public name of the

user namespace is `tag/NS/user`. This can be used to reference the namespace e.g. in attributes. 361

role 362 This is the type the tag should be mapped too. In a PDF 1.7 or earlier this is normally a type from the `pdf` set, in PDF 2.0 from the `pdf`, `pdf2` and `mathml` set. It can also be a user type, or a still unknown type. The PDF format allows mapping to be done transitively. But you should be aware that tagpdf can't (or more precisely won't) check such unusual role mapping. It lies in the responsability of the author to ensure here that every type is correctly role mapped. 362

role-namespace 363 If the role is a known type the default value is the default namespace: `pdf2` for all types in this set, `pdf` for the type which exist only in PDF 1.7, `mathml` for the MathML types, and for previously defined user types whatever namespace has been set there. If the role is unknown, `user` is used and the code hopes that the type will be defined later. 363

unknown key 364 An unknown key is interpreted as a `tag/role`, this preserves the old syntax. So this two calls are equivalent: 364

```
\tagpdfsetup{add-new-tag = section/H1}
\tagpdfsetup{add-new-tag = {tag=section,role=H1}}
```

365 The exact effects of the key depends on the PDF version. With PDF 1.7 or older the namespace keys are ignored, with PDF 2.0 the namespace keys are use to setup the correct rolemaps. The `namespace` key is also used to define the default namespace if the type is used as a role or as tag in a structure. 365

9. “Real” space glyphs

366 TeX uses only spaces (horizontal movements) to separate words. That means that a PDF reader has to use some heuristic when copying text or reflowing the text to decide if a space is meant as a word boundary or e.g. as a kerning. Accessible document should use real space glyphs (U+0032) from a font in such places. 366

367 With the key interwordspace you can activate such space glyphs. 367

368 With pdftex this will simply call the primitive `\pdfinterwordspaceon`. pdftex will then insert at various places a char from a font called dummy-space. Attention! This means that at every space there are additional font switches in the PDF: from the current font to the dummy-space font and back again. This will make the PDF larger. As `\pdfinterwordspaceon` is a primitive function it can't be fine tuned or adapted. You can only turn it on and off and insert manually such a space glyph with `\pdffakespace`. 368

369 With luatex (in luamode) `interwordspace` is implemented with a lua-function which is inserted in two callbacks and marks up the places where it seems sensible to inter a space glyph. Later in the process the space glyphs are injected – the code will take the glyph from the current font if this has a space glyph or switch to the default latin modern font. The current

code works reasonable well in normal text. `interwordspace` can be used without actually tagging a document. 369

370 The key show-spaces will show lines at the places where in lua mode spaces are inserted and so can help you to find problematic places. For listings – which have a quite specific handling of spaces – you can find a suggestion in the example `ex-space-glyph-listings`. 370

371 *Attention:* Even with real spaces copy& pasting of code doesn't need to give the correct results: you get spaces but not necessary the right number of spaces. The PDF viewers I tried all copied four real space glyphs as one space. I only got the four spaces with the export to text or xml in the AdobePro. 371

372 `\pdffakespace` 372

373 This is in pdftex a primitive. It inserts the dummy space glyph. `tagpdf` defines this command also for luatex – attention if can perhaps insert break points. 373

10. Structure destinations

374 Standard destinations (anchors for internal links) consist of a reference to a page in the pdf and instructions how to display it—typically they will put a specific coordinate in the left top corner of the viewer and so give the impression that a link jumped to the word in this place. But in reality they are not connected to the content. 374

375 Starting with pdf 2.0 destinations can in a tagged PDF also point to a structure (to a `/StructElem` object). GoTo links can then additionally to the `/D` key which points to a standard page destination also point to such a structure destination with an `/SD` key. Programs that e.g. convert such a PDF to html can then create better links. (According to the reference, PDF-viewer should prefer the structure destination over the page destination, but as far as it is known this isn't done yet.) 375

376 Currently structure destinations (and GoTo links making use of it) could natively only be created with the dvipdfmx backend. With pdftex and lualatex it was only possible to create a restricted type which used only the "Fit" mode. Starting with TEXlive 2022 (earlier in miktex) both engines will knew new keywords which allow to create structure destination easily. 376

377 Support for this has been already added to the pdfmanagement and `tagpdf` will make use of it if possible. In most cases it should simply work, but one should be aware that as one now has a destination that is actually tied to the content it gets more important to actually consider the context and the place where such destinations are created. It now makes a difference if the destination is created before the structure is opened or after so in some cases code that place destinations should be changed to place them inside the structure they belong too. . One also has to consider the pages connected to the destinations: The structure destination is bound to the page where the structure *begins*, if this differ from the page of the page destination (e.g. if the destination is created by a `\phantomsection` in the middle of a longer paragraph) then be necessary to surround destinations with a dummy structure (a Span or an Artifact) to get the right page number. 377

11. Accessibility is not only tagging

A tagged PDF is needed for accessibility but this is not enough. As already mentioned there are more requirements:

- The language must be declared by adding a `/Lang xx-XX` to the PDF catalog or – if the language changes for a part of the text to the structure or the marked content. Setting the document language can be rather easily done with existing packages. With the new PDF resource management it should be done with `\pdfmanagement_add:nnn{Catalog}{Lang}{(en-US)}`. For settings in marked content and structure I will have to add keys.
- All characters must have an unicode representation or a suitable alternative text. With lualatex and open type (unicode) fonts this is normally not a problem. With pdflatex it could need

```
\input{glyphtounicode}
\pdfgentounicode=1
```

  and perhaps some `\pdfglyphtounicode` commands.
- Hard and soft hyphen must be distinct.
- Spaces between words should be space glyphs and not only a horizontal movement. See section 9.
- Various small infos must be present in the catalog dictionary, info dictionary and the page dictionaries, e.g. metadata like title.

If suitable I will add code for this tasks to this packages. But some of them can also be done already with existing packages like hyperref, hyperxmp, pdfx.

12. Debugging

While developing commands and tagging a document, it can be useful to get some info about the current structure. For this a show command is provided

`\ShowTagging{⟨key-val⟩}`

This command takes as argument a key-val list which implements a number of show options.

mc-data This key is relevant for luamode only. It shows the data of all mc-chunks created so far. It is accurate only after shipout, so typically should be issued after a newpage. The value is a positive integer and sets the first mc-shown. If no value is given, 1 is used and so all mc-chunks created so far are shown.

mc-current This key shows the number and the tag of the currently open mc-chunk. If no chunk is open it shows only the state of the absolute counter. It works in all mode, but the output in luamode looks different.

struct-stack This key shows the current structure stack. Typically it will contain at least `root` and `Document`. With the value `log` the info is only written to the log-file, `show` stops the compilation and shows on the terminal. If no value is used, then the default is `show`.

13. To-do

- Add commands and keys to enable/disable the checks.
- Check/extend the code for language tags.
- Think about math (progress: examples using luamml, associated files exists).
- Think about Links/Annotations (progress: mostly done, see section 6.2.6 and the code in `l3pdffield`)
- Keys for alternative and actualtext. How to define the input encoding? Like in Accsupp? (progress: keys are there, but encoding interface needs perhaps improving)
- Check twocolumn documents
- Examples
- Write more Tests
- Write more Tests
- Unicode
- Hyphenation char
- Think about included (tagged) PDF. Can one handle them?
- Improve the documentation (progress: it gets better)
- Tag as proof of concept the documentation (nearly done)
- Document the code better (progress: mostly done)
- Create dtx (progress: done)
- Find someone to check and improve the lua code
- Move more things to lua in the luamode
- Find someone to check and improve the rest of the code
- Check differences between PDF versions 1.7 and 2.0. (progress: WIP, namespaces done)
- bidi?

References

[1] Adobe Systems Incorporated. *Document management – Portable document format – Part 1: PDF 1.7*. 1st ed. July 1, 2008. URL: https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf (visited on 04/18/2021).

[2] Adobe Systems Incorporated. *PDF Reference, sixth edition*. 2006. URL: https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf.

[3] Dual Lab. *Next-Generation PDF*. URL: http://ngpdf.com/.

[4] International Standard. *ISO 32000-2:2020(en). Document management — Portable document format — Part 2: PDF 2.0*. 2nd ed. Dec. 2020. URL: https://www.iso.org/obp/ui/#iso:std:iso:32000:-2:ed-2:v1:en (visited on 04/18/2021).

[5] TeX User Group. *PDF accessibility and PDF standards*. URL: https://tug.org/twg/accessibility/.

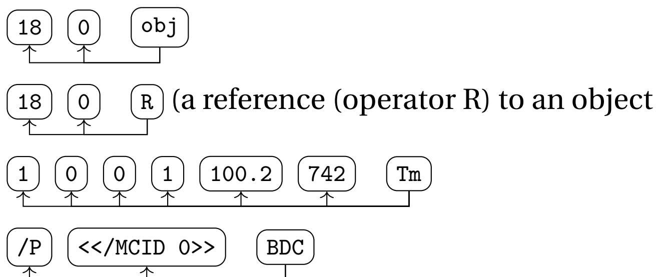
[6] veraPDF consortium. *veraPDF*. URL: http://verapdf.org/.

[7] Zugang für alle – Schweizerische Stiftung zur behindertengerechten Technologienutzung. *PDF Accessibility Checker (PAC 3)*. URL: http://www.access-for-all.ch/ch/pdf-werkstatt/pdf-accessibility-checker-pac.html (visited on 07/05/2018).

A. Some remarks about the PDF syntax

This is not meant as a full reference only as a background to make the examples and remarks easier to understand.

postfix notation PDF uses in various places postfix notation. This means that the operator is behind its arguments:



Names PDF knows a sort of variable called a “name”. Names start with a slash and may include any regular characters, but not delimiter or white-space characters. Uppercase and lowercase letters are considered distinct: `/A` and `/a` are different names. `/.notdef` and `/Adobe#20Green` are valid names.

Quite a number of the options of `tagpdf` actually define such a name which is later added to the PDF. I recommend *strongly* not to use spaces and exotic chars in such names. While it is possible to escape such names it is rather a pain when moving them through the various lists and commands and quite probably I forgot some place where it is needed.

Strings There are two types of strings: *Literal strings* are enclosed in round parentheses. They normally contain a mix of ascii chars and octal numbers:

```
(gr\374\377ehello[]\050\051).
```

Hexadezimal strings are enclosed in angle brackets. They allow for a representation of all characters the whole unicode ranges. This is the default output of lualatex.

```
<003B00600243013D0032>.
```

Arrays Arrays are enclosed by square brackets. They can contain all sort of objects including more arrays. As an example here an array which contains five objects: a number, an object reference, a string, a dictionary and another array. Be aware that despite the spaces `15 0 R` is *one* element of the array.

```
[0 15 0 R (hello) <</Type /X>> [1 2 3]]
```

Dictionaries Dictionaries are enclosed by double angle brackets. They contain key-value pairs. The key is always a name. The value can be all sort of objects including more dictionaries. It doesn't matter in which order the keys are given.

Dictionaries can be written all in one line:
```
<</Type/Page/Contents 3 0 R/Resources 1 0 R/Parent 5 0 R>>
```
but at least for examples a layout with line breaks and indentation is more readable:

```
<<
  /Type /Page
  /Contents 3 0 R
  /Resources 1 0 R
  /MediaBox [0 0 595.276 841.89]
  /Parent 5 0 R
>>
```

(indirect) objects These are enclosed by the keywords `obj` (which has two numbers as prefix arguments) and `endobj`. The first argument is the object number, the second a generation number – if a PDF is edited objects with a larger generation number can be added. As with pdflatex/lualatex the PDF is always new we can safely assume that the number is always 0. Objects can be referenced in other places with the `R` operator. The content of an object can be all sort of things.

streams A stream is a sequence of bytes. It can be long and is used for the real content of PDF: text, fonts, content of graphics. A stream starts with a dictionary which at least sets the `/Length` name to the length of the stream followed by the stream content enclosed by the keywords `stream` and `endstream`.

Here an example of a stream, an object definition and reference. In the object 2 (a page object) the `/Contents` key references the object 3 and this then contains the text of the page in a stream. `Tf`, `Tm` and `TJ` are (postfix) operators, the first chooses the font with the name `/F15` at the size 10.9, the second displaces the reference point on the page and the third inserts the text.

```
% a page object (shortened)
2 0 obj
  <<
    /Type/Page
    /Contents 3 0 R
    /Resources 1 0 R
    ...
  >>
endobj

%the /Contents object (/Length value is wrong)
3 0 obj
 <</Length 153 >>
 stream
  BT
   /F15 10.9 Tf 1 0 0 1 100.2 746.742 Tm [(hello)]TJ
  ET
 endstream
endobj
```

[435] In such a stream the BT–ET pair encloses texts while drawing and graphics are outside of such pairs. [435]

Number tree [436] This is a more complex data structure that is meant to index objects by numbers. In the core is an array with number-value pairs. A simple version of number tree which has the keys 0 and 3 is [436]

```
6 0 obj
<<
  /Nums [
         0 [ 20 0 R 22 0 R]
         3 21 0 R
       ]
>>
endobj
```

[437] This maps 0 to an array and 2 to the object reference 21 0 R. Number trees can be split over various nodes – root, intermediate and leaf nodes. We will need such a tree for the *parent tree*. [437]