The tagpdf package, v1.0j

Ulrike Fischer*

2026-11-15

This package is not meant for direct use in document preambles. It is loaded automatically if `\DocumentMetadata` is used.
The package started in 2018 as a tool to *research* tagging. It is now the base of the code developed in the `latex-lab` bundle for the Tagged PDF project (i.e., loaded by that code) https://latex3.github.io/tagging-project/.
The package is developed and improved in parallel with the LaTeX kernel, the code in the `latex-lab` bundle (part of the core LaTeX distribution), the `pdfmanagement` package (the LaTeX PDF management bundle) and the L3 programming layer (part of the LaTeX format). That means you must ensure that all these components are up-to-date and in sync with each other.
This package quite probably still contains some bugs. It is in some parts quite slow because the code currently prefers readability over speed. At some point in the future its code will be integrated into the LaTeX format and then this package will disappear.
Because of its function as a research and development tool it is important to understand that this package can still change in incompatible ways from one version to the next.
You need some knowledge about TeX, PDF and perhaps even Lua to use its commands directly.

Issues, comments and suggestions can be added as issues to these two github trackers:

https://github.com/latex3/tagging-project
or https://github.com/latex3/tagpdf

Contents

1 Introduction **3**
1.1 Tagging and accessibility 4
1.2 Engines and modes 4
1.3 References and target PDF version 5
1.4 Validation 5
1.5 Examples wanted! 6
1.6 Proof of concept: the tagging of the documentation itself 6

*fischer@troubleshooting-tex.de

2 Loading **7**
2.1 Modes and package options . . . 9
2.2 Setup and activation . . . 9

3 Tagging **13**
3.1 Tag names . . . 14
3.2 Three tasks . . . 14
3.3 Task 1: Marking the chunks: the mark-content-step . . . 15
3.3.1 Generic mode versus Lua mode in the mc-task . . . 17
3.3.2 Commands to mark content and chunks . . . 17
3.3.3 Retrieving data . . . 20
3.3.4 Luamode: global or not global – that is the question . . . 21
3.3.5 Tips . . . 22
3.3.6 Header and Footer . . . 22
3.3.7 Links and other annotations . . . 24
3.3.8 Math . . . 25
3.3.9 Split paragraphs . . . 26
3.3.10 Automatic tagging of paragraphs . . . 27
3.4 Task 2: Marking the structure . . . 28
3.4.1 Structure types . . . 28
3.4.2 Sectioning . . . 29
3.4.3 Commands to define the structure . . . 29
3.4.4 Updating structure keys . . . 33
3.4.5 Root structure . . . 33
3.4.6 Attributes and attribute classes . . . 33
3.4.7 Checking the structure . . . 34
3.5 Task 3: Tree management . . . 34
3.6 A fully marked up document body . . . 34
3.7 Interrupting the tagging . . . 35
3.8 Adding tagging to commands . . . 36

4 Alternative text, ActualText and text-to-speech software **37**

5 Standard types and new tags **37**
5.1 The `latex` namespace . . . 39
5.2 The `user` namespace . . . 39
5.3 Fallback RoleMap . . . 39
5.4 Mathml . . . 39

6 Checking parent-child rules **39**

7 “Real” space glyphs **41**

8 Structure destinations **41**

9 Storing and reusing boxes **42**
9.1 Boxes without tagging commands . . . 42
9.2 Boxes with tagging commands . . . 43
9.3 Detecting tagging commands . . . 43
9.4 Putting everything together . . . 44

10 Accessibility is not only tagging **44**

11 CSS style definition for derivation to html **45**

12 Debugging **46**

13 To-do **46**

14 History **47**
14.1 Changes in 0.3 . . . 47
14.2 Changes in 0.5 . . . 47
14.3 Changes in 0.6 . . . 47
14.4 Changes in version 0.61 . . . 47
14.5 Changes in version 0.8 . . . 48
14.6 Changes in version 0.81 . . . 48
14.7 Changes in version 0.82 . . . 48
14.8 Changes in version 0.83 . . . 49
14.9 Changes in version 0.90 . . . 49
14.10 Changes in version 0.92 . . . 49
14.11 Changes in version 0.93 . . . 50
14.12 Changes in version 0.94 . . . 50
14.13 Changes in version 0.95 . . . 50
14.14 Changes in version 0.96 . . . 50
14.15 Changes in version 0.98 . . . 50
14.16 Changes in version 0.98a . . . 51
14.17 Changes in version 0.98b . . . 51
14.18 Changes in version 0.98e . . . 51
14.19 Changes in version 0.98k . . . 51
14.20 Changes in version 0.98l . . . 51
14.21 Changes in version 0.99f . . . 52
14.22 Changes in version 0.99m . . . 52

References **52**

A Some remarks about the PDF syntax **53**

1 Introduction

For many years the creation of accessible, tagged PDF-files with LaTeX that 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 web page [10] dedicated to this topic.

In my opinion missing were 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, means to try out how core LaTeX commands behave if tagging is used. Without such experiments it is in my opinion quite difficult to get a feeling about what has to be done, which kernel changes are needed, and how packages should be adapted.

This package was developed to close this gap by offering *core* commands to tag a PDF[1]. My hope was that the knowledge gained by the use of this package would in the end allow to decide if and how code to do tagging should become part of the LaTeX kernel.

The code has been written so that it can be added as a module to the LaTeX kernel itself if it turned out to be usable. It therefore avoids patching commands from other packages. It was also not an aim of the package to develop patches to directly enable tagging in other packages. While in the end changes to various commands in many classes and packages will be needed to automatically get tagged PDF files, these changes should be done by 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 needs to look at packages like `tex4ht` or `bidi` or `hyperref` to see how difficult and sometimes fragile this is.

The package is now a part of the Tagged PDF project and triggered already various changes in the LaTeX kernel and the engines: There is a new PDF management, the new para hooks allow to automatically tag paragraphs, after changes in the output routine page breaks and header and footer are handled correctly, the engines now support structure destinations. More changes are in the latex-lab bundle and are loaded through the `tagging` keys.

I'm sure that `tagpdf` still has bugs. Bug reports, suggestions and comments can be added to the issue tracker on github either https://github.com/latex3/tagpdf or https://github.com/latex3/tagging-project.

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

1.1 Tagging and accessibility

While the package is named `tagpdf` the goal is also *accessible* PDF-files. Tagging is *one* (the most difficult) requirement for accessibility but there are others. I will mention some later on in this documentation, and – if sensible – also 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 short and easy to pronounce (and of course, it was always meant to be temporary).

1.2 Engines and modes

Theoretically, the package works with all engines, but the xelatex and the latex-dvips-route are basically untested and they also don't support real space glyphs so I don't recommend them. lualatex is the most powerful and safe modus and should be used for new documents; it is slower than pdflatex but requires fewer compilations. pdflatex works ok and can be used for legacy documents; it needs more compilations to resolve all cross references needed for the tagging.

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 and (since version 0.98k) with dvilualatex. Since version 0.99m, the Lua mode is forced if luatex is detected, otherwise generic mode is used.

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.

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

PDF is a page oriented format and so the ability of luatex to manipulate pages and nodes after the TEX-processing has finished is really useful here. Also with luatex characters are normally already given as Unicode.

The package uses quite a lot of labels (in generic mode more than with luamode). It is now based on the property module of the LATEX kernel. This module provides expandable references but the drawback is that (right now) they don't always give good rerun messages if they have changed. I advise to use the `rerunfilecheck` package as an intermediate work-around and when using pdflatex compile at least once or twice more often than normal.

1.3 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 they implemented only support for PDF 1.7.

In 2018 PDF 2.0 was released. The reference can now be bought at no cost through the PDF association.

PDF 2.0 has a number of features that are really needed for good tagging: it knows more structure types, it allows to add associated files to structures—these are small, embedded files that can, for example, contain the mathML or source code of an equation—it knows structure destinations, which allows to link to a structure. It knows the MathML namespace.

LATEX therefore targets PDF 2.0 and `tagpdf` has support for associated files, for name spaces and other PDF 2.0 features.

PDF 2.0 features are currently (beginning of 2025) still not well supported by PDF consumers, but some progress has been made. Foxit can handle MathML associated files and to some extent MathML structure elements and, together with development versions of NVDA and MathCat, reading of equations is already quite good. The PDF Accessibility Checker (PAC) no longer crashes if one tries to load a PDF 2.0 file. We recommend to use PDF 2.0 if possible and then to complain to the PDF consumer if something doesn't work.

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 paths; so for example associated files will always be added. But tests could be added in case this leads to incompatibilities.

1.4 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 a 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 PDF follows requirements of standards like PDF/UA *formally*[2].
- One must check how good the accessibility is *practically*.

Syntax validation and formal standard validation can be done with the validator veraPDF [13] which can also handle PDF 2.0 files. Other options (only for PDF 1.7 and older) are preflight of the (non-free) Adobe Acrobat and the free PDF Accessibility Checker (PAC 2026) [5]. A quite useful tool is "Next Generation PDF" [3], a browser application which converts a tagged PDF to

[2] The PDF/UA-2 standard for PDF 2.0 will hopefully be released begin of 2024.

html, allows to inspect its structure and also to edit the structure. For PDF 2.0 files there is also a checker based on the Arlington model from veraPDF.

A tool developed by the LaTeX team allows to extract the structure as XML and to validate it against a schema. This can be tested as `https://texlive.net/showtags`.

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

Validation woes

Sadly validators can not always be trusted.

At first lots of the requirements for PDF/UA-1 and PDF/UA-2 are human checks. This means that a validator either doesn't check it or has to use some heuristic.

At second LaTeX creates by default PDF 2.0 and not many validators handle that correctly.

1.5 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, contributions and corrections are welcome!

All examples should use the `\DocumentMetadata` key `uncompress` so that an uncompressed PDF is created and the internal objects and structures can be inspected and be compared by the l3build checks.

1.6 Proof of concept: the tagging of the documentation itself

Starting with version 0.6 the documentation itself has been tagged. The tagging wasn't (and isn't) in any way perfect. The validator from Adobe didn't complain, but PAC 3 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 had no idea what should be used for code in 2.0. Margin notes were simply wrong and there were tagging commands everywhere …

The tagging has been improved and automated over time in sync with improvements and new features in the LaTeX kernel, the latex-lab bundle and the PDF management code and is now much better. Only a few structures—mostly some from currently unsupported packages—still need manual tagging. But sadly the output of the validators don't quite reflect the improvements. The documentation now uses PDF 2.0 and while the newest PAC 2024 can at least open the file it can not validate properly the file. It doesn't fully supports PDF 2.0 (see figure 1). It also complains about the tabular header cells as it doesn't follow attribute classes, see figure 2. The Adobe validator has a bug and doesn't like the (valid) use of the `Lbl` tag for numbers outside lists and so complains about the tagging of the table of content entries (see figure 3).

But even if the documentation would pass all the tests of the validators: as mentioned above passing a formal test doesn't mean that the content is really good and usable. The user commands used for the tagging and also some of the patches used are still rather crude. So there is lot of space for improvement.

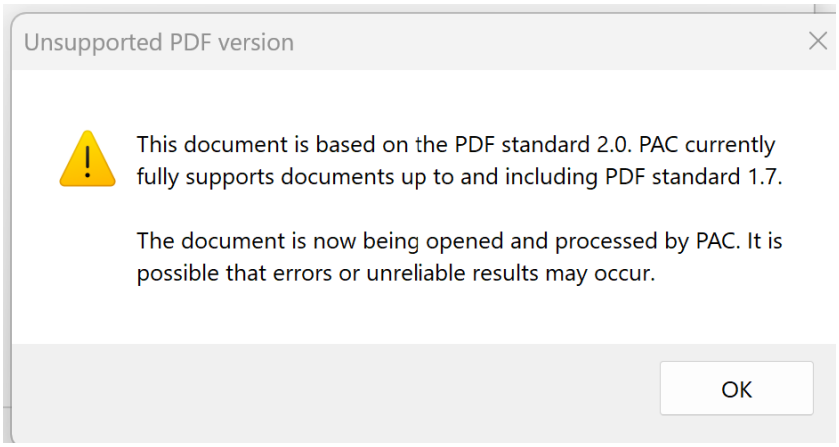


Figure 1: PAC 2024 reading a PDF 2.0 file

Be aware that to create the tagged version a current lualatex-dev and a current version of the `pdfmanagement` package is needed.

2 Loading

The package requires the new PDF management. With a current LaTeX (2025-06-01 or newer) the PDF management is loaded if you use the `\DocumentMetadata` command before `\documentclass`. The `tagpdf` package can then be loaded and activated by using the `tagging=on` key. The exact behavior of the `tagging` key is documented in `documentmetadata-support-doc.pdf` which is part of the `latex-lab` bundle.

Various parts of the code differentiate between PDF version 2.0 and lower versions. By default `\DocumentMetadata` will now set the PDF version to 2.0. If PDF 1.7 is wanted it is required to set the version early in the first `\DocumentMetadata` command so that `tagpdf` can pick up the correct code path.

```
\DocumentMetadata
 {
   tagging=on,
   pdfstandard=ua-2,   % pdfstandard can be set too
 }
\documentclass{article}
\begin{document}
some text
\end{document}
```

Deactivation
When loading `tagpdf` through the `tagging=on` keys, it is automatically activated. To deactivate it while still retaining all the other new code from the latex-lab files, you can use the `tagging=off` key.

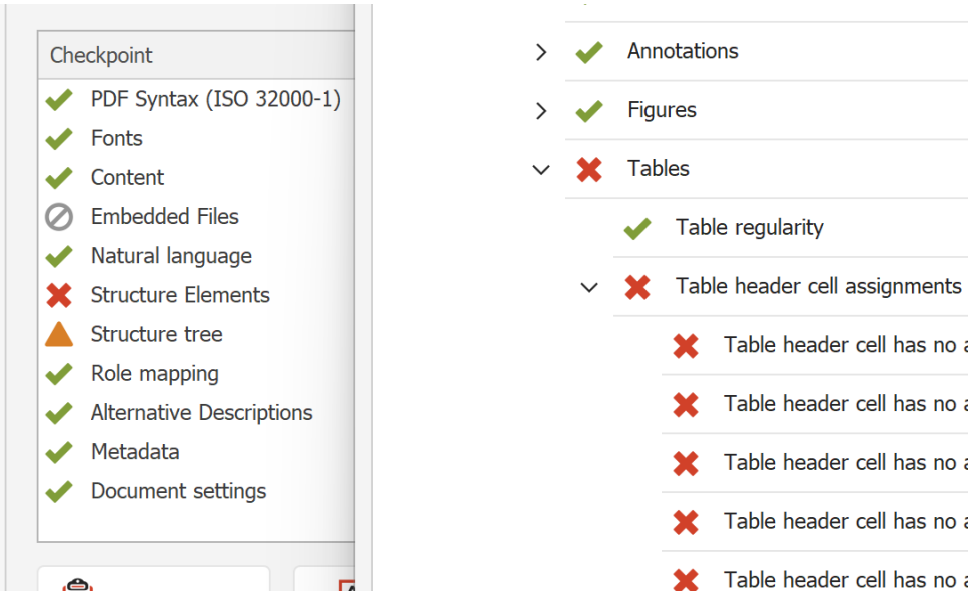


Figure 2: PAC 2024 Validation of header cells

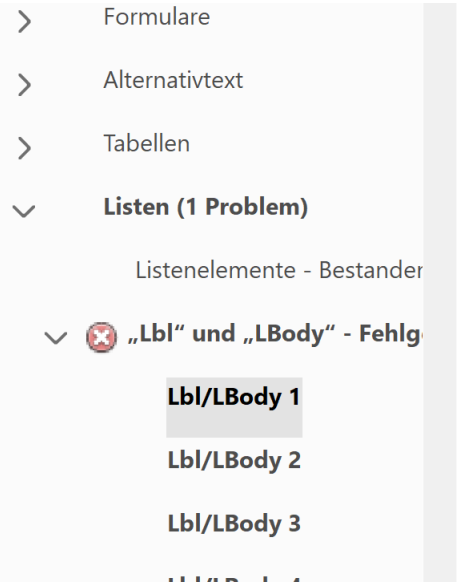


Figure 3: Adobe Acrobat complaining about the `Lbl` use

Starting with LaTeX 2025-11-01, you can also use `\DocumentMetadata{}`, which will then do the same as `\DocumentMetadata{tagging=off}`. This also deactivates the interword space code.

Loading as package needs activation!
It is not recommended anymore, but the package can also be loaded normally with `\usepackage` or `\RequirePackage`. But it still requires the PDF management, so you need to do something like this:

```
\RequirePackage{pdfmanagement}
\RequirePackage{tagpdf}
```

But it will then – apart from loading more packages and defining a lot of things – not do much. You will have to *activate* it with `\tagpdfsetup`.

The PDF management will in any case load `tagpdf-base`, a small package that provides no-op versions of the main tagging commands.

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`{⟨*true code*⟩}{⟨*false code*⟩}

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

2.1 Modes and package options

The package has two different modes: The **generic mode** works (in theory, currently only fully tested with pdflatex) probably with all engines, the **Lua mode** only with lualatex. The differences between both modes will be described later.

Starting with version 0.99m the mode is set automatically (Lua mode for luatex, generic mode otherwise). The package options do nothing anymore and will be removed in future versions.

2.2 Setup and activation

`\tagpdfsetup`{⟨*key-val-list*⟩}

This command sets up the general behavior of the package. The command should normally be 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 at least the following keys. More keys are defined in some of the latex-lab modules; see table 1 for an overview which also includes older, now deprecated names.

The command is deactivated before the class is loaded if the key `tagging=off` is used in `\DocumentMetadata` and then does nothing.

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

activate Like `activate/all`, *additionally* 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.

activate/struct-dest Boolean, initially true. Starting with version 0.93 `tagpdf` will automatically create structure destinations (see section 8) if `hyperref` is used and if the engine supports it. 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 have been used too.

activate/spaces Boolean. The key activates/deactivates the insertion of space glyphs, see section 7. In the luamode it only works if at least activate/mc has been used. It is on by default if you use `tagging=on` in `\DocumentMetadata`. The `tagging=off` key of `\DocumentMetadata` deactivates it. The old name of the key `interwordspace` is still supported but deprecated.

activate/softhyphen luamode only. The key activates and controls the replacing of hard hyphens from hyphenation by soft hyphens. It knows the values

- `false`, `off` Deactivates the automatic handling.
- `char` Replaces the hyphens by U+00AD if the font contains this glyph. Note that in some newer fonts U+00AD is an invisible character and so this could led to disappearing hyphens.
- `artifact` This keeps the hard hyphen U+002D but surrounds it by an Artifact-MC if tagging. This only works if the document is tagged. It requires that the active/tagunmarked key is not set to false.
- `true`, `on` Currently does the same as `artifact`.

If tagging is not active, the softhyphen handling is not activated by default and the hyphen stays a hard-hyphen. It is possible to use the `char` mode with

```
\DocumentMetadata{tagging=off,tagging-setup={activate/softhyphen=char},}
```

As `\tagpdfsetup` is deactivated in the document is not possible to change the mode locally in the document with it.

If tagging is active the default is the `artifact` mode. The mode can be changed with `\tagpdfsetup` at any time. Switching on and off the softhyphen is a global operation, a switch between artifact and char more is a local operation.

role/new-tag Allows to define new tag names, see section 5 for a description.

role/new-attribute This key takes two arguments and declares an attribute. See 3.4.6.

role/new-attribute* Normally attributes declared with the previous key are only written into the PDF if used in a structure. This starred version of the key ensures that the attribute is always written to the PDF.

role/map-tags This key allows to remap the structure tags. Currently it supports only two values: `false` (the default) and `pdf` which maps all tags to their standard PDF role, e.g. `itemize` will be mapped to `L`.

para/tagging Boolean. This activate/deactivates the automatic tagging of paragraphs, see 3.3.10 for more background. It uses the `para/begin` and `para/end` hooks. With more tagging support conditions will be added, that means 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:

para/tag String. This key changes the second tag used by the paratagging code. The default tag is `text`, a LaTeX specific tag that is role mapped to `P`. A useful local setting here can be `NonStruct`, which creates a structure "without meaning".

para/maintag String. This key changes the first tag used by the paratagging code. The default tag is `text-unit`, a LaTeX specific tag that is role mapped to `Part`.

page/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 ….

luamode **activate/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.

viewer/startstructure A structure number. If a `OpenAction` is set in the PDF Catalog (which is normally the case if `hyperref` is used) a structure destination pointing to the structure is added. The initial value is structure 1 (the `Document` structure), the default value is the current structure. The key can be used more than once, the last setting will win.

debug/uncompress Sets both the PDF compresslevel and the PDF objcompresslevel to 0 and so allows to inspect the PDF. Not really useful anymore as this can also be set in `\DocumentMetadata`.

debug This key knows a number of sub-keys to set various debug options.

debug/show This takes a comma list of keywords:

luamode `spaces/spacesOff`: That helps in Lua mode to see where space glyph will be inserted if activate/spaces is activated. This can also be activated with the now deprecated key `show-spaces`

`para/paraOff`: This (locally) activates/deactivates small red and green numbers in the places where the paratagging hook code is used.

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

Table 1: Overview of the keys for `\tagpdfsetup`

new key name	old key name	value	type	defined in
???	root-AF	string	code	tagpdf-struct
root-supplemental-file		string	code	tagpdf-struct

Table 1: Overview of keys for `\tagpdfsetup` (continued)

new key name	old key name	value	type	defined in
attach-css		boolean	bool	tagpdf-user
css-list		string	code	tagpdf-user
css-list-remove		string	code	tagpdf-user
css-list-add		string	code	tagpdf-user
activate	activate	string	meta	tagpdf
activate/all	activate-all	boolean	meta	tagpdf
activate/mc	activate-mc	boolean	bool	tagpdf
activate/softhyphen	—	string	code	tagpdf
activate/spaces	interwordspace	boolean	code	tagpdf-space
activate/struct	activate-struct	boolean	bool	tagpdf
activate/struct-dest	no-struct-dest	boolean	bool	tagpdf
activate/tagunmarked	tagunmarked	boolean	bool	tagpdf
activate/tree	activate-tree	boolean	bool	tagpdf
role/mathml-tags	mathml-tags	boolean	bool	tagpdf-roles
role/new-NS		three brace groups	code	tagpdf-roles
role/new-tag	add-new-tag	string	code	tagpdf-roles
role/new-attribute	newattribute	string	code	tagpdf-struct
role/new-attribute*	newattribute	string	code	tagpdf-struct
role/user-NS	—	string	code	tagpdf-roles
role/map-tags	—	choice	code	tagpdf-roles
debug/show	various	choice	code	various
debug/show=para	paratagging-show	—	code	tagpdf-user
debug/show=paraOff		—	code	tagpdf-user
debug/show=spaces	show-spaces	—	code	tagpdf-spaces
debug/show=spacesOff	—	—	code	tagpdf-spaces
debug/log	log	choice	code	tagpdf
debug/uncompress	uncompress	—	code	tagpdf
debug/parent-child-check		choice	code	tagpdf
viewer/pane/mathml	—	boolean	bool	latex-lab-math (dev)
viewer/pane/mathsource	—	boolean	bool	latex-lab-math (dev)
viewer/startstructure	—	number	code	tagpdf-tree
page/tabsorder	tabsorder	choice	code	tagpdf
page/exclude-header-footer	exclude-header-footer	choice	code	tagpdf-user
page/tag-header-footer		choice	code	tagpdf-user
para/tagging	paratagging	boolean	bool	tagpdf-user
para/tag	paratag	string	code	tagpdf-user
para/maintag	—	string	code	tagpdf-user
para/flattened	—	boolean	bool	tagpdf-user

Table 1: Overview of keys for \tagpdfsetup (continued)

new key name	old key name	value	type	defined in
math/alt/use	—	boolean	bool	latex-lab-math
math/mathml/write-dummy	—	—	code	latex-lab-math
math/mathml/sources	—	clist	clist	latex-lab-math
math/mathml/AF	—	boolean	bool	latex-lab-math
math/tex/AF	—	boolean	bool	latex-lab-math
math/setup	—	string	code	latex-lab-math
table/tagging	table-tagging	choice	code	latex-lab-table
table/header-rows	table-header-rows	clist	clist	latex-lab-table
text/lang	—	string	code	tagpdf-user

\tagtool{⟨*key-val*⟩}
\tag_tool:n{⟨*key-val*⟩}

These commands are deprecated. Tagging sockets (documented in source2e) should be used instead.

3 Tagging

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[3]:

```
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
```

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 guarantee that the stream contains the text in the order it should be read.

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.)

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

3.1 Tag names

In the following there will be many references to "tag names". In the PDF these are "PDF names": strings with a slash at the begin like `/LBody` and `/Em`. In the commands of `tagpdf` they are – with some exceptions for commands meant for developers – input without the slash.

Tag names are used a lot and to avoid to slow down the compilation they are not purified and checked. They are also normally not passed through some escaping command, but there can be code parts where it happens anyway. It lies in the responsibility of the user to use sensible names that do not break under such conditions.

The value should therefore be a string that can be used as a PDF name (by adding a slash at the begin) and that is not changed if it is passed through a command that escapes names. A good test is to use `\pdf_name_from_unicode_e:n`: if this only adds a slash the name should be fine. Some examples:

tag name	escaped with \pdf_name_from_unicode_e:n	comment
chapter	/chapter	good
chapter-title	/chapter-title	good
chapter.title	/chapter.title	good
chapter title	/chapter#20title	bad
chapter/title	/chapter#2Ftitle	bad
chapter(title)	/chapter#28title#29	bad
grüße	/gr#C3#BC#C3#9Fe	bad

3.2 Three tasks

To tag a PDF three tasks must be carried out:

mc-task

1. **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 of text is created automatically, e.g. the toc, references, citations, list numbers etc. and it is not always easy to mark them correctly.

struct-task

2. **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.

tree-task

3. **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 parent tree must be created to get a reference from the mc-chunks to the parent structure. A role map must be written. And a number of dictionary entries. All this is hopefully done automatically and correctly by the package ….

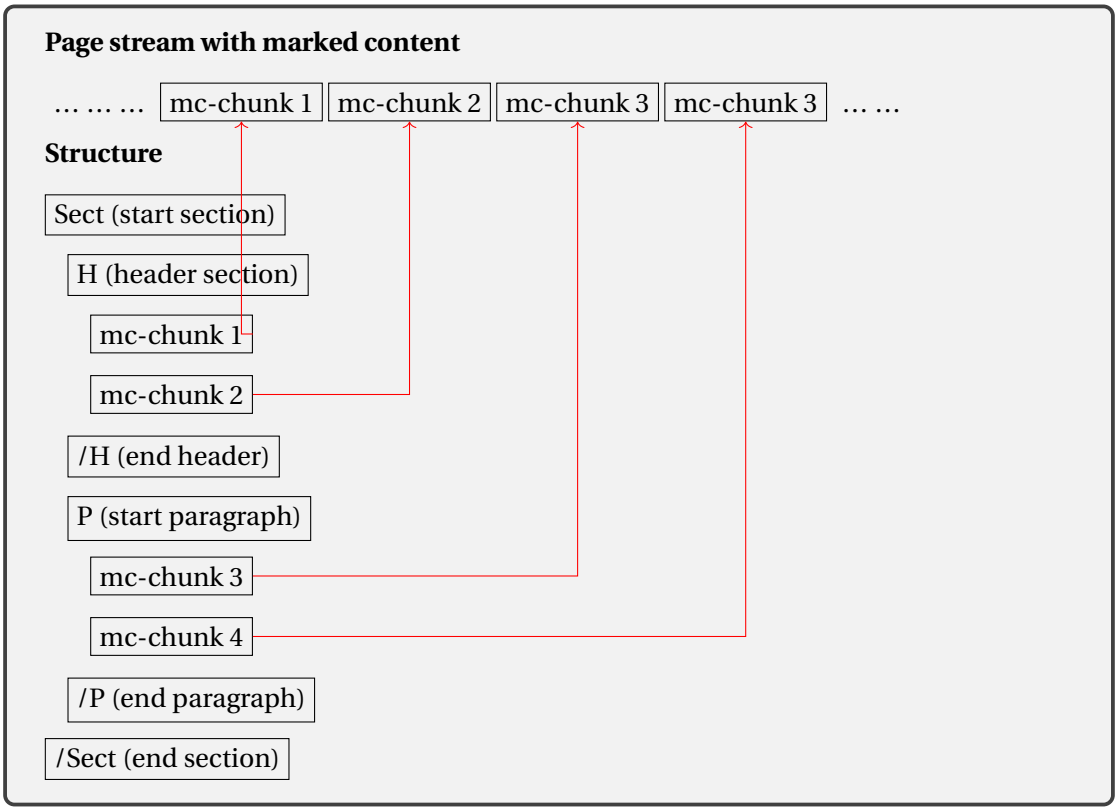


Figure 4: Schematical description of the relation between marked content in the page stream and the structure

3.3 Task 1: Marking the chunks: the mark-content-step

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:

Artifacts They are marked with of a pair of keywords, `BMC` and `EMC` which surrounds the text. `BMC` has a single prefix argument, the fixed 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.

```
/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[4]. 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.

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 there shouldn't be "holes" in the numbering system. (It is perhaps possible to handle a numbering scheme not starting with 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.

[4] 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.

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

7. 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 3.3.9.

3.3.1 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 (global) *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. Page breaks 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.

3.3.2 Commands to mark content and chunks

Generic mode only

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 3.3.9.

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

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.

In the generic mode the commands insert literals. These are whatsits and so can affect spacing. In Lua mode they set an attribute *globally*.

The key-val list understands the following keys:

tag This key is optional. By default the tag name of the surrounding structure is used, which normally should be fine. But if needed the name can be set explicitly with this key. The value of the key is typically one of the standard type listed in section 5 (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).

artifact 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.

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.

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

Lua mode The Lua mode will mark up everything unmarked as `artifact=notype`. You can suppress this behavior by setting the tagpdfsetup key `activate/tagunmarked` to false. See section 2.2.

stash 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 margin note 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.

label 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-`.

alt This key inserts an `/Alt` value in the property dictionary of the BDC operator. See section 4. 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). If the value is empty, nothing will happen.

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,alt=\myalttext}
```

actualtext This key inserts an `/ActualText` value in the property dictionary of the BDC operator. See section 4. 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). If the value is empty, nothing will happen.

That means that you can do something like in the following listing 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. There is also some discussion going on, if `/ActualText` can actually be used in a MC dictionary or if it should be in a separate BDC-operator.

lang This (experimental) key allows to add a `/Lang` entry on the MC. This probably is not conforming to the PDF specification but it works in Adobe reader. The value is a BCP-string, e.g. `en` or `de`. The key is not so useful as you typically want to the set '/Lang' entry on all MC inside a region or structure. Once the requirements of the PDF specification has been clarified the interface will be extended.

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{`⟨*label*⟩`}`
`\tag_mc_use:n{`⟨*label*⟩`}`

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.

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

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

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

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

`\tag_mc_reset_box:N`⟨*box*⟩

In Lua mode this command will process the given box and reset all mc related attributes in the box to the current values. This means that if the box is used all its contents will be a kid of the current structure. This should (probably) only be used on boxes which don't contain tagging commands. See below section 9 for more details.

3.3.3 Retrieving data

With more elaborate tagging the need arise to retrieve and store current data.

`\tag_get:n{`⟨*key word*⟩`}`

This (expandable) command returns the values of some variables. Currently, the working key words are

- `mc_tag`: the tag name of the current mc-chunk
- `struct_tag`: the tag name of the current structure (without role-mapping).
- `struct_id`: The ID of the current structure. This is a string and is returned including parentheses.
- `struct_num`: This returns a number and works also if only `tagpdf-base` has been loaded, but then doesn't give the same output: if `tagpdf` is loaded and tagging is active, `struct_num` gives the number of currently active structure, so it reverts to the parent number if a structure is closed. If only `tagpdf-base` is loaded nesting of structure is not tracked and so the command gives back the number of the last structure that has been created. In luatex this number can also be retrieved with the Lua function `ltx.tag.get_struct_num()`.
- `struct_counter`: This returns a number and works also if only `tagpdf-base` has been loaded. It gives back the state of the absolute structure counter and so the number of the last structure that has been created. This can be used to detect if in a piece of code there are structure commands. Be aware that this is a LaTeX counter and so is reset in some places. In luatex this number can also be retrieved with the Lua function `ltx.tag.get_struct_counter()`. The number of the next structure to be created is then `ltx.tag.get_struct_counter()` increased by one), this can also be retrieved with the function `ltx.tag.get_struct_num_next()`.
- `mc_counter`: This returns a number and works also if only `tagpdf-base` has been loaded. It gives back the state of the absolute mc-counter and so number of the last mc-chunk that has been created. This can be used to detect if in a piece of code there are mc-commands.
- `current_Sect`: (defined in latex-lab-sec) gives back the number of current Sect structure, it is, e.g., used by the footnote code.

`\tag_get:nN{`⟨*key word*⟩`}{`⟨*tl var*⟩`}`

This (not expandable) command returns the values of some variables and stores them into the tl var. Currently, the working key words are

- `struct_tag`: the tag name of the current structure (without role-mapping).
- `struct_num`: This returns a number.

`\tag_get:nnN{`⟨*number*⟩`} {`⟨*key word*⟩`}{`⟨*tl var*⟩`}`

This (not expandable) command returns the values of some variables of the structure `{`⟨*number*⟩`}` and stores them into the tl var. Currently, the working key words are

- `struct_C`: the attribute classes as a comma list.

3.3.4 Luamode: global or not global – that is the question

Lua mode

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.

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:

```
\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
```

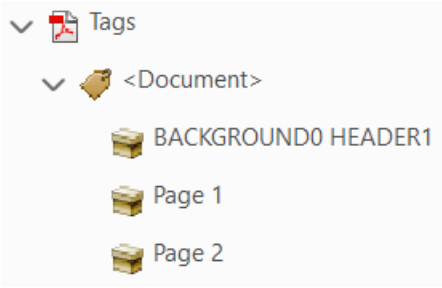
The reading order was then wrong, and the *emphasized text* moved in the structure at the end.

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

```
\DocumentMetadata{uncompress}
\documentclass{article}

\pagestyle{headings}
\begin{document}
\sectionmark{HEADER}
\AddToHook{shipout/background}{\put(5cm,-5cm){BACKGROUND}}
\tagmcbegin{tag=P}Page 1\newpage Page 2\tagmcend
\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 are now marked as artifacts (see below). 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.

3.3.5 Tips

- Mark commands inside floats should work fine (but need perhaps some compilation rounds in generic mode).
- 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 or 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 okay.
- If you use complicated structures and commands (breakable boxes like the one from `tcolorbox`, `multicol`, many footnotes) you will have to check the PDF.

3.3.6 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

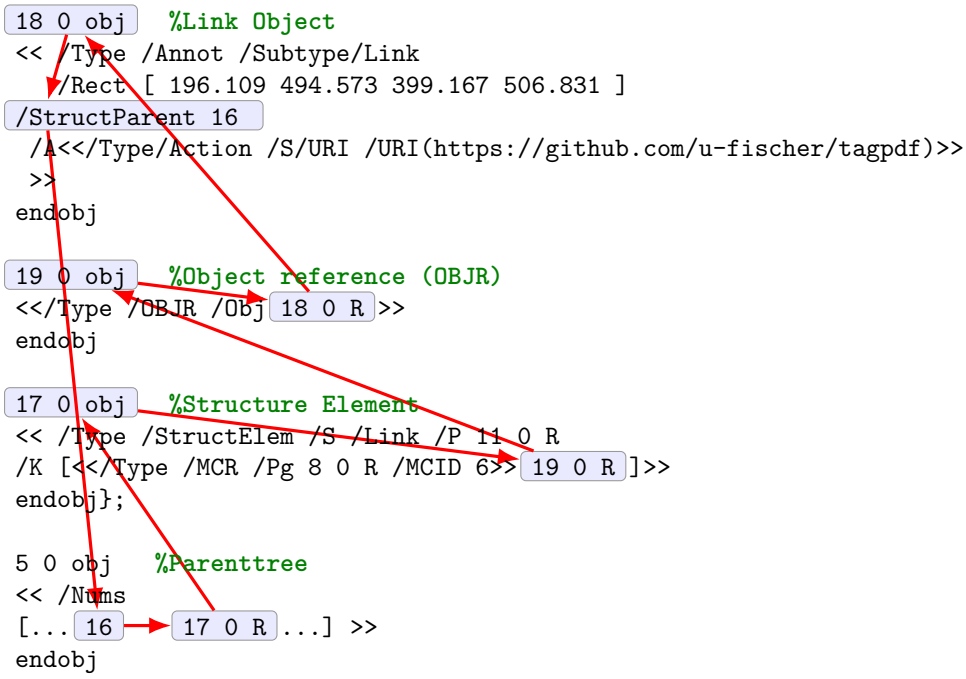


Figure 5: Structure needed for a link annotation

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 makes use of this info.

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.

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.

With the key page/exclude-header-footer the behavior can be changed: The value `false` disables the automatic tagging (the resulting tagging is then rather unpredictable and depends on the surrounding page), the value `pagination` adds additionally an `/Artifact` structure with the attribute `/Pagination`, but the contents is still not tagged.

The (experimental) key page/tag-header-footer surrounds the header with an `/Artifact` structure too but then reenables tagging. It currently only works if the header and footer contain a `\parbox`, so if e.g. the `fancyhdr` package is used.

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

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

3.3.7 Links and other annotations

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.

To connect such a annotation with the structure and so with surrounding or underlying text a specific structure has to be added, see 5: 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.

To support this, `tagpdf` offers currently two commands

`\tag_struct_parent_int:`

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.

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

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.

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 fulfill 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.

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.

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

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

```
\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:}
```

3.3.8 Math

Math is still a problem but some progress has been made. To tag math you have to surround it with a `Formula` structure. But the content of such a structure is handled by readers as a black box so additional data is needed for accessibility.

There are a number of theoretical options here:

1. One can add an alternative text (`/Alt`) or an `/ActualText` to the structure element either some text manually provided by the author or (with the math module in the latex-lab bundle) the LaTeX-source).

2. One can add an alternative text (`/Alt` or `/ActualText`) to the MC-chunks.

3. One can build inside the `Formula` structure element a tree with MathML structure elements—with PDF 2.0 this not require to declare new tags as the MathML name space is built-in.

4. One can in PDF 2.0 attach a MathML file and/or the LaTeX-source as associated file to the `Formula` structure (or to one or more MC-chunks).

The question is how these work in reality.

Option 1 and 2 give not too bad results with a screen reader, but can require manual work and if you are unlucky the reader drops important part of the math (like punctuation symbols). Exploring the equation is not possible.

Option 3 creates many structure elements. E.g. 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 in this direction has been done by Marcel Krüger in the luamml project. But up-to-now it was not possible to test the usability of this approach: With the exception of the html derivation with ngpdf no PDF-viewer/screen reader combination seems to make use of such structures. I'm not sure anyway 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.

The last option 4 has been implemented in the math module in the `latex-lab` bundle. Here happily a proof of concept was possible: With development versions of Foxit and the NVDA reader it was possible to access an attached MathML and get speech output from it [11, 12]. See also [7] for some examples and section 4 for some more remarks and tests.

3.3.9 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 0.92 there is code which resolves 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. 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.

Please note

- 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 (`longtable` is for example problematic).

3.3.10 Automatic tagging of paragraphs

Another feature that emerged from the LaTeX tagged PDF project are hooks at the begin and end of paragraphs. The tagging code makes use of these hooks to tag paragraphs. In the first version it added only one structure, but this proved to be not adequate:

Paragraphs in LaTeX can be nested, e.g., you can have a paragraph containing a display quote, which in turn consists of more than one (sub)paragraph, followed by some more text which all belongs to the same outer paragraph.

In the PDF model and in the HTML model that is not supported: the rules in PDF specification do not allow `P`-structures to be nested—a limitation that conflicts with real live, given that such constructs are quite normal in spoken and written language.

The approach we take (starting with march 2023) to resolve this is to model such "semantic" paragraphs with a structure named `semantic-para` and use `P` (under the name `text-block`) only for (portions of) the actual paragraph text in a way that the `P`s are not nested. As a result we have for a simple paragraph two structures:

```
<text-unit>
  <P>
   The paragraph text ...
  </P>
</text-unit
```

In the case of an element, such as a display quote or a display list inside the paragraph, we then have

```
<text-unit>
  <text>
   The paragraph text before the display element ...
```

```
  </text>
   <display element structure>
    Content of the display structure possibly involving inner <text-unit> tags
   </display element structure>
  <text>
   ... continuing the outer paragraph text
  </text>
</text-unit>
```

In other words such a display block is always embedded in a `<text-unit>` structure, possibly preceded by a `<text>`...`</text>` block and possibly followed by one, though both such blocks are optional. More information about this can be found in the documentation of `latex-lab-block`.

As a consequence the tagging code adds two structures if paratagging is activated. The code to tag display blocks extends this code to handle the nesting of lists and other display structures.

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 very confused. At the end of the document `tagpdf` checks if the number of outer and inner start and end paragraph structures created with the automatic paratagging code are equal and it will error if not.

The automatic tagging of paragraphs can be deactivated completely or only the outer level with the `\tagpdfsetup` key `para/tagging` and `para-flattened` or with the commands `\tagpdfparaOn` and `\tagpdfparaOff` or by changing the plugs of the tagging sockets – they are defined and described in source2e, section lttagging.

Nesting the activation and deactivation of the tagging of paragraphs can be quite difficult. For example if it is unclear if the inner code issues a `\par` or not it is not trivial to exclude an end hook for every excluded begin hook. In such cases it can be easier to use the `para/tag` key with the value `NonStruct` to convert some `P`-structures into `NonStruct`-structures without real meaning.

3.4 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 leaves of the tree – the marked contents chunks marked up with the `tagmc`-commands. The root of the tree is the `StructTreeRoot`.

3.4.1 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 5 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{
  role/new-tag = TAB/P,
  role/new-tag = FIG/P,
            }
```

3.4.2 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. It has been implemented in the sec module in latex-lab, but can break if sectioning commands are hidden inside boxes.

3.4.3 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 5. 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 one can refer to the structure. Currently the key writes a property whose name starts with `tagpdfstruct-` to the aux-file with the two attributes `tagstruct` (the structure number) and `tagstructobj` (the object reference) but also stores the name and the structure number into a prop for use in the current compilation. The

label is e.g. used by `\tag_struct_use:n` and by the `ref` key (which can refer to future structures).

parent With the parent key one can choose another parent. The value is a structure number which must refer to an already existing, previously created structure. Such a structure number can have been stored previously with `\tag_get:n`, but one can also use a label on the parent structure and then use `\property_ref:nn{tagpdfstruct-label}{tagstruct}` to retrieve it.

firstkid If this key is used the structure is added at the left of the kids of the parent structure (if the structure is not stashed). This means that it will be the first kid of the structure (unless some later structure uses the key too). This can be needed e.g. for a caption as the PDF reference requires it to be the first or last kid of its structure.

alt This key inserts an `/Alt` value in the dictionary of structure object, see section 4. 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). If the value is empty, nothing will happen.

That means that you can do something like this:

```
\newcommand\myalttext{\frac{a}{b}}
\tagstructbegin{tag=P,alt=\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 4. 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). If the value is empty, nothing will happen.

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}
```

The key can be used more than once and works then additively: new attributes are added to the existing once. See the next key if you want to overwrite existing attributes.

See also section 3.4.6.

attribute* This key first clears the attributes and then works like attribute. This will be rarely needed but is offered in case that additional settings for a structure are passed through a socket.

attribute-unnamed This takes a list of attributes contents (the part which is set in the second argument of `role/new-attribute`). Note that unlike attribute it can not reuse attribute objects, every use creates a new object. The argument is not expanded, but expansion can be forced (as with all keys) with `attribute-unnamed:o` or `attribute-unnamed:e`

attribute-unnamed* This clears the attributes and then works like attribute-unnamed

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 3.4.6.

title This key allows to set the dictionary entry `/T` (for a 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 `/T` 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 1.7, but the code currently doesn't check the PDF version, it is your responsibility to set it (this can be done with the `pdfversion` key in `\DocumentMetadata`).

root-AF This key allows to reference an associated file in the root structure element. The use of the key is no longer useful as ngpdf [3] will change its implementation and use files attached to the Catalog.

root-supplemental-file This is a variant of the previous key. It takes as argument a file name. It then embeds this file with `/AFRelationship /Supplement` and appends it as associated file to the structure root. This key is like `root-AF` no longer recommended. Use the following key instead.

catalog-supplemental-file This key should be used instead of `root-supplemental-file`. It takes as argument a file name. It then embeds this file with `/AFRelationship /Supplement` and appends it as associated file to the Catalog. ngpdf [3] will store a `.css` attached in this way and reference it in the head of the html. If a `html` is attached in this way, ngpdf will copy the content into the head of the derived html. This means that the content of such an html file should normally be some html snippet suitable for the head, e.g. some css-code inside `<style>` tags.

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 `AFinline`, but it expands the value once.

texsource This is like `AFinline-o`, but it creates a tex-file, with mime type `application/x-tex` and the AFRelationship `Source`. It also sets the /Desc key to a (currently) fix text to satisfy some validators.

mathml This is like `AFinline-o`, but it creates a xml-file, with mime type `application/xml` and the AFRelationship `Supplement`. It also sets the /Desc key to a (currently) fix text to satisfy some validators.

lang This key allows to set the language for a structure element. The value should be a bcp-identifier, e.g. `de-De`. It can also be set "from the outside" for all structures in the current group with `\tagpdfsetup` and the `text/lang` key.

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}`. It can be used more than once in the key/value argument and combines the references. See below in section 3.4.4 for an extended discussion about the `/Ref` array.

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 stuck to E).

phoneme This key sets the `/Phoneme` key, a pronunciation hint. The value should be a IPA text (the PDF spec allows other variants but this not yet implemented). This is a new key from PDF 2.0 but as far as I know not used by any viewer.

`\tagstructend`
`\tag_struct_end:`

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.

`\tagstructuse{`⟨*label*⟩`}`
`\tag_struct_use:n{`⟨*label*⟩`}`

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.

3.4.4 Updating structure keys

In a number of cases structures must be updated later. Such updates often can not simply replace values but must extend them in special ways. For this a generic commands exists. Currently it supports updating the `/S` (tag), `/Alt`, `/T` (title), `/C` (attribute classes), `/A` key (attributes) and the `Ref` key.

`\tag_struct_gput:nnn{`⟨*structure number*⟩`}{`⟨*keyword*⟩`}{`⟨*value*⟩`}`

The allowed ⟨*keywords*⟩ are `tag`, `alt`, `title`, `C`, `ref`, `ref_label` `ref_dest`, `ref_dest_parent`,`ref_num` and `attribute`.

The first four values *replace* the relevant entries in the structure.

The keywords related to the `/Ref` allow to *extend* its value (`/Ref` is an array), with four methods: directly as object reference (key word `ref`), through a label name set with the `label` key (key word `ref_label`), through a destination name if a `\MakeLinkTarget` has been used in the target structure (key words `ref_dest` and `ref_dest_parent`)—this also works if `hyperref` has not been loaded—and through the structure number, which has been stored e.g. in a label.

With the keyword `attribute` the attribute of the structure is extended. The value is the content of one attribute dictionary, so for example `/O /Layout /BBox [10 10 50 50]`.

3.4.5 Root structure

A document should have at least one structure which contains the whole document. A suitable tag is `Document`. Such a root is now always added automatically. Its type can be changed with the key `activate`.

3.4.6 Attributes and attribute classes

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:

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

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.

To use such attributes you must at first declare it in `\tagpdfsetup` with the key `role/new-attribute` or with the command `\tag_attr_new:nn`. Both take two arguments, a name and the content of the attribute. The name should be a sensible key name, it is converted to a PDF name with `\pdf_name_from_unicode_e:n`, so slashes and spaces are allow. The content should be a dictionary without the bracket.

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

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

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

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

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

3.4.7 Checking the structure

\tag_if_in:nTF{⟨*structure tag*⟩}

With this command it is possible to test if a structure with tag {⟨*structure tag*⟩} is a (direct or indirect) parent. {⟨*structure tag*⟩} must be a standard structure type, no role mapping is applied.

\tag_if_struct_exist:nTF{⟨*structure number*⟩}

With this command it is possible to test if a structure with number {⟨*structure number*⟩} exists.

3.5 Task 3: Tree management

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.

__tag_finish_structure:

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).

3.6 A fully marked up document body

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 environments.

```
\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}
```

3.7 Interrupting the tagging

Experience showed that it must be possible to interrupt tagging in some places. For example various packages do trial typesetting to measure text and this shouldn't create structures. There

are therefore a number of commands for various use cases[5]

Warning! Stopping tagging should be done only with care and when it is ensured that no code inside the stopped part gets confused. Most importantly currently tagging should not be stopped if a page break can occur or the output routine is called.

\tag_suspend:n{⟨*label*⟩}
\tag_resume:n{⟨*label*⟩}

These commands suspend and resume tagging in the current group by switching *local* booleans. They also stop the increasing of the counters which keep track of paragraphs if the correct wrapper commands are used.

Restarting tagging is normally only needed if groups can't be used and then must be done with care: `\tag_resume:n` should normally only restart tagging if the corresponding stop command actually stopped tagging. This is implement through a local counter which keeps track of the level.

The ⟨*label*⟩ can be used to identify the command in debugging message. The label is not expanded and so can for example be a single command token.

The commands are the L3-layer versions of `\SuspendTagging` and `\ResumeTagging` and will be available in the kernel with the 2024 November release.

```
\tag_suspend:n{\outercommand}
 ...
 \tag_suspend:n{\innercommand}
  ...
 \tag_resume:n{\innercommand}
 ...
\tag_resume:n{\outercommand}
```

\tag_stop:
\tag_start:
\tagstop
\tagstart
\tag_stop:n{⟨*label*⟩}
\tag_start:n{⟨*label*⟩}

These commands are now deprecated in favor or `\tag_suspend:n` and `\tag_resume:n` but are still provided for some time.

3.8 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 when defining commands you have to ensure that they correctly push and pop the mc-chunks where needed.

[5] it is quite possible that some of the commands will disappear again if we realize that they are not fitting!

4 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 does 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 `alt`) 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 both keys and text-to-speech software (the built-in 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.

5 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 `role/new-tag` key:

```
\tagpdfsetup{role/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 (`StructTreeRoot` is a bit special, it is not really a structure name but nevertheless listed here): StructTreeRoot, Document, Part, Sect, Div, Caption, 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, 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[6]:
   Art, BlockQuote, TOC, TOCI, Index, Private, Quote, Note, Reference, BibEntry, Code,
   and the following are new:
   Artifact, DocumentFragment, Aside, H7, H8, H9, H10, Title, FENote, Sub, Em, Strong.

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 `role/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`. See section 3.1 regarding the syntax of the value.

tag-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.

role 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` or `mathml` set. It can also be a user type, then this user tag must have been declared before. The PDF format allows mapping to be done transitively. But you should be aware that `tagpdf` can't (or more precisely won't) check if some unusual role mapping makes really sense, this lies in the responsibility of the author.

role-namespace The default value is the default namespace of the role: `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. With this key the value can be overwritten.

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

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

The exact effects of the keys depend 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.

[6] They still can be used in a PDF 2.0 document!

5.1 The `latex` namespace

Starting with version 0.98 work has started to setup specific latex tags. In PDF 2.0 in form of a special name space, with PDF 1.7 or older the tags are role mapped. This is work in progress and bound to change.

5.2 The `user` namespace

As mentioned above new tags are created by default in the user namespace. Namespaces are identified through an ID which is normally given as an URI. By default `tagpdf` creates an random URI for the user namespace but it is possible to change that:

```
\tagpdfsetup{role/user-NS = somestring}
```

will create use the URI `https://www.latex-project.org/ns/local/somestring` as ID for the namespace.

5.3 Fallback RoleMap

As mentioned above PDF 2.0 support name spaces for tags. This is quite nice. At first because it avoid name clashes, but also because it allow to build a cleaner model of the document structure.

But sadly support for PDF 2.0 is still quite scarce and while most PDF readers have no problems to open and render a PDF 2.0 file they don't "see" the role mapping if name spaces are used. Therefore since version 0.98t `tagpdf` adds in PDF 2.0 files additionally also a global `/RoleMap` dictionary as a fallback for such processors.

5.4 Mathml

In PDF 2.0 mathml tags have their own name space and can be freely used. In PDF 1.7. they can only be used if they are rolemapped to a standard type. By default they are not added to the `/RoleMap` dictionary, but this can be forced with `\tagpdfsetup{role/mathml-tags}`. Please note that this adds mathml at the end of the document and overwrites tags with the same name without warning.

6 Checking parent-child rules

The PDF references formulate various rules about whether a structure can be a child of another structure, e.g. a `Sect` can not be a child of `P`. In the PDF 1.7 reference this rules were rather vague, in the PDF 2.0 reference there is a quite specific matrix, which sadly misses some of the tags from PDF 1.7. The now released ISO norm 32005 addresses this problem and extends the matrix to cover tags from PDF 1.7 and 2.0 (but it still misses the `math` tag and `mathml` tags).

The rules in the matrix are not a simple allowed/not allowed. Instead some rules determine that structure elements can appear only once in a parent, or that additional requirements can be found in the descriptions of the standard structure types, e.g. `Caption` often has to be the first element in the parent structure, and elements like `Part` and `Div` inherit restrictions from parent structures. External standards like PDF/UA can add more rules.

Altogether this doesn't make it easy to check if a structure tree is conformant or not without slowing down the compilation a lot.

With version 0.98 some first steps to do checks (and to react to the result of a change) have been implemented. Some checks will led to warning directly, but the majority will only be visible if the log-level is increased.

Typical messages will look then like this

```
Package tagpdf Info: Checking Parent-Child 'pdf2:Document' --> 'latex:text-unit'
Package tagpdf Info: Parent-Child 'Document' --> 'Part'.
(tagpdf)             Relation is 1 (='0..n') on line ...

Package tagpdf Warning: Parent-Child 'pdf2:Document' --> 'pdf2:Ruby'.
(tagpdf)                Relation is not allowed! on line ...
(tagpdf)                struct 67, text-unit --> struct 68, Ruby
```

The descriptions of the parent and child are rather verbose as the checks have to take role mapping and name spaces into account. The result of a check is a number—negative if the relation is not allowed, positive if allowed. The text in the parentheses show the symbols used in the PDF-matrix.

Be aware

- This doesn't test all rules, it only implements (hopefully correctly) the matrix.
- There can be differences between PDF 1.7 and 2.0, e.g. `FENote` is role-mapped to `Note` in PDF 1.7 and then has different containment rules.
- The special tag `MC` stands for mc-chunks, so "real content" (the matrix has containment rules for this too).
- Currently there is only only negative number `-1` but that is bound to change, depending on if (and how) it is possible to "repair" a disallowed parent-child relation.
- Warnings can be wrong.
- The tags `Part`, `Div` and `NoStruct` are sometimes (not always) "transparent", that means that it depends on their parent if a child tag is allowed. The code tries to detect this, but if a structure using one of these tags is stashed it can't identify the parent and so won't be able to check the rule

Parent-child checks are not fast, so there is an option to disable them

```
\tagpdfsetup{debug/parent-child-check=off}
```

This will be speed up the compilation but should be used with care, parent-child checks are an important tool to detect faulty structures!

They can be reenabled with

```
\tagpdfsetup{debug/parent-child-check=on}
```

When using lualatex there is also the value `atend`. This disables the inline checks, but at the end of the document, the Lua code will go through the structures and check the rules.

There is also an online service [8] which validates the structure with an Relax NG schema developed by the LaTeX Team, which can be found at [9].

7 “Real” space glyphs

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.

With the key activate/spaces you can activate such space glyphs.

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`.

With luatex (in luamode) `activate/spaces` 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. `activate/spaces` can be used without actually tagging a document.

The key-value debug/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`.

Attention: Even with real spaces copy & pasting of code doesn't need to give the correct results: you get spaces but not necessarily 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 Adobe Pro.

`\pdffakespace`

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.

`\tag_spacechar_off:`
`\tag_spacechar_on:`

The commands allow to switch on and off the insertion of space chars. With pdftex they map to primitive `\pdfinterwordspaceoff` and and `\pdfinterwordspaceon` which insert a whatsits and so act globally. The luatex implementation uses an attribute which is also set globally to stay more or less consistent with pdftex. In dvi-mode the commands do nothing.

8 Structure destinations

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.

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.)

At first 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 knew new keywords which allowed to create structure destination easily and support has been already added to the PDF management and `tagpdf`. 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 it may be necessary to surround destinations with a dummy structure (a Span or an Artifact) to get the right page number.

9 Storing and reusing boxes

TeX allows to store material in boxes and to use these box once or multiple times in other places. This poses some challenges to tagging. The listings in the following examples uses low-level TeX box commands to avoid that changes in the LaTeX commands that improve tagging interfere in case you want to test this. To keep the examples short they don't show the needed `\ExplSyntaxOn/\ExplSyntaxOff`.

9.1 Boxes without tagging commands

If no tagging commands were used (or if they were inactive) when the box was stored then there is no problem to use this box with pdfLaTeX/generic mode in various places. So

```
\newbox\mybox
The\setbox\mybox\hbox{yellow} duck

The \box\mybox{} sun
```

will produce (assuming para tagging is activated) the paragraph structures "The duck" and "The yellow sun".

With LuaLaTeX/Lua mode this is different: The nodes in the box will have the mc-attribute value attached which were active when the box was saved and this value is recorded as kid of the first paragraph. So when the Lua code later wanders through the box to find all kids of structure it will also find the content of the `\usebox`. This means with LuaLaTeX we get the two paragraph structures "The duck yellow" and "The sun".

The solution here is to reset the attributes before using the box:

```
The\setbox\mybox\hbox{yellow} duck

The \tag_mc_reset_box:N\mybox\box\mybox{} sun
```

The box can in both modes be used without problems many times.

9.2 Boxes with tagging commands

We assume in the following that the box contains only well balanced tagging commands and no parts that are "untagged". It should be possible to copy the whole box inside a `\tagstructbegin`/`\tagstructend` pair. So the following is fine as box content

```
box=\tagstructbegin{...}\tagmcbegin{} balanced content\tagmcend\tagstructend
box=
  \tagmcbegin{}text\tagmcend
  \tagstructbegin{...}\tagmcbegin{} balanced content\tagmcend\tagstructend
  \tagmcbegin{}text\tagmcend
```

but this not (this case could probably be handled nevertheless with a bit care at least in Lua mode)

```
box= text\tagmcend\tagstructbegin{...}...\tagstructend\tagmcbegin{}text
```

and this is absolutely unusable:

```
box= text\tagmcend\tagstructbegin{...}\tagmcbegin{}text
```

We also assume that we want to move the structure of the box to the place where the box is used (if the structure should stay where the box is saved, simply save it and that will happen). For this we must add a structure that we can stash and label.

```
\tag_mc_end_push: % interrupt an open mc
\tagstructbegin{tag=NonStruct,stash}
 \edef\myboxnum{\tag_get:n{struct_num}} % store structure number
 \setbox\mybox\hbox %or \vbox or ...
   {content}
\tagstructend
\tag_mc_begin_pop:n{}% restart open mc
```

At the place where the box is then used we also have to inject this structure:

```
\tag_mc_end_push: % interrupt an open mc
\tag_struct_use_num:n {\myboxnum} % use structure
\box\mybox              % use box
\tag_mc_begin_pop:n{}% restart open mc
```

With pdfLaTeX Boxes with tagging commands can currently be used only once. The tagging commands set labels and reusing the box gives multiple label warnings.

With LuaLaTeX it is possible to reset the attributes as done with the untagged box and then to reuse at least the content.

9.3 Detecting tagging commands

It is possible to detect if a box contains tagging commands by comparing the state of the mc and structure counter:

```
\def\statebeforebox{\inteval{\tag_get:n{struct_counter}+\tag_get:n{mc_counter}}}
\setbox\mybox ...
%compare numbers against \statebeforebox
```

9.4 Putting everything together

To tag boxes that can be both (without tagging commands or with balanced tagging commands) the following strategy can be used:

- when storing the box put around it a structure as needed by the tagged variant:

```
\tag_mc_end_push: % interrupt an open mc
\tagstructbegin{tag=NonStruct,stash}
\edef\myboxnum{\tag_get:n{struct_num}} % store structure number
 \def\statebeforebox{\inteval{\tag_get:n{struct_counter}+\tag_get:n{mc_counter}}}
 \setbox\mybox\hbox %or \vbox or ...
   {content}
 %check if there is tagging content and store that
\tagstructend
\tag_mc_begin_pop:n{}% restart open mc
```

- when using the box the first time
  - if it has no tagging commands then reset the attribute and use the box.

    ```
    \ExplSyntaxOn
    \let\tagmcresetbox\tag_mc_reset_box:N %make a copy
    \ExplSyntaxOff
    ...
    The \tagmcresetbox\mybox\box\mybox{} sun
    ```

    The stashed `NonStruct` structure is then thrown away.
  - if there is a structure then use the stashed structure

    ```
    \tag_mc_end_push: % interrupt an open mc
    \tag_struct_use_num:n {\myboxnum} % use structure
    \box\mybox              % use box
    \tag_mc_begin_pop:n{}% restart open mc
    ```

- if the box is used a second time then throw an error with pdfLaTeX. With LuaLaTeX reset the attributes and issue a warning.

10 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 done with the `lang` option of `\DocumentMetadata`. For settings in marked content and structure the `lang` key can be used too.
- All characters must have a 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 additional `\pdfglyphtounicode` commands.

- Hard and soft hyphen must be distinct. In luamode this is now handled through the `activate/softhyphen` key. For pdftex no solution is known.
- Spaces between words should be space glyphs and not only a horizontal movement. See section 7.
- Various small infos must be present in the catalog dictionary, info dictionary and the page dictionaries, e.g. metadata like title. This can be done with the options of `\DocumentMetadata`. See the documentation of `l3pdfmeta` for details.

11 CSS style definition for derivation to html

Derivation to html ([6] implemented by, e.g., ngpdf) can be improved by attaching CSS style definitions in associated files with relationship supplement to the StructTreeRoot.

Such CSS style definitions can be given in two ways:

- In files with the extension `.css`. Such files should contain only CSS style definitions. A derivation processor like ngpdf will store these files and include them with an `<link rel=stylesheet href=..` in the head of the html.
- In files with the extension `.html`. Such files should contain CSS style definitions inside one (or more) `<style>...</style>` html tags. The content of these files are copied directly into the head of the derived html.

By default (if tagging is active) `tagpdf` embeds now such CSS style definitions. Currently the list of files is rather short and consists of two files (with extension .html and `<style>...</style>` html tags) which are provided by the `tagpdf` package:

- `latex-align-css.html` which improves the styling of `amsmath` alignments tagged with MathML.
- `latex-list-css.html` which improves the style of list environments.

A number of configuration keys for `\tagpdfsetup` are provided

attach-css Boolean, initially true. It is possible to suppress the embedding of these files by setting this key to `false`, `attach-css=true` or `attach-css` reverts this again.

css-list With this key developers can overwrite the list, e.g. with `css-list={file1,file2}` (where all files should have either the extension `html` or `css`. `css-list=` clears the list (and so suppresses the embedding too).

css-list-remove This key allows to remove a file from the list, e.g. `css-list-remove=latex-list-css.html`.

css-list-add this key allows to add more files to the list, e.g., `css-list-add=my-fancy-align-css.html`. It is also possible to attach a .css-file in this way.

These keys do not affect files added directly with root-supplemental-file.

The files in this list are attached at the end of the compilation (and so normally after the files attached with root-supplemental-file) but you shouldn't rely on this or on a specific order of the embedding in the html.

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-list*⟩`}`

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 new page. 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`.

debug/structures This key is only available if the package `tagpdf-debug` has been loaded too. It takes as value a number (the default is 0), and shows on the terminal and in the log information about all structures with a number equal or larger than the number. The data avoids to show PDF object numbers to make it more usable for test suites.

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 3.3.7 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?

14 History

This section lists important changes during the development of the package. More can be found in the `CHANGELOG.MD` and by checking the git commits.

14.1 Changes in 0.3

In this version I improved the handling of alternative and actual text. See section 4. This change meant that the package relies 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.

14.2 Changes in 0.5

I added code to handle attributes and attribute classes, see section 3.4.6 and corrected a small number of code errors.

I added code to add "real" space glyphs to the PDF, see section 7.

14.3 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 attributes 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.

14.4 Changes in version 0.61

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

14.5 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 `role/new-attribute` (old key name: `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.

14.6 Changes in version 0.81

- Hook code to tag links (URI and GoTo type) have been added. So normally they should simply work if tagging is activated.
- Commands and keys to allow automatic paragraph tagging have been added. See section 3.3.10. 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.
- A key to test if local or global setting of the mc-attributes in luamode is more sensible, see 3.3.4 for more details.
- New commands to store and reset mc-tags.
- PDF 2.0 namespaces are now supported.

14.7 Changes in version 0.82

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

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

14.8 Changes in version 0.83

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

14.9 Changes in version 0.90

- Code has been cleaned up and better documented.
- **More engines supported** The generic mode of `tagpdf` now works (theoretically, it is not much tested) with all engines supported by the PDF management. So compilations with XeLaTeX or with dvips should work. But it should be noted that these engines and backends don't support the `interwordspace` 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).

Breaking change!

- **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 3.3.4 for more details.
- **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 `activate/spaces` now works also if tagging is not active.
- The key `activate` has been added, it open the first structure, see above.

14.10 Changes in version 0.92

- support for page breaks in pdftex has been added, see section 3.3.9,
- header and footer are tagged as artifacts automatically, see section 3.3.6.
- keys `alttext-o` and `actualtext-o` has been removed. `alttext` and `actualtext` will now expand once.

14.11 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 planned changes in LaTeX: 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 the package.
- 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 `\tagpdfifluatexTF` and `\tagpdfifpdftexTF` have been removed.

14.12 Changes in version 0.94

In this version a small package, `tagpdf-base` has been added. It provides no-op versions of the main expl3 user commands for packages that want to support tagging but can't be sure if the `tagpdf` package has been loaded.

14.13 Changes in version 0.95

Small bug fixes.

14.14 Changes in version 0.96

- The `alttext` key has been renamed to `alt`, the other key name exists as alias.
- The new command `\tag_struct_object_ref:n` allows to create the object reference of a structure.
- a new key `parent` has been added to allow structures to choose their parent structure.
- a new option `paratag` allows to change the tag name used for the automatically tagged paragraphs.
- the commands `\tag_start:`, `\tag_stop:`, `\tag_stop:n` and `\tag_start:n` allow to stop and start tagging (for example in trial typesetting).
- Small bug fixes.

14.15 Changes in version 0.98

- The declarations of tag namespaces have been externalized and are now read from files when `tagpdf` is loaded.
- The PDF format (and some of the standards) declare various parent-child rules for structure tags. A first step to implement this rules and check if they are fulfilled have been done. More information can be found in section 6.

- As a side effect of the new rule checking, the requirements for new tags have been tightened: Adding a new tag with add-new-tag now requires that the target role is defined. Unknown roles error.
- `\tagmcbegin` no longer requires that a tag is set, instead if will pick up the tag name from the surrounding structure.
- Structure destination are now created also with PDF< 2.0. They shouldn't harm and can improve the html export.

14.16 Changes in version 0.98a

Small bug fixes in code and documentation.

14.17 Changes in version 0.98b

The main change is from now on every structure has an ID and an IDtree is added. The ID of a structure can be retrieved with `\tag_get:n` see 3.3.3.

14.18 Changes in version 0.98e

- The main change is that the automatic paratagging uses now a two-level structure. This accompanies development in the LaTeX github in the `latex-lab` package regarding the tagging of blocks like lists or verbatim. See 3.3.10 and also `latex-lab-block-tagging.dtx` for more background.
- The command `tag_struct_end:n` has been add to improve debugging.

14.19 Changes in version 0.98k

The luamode has been adapted and now allows also the compilation with dvilualatex. By default it will insert specials for `dvips` into the dvi. But be aware that `dvips` can normally not be used as it can't handle open type fonts, and extended version would be needed which isn't in texlive yet. It is also possible to use `dvipdfmx` as backend (which already has support for open type fonts), for this you need to use `backend=dvipdfmx` in the `\DocumentMetadata` command. Real space chars will work, but are currently not taken from the current font. This will be improved in the next luaotfload version. The compilation with dvilualatex is not much tested yet.

14.20 Changes in version 0.98l

In 2023 the primitives to write literal code into the PDF have been extended in all engines and now allow to delay the expansion of their argument to the shipout. This made it possible to greatly simplify and speed up the code used in generic mode to number the MC-chunks. In most cases building the structure should now need only two or three compilations. The new code requires a current pdfmanagement-testphase and is then used automatically if the new engines are detected.

14.21 Changes in version 0.99f

Deprecated `\tag_start:`, `\tag_stop:`, `\tag_stop:n` and `\tag_start:n` in favor of `\tag_suspend:n` and `\tag_resume:n`.

14.22 Changes in version 0.99m

Removed/deactivated package options `luamode` and `genericmode`. Added first support for pronunciation hints, the `phoneme` key.

References

[1] Adobe Systems Incorporated. *Document management – Portable document format – Part 1: PDF 1.7*. 1st ed. July 1, 2008. URL: https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/PDF32000_2008.pdf (visited on 02/17/2024).

[2] Adobe Systems Incorporated. *PDF Reference, sixth edition*. 2006. URL: https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/pdfreference1.7old.pdf (visited on 02/17/2024).

[3] Dual Lab. *Next-Generation PDF*. URL: https://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] *PDF Accessibility Checker (PAC 2026)*. URL: https://pac.pdf-accessibility.org/de (visited on 09/19/2026).

[6] PDF Association. *Deriving HTML from PDF*. An update is in development. 2019. URL: https://pdfa.org/wp-content/uploads/2019/06/Deriving_HTML_from_PDF.pdf (visited on 03/05/2025).

[7] LaTeX Project. *MathML Associated Files Examples*. URL: https://github.com/latex3/tagging-project/discussions/56 (visited on 02/19/2024).

[8] LaTeX Project. *PDF Structure Tree Display and Validation Upload Tagged PDF*. URL: https://texlive.net/showtags.

[9] LaTeX Project. *show_pdf_tags script and Relax NG schema*. URL: https://github.com/latex3/pdf_structure.

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

[11] Roman Toda et al. "PDF Document Object Model Support for Math". In: *The 5th International Workshop on Digitization and E-Inclusion in Mathematics and Science 2024*. The DEIMS2024 Organizing Committee, 2024. URL: https://workshop.sciaccess.net/deims2024/DEIMS2024_Proceedings.zip.

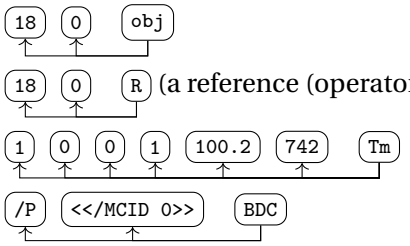
[12] Frank Mittelbach und Ulrike Fischer. "Enhancing LATEX to Automatically Produce Tagged and Accessible PDF". In: *The 5th International Workshop on Digitization and E-Inclusion in Mathematics and Science 2024*. The DEIMS2024 Organizing Committee, 2024. URL: https://workshop.sciaccess.net/deims2024/DEIMS2024_Proceedings.zip.

[13] veraPDF consortium. *veraPDF*. URL: https://verapdf.org/.

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).`

Hexadecimal 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
```

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

Number tree 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

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

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*.