

An Introduction to PDF with CamIPDF

John Whittington
August 12, 2013



Coherent Graphics Ltd

For bug reports, feature requests and comments, email
contact@coherentgraphics.co.uk

©2010-2013 Coherent Graphics Limited. All rights reserved.

Adobe, Acrobat, Adobe PDF, Adobe Reader and PostScript are registered trademarks of
Adobe Systems Incorporated.

A First Program

You will require `ocamlfind` to be installed on your system. It comes with any modern installation of OCaml.

To build CamlPDF, navigate to the source directory and type

```
make
```

You can then install CamlPDF:

```
make install
```

As an alternative, you may use the OPAM package manager, if you have it installed:

```
opam install camlpdf
```

To build the examples, navigate to the examples folder and type

```
make
```

Now run a simple example to build the file `hello.pdf`

```
./pdfhello
```

Now we can enter the top level:

```
ocaml
Objective Caml
# #use "topfind";;
- : unit = ()
Findlib has been successfully loaded. Additional directives:
#require "package";;      to load a package
#list;;                  to list the available packages
#camlp4o;;               to load camlp4 (standard syntax)
#camlp4r;;               to load camlp4 (revised syntax)
#predicates "p,q,...";;  to set these predicates
Topfind.reset();;        to force that packages will be
reloaded
#thread;;                 to enable threads

- : unit = ()
# #require "camlpdf";;
/Users/john/.opam/4.00.1/lib/ocaml/unix.cma: loaded
```

```
/Users/john/.opam/4.00.1/lib/ocaml/bigarray.cma: loaded
/Users/john/.opam/4.00.1/lib/camlpdf: added to search path
/Users/john/.opam/4.00.1/lib/camlpdf/camlpdf.cma: loaded

#
```

The `Pdfread` module allows us to load PDF files into memory. The raw PDF data is parsed into a structured OCaml value of type `Pdf.pdfdoc`:

```
# let pdf = Pdfread.pdf_of_file None None "hello.pdf";;
val pdf : Pdf.t =
  {Pdf.major = 1; Pdf.minor = 1; Pdf.root = 2;
   Pdf.objects =
    {Pdf.maxobjnum = 4; Pdf.parse = Some <fun>;
     Pdf.pdfobjects = <abstr>;
     Pdf.object_stream_ids = <abstr>};
   Pdf.trailerdict =
    Pdf.Dictionary
    [("/Root", Pdf.Indirect 2);
     ("/ID",
      Pdf.Array
      [Pdf.String "?W???`??w\r?\001??";
       Pdf.String "?W???`??w\r?\001??"]);
     ("/Size", Pdf.Integer 4)]}
```

Looking at the parts of the `Pdf.t` record type:

- `Pdf.major` and `Pdf.minor` - the parts of the PDF version number. Here, PDF Version 1.1
- `Pdf.root` - the object number of the 'root object' of the PDF (A PDF is a directed graph of objects, indexed by number)
- `Pdf.objects` - the PDF objects
- `Pdf.trailerdict` - the trailer dictionary. This is a distinguished PDF object containing a number of commonly used per-file items. `Pdf.trailerdict` has type `Pdf.pdfobject`, which represents all possible PDF data.

Diversion: A Look at `hello.pdf`

Here is the contents of the file `hello.pdf`, as you might see it in a text editor, annotated with some explanatory comments.

```

%PDF-1.1 Header
%%$^@
1 0 obj Object 1...
<< /Type /Pages /Kids [ 3 0 R ] /Count 1 >> ...which is the catalogue of pages
endobj
2 0 obj This object is a stream, which is a dictionary plus some binary data
<< /Length 102 >>
stream Usually compressed, but plain here for ease of reading
1.000000 0.000000 0.000000 1.000000 50.000000 770.000000 cm
BT /F0 36.000000 Tf (Hello, World!) Tj ET The page content, a bit like PostScript
endstream
endobj
3 0 obj The page object,
<< /Type /Page
  /Parent 1 0 R The syntax "1 0 R" means a reference to Object 1
  /Resources
    << /Font The font dictionary
      << /F0
        << /Type /Font /Subtype /Type1 /BaseFont /Times-Italic >>
      >> >>
    /MediaBox [ 0.000000 0.000000 595.275591 841.889764 ] The page dimensions
    /Rotate 0
  /Contents [ 2 0 R ] >> Reference to contents in object 2
endobj
4 0 obj
<< /Type /Catalog /Pages 1 0 R >> The root object
endobj
xref The cross-reference table, listing the byte offsets of each object for random access.
0 5
0000000000 65535 f
0000000015 00000 n
0000000074 00000 n
0000000227 00000 n
0000000449 00000 n
trailer The trailer dictionary
<< /Size 5 /Root 4 0 R /ID [ (..)ZŁzE) (..?ZŁzE) ] >>
startxref
498 The trailer
%%EOF

```

Saving the Document

The Pdf.t data type is a record of mutable values. Let's change the PDF Version number and write the file.

```

# pdf.Pdf.minor <- 2;;
- : unit = ()
# Pdfwrite.pdf_to_file pdf "hello2.pdf";;

```

```
- : unit = ()
```

Next Steps

The objects in a PDF document are of type Pdf.pdfobject:

```
type stream = Stream data. Either in memory or still in the file
  | Got of Utility.bytestream
  | ToGet of Pdfio.input * int * int input, offset, length

type pdfobject =
  | Null
  | Boolean of bool
  | Integer of int
  | Real of float
  | String of string
  | Name of string
  | Array of pdfobject list
  | Dictionary of (string * pdfobject) list
  | Stream of (pdfobject * stream) ref Stream data (see above)
  | Indirect of int A reference to another object
```

For instance the PDF object in the file:

```
3 0 obj
<< /Type /Page
  /Parent 1 0 R
  /MediaBox [ 0.000000 0.000000 595.275591 841.889764 ]
  /Rotate 0
  /Contents [ 2 0 R ]
>>
end
```

is represented as object number 3 with the Pdf.t instance:

```
Dictionary
["/Type", Name "/Page";
"/Parent", Indirect 1;
"/MediaBox",
  Array [Real 0.; Real 0.; Real 595.275591; Real 841.889764];
"/Rotate", Integer 0;
```

```
"/Contents", Array [Indirect 2]]
```

Working with Pages

Introduce a command to show the current document, using whatever command opens (or updates) a PDF view on your system:

```
let show pdf =  
  Pdfwrite.pdf_to_file pdf "temp.pdf";  
  ignore (Sys.command "open temp.pdf");; Customize here
```

The Pdfpage module deals with PDF pages. We can get the list of pages from a document:

```
# let pages = Pdfpage.pages_of_pagetree pdf;;  
val pages : Pdfpage.t list =  
  [{Pdfpage.content = [Pdf.Indirect 4];  
   Pdfpage.mediabox =  
     Pdf.Array  
       [Pdf.Integer 0; Pdf.Integer 0; Pdf.Real 595.275590551;  
        Pdf.Real 841.88976378];  
   Pdfpage.resources =  
     Pdf.Dictionary  
       [("/Font",  
        Pdf.Dictionary  
          [("/F0",  
           Pdf.Dictionary  
             [("/Type", Pdf.Name "/Font");  
              ("/Subtype", Pdf.Name "/Type1");  
              ("/BaseFont", Pdf.Name "/Times-Italic")])])]);  
   Pdfpage.rotate =  
     Pdfpage.Rotate0; Pdfpage.rest = Pdf.Dictionary []}]
```

Each page is a record containing five things:

- Pdfpage.content An ordered list of pdf objects representing the one or more streams containing the graphical data for the page.
- Pdfpage.mediabox The page dimensions
- Pdfpage.resources The resources dictionary for a page, which contains the fonts, colour spaces and so on for the page.
- Pdfpage.rotate The viewing rotation for the page.
- Pdfpage.rest The rest of the page dictionary (i.e that which has not been separated into the items above).

Let's change the viewing rotation to 90 degrees:

```
# let page = {(List.hd pages) with Pdfpage.rotate = Pdfpage.Rotate90};;
val page : Pdfpage.t = ...
# let pdf = Pdfpage.change_pages false pdf [page];;
val pdf : Pdf.t = ...
# show pdf;;
- : unit
```

Now change the rotation back: we're going to work with graphics next, and the viewing rotation would confuse:

```
# let page = List.hd pages;;
val page : Pdfpage.t = ...
# let pdf = Pdfpage.change_pages false pdf [page];;
val pdf : Pdf.t = ...
# show pdf;;
- : unit
```

Graphics and Text

The Pdfops module represents the graphical content of each page, which is formed of PostScript-like operators which draw the page. Let's get the operator list from the page:

```
# let ops =
  Pdfops.parse_operators
    pdf page.Pdfpage.resources page.Pdfpage.content;;
val ops : Pdfops.t list =
[Pdfops.Op_cm
 {Transform.a = 1.; Transform.b = 0.;
  Transform.c = 0.; Transform.d = 1.;
  Transform.e = 50.; Transform.f = 770.};
 Pdfops.Op_BT;
 Pdfops.Op_Tf ("/F0", 36.);
 Pdfops.Op_Tj "Hello, World!";
 Pdfops.Op_ET]
```

The Op_cm operator alters the graphics matrix to position the text. Op_BT and Op_ET mark the beginning and end of a text section. Op_Tf chooses 36pt Times Italic (which is font F0 in the page's font dictionary in its resources) and Op_Tj paints the text.

Let's add operators to underline the text – Op_m to move, Op_l to draw a line and Op_S to stroke the path. We calculate the width of the underline using the Pdfstandard14 and

`Pdftext` modules to get the raw width of the string in millipoints, adjusting for font size and converting to points.

```
# let width =
  Pdfstandard14.textwidth false Pdftext.TimesItalic "Hello, World!";;
val width : int = 5555

# let actual_width = float width *. 36. /. 1000.;;
val actual_width : float = 199.98

# let ops' =
  ops @
  [Pdfops.Op_m (0., 0.);
   Pdfops.Op_l (actual_width, 0.);
   Pdfops.Op_S];;
val ops' : Pdfops.t list = ...
```

and make the new content stream:

```
# let stream = Pdfops.stream_of_ops ops';;
val stream : Pdf.pdfobject =
  Pdf.Stream
  {contents =
    (Pdf.Dictionary [("/Length", Pdf.Integer 72)], Pdf.Got <abstr>)}
```

and add it to the page, and replace the page in the PDF.

```
# let page' = {page with Pdfpage.content = [stream]};;
val page' : Pdfpage.t = ...

# let pdf = Pdfpage.change_pages false pdf [page'];;
val pdf : Pdf.t = ...
```

and show it:

```
# show pdf;;
- : unit ()
```

Next Steps

CamlPDF is a large piece of software. A good way to get to know it is to study the examples shipped with CamlPDF:

<code>pdfhello.ml</code>	Build a "Hello, World!" PDF from scratch
<code>pdfdecomp.ml</code>	Command line utility to decompress a PDF
<code>pdfmerge.ml</code>	Command line utility to merge PDF files
<code>pdfdraft.ml</code>	Command line utility to make draft documents
<code>pdfctest.ml</code>	Reads and interprets a file to test CamlPDF's major functionality
<code>pdfencrypt.ml</code>	Command line utility to encrypt a PDF file

Summary of CamlPDF modules:

Module	Description
<code>Pdfutil</code>	General Functions.
<code>Pdfio</code>	Generic Input/Output from/to channels, strings, files etc.
<code>Pdftransform</code>	Affine Transformations in Two Dimensions
<code>Pdfunits</code>	Units and Unit Conversion
<code>Pdfpaper</code>	Media Sizes
<code>Pdf</code>	Representing PDF Files in Memory
<code>Pdfcrypt</code>	Decrypting PDF files
<code>Pdfflate</code>	Interface to miniz.c via Zlib-like functions.
<code>Pdfcodec</code>	Encoding and Decoding PDF Streams
<code>Pdfwrite</code>	Writing PDF Files
<code>Pdfgenlex</code>	A very fast lexer for very basic tokens.
<code>Pdfread</code>	Reading PDF Files
<code>Pdfjpeg</code>	PDF Jpeg Support
<code>Pdfops</code>	Parsing PDF Graphics Streams
<code>Pdfdest</code>	Destinations
<code>Pdfmarks</code>	Bookmarks
<code>Pdfpagelabels</code>	Page Labels
<code>Pdfpage</code>	Page-level functionality
<code>Pdfannot</code>	Annotations
<code>Pdffun</code>	Parsing and Evaluating PDF Functions.
<code>Pdfspace</code>	Colour Spaces
<code>Pdfimage</code>	Extract Images.
<code>Pdfafm</code>	Parse Adobe Font Metrics files
<code>Pdfafmdata</code>	AFM Data for the standard 14 fonts
<code>Pdfglyphlist</code>	Glyph Lists
<code>Pdfctext</code>	Parsing fonts and extracting text from content streams and PDF strings
<code>Pdfstandard14</code>	Standard PDF Fonts
<code>Pdfgraphics</code>	Structured Graphics.
<code>Pdfshapes</code>	Basic Shapes
<code>Pdfdate</code>	Representing and Parsing PDF Dates
<code>Pdfocg</code>	Optional Content Groups.
<code>Pdfcff</code>	Convert a CFF Type 1 Font to a Type 3 Font.
<code>Pdftype1</code>	Convert an PostScript Type 1 Font to a Type 3 Font.
<code>Pdftruetype</code>	Convert a TrueType font to a Type 3 Font.
<code>Pdftype0</code>	Type 0 font support
<code>Pdfmerge</code>	Merge PDF files, optionally rotating some pages.

The HTML documentation for CamlPDF is built in `doc/html/camlpdf` when CamlPDF

is built.

Further Reading

The author's book is a suitable introduction to the PDF file format:

<http://shop.oreilly.com/product/0636920021483.do>

For any serious work, you will need the PDF Reference Manual

http://www.adobe.com/devnet/acrobat/pdfs/PDF32000_2008.pdf

For an introduction to OCaml, the author's book is available:

<http://ocaml-book.com> or at Amazon.com

