# 1 The LUATEX PDF backend

## 1.1 Introduction

The original design of TEX has a clear separation between the front- and backend code. In principle shipping out a page boils down to traversing the to be shipped out box and translate the glyph, rule, glue, kern and list nodes into positioning just glyphs and rules on a canvas. The dvi backend is therefore relatively simple, as the dvi output format delegates the details of font inclusion and such into the final format to other programs; it just describes the pages.

Because we eventually want color and images too there is a mechanism to pass additional information to postprocessing programs. One can insert `\special`s with directives like `insert image with name foo.jpg here`. The frontend as well as the backend are not really interested in what goes into a special; the dvi postprocessor of course is.

The pdf backend is more complex as it immediately produces the final result and as such it offers possibilities to insert verbatim code (`\pdfliteral`), images (`\pdfximage` cum suis), reuse typeset content (`\pdfxform` cum suis), annotations, destinations, threads and all kind of objects so there are all kind of `\pdf...` commands. The way these were implemented prior to 0.82 violates the separation between front- and backend, an inheritance from pdfTEX. Additional features like protrusion and expansion add to that entanglement. However, because pdf is an evolving standard occasionally we need to adapt the related code. A separation of code makes sure that the frontend can become stable (and hopefully frozen) at some point.[1]

In LuaTEX we already had started making the separation again, like a cleaner implementation of font expansion, but all these `\pdf...` commands were still all over the place, leading to fuzzy dependencies, checks for backend modes, etc. so a logical step was to clean this up. That way we can let LuaTEX get a cleaner core constructed from traditional TEX, extended with $\varepsilon$-TEX, Aleph/Omega, and LuaTEX functionality.

[1] In practice nowadays that happens seldom because the pdf that LuaTEX produces is rather simple and there are ways to adapt to the standard.

## 1.2 Extensions

A first step was to turn generic (i.e. independent from a backend) functionality, still sort of bound to Aleph and pdfTEX, into core functionality. A second step was to reorganize the backend specific pdf code, i.e. move it out of the core and into the extension group of commands. This extension group is kind of special and originates in traditional TEX. It is the way to add your own functionality to TEX the program.

As example Don Knuth added 4 extensions: `\openout`, `\closeout`, `\write` and `\special`. The Aleph and pdfTEX engines put some functionality in extensions and some in the core. This has to do with the fact that dealing with variables in the extensions is not really handy as they are then seen as (unexpandable) commands instead of integers, token lists etc. The write related commands being there is just a side effect of demonstrating the mechanism, because everything related to reading from files is in the core. There is one property that sort of forces us to keep the writers there, and that's the `\immediate` prefix.[2]

In the process of separating we reshuffled the code base a bit and the current use of the extensions mechanism still suits as an example but also provides us backward compatibility. However, new backend primitives will not be added there but in specific plugins (if needed at all).

## 1.3 Concepts

The pdf backend has introduced two concepts into the core: (reuseable) images and (reusable) content (wrapped in boxes). In good TEX tradition these were implemented as whatsits (a node type for extensions) but this also introduced an anomality in the handling of such nodes. Say that we have a loop over a node list where we need to check dimensions of nodes. We then get something like:

```
while n do
    if n.id == glyph then
        -- wd ht dp
    elseif n.id == rule then
        -- wd ht dp
    elseif n.id == kern then
```

[2] Unfortunately we're stuck with that one (a `deferred` keyword would have been handier, especially because some backend related commands also can be immediate.

```
        -- wd
    elseif n.id == glue then
        -- size stretch shrink
    elseif n.id == whatsits then
        if n.subtype == pdfxform then
            -- wd ht dp
        elseif n.subtype == pdfximage then
            -- wd ht dp
        end
    end
    n = n.next
end
```

So each time we need to check these two whatsit types. But as these two concepts are rather generic there is no need to implement it this way. Of course the backend has to provide the inclusion and reuse, but the frontend is rather agnostic for this. And at the input end, in specifying these two injects, we only have to make sure we pass the right information (so the scanner might differentiate between backends).

In LuaTEX these two concepts have been promoted to core features:

```
\saveboxresource              \pdfxform
\saveimageresource            \pdfximage
\useboxresource               \pdfrefxform
\useimageresource             \pdfrefximage
\lastsavedboxresourceindex    \pdflastxform
\lastsavedimageresourceindex  \pdflastximage
\lastsavedimageresourcepages  \pdflastximagepages
```

The index is to be considered just some number and is whatever the backend plugin decides to use as identifier. These are no longer whatsits either, but a special type of rule: after all, TEX is only interested in dimensions so the previous code can be simplified to:

```
while n do
    if n.id == glyph then
        -- wd ht dp
    elseif n.id == rule then
        -- wd ht dp
    elseif n.id == kern then
        -- wd
    elseif n.id == glue then
```

```
        -- size stretch shrink
    end
    n = n.next
end
```

A similar upgrade has been done with direction nodes that also were whatsits. These are now normal nodes so instead of consulting whatsit subtypes we can now just check the id of a node. It will be clear that both these changes from whatsits to normal nodes already simplifies the code base.

The only consequence for the already existing rule (which in fact is also something that needs to be dealt with in the backend, depending on the target format) is that it now has subtype 0 while the box resource has subtype 1 and the image subtype 2.

If you still want to use the pdfTEX names you can do this:

```
\let\pdfxform           \saveboxresource
\let\pdflastxform       \lastsavedboxresourceindex
\let\pdfrefxform        \useboxresource

\let\pdfximage          \saveimageresource
\let\pdflastximage      \lastsavedimageresourceindex
\let\pdflastximagepages\lastsavedimageresourcepages
\let\pdfrefximage       \useimageresource
```

There are more commands promoted to core commands. The pdfTEX counterparts are:

```
\let\pdfpagewidth       \pagewidth
\let\pdfpageheight      \pageheight

\let\pdfadjustspacing   \adjustspacing
\let\pdfprotrudechars   \protrudechars
\let\pdfnoligatures     \ignoreligaturesinfont
\let\pdffontexpand      \expandglyphsinfont
\let\pdfcopyfont        \copyfont

\let\pdfnormaldeviate  \normaldeviate
\let\pdfuniformdeviate \uniformdeviate
\let\pdfsetrandomseed  \setrandomseed
\let\pdfrandomseed     \randomseed
```

```
\let\ifpdfabsnum       \ifabsnum
\let\ifpdfabsdim       \ifabsdim
\let\ifpdfprimitive    \ifprimitive

\let\pdfprimitive      \primitive

\let\pdfsavepos        \savepos
\let\pdflastxpos       \lastxpos
\let\pdflastypos       \lastypos

\let\pdftexversion     \luatexversion
\let\pdftexrevision    \luatexrevision
\let\pdftexbanner      \luatexbanner

\let\pdfoutput         \outputmode
\let\pdfdraftmode      \draftmode

\let\pdfpxdimen        \pxdimen

\let\pdfinsertht       \insertht
```

I won't mention the ones that are gone. These were experimental anyway and can easily be provided in TEX (using Lua).

## 1.4 Commands

There are many commands that start with `\pdf` and in the past many have been added, some have been renamed, others removed. Instead of the many we now have just one: `\pdfextension`. This is used as:

```
\pdfextension literal {1 0 0 2 0 0 cm}
\pdfextension obj     {/foo (bar)}
```

So, we pass a keyword that tells what to scan for and what to do with it. A backward compatible interface is easy to write. It delegates a bit more management of these `\pdf` commands to the macro package but the responsibility for dealing with such low level, error-prone calls is there anyway. The full list of extensions is given here. The scanning after the keyword is the same as for pdfTEX.

```
\protected\def\pdfliteral       {\pdfextension literal }
\protected\def\pdfcolorstack    {\pdfextension colorstack }
```

```
\protected\def\pdfsetmatrix     {\pdfextension setmatrix }
\protected\def\pdfsave          {\pdfextension save\relax}
\protected\def\pdfrestore       {\pdfextension restore\relax}
\protected\def\pdfobj           {\pdfextension obj }
\protected\def\pdfrefobj        {\pdfextension refobj }
\protected\def\pdfannot         {\pdfextension annot }
\protected\def\pdfstartlink     {\pdfextension startlink }
\protected\def\pdfendlink       {\pdfextension endlink\relax}
\protected\def\pdfoutline       {\pdfextension outline }
\protected\def\pdfdest          {\pdfextension dest }
\protected\def\pdfthread        {\pdfextension thread }
\protected\def\pdfstartthread   {\pdfextension startthread }
\protected\def\pdfendthread     {\pdfextension endthread\relax}
\protected\def\pdfinfo          {\pdfextension info }
\protected\def\pdfcatalog       {\pdfextension catalog }
\protected\def\pdfnames         {\pdfextension names }
\protected\def\pdfincludechars  {\pdfextension includechars }
\protected\def\pdffontattr      {\pdfextension fontattr }
\protected\def\pdfmapfile       {\pdfextension mapfile }
\protected\def\pdfmapline       {\pdfextension mapline }
\protected\def\pdftrailer       {\pdfextension trailer }
\protected\def\pdfglyphtounicode{\pdfextension glyphtounicode }
```

## 1.5 Variables

There are also lots of variables that influence the pdf backend. The most important one is of course the one that sets the output mode. Well, that one is gone and replaced by `\outputmode`. A value of 1 means that we produce pdf.

One complication of variables is that (if we want to be compatible) we need to have them as real TEX registers. However, most of them are optional so an easy way out is to simply not define them in the engine. In order to be able to deal with them as registers (backward compatible), we define them as follows:

```
\edef\pdfcompresslevel      {\pdfvariable compresslevel}
\edef\pdfobjcompresslevel   {\pdfvariable objcompresslevel}
\edef\pdfdecimaldigits      {\pdfvariable decimaldigits}
\edef\pdfgamma              {\pdfvariable gamma}
\edef\pdfimageresolution    {\pdfvariable imageresolution}
```

```
\edef\pdfimageapplygamma    {\pdfvariable imageapplygamma}
\edef\pdfimagegamma         {\pdfvariable imagegamma}
\edef\pdfimagehicolor       {\pdfvariable imagehicolor}
\edef\pdfimageaddfilename   {\pdfvariable imageaddfilename}
\edef\pdfpkresolution       {\pdfvariable pkresolution}
\edef\pdfinclusioncopyfonts {\pdfvariable inclusioncopyfonts}
\edef\pdfinclusionerrorlevel{\pdfvariable inclusionerrorlevel}
\edef\pdfreplacefont        {\pdfvariable replacefont}
\edef\pdfgentounicode       {\pdfvariable gentounicode}
\edef\pdfpagebox            {\pdfvariable pagebox}
\edef\pdfminorversion       {\pdfvariable minorversion}
\edef\pdfuniqueresname      {\pdfvariable uniqueresname}

\edef\pdfhorigin            {\pdfvariable horigin}
\edef\pdfvorigin            {\pdfvariable vorigin}
\edef\pdflinkmargin         {\pdfvariable linkmargin}
\edef\pdfdestmargin         {\pdfvariable destmargin}
\edef\pdfthreadmargin       {\pdfvariable threadmargin}

\edef\pdfpagesattr          {\pdfvariable pagesattr}
\edef\pdfpageattr           {\pdfvariable pageattr}
\edef\pdfpageresources      {\pdfvariable pageresources}
\edef\pdfxformattr          {\pdfvariable xformattr}
\edef\pdfxformresources     {\pdfvariable xformresources}
\edef\pdfpkmode             {\pdfvariable pkmode}
```

You can now initialize them (although there is no real need as the values shown are the initial values anyway):

```
\pdfcompresslevel         9
\pdfobjcompresslevel      1
\pdfdecimaldigits         3
\pdfgamma              1000
\pdfimageresolution      71
\pdfimageapplygamma       0
\pdfimagegamma         2200
\pdfimagehicolor          1
\pdfimageaddfilename      1
\pdfpkresolution         72
\pdfinclusioncopyfonts    0
\pdfinclusionerrorlevel   0
\pdfreplacefont           0
```

```
\pdfgentounicode          0
\pdfpagebox               0
\pdfminorversion          4
\pdfuniqueresname         0

\pdfhorigin              1in
\pdfvorigin              1in
\pdflinkmargin           0pt
\pdfdestmargin           0pt
\pdfthreadmargin         0pt
```

Their removal from the frontend cleaned up the code and by making them registers they are still compatible. A call to `\pdfvariable` will define an internal register that keeps the value (of course this value can also be influenced by the backend itself). Although they are real registers they live in a protected namespace:

```
\meaning\pdfcompresslevel
```

this gives:

```
macro:->[internal backend integer]
```

It's really unfortunate that we have to be compatible because a setter and getter would be nicer. I still consider writing the extension primitive in Lua using the token scanner but it might not be possible to stay compatible then. This is not so much an issue for ConTEXt which always had backend drivers but other macro packages have users that expect the primitives (or counterparts) to be available.

The backend can report back some properties and these were also accessible via `\pdf...` primitives. Because these are read-only variables another primitive deals with them: `\pdffeedback`. That one can be used to define compatible alternatives:

```
\def\pdflastlink       {\numexpr\pdffeedback lastlink\relax}
\def\pdfretval         {\numexpr\pdffeedback retval\relax}
\def\pdflastobj        {\numexpr\pdffeedback lastobj\relax}
\def\pdflastannot      {\numexpr\pdffeedback lastannot\relax}
\def\pdfxformname      {\numexpr\pdffeedback xformname}
\def\pdfcreationdate           {\pdffeedback creationdate}
\def\pdffontname       {\numexpr\pdffeedback fontname\relax}
\def\pdffontobjnum     {\numexpr\pdffeedback fontobjnum\relax}
```

```
\def\pdffontsize       {\dimexpr\pdffeedback fontsize\relax}
\def\pdfpageref        {\numexpr\pdffeedback pageref\relax}
\def\pdfcolorstackinit         {\pdffeedback colorstackinit}
```

The variables are internal ones, so they are anonymous. When you ask for the meaning of a few previously defined ones:

```
\meaning\pdfhorigin
\meaning\pdfcompresslevel
\meaning\pdfpageattr
```

you will get:

```
macro:->[internal backend dimension]
macro:->[internal backend integer]
macro:->[internal backend tokenlist]
```