Technology Preview - PDF Interpreter

For the past few years the Ghostscript development team has, amongst many other things, been working on a ground-up rewrite of the PDF interpreter. While the work is not yet completed we want to make an early announcement so that our customers and free users can be ready for the switch to the new interpreter when it comes.

This document is intended to try and answer what we expect to be the most frequently asked questions, and highlight the changes.

Why the change?

The original PDF interpreter, as currently supplied with Ghostscript, is written in PostScript. When the original implementation was done this made good sense; the graphics model of PostScript and PDF was compatible and the PDF syntax is (or at least was) broadly similar to PostScript. Indeed that original PDF interpreter has served us well for decades.

However there are problems, mainly invisible to our users but nevertheless still present. PostScript has been described, with some justification, as a ‘write-only’ language and, being now an elderly language, is a rare skill for developers making it quite hard to recruit new engineers with PostScript programming skills. Not all of the Artifex development team are experienced PostScript programmers and even for those of us skilled in the language, the PDF interpreter code is now so large and arcane that it is difficult to fully understand some aspects of the PostScript program which performs the PDF interpretation.

In addition the PDF specification has continued to evolve, whereas the PostScript language has remained static. PDF has added features like transparency, which have no equivalent in PostScript, and the only way for us to support these has been to add special, often undocumented, PostScript extensions. These extensions have proven to be a security problem in the past and we would like to remove our PDF interpreter’s dependence on them.

It has also become increasingly evident that many PDF producers do not create PDF files that conform to the specification. Since there is no means to ‘verify’ that a PDF file conforms, creators fall back on using Adobe Acrobat, the de facto standard. If Acrobat will open the file then it must be OK! Sadly it turns out that Acrobat is really very tolerant of badly formed PDF files and will try to open them. Often it silently repairs the file in the background; the first time an alert user would be aware of this is when Acrobat offers to ‘save changes’ to a file the user has not modified, frequently Acrobat doesn’t even do that.

Because Acrobat will open these files, there is considerable pressure for Ghostscript to do so as well, though we do try to at least flag warnings to the user when something is found to be incorrect, giving the user a chance to intervene.

But Ghostscript's PDF interpreter is, as noted, written in PostScript, and PostScript is not a great language for handling error conditions and recovering. In general when something goes wrong in a PostScript program the expectation is that the PostScript interpreter will generate an error message and stop. It is possible to do better, but it is not trivial. As time has gone on, and we have encountered more and more PDF files with ever more unexpected deviations from the specification, it has become harder and harder to come up with new strategies to work around these faults without re-introducing previously fixed problems or failing to process compliant files. It is also true that many of these workarounds have led to decreased performance when processing all PDF files, not just the malformed ones.

Finally, because the PDF interpreter is written in PostScript, there is no way to divorce it from Ghostscript and its PostScript interpreter. This has performance implications (starting up a PostScript interpreter is quite a complex process), imposes a resource overhead in that we need both the PostScript interpreter and a complex PostScript program before we even start to interpret the PDF file, and exposes us to potential security issues due to the use of non-standard PostScript extensions, and the possibility of being forced to run PostScript XObjects (long since deprecated) in a PDF file, which again permits some possible security faults.

What's new?

The new PDF interpreter is entirely written in C, but interfaces to the same underlying graphics library as the existing PostScript interpreter. So operations in PDF should render exactly the same as they always have (this is affected slightly by differing numerical accuracy), all the same devices that are currently supported by the Ghostscript family, and any new ones in the future, should work seamlessly.

Because the interpreter no longer relies on PostScript, however, it can be divorced from it. It is now possible to create a stand-alone PDF interpreter, GhostPDF, and it is integrated as a separate module in the language-switching product GhostPDL.

This offers us some advantages in that the memory footprint is smaller, and the startup time of the stand-alone PDF interpreter is less than starting up the PostScript interpreter.

That said, we do recognise that people are used to being able to process PDF files through Ghostscript, and indeed over the years we have offered customers and free users a wide range of solutions which are based on the fact that the PDF interpreter is currently written in PostScript, and its behaviour can be controlled or influenced from the PostScript environment.

So one of the goals of this project was to enable the C PDF interpreter to be integrated into the PostScript environment in such a way that PostScript can be used to influence the graphics state of the PDF interpreter, and PostScript functionality like BeginPage and EndPage continue to function with it. And of course not forgetting that initial point, Ghostscript today can process PDF files and our users will expect that ability to continue. I'll set out some of the means for that below.

When will the changes occur?

We are not yet entirely certain. We had hoped to have this alpha version of the code in the 9.54.0 release, but simply ran out of time to finish up some basic functionality. We currently intend for a beta in the next release (September 2021), then a release where the new PDF interpreter is complete

and shipped as part of Ghostscript, but the old interpreter will continue to be present, and will be used in preference to the new one. For the release after that we will swap the dominance, the new interpreter will be used for PDF files but it will be possible to switch back to the old one if there are problems. Currently that is tentatively scheduled for September 2022 though we would like to bring that forward if we can. After that we intend to remove the old PDF interpreter completely.

That’s a long time, why should I care?

Well, the earlier you experiment with the code, the earlier you can tell if the changes are going to cause you a problem, whether that be something relatively simple that you can alter your workflow to accommodate or more seriously if you need to take it up with us and have us alter some aspect of our plans.

We do intend to replace the old PostScript-based PDF interpreter but we want it to be as easy as possible for our customers to change over. The earlier you can flag up problems the better. Do please try it out and let us know.

What currently doesn’t work?

Right now the biggest hole is support for certain kinds of font, we are confident we can add that it is just taking longer than we hoped. So some files will render with a fallback font (or possibly missing text, or incorrect text).

We do anticipate problems with the pdfwrite family of devices, again mostly concerned with fonts and text. The pdfwrite device has evolved over the years and is only rather loosely coupled to the graphics library API with numerous exceptions, back doors and hidden assumptions. We are working through those but this is the area we expect the majority of problems to arise.

Other than that we think pretty much everything should work, and a few things work in the new interpreter that don’t in the old one; some annotation types are preserved that were not before, some files will work that previously stopped with errors, and some files with problems will render more content than they previously did. If you find any switch that doesn’t work, or doesn’t do what it used to do, we’d definitely like to hear about it. At this point it probably isn’t worth raising issues about text, especially if it is with the pdfwrite family.

Getting the new code

We are not (yet) putting the new code in the regular release, instead we are making a new source archive available, which includes the source for the new PDF interpreter and makefiles to build it.

From https://github.com/ArtifexSoftware/ghostpdl-downloads/releases/tag/gpdf_alpha1 download ghostpdl-9.55.0-gpdf_alpha1.tar.gz, extract the source files and then use the usual ./autogen.sh and make invocation (on Linux) or on Windows build ghostpdl and ghostscript from the solution. This should generate a Ghostscript executable which uses the new interpreter, and a new executable 'gpdf' or 'gpdfwin64/gpdfwin32' on Windows.

There are also 64 and 32-bit Windows installers; gs9550w64-gpdf_alpha1.exe and gs9550w32-gpdf_alpha1.exe in the same location.

Alternatively, if you are using Git, you can checkout the 'pdfi' branch from our Git repository, but be aware that this is a working branch, it should always build but it might sometimes go backwards in functionality as we work on it.

Using the new code

If you are using Ghostscript then just use gs or gswin32/gswin64(c) as you would normally. If you want to experiment with the interface between PostScript and PDF that is documented later.

If you just want to try out the new PDF interpreter as a stand-alone application then use gpdf (Linux) or gpdfwin32/gpdfwin64 (Windows). The Windows gpdfwinxx works the same as the Ghostscript command-line executable for Windows, there is currently no equivalent of the windowed executable.

Command line switches should work in both cases the same as they do in Ghostscript right now. Please note that the gpdf executable does **not** permit you to use the pdfmark operator (or otherwise send arbitrary PostScript to the interpreter using the -c switch). The pdfmark operator is a PostScript operator and therefore requires you to use the PostScript interpreter.

Obviously the gpdf interpreter will not execute PostScript XObjects embedded in PDF files, for the same reason.

Using the PDF interpreter from PostScript

The new code has been integrated following the old PDF interpreter; if all you want to do is process a PDF file then simply putting the file on the Ghostscript command line is sufficient. Also the definition of the PostScript 'run' operator works with the new PDF interpreter, so you can still use code such as '(/home/myfile.pdf) run'.

The sections below cover those cases where more control is needed.

Existing PostScript-based PDF interpreter

We'll start by looking at the existing implementation, and describe those parts of it which appear to be intended as 'public' (or at least, what we think were intended as public). These functions still exist in the new implementation, but they are implemented in terms of the new PostScript operators described later, which use the new PDF interpreter, rather than by calling the existing PostScript-based interpreter.

Currently pdf_main.ps redefines the PostScript run operator to examine the input and try to determine if it's a PDF file. If it is, and the input is from stdin, we send the whole input to a temporary file on disk then continue by calling the PostScript 'run' operator, which can differentiate between a PostScript and PDF file. For PDF input when the file is complete, or if the input was already from a file, we call the function runpdf.
That function starts by calling process_trailer_attrs (which appears to deal with Outlines etc see .**PDFMetadata** below) then runpdfpagerange which turns off the built-in FirstPage/LastPage device and generates a list of page numbers to process.
We then call dopdfpages which executes pdfshowpage for each page in the list which (finally!) uses pdfshowpage_init, pdfshowpage_setpage and pdfshowpage_finish to actually draw the page.

According to comments in the PostScript code, the following routines are used by customers:

```
pdfshowpage_init     % <pagedict>
pdfshowpage_setpage  % <pagedict>
pdfshowpage_finish   % -
```

The 'pagedict' argument for these procedures is gathered by calling pdfgetpage, which returns a PostScript representation of the PDF page dictionary for a given page number.

dict **pdfshowpage_init** *dict*
Looks for the existence of DSCPageCount in a dictionary argument and increments it by 1 then 'store's it. Does not use the supplied dict argument.

dict **pdfshowpage_setpage** *dict*
Does some weirdness with Orientation for pdfmarks. Gets a Box from the (supplied) page dictionary and either sets the PageSize to that, or, if **PDFFitPage** is true, preserves the existing PageSize but scales the content to fit the existing page. If the user has set **UseOutputIntent** then check the page to see if it has such an intent and use it if it does.

dict **pdfshowpage_finish** *dict*
Badly misnamed really... If .**writepdfmarks** is true then we are going to pdfwrite and we do the following: copy PageLabels if there are any, copy all the *Box entries. Then execute the page content stream(s), execute the annotations (honouring ShowAnnots, PreserveAnnots et al), if there is an AcroForm on the page, and **ShowAcroForm** is true, draw the form, call endpage,

Each of these uses 'pagedict' and I propose to replace the existing code in 'dopdfpages' with a loop which calls PDFPageInfo to retrieve a page dictionary, then uses the routines defined above to render the pages, mimicking the existing behaviour.

New Interpreter

Proposed new PostScript operators for the PDF interpreter

dict **.PDFInit** *PDFcontext*
dict is an optional dictionary that contains any interpreter-level switches, such as PDFDEBUG, this is used to set the initial state of the PDF interpreter. The return value is a PDFcontext object which is an opaque object to be used with the other PDF operators.

filename PDFcontext .**PDFFile** -
filename is a string containing a fully qualified path to the PDF file to open (must be accessible!)

stream PDFcontext .**PDFStream** -
stream must be an already open **disk**-based file that the PDF interpreter should use as its input.

PDFcontext .**PDFClose** -
If the context contains an open PDF file which was opened via the .PDFfile operator, this closes the file. Regardless of the source it then shuts down the PDF interpreter and frees the associated memory.

PDFcontext .**PDFInfo** *dict*
PDFcontext is a PDFcontext object returned from a previous call to .PDFInit.
Returned dictionary contains various key/value pairs with useful file level information

```
/NumPages int
/Creator string
/Producer string
/IsEncrypted boolean
```

PDFcontext .**PDFMetadata -**
PDFcontext is a PDFcontext object returned from a previous call to .PDFInit.
For the benefit of high level devices, this is a replacement for 'process_trailer_attrs' which is a seriously misnamed function now. This function needs to write any required output intents, load and send Outlines to the device, copy the Author, Creator, Title, Subject and Keywords from the Info dict to the output device, copy Optional Content Properties (OCProperties) to the output device, if an AcroForm is present send all its fields and link widget annotations to fields, and finally copy the PageLabels. If we add support for anything else, it will be here too.

PDFcontext int .**PDFPageInfo** *dict*
PDFcontext is a PDFcontext object returned from a previous call to ,PDFInit.
The integer argument is the page number to retrieve information for.
Returns a dictionary with the following key/value pairs:

```
/UsesTransparency true|false
/SpotColours array of names, may be empty
/MediaBox [llx lly urx ury]
/HasAnnots true|false
```

/FontsUsed array of names, may be empty.

May also contain (if they are present in the Page dictionary)
/ArtBox [llx lly urx ury]
/CropBox [llx lly urx ury]
/BleedBox [llx lly urx ury]
/TrimBox [llx lly urx ury]
/UserUnit *int*

PDFcontext int .**PDFDrawPage** -
PDFcontext is a PDFcontext object returned from a previous call to .PDFInit.
The integer argument is the page number to be processed.
Interprets the page content stream(s) of the specified page using the current graphics state

PDFcontext int .**PDFDrawAnnots** -
PDFcontext is a PDFcontext object returned from a previous call to .PDFInit.
The integer argument is the page number to be rendered.
Renders the Annotations (if any) of the specified page using the current graphics state
For correct results, the graphics state when this operator is run should be the same as when **PDFDrawPage** is executed.

Finally, here is a simple example program to make use of the new PDF interpreter. This PostScript program opens two PDF files simultaneously, then reads pages from each in turn. Used with this command line:

gs -sDEVICE=pdfwrite -o new.pdf –permit-file-read=/temp/ test.ps

it will create a new PDF file containing all the pages from the input PDF files, interleaved. Notice the use of –permit-file-read to allow the PostScript program to open the input PDF files.

```
%!
% simple program to read two PDF files simultaneously
% and interleave them on output.
%

userdict begin
/MyDict 20 dict def
MyDict begin

% First things first, create two PDF contexts, we
% need one for each input PDF file
%
/File1_Context .PDFInit def
/File2_Context .PDFInit def

% Now open the files on disk, using each context
%
(/temp/test1.pdf) File1_Context .PDFFile
(/temp/test2.pdf) File2_Context .PDFFile

% Now loop around showing each page of the input from
% file 1, followed by the same page from file 2.
%
0 1 3 {
  dup                 % copy the loop counter
  File1_Context exch  % stack: counter context counter
  .PDFDrawPage        % draw page 'n'. Stack: counter
  showpage            % finish the page
  File2_Context exch  % stack: context counter
  .PDFDrawPage
  showpage            % finish the page
} for

% Close each of the PDF contexts, which will
% also close the PDF files
%
File1_Context .PDFClose
File2_Context .PDFClose

% and finally tidy up our dictionary
end                     % MyDict
```

The two input PDF files were created using Ghostscript as well, from two simple PostScript programs. The command lines to create the example input PDF files are:

gs -sDEVICE=pdfwrite -o test1.pdf test1.ps
gs -sDEVICE=pdfwrite -o test2.pdf test2.ps

And the PostScript programs:

```
%!
% Test1.ps draw some simple content
%
1 0 0 setrgbcolor
180 500 moveto
/Helvetica findfont 100 scalefont setfont
(File 1) show
150 250 moveto
(Page 1) show
showpage
1 0 0 setrgbcolor
180 500 moveto
/Helvetica findfont 100 scalefont setfont
(File 1) show
0 1 0 setrgbcolor
150 250 moveto
(Page 2) show
showpage
1 0 0 setrgbcolor
180 500 moveto
/Helvetica findfont 100 scalefont setfont
(File 1) show
0 0 1 setrgbcolor
150 250 moveto
(Page 3) show
showpage
1 0 0 setrgbcolor
180 500 moveto
/Helvetica findfont 100 scalefont setfont
(File 1) show
0 0 0 setrgbcolor
150 250 moveto
(Page 4) show
showpage
```

```
%!
% Test2.ps draw some simple content
%
0 1 0 setrgbcolor
180 500 moveto
/Helvetica findfont 100 scalefont setfont
(File 2) show
1 0 0 setrgbcolor
150 250 moveto
(Page 1) show
showpage
0 1 0 setrgbcolor
180 500 moveto
/Helvetica findfont 100 scalefont setfont
(File 2) show
0 1 0 setrgbcolor
150 250 moveto
(Page 2) show
showpage
0 1 0 setrgbcolor
180 500 moveto
/Helvetica findfont 100 scalefont setfont
(File 2) show
0 0 1 setrgbcolor
150 250 moveto
(Page 3) show
showpage
0 1 0 setrgbcolor
180 500 moveto
/Helvetica findfont 100 scalefont setfont
(File 2) show
0 0 0 setrgbcolor
150 250 moveto
(Page 4) show
showpage
```