



author Esger Renkema
 contact minim@elrenkema.nl

This is a modern plain format for the LuaTeX engine, adding improved low-level support for many LuaTeX extensions and newer PDF features. While it can be used as drop-in replacement for plain TeX, it probably is most useful as a basis for your own formats.

Most features included in the format are provided by separate packages that can be used on their own; see the packages

- minim-mp** for mplib (MetaPost) support
- minim-math** for Unicode mathematics
- minim-pdf** for hypertext and Tagged PDF
- minim-xmp** for XMP (metadata) inclusion

The documentation for the above packages will be replicated in separate chapters below.

You can use this package by simply saying `\input minim`; this will load the file `minim.tex`. For building your own format files, you can re-use the file `minim.ini`: if you define `\fmtname` before inputting this file, no `\dump` will be performed.

Contents

Compatibility.....	1
Licence.....	1
Metapost	2
Metapost instances.....	2
Running tex from within metapost.....	3
Running lua from within metapost.....	3
Tiling patterns.....	4
Other metapost extensions.....	4
Lua interface.....	5
PDF resource management.....	5
Debugging.....	6
Extending metapost.....	6
Mathematics	7
Styles and alphabets.....	7
Character variants.....	8
Setting up fonts.....	8
Shorthands and additions.....	9
Best practices.....	10
Hypertext	11
Hyperlinks.....	11
Bookmarks.....	11
Page labels.....	11
PDF/A.....	11
Embedded files.....	12
Lua module.....	12

Tagged PDF	13
Purpose, limitations and pitfalls	13
General overview	14
Marked content items	14
Artifacts	14
Document structure	15
Structure element aliases	15
Manipulating the logical order	15
Structure element options	16
Languages	16
Helper macros	17
Metadata	18
Setting metadata	18
Getting metadata	19
Supported metadata keys	19
Adding new keys and schemas	19
Generated XMP	20
Format files	21
Register allocation	21
Callbacks	22
Miscellaneous functions	23

Compatibility

One central design goal of `minim` is to be as unobtrusive as possible: you should be able to safely ignore any function you do not want to use. Please get in touch if you find this not the case.

Particular care has been taken to be compatible with `lualatex`. All overlapping functions should produce the same results and `lualatex` can be loaded either before or after `minim`.

One point of incompatibility exists between `tikz/pgf` and the pattern functionality of `minim-mp`, due to conflicting implementations of pdf resource management. If you do not use filling patterns, however, the two packages can be used together.

Licence

This package may be distributed under the terms of the European Union Public Licence (EURL) version 1.2 or later. An english version of this licence has been included as an attachment to this file; copies in other languages can be obtained at

<https://joinup.ec.europa.eu/collection/eupl/eupl-text-eupl-12>

Metapost



This package offers low-level mplib integration for plain luatex. The use of multiple simultaneous metapost instances is supported, as well as running tex or lua code from within metapost. In order to use it, simply say `\input minim-mp.tex`.

After this, `\directmetapost [options] { mp code }` will result in a series of images corresponding to the `beginfig ... endfig` statements in your mp code. Every image will be in a box of its own.

Every call to `\directmetapost` opens and closes a separate metapost instance. If you want your second call to remember the first, you will have to define a persistent metapost instance. This will also give you more control over image extraction. See below under „Metapost instances”. The `options` will also be explained there (for simple cases, you will not need them).

The logging of the metapost run will be included in the regular log file. If an error occurs, the logging will also be shown on the terminal.

This package can also be used as a stand-alone metapost compiler. Saying

```
luatex --fmt=minim-mp your_file.mp
```

will create a pdf file of all images in `your_file.mp`, in order, with page sizes adjusted to image dimensions.

Metapost instances

For more complicated uses, you can define your own instances by saying `\newmetapostinstance [options] \id`. An instance can be closed with `\closemetapostinstance \id`. These are the options you can use:

Option	Default	Description
<code>jobname</code>	<code>':metapost:'</code>	Used in error messages.
<code>format</code>	<code>'plain.mp'</code>	Format to initialise the instance with.
<code>math</code>	<code>'scaled'</code>	One of <code>scaled</code> , <code>decimal</code> or <code>double</code> .
<code>seed</code>	<code>nil</code>	Random seed for this instance.
<code>catcodes</code>	<code>0</code>	Catcode table for <code>btex ... etex</code> .
<code>env</code>	copy of <code>_G</code>	Lua environment; see below.

Now that you have your own instance, you can run chunks of metapost code in it with `\runmetapost \id { code }`. Any images that your code may have contained will have to be extracted explicitly. This is possible in a number of ways, although each image can be retrieved only once.

`\getnextmpimage \id` – Writes the first unretrieved image to the current node list. There, the image will be contained in a single box node.

`\getnamedmpimage \id {name}` – Retrieves an image by name regardless of its position, and writes it to the current node list.

`\boxnextmpimage \id box-nr` – Puts the next unretrieved image in box `box-nr`. The number may be anything tex can parse as a number.

`\boxnamedmpimage \id box-nr {name}` – Puts the image named `name` in box `box-nr`.

Say `\remainingmpimages \id` for the number of images not yet retrieved. Finally, as a shorthand, `\runmetapostimage \id { code }` will add `beginfig`

... `endfig` to your code and write the resulting image immediately to the current list.

Running tex from within metapost

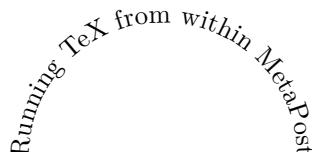
You can include tex snippets with either `maketext "tex text"` or `btex ... etex` statements. The tex code will be executed in the current environment without an extra grouping level. The result of either statement at the place where it is invoked is an image object of the proper dimensions that can be moved, scaled, rotated and mirrored. You can even specify a colour. Its contents, however, will only be added afterwards and are invisible to metapost.

Arbitrary tex statements may be included in `verbatimtex ... etex`, which may occur anywhere. These `btex` and `verbatimtex` statements are executed in the order they are given.

You can also use metapost's `infont` operator, which restricts the text to be typeset to a single font, but returns an `picture` containing a `picture` for each character. The right-hand argument of `infont` should either be a (numerical) font id or the (cs)name of a font.

One possible use of the `infont` operator is setting text along curves:

```
beginfig(1)
  save t, w, r, a; picture t;
  t = "Running TeX from within MetaPost" infont "tenrm";
  w = xpart lrcorner t = 3.141593 r;
  for c within t :
    x := xpart (llcorner c + lrcorner c)/2;
    a := 90 - 180 x/w;
    draw c rotatedaround((x,0), a)
        shifted (-r*sind(a)-x, r*cosd(a));
  endfor
endfig;
```



Running TeX from within MetaPost

Running lua from within metapost

You can call out to lua with `runscript "lua code"`. For this purpose, each metapost instance carries around its own lua environment so that assignments you make are local to the instance. (You can of course order the global environment to be used by giving `env = _G` as option to `\newmetapostinstance`.)

If your lua snippet returns nothing, the `runscript` call will be invisible to metapost. If on the other hand it does return a value, that value will have to be translated to metapost. Numbers and strings will be returned as they are (so make sure the string is surrounded by quotes if you want to return a metapost string). You can return a point or colour by returning an array of two to four elements. For other return values, `tostring()` will be called.

Do keep in mind that metapost and lua represent numbers in different ways and that rounding errors may occur. For instance, metapost's `decimal epsilon` returns 0.00002, which metapost understands as 1/65536, but lua as

1/50000. Use the metapost macro `hexadecimal` instead of `decimal` for passing unambiguous numbers to lua.

Additionally, you should be aware that metapost uses slightly bigger points than tex, so that `epsilon` when taken as a dimension is not quite equal to `1sp`. Use the metapost macro `scaledpoints` for passing to lua a metapost dimension as an integral number of scaled points.

Tiling patterns

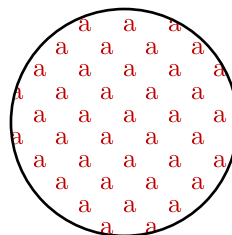
The `withpattern(<name>)` added to a `fill` statement will fill the path with a pattern instead of a solid colour. If the patterns contains no colour information of itself, it will have the colour given by `withcolor`. Stroking operations (the `draw` part) will not be affected. Patterns will always look the same, irrespective of any transformations you apply to the picture.

To define a pattern, sketch it between `beginpattern(<name>) ... endpattern(xstep, ystep)`; where `<name>` is a suffix and `(xstep, ystep)` are the horizontal and vertical distances between applications of the pattern. Inside the definition, you can draw the pattern using whatever coordinates you like; assign a value to the `matrix` transformation to specify how the pattern should be projected onto the page. This `matrix` will also be applied to `xstep` and `ystep`.

You can also change the internal variable `tilingtype` and the normal variable `painttype`, although the latter will be set to 1 automatically if you use any colour inside the pattern definition. Consult the pdf specification for more information on these parameters.

You can use text inside patterns, as in this example:

```
% define the pattern
picture letter; letter = maketext("a");
beginpattern(a)
  draw letter rotated 45;
  matrix = identity rotated 45;
endpattern(12pt,12pt);
% use the pattern
beginfig(1)
  fill fullcircle scaled 3cm withpattern(a) withcolor 3/4red;
  draw fullcircle scaled 3cm withpen pencircle scaled 1;
endfig;
```



A small pattern library is available in the `minim-hatching.mp` file; see the accompanying documentation sheet for an overview of patterns. Tiling patterns cannot be used together with tikz/pgf; see below under ‘Resource management’.

Other metapost extensions

There is currently no support for the `glyph of` operator.

You can set the baseline of an image with `baseline(p)`. There, `p` must either be a point through which the baseline should pass, or a number (where an `x` coordinate of zero will be added). Transformations will be taken into account, hence the specification of two coordinates. The last given baseline will be used.

Previously-defined box resources can be included with `boxresource(nr)`. The result will be an image object with the proper dimensions. This image can be

transformed in any way you like, but you cannot inspect the contents of the resource within metapost.

You can write to tex's log directly with `texmessage "hello"`.

You can write direct pdf statements with `special "pdf: statements"` and you can add comments to the pdf file with `special "pdfcomment: comments"`. Say `special "latelua: lua code"` to insert a `late_lua` whatsit. All three specials can also be given as pre- or postscripts to another object. In that case, they will be added before or after the object they are attached to.

Lua interface

In what follows, you should assume `M` to be the result of

```
M = require('minim-mp')
```

as this package does not claim a table in the global environment for itself.

You can open a new instance with `nr = M.open {options}`. This returns an index in the `M.instances` table. Run code with `M.run (nr, code)` and close the instance with `M.close (nr)`. Images can be retrieved only with `box_node = M.get_image(nr, [name])`; omit the `name` to select the first image. Say `nr_remaining = M.left(nr)` for the number of remaining images.

Each metapost instance is a table containing the following entries:

<code>jobname</code>	The jobname.
<code>instance</code>	The primitive metapost instance.
<code>results</code>	A linked list of unretrieved images.
<code>status</code>	The last exit status (will never decrease).
<code>catcodes</code>	Number of the catcode table used with <code>btex ... etex</code> .
<code>env</code>	The lua environment for <code>runscript</code> .

PDF resource management

This package can add `/Pattern` and `/ColorSpace` entries to all page and xform resource dictionaries. Both refer to a single, global dictionary shared by all pages. Support for other keys may be added in the future.

At the moment, this implementation only serves tiling pattern support; the mechanism will be enabled automatically at the first use of a tiling pattern (merely defining a pattern will not enable it) and is of little use for anything else. The relevant tables, should you want to expand on it yourself, are `M.colourspaces` and `M.patterns`; see the source file for additional instructions.

Since pdf resource management must be done exactly once, this package may clash with other graphics packages doing the same. In particular, `minim`'s resource management will cause double (and thus invalid) entries in pages' attribute dictionaries when used together with `tikz` or `pgf`. They can be used together, however, if you do not use `minim`'s tiling patterns.

Debugging

You can enable (global) debugging by saying `debug_pdf` to metapost or `M.enable_debugging()` to lua. This will write out a summary of metapost object information to the pdf file, just above the pdf instructions that object was translated into. For this purpose, the pdf will be generated uncompressed. Additionally, a small summary of every generated image will be written to log and terminal.

Extending metapost

You can extend this package by adding new metapost specials. Specials should have the form `"identifier: instructions"` and can be added as pre- or postscript to metapost objects. A single object can carry multiple specials and a `special "..."` statement is equivalent to an empty object with a single prefix.

Handling of specials is specified in three lua tables: `M.specials`, `M.prescripts` and `M.postscripts`. The `identifier` above should equal the key of an entry in the relevant table, while the value of an entry in one of these tables should be a function with three parameters: the internal image processor state, the `instructions` from above and the metapost object itself.

If the `identifier` of a prescript is present in the first table, the corresponding function will replace normal object processing. Only one prescript may match with this table. Functions in the the other two tables will run before or after normal processing.

Specials can store information in the `user` table of the picture that is being processed; this information is still available inside the `finish_mpfigure` callback that is executed just before the processed image is surrounded by properly-dimensioned boxes.

The `M.init_files` table contains the list of metapost files that new instances are initialised with. The actual format will be loaded after the files in this table.

Mathematics



This package gives a simple and highly-configurable way of using unicode and OpenType mathematics with plain LuaTeX, making use of most of the latter engine's new capabilities in mathematical typesetting. Also included are proper settings and definitions for nearly all unicode mathematical characters, as well as a few shorthands and helper macros that seemed useful additions.

Load the package by saying `\input minim-math.tex`; this will set up luatex for using opentype mathematical fonts and unicode math input. It will not, however, select mathematical fonts for you. That you will have to do for yourself; see below for instructions.

Styles and alphabets

For some (mostly alphabetical) characters, multiple variants are available, e.g. $AAA\AA\mathcal{A}$. You can (locally) override the default style of these with `\mathstyle {style}` (equivalent to the old `\bf`, `\rm` etc.) or with one of the shorthands that apply the style to their argument only:

Shorthand	Value of style	Result
<code>\mup</code>	up/rm	ABC
<code>\mit</code>	it	<i>ABC</i>
<code>\mbf</code>	bf	ABC
<code>\mbfit</code>	bfit	<i>ABC</i>
<code>\mbb</code>	bb	$\mathbb{A}\mathbb{B}\mathbb{C}$
<code>\frac</code>	frac	$\mathfrak{A}\mathfrak{B}\mathfrak{C}$
<code>\bfrac</code>	bfrac	$\mathfrak{A}\mathfrak{B}\mathfrak{C}$
<code>\scr</code>	cal/scr	$\mathcal{A}\mathcal{B}\mathcal{C}$
<code>\bfscr</code>	bfscr	$\mathcal{A}\mathcal{B}\mathcal{C}$

Styles without shorthand are `sans/sf`, `sfit`, `sfbf`, `sfbfit`, `tt/mono` and finally the special value `clear` for using the default style. You can use the shorthands directly in sub- and superscripts: $v^{\text{scr}}F$ will result in $v^{\mathcal{F}}$.

While math families are not used anymore for switching between styles (see below), you still can use `\fam` with the values 0, 1, 2, 4, 5, 6 or 7 for doing so. This means that plain tex's `\rm`, `\it`, `\cal`, `\sl`, `\bf` and `\tt` can still be used (at least in math mode).

Please note that `\mup` is not the right choice for upright multiletter symbols or operators: you should use `\mord` or `\mop` instead (see near the end of this chapter). For nonmathematical text, you should use `\text` instead of `\mup`.

The default properties of characters can be set with one of the following three commands:

```
\mathmap {character list} {style}  
\mathclass {character list} {class}  
\mathfam {character list} nr
```

There, `style` is one of the above and `class` is the name of a class as below. Finally, the `character list` should be a comma-separated list with elements of one of the following forms:

1. a list of characters, like `abc` or `\partial` or \mathbb{R} ;
2. a character range, like ``A-`Z`, `65-90` or `"41-"5A`;

3. one of the alphabets [Ll]atin, [Gg]reek, or digits;
4. one of the style groups bold, boldgreek, sans, sansgreek, mono, blackboard, fraktur or script;
5. the name of a math class: ord, op, bin, rel, fence, open, close or punct.

Note that unicode is somewhat irregular in its encoding of mathematical letters; this is taken into account when using ranges as under (2) above. Thus, `\mscra-\mscrz` really gives you all lowercase script characters, despite e.g. *e* being well outside that range.

The default style settings are `\mathmap {latin, greek, Latin}{it}`. Since the math family setting is not used anymore for selecting different styles, the default family of every symbol is zero. Instead, you can use `\mathfam` for mixing fonts (see below). The `class` option to `\mathclass` should be one of the names under 5.

Character variants

You can change the default appearance of several greek characters with `\usemathvariant {chars}` or `\usemathdefault {chars}`, where `chars` is a list of normal greek characters. As in unicode but against tex's tradition, the variants are $\epsilon\vartheta\Theta\iota\varpi\varrho\phi$ and the defaults $\epsilon\theta\Theta\kappa\pi\rho\varphi$. The macros `\varepsilon` etc. have been updated to reflect the unicode variants.

The appearance of root symbols can be set with `\closedroots` ($\sqrt{2}$) and `\normalroots` ($\sqrt[2]{2}$, the default).

Say `\unicodedots` to use the unicode dots characters (\dots) and `\traditionaldots` to construct these characters from periods (\cdots , the default). Both settings affect the meaning of both the actual characters and the `\xdots` macros ($x \in \{1, v, c, a, d\}$). Unlike in traditional plain tex, the traditional dots are available in script sizes, too.

Say `\decimalcomma` and have commas appear as 1,2 instead of 1.2 (`\nodecimalcomma` restores the default). The explicit `\comma`, like `\colon`, will always be punctuation.

The behaviour of limits on integral signs can be set by redefining `\intlimits` (the default is `\let \intlimits = \nolimits`).

If you want to change the meaning (inside math mode) altogether for a single character, you can use the commands `\mathdef` and `\mathlet`. For example, by default, you can use the letter `h` for the reduced planck constant \hbar ; this has been made so with `\mathdef h {\hbar}` (you could also have said `\mathlet h \hbar`).

Setting up fonts

The minimum you need do to set up a mathematical font is this:

```
\font\tenmath
    {Latin Modern Math:modes=base;script=math;ssty=0} at 10pt
\font\tenmaths
    {Latin Modern Math:modes=base;script=math;ssty=1} at 7pt
\font\tenmathss
    {Latin Modern Math:modes=base;script=math;ssty=2} at 5pt
\textfont      0 = \tenmath
```

```
\scriptfont 0 = \tenmaths
\scriptscriptfont 0 = \tenmathss
```

Note that you only have to set up the font for a single family: opentype mathematical fonts typically contain all necessary variants of all mathematical characters. Therefore, the `\fam` setting has been made a no-op (use `\setfam` if you really need the old primitive) and the default family of all symbols has been set to zero.

As mentioned above, you can still change the family number of specific characters and this allows you to mix mathematical fonts. For instance, if you dislike the current blackboard bold characters, just assign a second font to family 1 and say `\mathfam {blackboard} 1`. Less useful are the parameters `\accentfam`, `\radicalfam` and `\extensiblefam` that control the family of all accents, radicals and extensibles.

Shorthands and additions

You can use `\text` for adding nonmathematical text to your equations. It will behave well in sub- and superscripts: `\text{word}^{\text{word}}` gives $\text{word}^{\text{word}}$. By default, the font used is the normal mathematical font. You can change this by setting the `\textfam` parameter to some nonzero value and assigning a different font to that family (see above). You probably want to do this, since most commonly-used mathematical fonts do not include a normal kerning table.

All the usual arrows can be made extensible by prefacing them with an `x`, including `\xmapsto` and `\xmapsfrom`. Alternatively, you can use `\rightarrow` etc. as shorthands. Additionally, you can use the following:

Shorthand	Result
<code>\bra x, \ket y</code>	$\langle x , y\rangle$
<code>\braket x y</code>	$\langle x y\rangle$
<code>\norm x, \Norm x</code>	$ x , \ x\ $
<code>x \stackrel{?}{=} y</code>	$x \stackrel{?}{=} y$
<code>x \stackbin{a}{+} y</code>	$x \stackbin{a}{+} y$
<code>f\inv</code>	f^{-1} (cf. f^{-1})
<code>a \xrightarrow[down]{up} b</code>	$a \xrightarrow[down]{up} b$
<code>a \xeq[down]{up} b</code>	$a \xrightarrow[down]{up} b$
<code>\frac12, \tfrac12, \dfrac12</code>	$\frac{1}{2}, \frac{1}{2}, \frac{1}{2}$

Also new are the operators `\Tr`, `\tr`, `\Span`, `\GL`, `\SL`, `\SU`, `\U`, `\SO`, `\O`, `\Sp`, `\im`, `\End`, `\Aut`, `\Dom` and `\Codom`. You can define new operators with `\newmathop` and `\newlargemathop`: `\newmathop{op}` will define the new operator `op`. For single use of an upright symbol, operator or large operator you can use `\mord`, `\mop` and `\mlop`. The difference between `\mord` and `\mup` is that `\mord` also applies the correct symbol spacing.

The accents `\overbrace`, `\underbracket` etc. allow a label between square brackets: `$$\underbrace{[=1]{(x^2+y^2)}}{=1}$$` gives

$$\underbrace{(x^2 + y^2)}_{=1}$$

Best practices

The following remarks on mathematical typesetting have no relation to the contents of this package; I have included them because I find them hard to remember.

1. `\eqalign` gives a vertically centered box and can occur many times in an equation, while `\eqalignno` and `\leqalignno` span whole lines (put the equation numbers in the third column). All assume the relation (or operator) appears at the right hand side of the ampersand.
2. The command `\displaylines` can only have one column that spans the whole line (and you will have to add the equation number by hand).
3. Further alignment commands are `\cases`, `\matrix`, `\pmatrix` (with parentheses) and `\bordermatrix` (includes labels for lines and columns).
4. Finetuning alignments can be done with `\smash`, `\phantom`, `\hphantom` and `\vphantom`.
5. Small matrices like $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ can be made by misusing `\choose` or `\atop`.
6. If you start a line with a binary operator, put a `{}` before it: this way, tex recognises it as such.
7. Thin spaces (`\,`) should be inserted: before dx , before units, after factorials and after `\dots` if those are followed by a closing parenthesis.
8. Whether the differential operator should be set upright or not is as of yet an open question in mathematics.
9. You should prefer `\bigr` and `\bigl` etc. over `\big`, `\Big`, `\bigg` and `\Bigg`.
10. An overview of mathematical symbols, with control sequences and their availability in different fonts, can be found in [unimath-symbols.pdf](#), which is part of the unicode-math package.

Hypertext



This chapter and the next document the support of the modern pdf features provided by the `minim-pdf` package. Load it by saying `\input minim-pdf`. The next chapter concerns the creation of tagged pdf; all other features of the package are described here.

Hyperlinks

For most simple cases, you can use `\hyperlink [name {...} | url {...}] ... \endlink` for linking to a named destination in your own document or to an external hyperlink respectively. There is no support for nonsimple cases.

A named destination can be created with `\nameddestination {...}` (also in horizontal mode, unlike the backend primitive) and if you cannot think of a name, `\newdestinationname` should generate a unique one. If you need the latter twice, `\lastdestinationname` gives the last generated name.

Bookmarks

Bookmarks can be added with `\outline [open] [dest {name}] {title}`. Add `open` to have the bookmark appear initially open and say `dest {name}` for having it refer to a specific named destination (otherwise, a new one will be created where the `\outline` command appears).

A bookmark is automatically associated with the current structure element and the hierarchy of structure elements determines the nesting of bookmarks. Therefore, if you want nested bookmarks, you *must* precede the `\outline` command with a declaration of the current structure element, even if you have otherwise disabled tagging. See the next chapter on how to do this.

Page labels

If the page numbers of your document are not a simple sequence starting with 1, you can use `\setpagelabels [pre {prefix}] style nr` for communicating this to the pdf viewer. This command affects the page labels from the next page on: `nr` should be the numerical page number of that page. The `prefix` is prepended to each number and the `style` must be one of `decimal`, `roman`, `Roman`, `alphabetic`, `Alphabetic` or `none`. In the last case, only the prefix is used.

PDF/A

You can declare pdf/a conformance with `\pdfalevel xy`, with version $x \in \{1, 2, 3\}$ and conformance level $y \in \{a, b, u\}$. This will set the correct pdf version and `pdfaid` metadata. If the conformance level is ‘a’, tagging will be enabled (see the next chapter). Finally, a default RGB colour profile will be included. The conformance level can be queried from the `\pdfaconformancelevel` register.

Note that merely declaring conformance will not make your document pdf/a compliant, and that `minim` will not warn you if it is not. However, the features described in this chapter and the next should be enough to make pdf/a compliance possible.

Also note that there currently is no documented way of choosing a different colour profile from the default (i.e. the default rgb profile provided by the colorprofiles package). Should you need do that, you will have to do so manually, after disabling the automatic inclusion by saying `\expandafter\let \csmname minim:default:rgb:profile\endcsname = \relax`.

Finally, note that pdf/a requires that spaces are represented by actual space characters and that discretionary hyphens are marked as soft hyphens (U+00AD). Since both features benefit accessibility and text extraction in general, they are enabled by default. You can disable them by setting `\writehyphensandspaces` to a nonpositive value.

Embedded files

You can attach (associate) files with `\embedfile <options>`. The file will be attached to the current structure element (see the next chapter) unless the `global` option is given: then it will be added to the document catalog. Arguments consisting of a single word can be given without braces and exactly one of the options `file` or `string` must be present.

<code>file</code>	<code>{...}</code>	The file to embed.
<code>string</code>	<code>{...}</code>	The string to embed.
<code>global</code>		Attach to the document catalog.
<code>uncompressed</code>		Do not compress the file stream.
<code>mimetype</code>	<code>{...}</code>	The file's mime type.
<code>moddate</code>	<code>{...}</code> *	The modification date (see * below).
<code>desc</code>	<code>{...}</code>	A description (the <code>/Desc</code> key).
<code>relation</code>	<code>{...}</code>	The <code>/AFRelationship</code> value as defined in pdf/a-3.
<code>name</code>	<code>{...}</code>	The file name (only required when writing a <code>string</code>).

* The modification date must be of the form `yyyy[-m[m] [-d[d]]]`. A default `moddate` can be set with `\setembeddedfilesmoddate {default}`. The `default` date will be expanded fully at the time of embedding. With the `minim-xmp` package, a useful setting is `\setembeddedfilesmoddate {\getmetadata date}`.

Lua module

The interface of the lua module (available via `local M = require('minim-pdf')`) is not stable yet, and may change. One function of interest, however, is `M.pdf_string(...)`, which converts a lua string to a pdf string. The surrounding `<>` or `()` characters are included in the return value.

Tagged PDF



This chapter is a continuation of the previous and describes the parts of `minim-pdf` that concern the creation of tagged pdf. All features in this chapter must be explicitly enabled by setting `\writedocumentstructure` to a positive value. This will be done automatically if you declare pdf/a conformance (see above).

This part of the package is rather low-level and this chapter rather technical. For a more general introduction to and discussion of tagged pdf, please read the (excellent) manual of latex's `tagpdf` package.

Purpose, limitations and pitfalls

The main purpose of this package is semi-automatically marking up the (hierarchical) structure of your document, thereby creating so-called tagged pdf. The mechanism presented here is not quite as versatile as the pdf format allows. The most important restriction is that all content of the document must be seen by tex's stomach in the *logical* order.

Furthermore, while the macros in this package are sophisticated enough that tagging can be done without any manual intervention, it is quite possible and rather easy to generate the wrong document structure, or even cause syntax errors in the resulting pdf code. You should always check the result in an external application.

This is the full list of limitations, pitfalls and shortcomings:

1. Document content must be seen by tex in its logical order (although you can mark out-of-order content explicitly if you know what you are doing; see below).
2. Artifacts cannot be split across pages. A pagebreak inmidst an artifact will cause incorrect pdf without error or warning.
3. The contents of `\localleftbox` and `\localrightbox` must be marked manually, probably as artifact.
4. You must mark page header, page footer and footnote rule yourself; no default is set.
5. There currently is no way of marking xforms or other pdf objects as content items of themselves.
6. The content of xforms (i.e. pdf objects created by `\useboxresource`) should not contain tagging commands.
7. Likewise, you should be careful with box reuse: it might work, but you should check.
8. The use of structure element attributes is currently not supported except in a limited number of cases.
9. This package currently only supports pdf 1.7 tagging and is not yet ready for use with pdf 2.0.

In order to help you debugging, some errors will refer you to the resulting pdf file. If you get such errors, decompress the pdf and search for the string 'Warning:'. It will appear in the pdf stream at the exact spot the problem occurs.

General overview

When speaking about tagging, we have to do with two (or perhaps three) separate and orthogonal tagging processes. The first is the creation of a hierarchical *document structure*, made up of *structure elements* (SEs). The document structure describes the logical structure of a document, made up of chapters, paragraphs, references etc. The second tagging process is the tagging of *marked content items* (MCIs): this is the partition of the actual page contents into (disjoint) blocks that can be assigned to the proper structure element. Finally, as a separate process, some parts of the page can be marked as *artifacts*, excluding their content from both content and structure tagging.

When using this package, artifacts and structure elements (excluding paragraphs; see below) must be marked explicitly, while marked content items will be created, marked and assigned automatically. There is some (partial and optional) logic for automatically arranging structure elements in their correct hierarchical relation.

The mechanism through which this is achieved uses attributes and whatsits for marking the contents and borders of SEs, MCIs and artifacts. At the end of the output routine, just before the pdf page is assembled, this information will be converted into markers inserted in the pdf stream.

Marked content items

Content items are automatically delineated at page, artifact and structure element boundaries and terminated at paragraph or display skips. This should relieve you from any manual intervention. However, if you run into problems, the commands below might be helpful.

Use of `ActualText`, `Alt` or `Lang` attribute on MCIs, while allowed by the pdf standard, is not supported by this package. You should set these on the structure element instead.

The beginning and ending of a content item can be forced with `\startcontentitem` and `\stopcontentitem`, while `\ensurecontentitem` will only open a new content item if you are currently outside any. If you need some part to be a single content item, that can use `\startsinglecontentitem ... \stopsinglecontentitem`. This will disable all SE and MCI tagging inside.

Tagging (both of MCIs and SEs) can be disabled and re-enabled locally with `\stoptagging` and `\starttagging`.

Artifacts

Artifacts can be marked in two ways: with `\markartifact {type} {...}` or with `\startartifact {type} ... \stopartifact`. The `type` is written to the pdf attribute dictionary directly, so that if you need a subtype, you can write e.g. `\startartifact {Pagination /Subtype/Header} etc.` Do make sure your artifact does not contain a page break, as this will result in invalid output.

Inside artifacts, other structure content markers will be ignored. Furthermore, this package makes sure artifacts are never part of marked content items, automatically closing and re-opening content items before and after the artifact. While the pdf standard does not require the latter, not enforcing this seems to confuse some pdf software.

Document structure

Like artifacts, structure elements can be given as `\markelement {Tag} {...}` or `\startelement {Tag} ... \stopelement {Tag}`. Here, in many cases the `\stopelement` is optional: whenever opening an element would cause a nesting of incompatible Tags, the current element will be closed until such a nesting is possible. Thus, opening a TR will close the previous TR, opening an H1 will automatically close any open inline or block structure elements, opening a TOCI will close all elements up until the current TOC etc. etc.

As a special case, the tags `Document`, `Part`, `Art`, `Sect` and `Div` (and their aliases) will try and close all open structure elements up to and including the last structure element with the same tag. (An alias will of course only match the same alias.)

While the above can greatly reduce the effort of tagging, the logic is neither perfect nor complete. You should always check the results in an external application. Particular care should be taken when ‘skipping’ structure levels: the sequence chapter – subsection – section will result in the section beneath the subsection. If you are in doubt about an element being closed already, you can use `\ensurestopelement {Tag}` instead of `\stopelement` to prevent an error being raised.

All these helpful features can also be disabled by setting `\strictstructuretagging` to a positive value. Then, every structure element will have to be closed by an explicit closing tag, as in xml. In this case, `\stopelement` and `\ensurestopelement` will be equivalent.

By default, P structure elements are inserted automatically at the start of every paragraph. The tag can be changed with `\nextpartag {Tag}`; leaving the argument empty will prevent marking the next paragraph. Auto-marking paragraphs can be (locally) disabled or enabled by saying `\markparagraphsfalse` or `\markparagraphstrue`.

Structure element aliases

New structure element tags can be created with `\addstructuretype [options] Existing Alias`. This will create a new structure tag named `Alias` with the same properties as `Existing`. The properties can be modified by specifying `options`: these will set values of the corresponding entry in the `structure_types` table (see the lua source file for this package). Any aliases you declare will be written to the pdf’s `RoleMap` only if they have actually been used.

Manipulating the logical order

With the process outlined above, the logical order of structure elements has to coincide with the order in which the SEs are ‘digested’ by tex. This, together with the marked content items being assigned to structure elements in their order of appearance, lies behind the restriction that logical and processing orders should match.

With manual intervention, this restriction can be relaxed somewhat. Issuing the pair `\savecurrentelement ... \continueelement` will append the MCIs following `\continueelement` to the SE containing `\savecurrentelement`. Since the assignments made here are global, this process cannot be nested; in more complicated situations you should therefore use `\savecurrentelementto\name`

... `\continueelementfrom\name` which restores the current SE from a named identifier `\name`.

Structure element options

The `\startcontentitem` command allows a few options that are not mentioned above: its full syntax is `\startcontentitem <options> {Tag}`. The three most useful options are `alt` for setting an alt-text (the `/Alt` entry in the structure element dictionary), `actual` for a text replacement (`/ActualText`) and `lang` for the language (`/Lang`; see the next section). The alternative and actual texts can also be given after the fact with `\setalttext{...}` and `\setactualtext{...}`. These apply to the current structure element.

Setting structure element attributes is not supported at this moment, except the placement attributes `block` and `inline`, which can be given as options.

Languages

If you do not specify a language code for a structure element, its language will be determined automatically. In order for this to work, you must associate a language code to every used language; you can do so with `\setlanguagecode name code`, where `name` must be an identifier used with `\uselanguage {name}` and `code` must be a two or three-letter language code, optionally followed by a dialect specification, a country code, and/or some other tag. Note that the language code is associated to a language *name*, not to the numerical value of the `\language` parameter. This allows you to assign separate codes to dialects.

There is a small set of default language code associations, which can be found in the file `minim-languagecodes.lua`. It covers most languages defined by the `hyph-utf8` package, as well as (due to their ubiquitous use) many ancient languages.

An actual language change introduced by `\uselanguage` will not otherwise be acted upon by this package. Therefore, you will probably want to add `\startelement{Span}` after every in-line invocation of `\uselanguage`.

You can set the document language with `\setdocumentlanguage language-code`. If unset, the language code associated with the first `\uselanguage` statement will be used, or else `und` (undetermined). The only function of the document language is that it is mentioned in the pdf catalog: it has no other influence.

New languages can be declared with `\newnamedlanguage {name} {lhm} {rhm}` and new dialects with `\newnameddialect {language name} {dialect name}`. Dialects will use the same hyphenation patterns (and will indeed have the same `\language` value) as their parent languages; newly declared languages will start with no hyphenation patterns. Do note that you will probably also have to specify language codes for new languages or dialects.

This package ensures the existence of the `nohyph`, `nolang`, `uncoded` and `undetermined` dummy languages, all without hyphenation.

Helper macros

For marking up an entry in a table of contents, you can use the macro `\marktocentry {dest} {lbl} {title} {filler} {pageno}`, which should insert all tags in the correct way. (The `dest` is a link destination and can be empty; the `lbl` is a section number and can also be empty.)

For marking up tables, a whole array of helper macros is available. First, `\marktable` should be given *before* the `\halign`. Then, in the template, the first cell should start with `\marktablerow` and each subsequent cell with `\marktablecell`. If your table starts with a header, insert `\marktableheader` before it and `\marktablebody` after. Before a table footer, insert `\marktablefooter`.

For greater convenience, insert just `\automarktable` before the `\halign`. Then you can leave out all the above commands (unless you `\omit` a template of course). This assumes the table has a single header row and more than one column. If you use a table for typesetting a list, you can use `\marktableaslist` instead, which marks the first column as list label and the second column as list item. Of course, this only works with two-column tables.

Finally, you can auto-tag equations as formulas by specifying `\autotagformulas`. This is especially dangerous, because sometimes equations are used for lay-out and should not be marked as such. After the latter command, auto-tagging can be switched off and on with `\stopformulatagging` and `\startformulatagging`.

Both `alt` and `actualtext` of the `Formula` structure element will be set to the (unexpanded) source code of the equation, surrounded by the appropriate number of dollar signs. Furthermore, if `\pdfaconformancelevel` equals three, the source of the formula will be attached in an embedded file with the `/AFRelation` set to `Source`. The name of this file can be changed by redefining `\formulafilename` inside the equation.

Metadata



This package enables simple XMP (eXtensible Metadata Platform) packet inclusion and will automatically generate pdf/a extension schemas. Use it by saying `\input minim-xmp.tex`. Thereafter, you can use `\setmetadata key {value}` and `\getmetadata key` for setting and retrieving document-level metadata values.

You do not need this package if you have your metadata ready-made in a separate file, for then you can simply say

```
\immediate\pdfextension obj uncompressed
  stream attr {/Type/Metadata /Subtype/XML}
  file {your-file.xmp}
\pdfextension catalog
  {/Metadata \pdffeedback lastobj 0 R}
```

Setting metadata

Metadata fields that contain (ordered or unordered) lists will be split on the `\metadataseparator` character; this is a semicolon by default. Alternatively, you can just make multiple assignments: these will be concatenated.

Where applicable, language alternatives can be given like `\setmetadata /dc:title {...}` or `\setmetadata /{de_DE} dc:title {...}`. Braces are necessary in the second case because the catcode of the underscore is not 11 or 12. When no alternative is given, the value `x-default` will be used.

Instead of using `\setmetadata`, multiple fields can be set in one go with `\startmetadata`. This way is particularly useful when assigning structured data to a key (see later on). In this example, `key1` contains a normal value, `key2` language alternatives and `key3` structured data:

```
\startmetadata
  key1 {...}
  key2 {... (default) ...}
  /alt1 {...}
  /alt2 {...}
  key3
  /field1 {...}
  /field2 {...}
stopmetadata
```

Since metadata values are read by lua as text, linebreaks and paragraphs are not preserved. You can work around this by saying `{\def\par{\Uchar"A\Uchar"A}\setmetadata abstract {...}}`.

Getting metadata

Metadata values can be retrieved again with `\getmetadata key`. This command is fully expandible.

If the data is a list, it will be returned according to the current value of `\metadataseparator`. If the data has language alternatives, the x-default value will be returned: the alternatives are accessible by `\getmetadata /lang key`.

For structured types (discussed below), `\getmetadata /field key` will return the value of a single field and `\getmetadata key` will return all fields as `/{field1} {value1} /{field2} {value2} ...` (this can be used again as input to `\startmetadata`).

Supported metadata keys

Initially, the `\setmetadata` and `\getmetadata` recognise all pdf/a compatible fields in the `pdf`, `pdfaid`, `dc`, `xmp`, `xmpMM` and `xmpRights` namespaces. Keys should be prefixed with their namespace, e.g. `dc:creator` or `xmp:CreatorTool`. Note that the `dc` namespace has lower-case fields; field names are case-sensitive.

Because the precise details of the above metadata namespaces can be confusing, you might want use one of the aliases `author` (`dc:creator`), `title` (`dc:title`), `date` (`dc:date` and `xmp:CreateDate`), `language` (`dc:language`), `keywords` (`dc:subject` and `pdf:Keywords`), `publisher` (`dc:publisher`), `abstract` (`dc:description`), `copyright` (`dc:rights`), `version` (`xmpMM:VersionID`) and `identifier` (`dc:identifier`). New aliases can be defined in the `aliases` table of the lua module.

Adding new keys and schemas

New metadata namespaces can be added in the following way:

```
require('minim-xmp').add_namespace(  
  'Example namespace', 'colours',  
  'http://example.com/minim/colours/', {  
    -- metadata keys  
    Favourite = { 'Colour', 'The author's favourite colour' },  
  }, {  
    -- value types  
    Colour = { 'RGB Colour', {  
      R = { 'Integer', 'Red component' },  
      G = { 'Integer', 'Green component' },  
      B = { 'Integer', 'Blue component' }  
    }, prefix = 'c' },  
  })
```

This will setup a namespace with prefix `colours` and one key: `Favourite`, of type `Colour`. That value type happens to be a structured type with three fields, which are also described. You can now use this namespace as

```
\startmetadata colours:Favourite /R 0 /G 0 /B 255 stopmetadata
```

or the equivalent but more verbose

```
\setmetadata/R colours:Favourite 0  
\setmetadata/G colours:Favourite 0  
\setmetadata/B colours:Favourite 255
```

after which the generated XMP code will be

```
<rdf:Description rdf:about=""
  xmlns:colours="http://example.com/minim/colours/"
  xmlns:c="http://example.com/minim/colours/">
  <colours:Favourite rdf:parseType="Resource">
    <c:B>255</c:B>
    <c:G>0</c:G>
    <c:R>0</c:R>
  </colours:Favourite>
</rdf:Description>
```

You probably will not need defining your own value types, so in most cases you should omit the fifth argument to `add_namespace`. If you do define a new value type, you can specify its prefix if it differs from the namespace prefix (as is done above) and likewise its uri identifier if it differs from the namespace URI.

List types can be given as `'Bag TypeName'` or `'Seq TypeName'`; language alternatives as `'Lang Alt'`. All other types will be treated as `'Text'`, though for `'Boolean'`, `'Integer'` and `'Date'` some validation is performed when setting values.

Additional metadata value type handling can be defined in the `getters` and `setters` tables of the lua module. Appropriate entries to those tables will be generated automatically for new structured types (which is why you could set the colour like we did above). Value types without fields, however, will be stored and retrieved as if they were `Text` until you provide another way.

Generated XMP

All metadata will be written to the PDF file uncompressed.

The `\metadatamodification` setting controls whether XMP packets will be marked as read-only (value 0; default) or writeable (value 1). Writeable XMP packets will be padded with about 2 kB of whitespace.

If the document-level metadata contains values in the `pdfaid` namespace, metadata extension schemas will be appended to the document-level metadata packet automatically. These extension schemas will only include keys that have been set somewhere, though they need not have been set in the document-level metadata. No extension schemas are generated for the built-in namespaces, as they are already included in the pdf/a standards.

This chapter describes the programming helper modules on which all the above modules depend. It mainly concerns register allocation, callback management and format file inclusion.

They can be loaded separately by saying `\input minim-alloc`; thereafter, you can use `local M = require('minim-alloc')` to access the lua interface. In this chapter, when discussing lua functions, you are assumed to have issued the latter statement, so that the table `M` refers to the contents of this module.

The callback-related code lives in a separate file that can and must be loaded separately as `local C = require('minim-callbacks')`. This is the only file in this collection that does not itself depend on the `minim-alloc` module.

There is a large functional overlap between this module and the `lualatex` package. You can use both at the same time, however, and the order in which you load both packages should not matter.

Format files

A major motivation for writing this module (and not, instead, depending on `l\lualatex.tex`) is the ability to write lua-heavy code that can be safely included in format files. For this purpose, the register allocation functions described below allow ensuring that the allocation is made only once.

Apart from registers, you need only do two more things to make your code format file safe. The first is saying `M.remember('your-file.lua')` somewhere, anywhere. This will mark your file for inclusion in the format. At the start of every job, all remembered files will be executed (in order) and their return values will be stored to be retrieved whenever you say `require('your-file')`. Note that while this feature does not improve speed in any meaningful way, it will ensure the lua file used by the format is identical to the one used to create it.

It does mean, however, that your file may be executed twice: once when building the format and once when the format is used. In most cases (e.g. callback registration) this is exactly what you want. Sometimes however, you may need to store variable (configurable) data in the format file. You can do this by saying `local t = M.saved_table('identifier', default-table)`. This will retrieve the table from the format file if possible; otherwise, it will return `default-table` and mark it to be saved to the format. A missing second argument is equivalent to an empty table. Saved tables may only contain (arbitrary but non-cyclic nestings of) tables, numbers and strings.

Register allocation

For allocating the new luatex registers, you can use the following: `\newfunction`, `\newattribute`, `\newwhatsit`, `\newluabytecode`, `\newluachunkname`, `\newcatcodetable` and `\newuserrule`. Note the one difference with `l\lualatex`, which has `\newluafunction` instead. (The reason for this is that `l\lualatex`, instead of a more sensible method, uses this macro for determining whether it has been read before.) Internally, the very same counts are used for keeping track of register allocation as in `l\lualatex`. Their effect should therefore be identical in all circumstances, with one exception: no bounds checking is performed on any allocation macro defined by `minim`. Please do not go and use more than sixty five thousand different `whatsits`.

All the above and all traditional registers can be allocated from within lua as well, using `M.new_count('name')`, `M.new_whatsit('name')` etc. All return the allocated number. The (optional) string `name` prevents the same allocation from being made twice: if another register has been retrieved with the same name, the number of that register will be returned. You will need this when you want to allow your lua code to be included in a format file; it has nothing to do with the tex-side `\countdef` and the like.

For the new allocation macros listed above and (as a special case) for `\newbox`, after saying `\newwhatsit\name`, the call `M.new_whatsit('name')` will return the number of `\name`. For the other (older) allocation macros, this is not the case. After all, because of the `\countdef` etc. included in `\newcount` etc. you can already use `tex.count['name']` etc. for retrieving tex-side allocations from lua. The exceptions to this are `\newbox`, which is why it is included with the new macros, and `\newattribute`, for which you can use both methods.

Besides `\newluachunkname\name`, you can also use `\setluachunkname \name {actual name}` to enter the value of the name directly.

Finally, for the registers for which etex defines a local allocation macro (and for those only), you can use `M.local_count()` etc. These allocation functions take no parameters.

Callbacks

As noted at the beginning of this chapter, the callback functions are only available after you say `local C = require('minim-callbacks')`.

This module will override the primitive callback functions with its own `C.register`, `C.find` and `C.list`; the original primitive functions can be found in the `C.primitives` table.

The simple function of this module is allowing multiple callbacks to co-exist. Different callbacks call for different implementations, and some callbacks can only contain a single function. Any callbacks that are already assigned before loading this module will be preserved; this includes the l^uatex callback mechanism if it has already been installed.

You can create your own callbacks with `C.new_callback(name, type)`. The `type` should be one of the types mentioned below or `'single'` for callbacks that allow only one function. If the `name` is that of a primitive callback, new registrations will target your new callback. You can call the new callback with `C.call_callback(name, ...)`, adding any number of parameters.

Callbacks of type `node` operate on a node list: for these, all registered functions will be called in order, each function receiving the result of the last. After one function returns `false`, no others will be called. Callbacks of this type are `pre_linebreak_filter`, `post_linebreak_filter`, `hpack_filter`, `vpack_filter`, `pre_output_filter` and `mlist_to_mlist`. There is no way of unregistering callbacks of this type.

Similarly, for the `data` callbacks `process_input_buffer`, `process_output_buffer` and `process_jobname`, all registered functions will be called in order on the output of the previous. Returning `false` will in this case result in the output of the previous function passing to the next.

For `stack` callbacks, a stack is kept and only the top function on the stack will be called. These are `mlist_to_hlist`, `hpack_quality`, `vpack_quality`, `hyphenate`, `linebreak_filter`, `buildpage_filter` and `build_page_insert`. Register `nil` at the callback to pop a function off the stack.

Finally, for the `simple` callbacks `contribute_filter`, `pre_dump`, `wrapup_run`, `finish_pdffile`, `finish_pdfpage`, `insert_local_par`, `ligaturing`, `Kerning` and `process_rule`. all registered functions are called in order with the same arguments.

The new `mlist_to_mlist` callback is called before `mlist_to_hlist` and should not convert noads to nodes.

If you create a new callback with a number for a name, that callback will replace the `process_rule` callback when its number matches the index property of the rule.

Miscellaneous functions

This section describes functions and macros that are internal to this package, but might be of general usefulness.

For instance, you might find the function `M.table_to_text(table)` useful when debugging lua code. The small functions `M.msg(...)`, `M.log(...)` and `M.err(...)` include a call to `M.string.format`; additionally, `M.amsg(...)` and `M.alog(...)` do not start a new line.

Very useful is `M.luadef('csname', function, ...)` for defining primitive-like tex macros from lua: `function` can be any function (it will be assigned a lua function register) and at the place of the dots you may append `'protected'` and/or `'global'`.

Both `M.unset` and `\unset` contain the value `-0x7FFFFFFF` that can be used for clearing attributes.

On the tex side, `\voidbox`, `\ignore`, `\spacechar`, `\unbrace`, `\firstoftwo` and `\secondoftwo` should be self-explanatory and uncontroversial additions. For looking ahead, you can use `\nextif \token {executed if present} {executed if not}` or its siblings `\nextifx` and `\nextifcat`. For defining macros with optional arguments, `\withoptions[default]{code}` will ensure something within square brackets follows `code`.

Finally, `\splitcommalist {code} {list}` will apply `code` to every nonempty item on a comma-separated `list`. Items of the list will be re-tokenised and have surrounding spaces removed. This macro is fully expandable.

Because of their usefulness and simplicity, these macros have been made available without special characters in their names; I hope you can tolerate their presence. Please let me know if their names clash with something important.