

BBE

merging, embedding, fixing
and messing a bit around
in context lmtx

Introduction

The three graphic formats that make most sense for inclusion in pdf are png, jpg, and pdf. The easiest of these is jpg because basically the binary blob gets transferred to the result file. A png graphic might need more work because what actually is supported is basic png inclusion. It means that often the image data has to be unpacked and split into pdf counterparts that get embedded. The pdf format is quite convenient because basically we only need to copy the used objects to the result, so when those objects are for instance png encoded images, we gain runtime, but when we're talking pages of documents it might take some more. Nevertheless, in practice it is still quite efficient.

This manual describes how to manipulate pdf files that are not behaving well or from which pages are to be embedded another pdf file within constraints. We discuss how to cleanup and/or embed fonts, fix colors, get rid of interfering resources and fix the page stream.

Many thanks to Massimiliano Farinella for teaming up to make this all work better and conducting extensive tests on complex documents.

Hans Hagen
Hasselt NL
January 2024⁺⁺

Embedding pdf files

Here we focus on pdf inclusion where we have several scenarios to deal with:

- A straightforward inclusion of a single page pdf file.
- Inclusion of a specific page from a pdf file.
- Inclusion of several pages from a pdf file.
- Inclusion of one or more pages from several pdf files.

To this we can add:

- Inclusion of one or more pages from pdf files that are generated independently (subruns) for instance in the process of writing a manual about something ConTeXt. Think of externally processed buffers.

The most natural way to include pages is to use the `\externalfigure` command but later we will see that there are more ways to manipulate pdf files.

```
\externalfigure[myfile.pdf] [page=4]
```

If you have problems with the inclusion that originate in the compact features discussed here you can say:

```
\externalfigure[myfile.pdf] [page=4,compact=]
```

but also make sure to tell us what goes wrong so that we can fix it. We can't predict what pdf is fed into the machinery.

Embedding multiple pdf files and sharing common content

When we include more than one page from a file, we only need to embed shared objects once. Of course it demands some object management but that has to be done anyway. We could share objects across files but that demands more memory and runtime and the savings are likely to be small, with one exception: fonts. It would be nice if we can embed missing fonts and also merge fonts that are the same. This can make the result much smaller, especially when we're talking of including examples of typesetting in a manual that uses the same fonts.

Another aspect of inclusion is the quality of the to be embedded page. Here you can think of errors in the page stream, color spaces that don't match, missing properties, invalid metadata, etc. Often there's not much we can do about it, but sometimes we can. However, it has to happen under user control and the outcome has to be checked, although often a visual check is good enough.

The compact parameter and fonts merging

The compact parameter of `\externalfigure` controls the embedding of pdf content. When set to 'yes' it will merge fonts but only when the file is produced by ConTeXt LMTX. The reason for not checking all fonts by default comes from the fact that references from the page stream to glyphs in the font depend on the application that made the pdf. In some cases the mapping is using the original glyph index, but one can never be sure. Using the `tounicode` map to go from page stream index to glyph index is also not reliable because multiple glyphs can have the same Unicode slot and when font features are applied (say small caps) you actually don't know that.

The mentioned yes option is a preset that has been defined like:

```
\startluacode
graphics.registerpdfcompactor ( "yes", {
  merge = {
    lmtx = true,
  },
} )
\stopluacode
```

Another preset is merge:

```
\startluacode
graphics.registerpdfcompactor ( "merge", {
  merge = {
    type0    = true,
    truetype = true,
    type1    = true,
    lmtx     = true,
  },
} )
\stopluacode
```

Currently we don't support Type3 optimization. It is doable but probably not worth the effort.

We can also force embedding of fonts that are not included in the document that we get the page from. This is unlikely unless you have old documents.

```
embed = {
  type0    = true,
  truetype = true,
  type1    = true,
}
```

References to glyphs in the page stream use an eight bit string encoding or an hexadecimal byte pairs. Depending on the font type we have up to 256 references (using one character or two hex bytes) or at most 65536 references (using two characters or 4 hex bytes). We normalize everything to hex encoding. That way we get rid of the ugly escapes and exceptions in page stream glyph string.

There are two trackers:

```
\enabletrackers[graphics.fonts]
\enabletrackers[graphics.fixes]
```

The first one reports what is done with fonts. When embedding or merging is not possible you can try to remap the found font onto one on your system. Here are some examples:

```
graphics.registerpdfont {
  source = "arial",
  target = "file:arial.ttf",
}
graphics.registerpdfont {
  source = "arial-bold",
  target = "file:arialbd.ttf",
}
graphics.registerpdfont {
  source = "arial,bold",
  target = "file:arialbd.ttf",
}
graphics.registerpdfont {
  source = "helvetica",
  target = "file:arial.ttf",
  -- unicode = true,
}
graphics.registerpdfont {
  source = "helvetica-bold",
  target = "file:arialbd.ttf",
  -- unicode = true,
}
graphics.registerpdfont {
  source = "courier",
  target = "file:cour.ttf",
}
graphics.registerpdfont {
  source = "ms-pgothic",
  unicode = true, -- via unicode (false for composite)
}
```

The unicode key needed when you get rubbish due to the indices in the page stream being different from glyph indices in the used font. In that case we go via the `tunicode` vector which works ok for the average simple document not using special font features. There is some trial and error involved but that is probably worth the effort when you have to manipulate many documents.

Manipulating properties other than fonts

There are two activities when we compact: fonts and content. When content is handled additional parsing of the page stream has to happen. What gets processed is determined by the identify table:

```
identify = {
  content    = true,
  resources  = true, -- needs checking
  page       = true, -- needs checking
}
```

although this is equivalent:

```
identify = "all"
```

As a proof of concept we can recolor an included file. Of course this assumes a rather simple use of color. Here is an example:

```
\startluacode
  graphics.registerpdfcompactor ( "preset:demo-1", {
    identify = {
      content  = true,
      resources = true,
      page     = true,
    },
    merge = {
      type0    = true,
      truetype = true,
      type1    = true,
      lmtx     = true,
    },
    recolor = {
      viagray = { 1, 0, 0 },
      -- viagray = { 0, 1, 0 },
      -- viagray = { 0, 1, 0, .5 },
      -- viagray = { .75 },
    }
  } )
\stopluacode
\setupexternalfigures[compact=preset:demo-1]
\startTEXpage
  \startcombination[3*4]
    {\externalfigure[test-000.pdf][frame=on]}      {\LUAMETATEX\ 0}
    {\externalfigure[test-001.pdf][frame=on]}      {\LUATEX\ 1}
    {\externalfigure[test-002.pdf][frame=on]}      {\LUATEX\ 2}
    {\externalfigure[test-003.pdf][frame=on,page=1]} {\LUATEX\ 3.1}
    {\externalfigure[test-003.pdf][frame=on,page=2]} {\LUATEX\ 3.2}
    {\externalfigure[test-003.pdf][frame=on,page=3]} {\LUATEX\ 3.3}
    {\externalfigure[test-004.pdf][frame=on,page=1]} {\PDFTEX\ 4.1}
    {\externalfigure[test-004.pdf][frame=on,page=2]} {\PDFTEX\ 4.2}
    {\externalfigure[test-004.pdf][frame=on,page=3]} {\PDFTEX\ 4.3}
    {\externalfigure[test-005.pdf][frame=on,page=1]} {\PDFTEX\ 4.1}
    {\externalfigure[test-005.pdf][frame=on,page=2]} {\PDFTEX\ 5.2}
    {\externalfigure[test-005.pdf][frame=on,page=3]} {\PDFTEX\ 5.3}
  \stopcombination
\stopTEXpage
```

In figure 1 we make a single page document that embeds 12 pages from six files made by several engines. The six files have a total of about 114K but the single page combination is only 19K. The test files are:

```
% \nopdfcompression

\starttext
  \startTEXpage[offset=1ex]
    test \type {some} more
  \stopTEXpage
\stoptext
```

So this one is an LMTX produced file. The next two files:

```
% engine=luatex
% \nopdfcompression
\starttext
  \startTEXpage[offset=1ex]
    test \type {test}
  \stopTEXpage
\stoptext
```

and

```
% engine=luatex
% \nopdfcompression
\starttext
  \startTEXpage[offset=1ex]
    last \type {last}
  \stopTEXpage
\stoptext
```

are done by Lua \TeX with MkIV and

```
% engine=luatex
% \nopdfcompression
\starttext
  \startTEXpage[offset=1ex]
    rest \type {rest}
  \stopTEXpage
  \startTEXpage[offset=1ex]
    whatever \type {whatever}
  \stopTEXpage
  \startTEXpage[offset=1ex]
    more \type {more}
  \stopTEXpage
\stoptext
```

as well as

```
% engine=pdfTeX
% \nopdfcompression
\starttext
  \startTEXpage[offset=1ex]
    rest \type {rest}
  \stopTEXpage
  \startTEXpage[offset=1ex]
    whatever \type {whatever}
  \stopTEXpage
```

```

    \startTEXpage[offset=1ex]
      more \type {more}
    \stopTEXpage
\stoptext

and

% engine=pdftex

% \nopdfcompression

\starttext
  \startTEXpage[offset=1ex]
    rest \type {rest}
  \stopTEXpage
  \startTEXpage[offset=1ex]
    whatever \type {whatever}
  \stopTEXpage
  \startTEXpage[offset=1ex]
    more \type {more}
  \stopTEXpage
\stoptext

```

are typeset with pdf_{TE}X and MkII so they have the Type1 instead of the OpenType Latin Modern file embedded (in fact, the MkII and MkIV files use the twelve point variant and LMTX the upscaled ten point), so if those were the same we would have an even smaller final file.

test some more	test test	last last
luameta _{TE} X 0	Lua _{TE} X 1	Lua _{TE} X 2
rest rest	whatever whatever	more more
Lua _{TE} X 3.1	Lua _{TE} X 3.2	Lua _{TE} X 3.3
rest rest	whatever whatever	more more
pdf _{TE} X 4.1	pdf _{TE} X 4.2	pdf _{TE} X 4.3
rest rest	whatever whatever	more more
pdf _{TE} X 4.1	pdf _{TE} X 5.2	pdf _{TE} X 5.3

Figure 1 An example of content manipulation.

A useful manipulation is removing tags. The fact that the content is tagged doesn't mean that tagging has any use, certainly not if it relates to editing specific for some application. Maybe at some point I'll add a re-tagging option but for now we just strip:

```

strip = {
  marked   = true,
  -- group = true,

```

```
-- extgstate = true,
}
```

The other two are sort of special and might be needed too, especially when for instance the states are just there because the producer wasn't clever enough to leave them out when not applicable.

It happens that producers use color while actually gray scales are meant. In that case one can use these:

```
reduce = {
  color = true, -- both rgb and cmyk
  rgb   = true,
  cmyk  = true,
}
```

reduce converts to gray scale all the rgb colors that have the same values for r, g and b and rgb = true or color = true).

The same goes for every cmyk color where c, m, m are the same and when cmyk = true or color = true. In this case the common component component is added to the k component. For example, .2 .2 .2 .5 K becomes .2 + .5 = .7 G, while .5 .5 .5 .7 K becomes 1 G, because the sum is limited to 1.

Using a gray scale is more efficient and in the case of cmyk a sloppy .5 .5 .5 0 K quite likely is meant to be 0 0 0 0.5 K or just .5 G.

Remapping rgb to cmyk (or gray if applicable) is done with:

```
convert = {
  rgb = true,
  -- cmyk = true,
}
```

and of course one can also remap cmyk to rgb.

I want to stress that manipulating the content stream has some limitations. For instance because objects are shared including a page a second time will reuse the already converted page. However, you can try the next trick:

```
\startluacode
  graphics.registerpdfcompactor ( "preset:demo-2", {
    identify = "all",
    merge    = { lmtx = true },
    recolor  = { viagrays = { 0, 1, 0 } },
  } )
  graphics.registerpdfcompactor ( "preset:demo-3", {
    identify = "all",
    merge    = { lmtx = true },
    recolor  = { viagrays = { 0, 0, 1 } },
  } )
\stopluacode
\setupexternalfigures[compact=preset:demo-1]
\startTEXpage
  \startcombination[2*1]
    {\externalfigure
      [test-000.pdf]
      [frame=on,compact=preset:demo-2,width=6cm,object=no,arguments=1]}
```



```

        {demo-2}
    {\externalfigure
        [test-000.pdf]
        [frame=on,compact=preset:demo-3,width=6cm,object=no,arguments=2]}
        {demo-3}
    \stopcombination
\stopTEXpage

```

In figure 2 we see that indeed a different compactor is used. We need to disable sharing by setting `object` to `no`. However, this will still share some but we abuse the `arguments` key to create a different sharing hash (normally that key is used to pass arguments to converters).

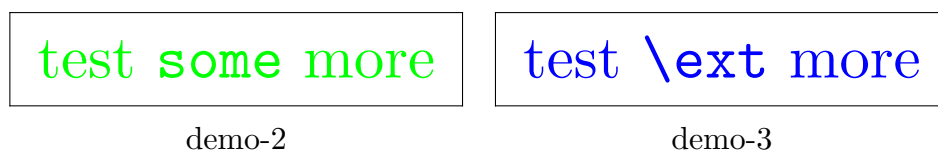


Figure 2 An example of manipulation content twice.

In cases where color conversion is problematic (or critical) you can remap specific colors. Especially `cmymk` is sensitive for conversion because there we have four color components while in `rgb` we have only three. Also watching on a display (`rgb`) is different from looking at a print (`cmymk`) and who knows what transfer function gets applied in the former. Here is how remapping works:

```

local cmykmap = {
    { 100, 100, 55, 0, 57, 0, 22, 40.8 }
}
graphics.registerpdfcompactor ( "preset:demo-5", {
    identify = "all",
    merge    = { lmtx = true },
    convert  = { cmyk = cmykmap },
} )

```

Here the entries in a `cmymk` map are:

```
{ factor, c, m, y, k, r, g, b }
```

In this case values are multiplied by 100 which makes sure that we catch rounding errors in the pdf definitions. Keep in mind that colors in many applications have at most 256 values per component. Also, even quality lcd displays can use less than eight bits per component.

In figure 3 we show an example. The file used looks like:

```

% \nopdfcompression

\starttext
\definecolor[one][c=1.000,m=0.550,y=0.000,k=0.57]
\definecolor[two][r=0.000,g=0.22,b=0.408]          % h=003868
\startTEXpage[offset=1ex]
    \blackrule[color=one,height=2cm,width=2cm,depth=0cm]
    \blackrule[color=two,height=2cm,width=2cm,depth=0cm]
\stopTEXpage
\stoptext

```

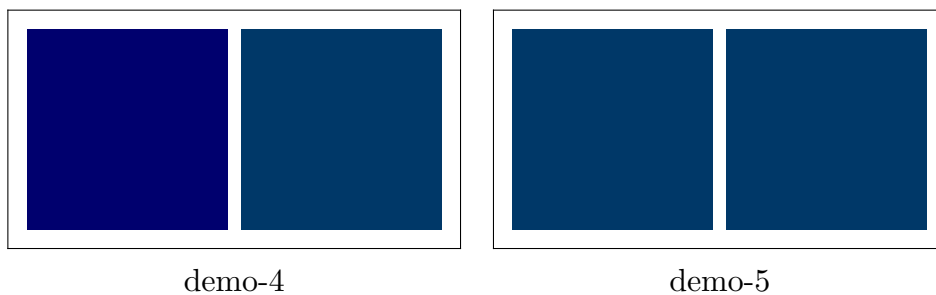


Figure 3 An example of remapping cmyk colors.

In case we wanted to map single rgb values to cmyk, we would define an analogous map:

```
local rgbmap = {
  { 100, 0, 22, 40.8, 100, 55, 0, 57 } -- factor, r, g, b, c, m, y, k
}
graphics.registerpdfcompactor ( "preset:demo-8", {
  identify = "all",
  merge    = { lmtx = true },
  convert  = { rgb = rgbmap },
} )
```

Here the entries in a rgb map are:

```
{ factor, r, g, b, c, m, y, k }
```

In figure 4 we show an example.

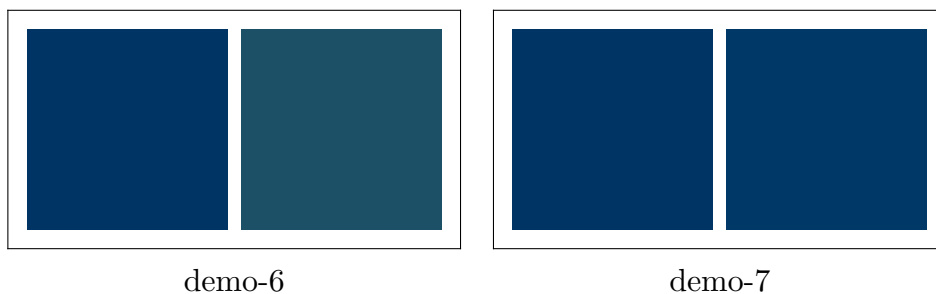


Figure 4 An example of remapping rgb colors.

The fixpdf script

I want to stress that you need to check the outcome. Often a visual check is enough. Extending the compactor beyond what MkIV provided was to a large extent facilitated by a cooperation with Tan, Syabil M. and Ser, Zheng Y. of 'Team Ramkumar' who did extensive testing and gave enjoyable feedback. In the process a test script was made that can help with experiments. We assume that qpdf, mutool and graphicmagick and verapdf are installed. Massimiliano Farinella applied these mechanism to large complex files from InDesign and InkScape that needed fixing and in the process the code got extended and improved.¹

```
mtxrun --script fixpdf --uncompress          foo
mtxrun --script fixpdf --convert      --compactor=preset:test foo
mtxrun --script fixpdf --validate      foo
mtxrun --script fixpdf --check        foo
```

¹ Feel free to send us files that give problems so that we can look into it.

```
mtxrun --script fixpdf --compare --resolution=300 foo
```

Here we produce an uncompressed version (so that we can see what we deal with), convert the original into a new one, validate (and check the outcome) and create a version for visual comparison. It's just an example of usage and here the focus was on fixing existing documents (six digit numbers so the workflow needs to be carefully checked) and not so much on single page inclusion.

However, this script and setup is somewhat complex so we also provide an alternative in the 'extras' namespace:

```
context --extra=fixpdf --compactor=mine:test --extrastyle=foo somefile.pdf
```

Additional options are `--notracing` and `--nocompression`. A compactor can be defined in a file with the name `compactors-mine.lua` that looks like this. Check out `compactors-preset.lua` for examples.

```
local fonts = {
  { source = "arial",           target = "file:arial.ttf" },
  { source = "arial-bold",     target = "file:arialbd.ttf" },
  { source = "arial,bold",     target = "file:arialbd.ttf" },
  { source = "helvetica",      target = "file:arial.ttf" },
  { source = "helvetica-bold", target = "file:arialbd.ttf" },
  { source = "courier",        target = "file:cour.ttf" },
  { source = "wingdings",      target = "wingding" },
  { source = "times-roman",    target = "file:times.ttf" },
  { source = "timesnewromanpsmt", target = "file:times.ttf" },
  { source = "timesnewromanpsitalicmt", target = "file:timesi.ttf" },
  { source = "timesnewromanps-italicmt", target = "file:timesi.ttf" },
  { source = "timesnewroman,italic", target = "file:timesi.ttf" },
  { source = "timesitalic",    target = "file:timesi.ttf" },
  { source = "times-italic",   target = "file:timesi.ttf" },
  { source = "timesnewromanpsboldmt", target = "file:timesbd.ttf" },
  { source = "timesnewromanps-boldmt", target = "file:timesbd.ttf" },
  { source = "timesnewroman,bold", target = "file:timesbd.ttf" },
  { source = "timesbold",      target = "file:timesbd.ttf" },
  { source = "times-bold",     target = "file:timesbd.ttf" },
}

return {
  name = "compactors-preset",
  version = "1.00",
  comment = "Definitions that complement pdf embedding.",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  compactors = {
    ["test"] = {
      fonts = fonts,
      embed = {
        type0 = true,
        truetype = true,
        type1 = true,
      },
      merge = {
        type0 = true,
        truetype = true,
      }
    }
  }
}
```

```
        type1    = true,
        LMTX     = true,
    },
    strip = {
        marked   = true,
    },
    cleanup = {
        pieceinfo = true,
        procset   = true,
        cidset    = true,
    },
    }
},
}
```

A file like this is easier than registering in a Lua snippet. It's also more future proof. The somewhat weird font list is normally build up as we test and is often rather specific for a specific set of files.