

A PDF Primer for APLers

by Adrian Smith (adrian@causeway.co.uk)

Background

Adobe's "Portable Document Format" is now nearly 10 years old and has become a very well-supported way of transmitting a document between any two computers so that "What You See is What They Get". In this way it differs fundamentally from HTML, where the person viewing the document has a great deal of control over the appearance. A PDF will look the same on a PC, a Mac, or a whole variety of Unix boxes.

Almost everyone seems to have downloaded Adobe's free PDF Reader application, so there is rarely a problem in posting documents on your website in this format, for example to provide a 'printed manual' in a downloadable form. However, PDFs have one very strong selling point just now – the format is sufficiently simple that there is nowhere to hide a virus. You can safely mail them as attachments without endangering your friends.

There are plenty of tools out there which can take a PostScript file and magically 'distil' it into the PDF format. Adobe Acrobat is by far the best and does an extremely competent job. However it is quite expensive and adds considerably to the complexity of any batch-production process (for example you might want to email a few hundred reports) where you need to create lots of little PostScript files and wait around for Distiller to do its magic.

What is less well known is that PDF is actually a published format, and can be written as pure ASCII text. The construction process is by no means as easy as HTML, but (in principle) any environment which can create a text file can generate a valid PDF. Naturally, this includes APL+Win, as this short session will demonstrate. What I cannot do in 45 minutes is to show you everything you might need to construct PDF files entirely for yourself. If you really want to implement this in production code, you will still have to do some serious research. Hopefully, I can help you to look in the right places.

Let's Take One Apart ...

There is no better way to learn any format than to take it apart, and that is what the rest of this session will be about. What I have done is made the simplest PDF I possibly could using Causeway's NewLeaf workspace and some code like

```
n|Use letter
n|Place 'Hello, world'
PG+n|Close
pdf+MakePDF PG
```

This writes 'Hello, world' in 12pt Times-Roman at the top of a sheet of US Letter paper. It then converts the NewLeaf intermediate format into a simple character vector of PDF code, which is actually a mere 991 bytes long. I have intentionally used linefeed-delimited vectors for the final result, as they are easy to browse with)edit and can be written directly to file in this form. To see it on screen, simply save it to file and load it into the Adobe viewer.

HelloWorld example

```
%PDF-1.1
%©Causeway

1 0 obj  % document info
<<
  /Creator (Causeway NewLeaf)
  /Producer (Causeway PDF Filter)
  /CreationDate (D:200110071818)
>>
endobj

2 0 obj  % Root node
<<
  /Type /Catalog
  /Pages 4 0 R
>>
endobj

% ===== Resources =====

3 0 obj  % Font
<<
  /Type /Font
  /Subtype /Type1
  /Name /F1
  /BaseFont /Times-Roman
  /Encoding /WinAnsiEncoding
>>
endobj

% ===== Content =====

4 0 obj  % Page list
<<
  /Type /Pages
  /MediaBox [0 0 612 792]
  /Count 1
  /Kids [5 0 R ]
  /Resources
<<
  /ProcSet [/PDF /Text]
  /Font << /F1 3 0 R >>
>>
endobj

5 0 obj  % Page 1
<<
  /Type /Page
  /Parent 4 0 R
  /Contents 6 0 R
>>
endobj

6 0 obj  % Stream
<< /Length 62 >>
stream
1 J 1 j
BT
/F1 12 Tf
1 0 0 1 72 705.6 Tm (Hello, world) Tj
ET
endstream
endobj

xref
0 7
0000000000 65535 f
0000000021 00000 n
0000000155 00000 n
0000000252 00000 n
0000000405 00000 n
0000000571 00000 n
0000000649 00000 n

trailer
<<
  /Size 7
  /Info 1 0 R
  /Root 2 0 R
>>
startxref
772
%%EOF
```

As you can see, PDFs tend to be quite long and thin, which is why I have switched to double-column to show it! Have a careful look at the overall structure (a highlighter pen could be very handy here) and some things will slowly begin to make sense. Some things I will just have to explain, like that **xref** block at the end. Ignore this for the moment, and observe that there seem to be lots of logical blocks delimited by “# 0 obj endobj” markers, where # is an identifying number that starts at 1.

Let’s begin at the beginning and explain the structure as we go. Actually , we should perhaps begin at the end (PDFs are actually designed to be read from the back) but that would be too confusing. So from the top it is:

```
%PDF-1.1
%©Causeway
```

That first line is mandatory and tells the reader that we are working within quite an old standard. This will all work fine in Acrobat Reader 2, and I normally use 4, although 5 is fine too. The second line has a gratuitous ‘hibit’ character which is designed to stop mail programs from treating the file as text, and generally trashing it.

```

1 0 obj    % document info
<<
  /Creator (Causeway NewLeaf)
  /Producer (Causeway PDF Filter)
  /CreationDate (D:200110071818)
>>
endobj

```

This section is entirely optional, but it shows up in the ‘document properties’ screen and I always feel it is a good idea to populate these things. It also illustrates several other points:

- Each object has a unique number and a version number (always 0). Comments are preceded by % symbols and the viewer ignores from the comment to the end of that line.
- Objects typically consist of dictionaries which are delimited by double angle-brackets. A dictionary consists of one or more name-value pairs; names are case-sensitive and always have a leading slash symbol. This is very easy to forget!
- Strings (any text) are placed between parentheses. If you want parentheses within a text string, you must escape them with a backslash. Of course the same applies to the backslash itself!
- The layout is quite immaterial, but I find it helps to indent things like this.

I suppose I should show you some APL code ... here are the functions which created that info block:

```

▽ pdfΔMakeInfo;obj
[1]  A Make document info object
[2]  obj←('/Creator' '/Producer'),[1.5] '(Causeway NewLeaf)'
      '(Causeway PDF Filter)'
[3]  A Date in ANSI standard format (local time)
[4]  obj←obj;'/CreationDate'('(D:',(,'ZI4,ZI2,ZI2,ZI2,ZI2'□FMT
      1 5p□TS),')')
[5]  'document info'pdfΔcatobj pdfΔMakeDict obj
▽

▽ dict←pdfΔMakeDict pvm
[1]  A Make a dictionary out of content (2-column matrix of pv pairs)
[2]  pvm[;2]←' ','pvm[;2] ◇ dict←c[2]pvm
[3]  dict←ε(cnl,' '),dict
[4]  dict←'<<',dict,cnl,'>>',cnl
▽

```

Now for the ancestor of the rest of the document ...

```

2 0 obj    % Root node
<<
  /Type /Catalog
  /Pages 4 0 R
>>
endobj

```

If you turn back to the complete listing, you will see that this is pointed to from the **trailer** section of the file – as I said, these are designed to be opened at the back, like any good newspaper!

What this is telling us is that it is the ‘catalog’ for the entire file and that the only other thing of interest is some pages, which we should look for in object number -4. There are plenty of other things that might be catalogued, such as outlines, but I did say this was a simple one to get us started! On to the next

```
% ===== Resources =====  
  
3 0 obj  % Font  
<<  
  /Type /Font  
  /Subtype /Type1  
  /Name /F1  
  /BaseFont /Times-Roman  
  /Encoding /WinAnsiEncoding  
>>  
endobj
```

Usually this would just list all the fonts which you are using in the document – here there is just one. This block needs explaining in detail, as fonts are crucial to the success of PDF publishing.

- It is a font (obvious), but specifically it is an Adobe Type-1 PostScript font. We shall refer to it with the mnemonic ‘F1’ from now on and the Reader can use Times-Roman to render it. This is a very important feature of PDF documents – if you restrict yourself to the 14 in-built fonts (Times, Helvetica, Courier, Symbol, ZapfDingbats) then your document will be easily readable by anybody. What is more, you do not need to include the definition of the font (typically around 40K each) so your PDF is very small.
- It is encoded to the Windows standard. This is not the default, so we should always include this entry too.

If you have used several fonts, there will be several very similar objects to define them. You can (of course) bundle in PostScript fonts here, but do not even think about bundling TrueType fonts. They go into the PDF as bitmaps and look terrible in the viewer. To be avoided at all costs!

On to the real meat now!

```
% ===== Content =====  
  
4 0 obj  % Page list  
<<  
  /Type /Pages  
  /MediaBox [0 0 612 792]  
  /Count 1  
  /Kids [5 0 R ]  
  /Resources  
<<  
    /ProcSet [/PDF /Text]  
    /Font << /F1 3 0 R >>  
>>  
>>  
endobj
```

This lists all the pages in the document, and defines any resources which are common to them. You can insert these settings for each page, but I cannot see that the paper -size will change mid-stream for anything I will generate, so I put them here to save space. Again, working through the detail:

- The *MediaBox* is what most of us would call the paper-size. Divide by 72 and you will recognise US Letter. Note the notation for an array – you delimit it with square brackets. PDF has heterogeneous and nested arrays, just like APL.
- The *Count* is the page count, and the *Kids* array is a set of object references to each page. This entry can grow quite large for a big document! Each reference gives the object number and version followed by R which is the **Reference** operator.
- The *Resources* entry is itself a dictionary (these can be nested too) and lists the PDF procedure sets which will be needed to render our pages (these are very standard) and any fonts which will be needed (again referencing the font objects by number)

So now we have almost reached the metal! You might confidently expect to see ‘Hello, world’ somewhere in the next object, but not quite. This onion has yet one more layer to peel:

```
5 0 obj    % Page 1
<<
  /Type /Page
  /Parent 4 0 R
  /Contents 6 0 R
>>
endobj

6 0 obj    % Stream
<< /Length 62 >>
stream
1 J 1 j
BT
/F1 12 Tf
1 0 0 1 72 705.6 Tm (Hello, world) Tj
ET
endstream
endobj
```

The *Page* object connects back to its parent (the *Pages* object) and points forward to a sibling which actually has the content we are looking for. This starts off with a dictionary to say how long it is (exact byte-count, including the line-feed characters from *stream* to *endstream*, but not including these two words) and then we have some marvellous mumbo-jumbo that actually puts the marks on the paper. Let’s pause for breath and consider this piece very care fully indeed!

- The first thing I do is set all line-ends and line-joins to ‘rounded’ which fits with the way Windows does it and in general makes for nicer looking tables and line-graphs. In this particular PDF, it has no effect whatsoever! Note that we need to think left-to-right here – all the PDF functions take their arguments on the left, so **J** and **j** are the functions.
- **BT** is a niladic function which sets us into text mode. This is one of the slightly strange things about PDF which must be a viewer optimisation – you can’t mix text and graphics in the same block.

- **Tf** is the function to set the working font. Here it is given two arguments, the font name and font size (in points). Remember that we named it in that font resource object a little while ago.
- **Tm** positions us on the page. Ignore the 1 0 0 1 (this sets stretch and skew) and notice the last two numbers 72 705.6 which are the (x,y) co-ordinates of the start of the text baseline. These are in points, measured from the *lower* left-hand corner of the paper. NewLeaf has a default left margin of 1" which leads directly to the number 72 here. The top margin is also 72pt, and if you fire up an APL interpreter and try $(792-72) - 12 \times 1.2$ you will discover where that second number came from – 792 is the height of the paper, and 1.2 is the default ‘leading’ so a 12pt font is vertically spaced on rails 14.4pt apart.
- **Tj** actually writes the text. Its argument is a simple string, in this case the text we started with. Do remember to ‘escape’ parentheses in all strings here!
- **ET** ends the text block, which would be necessary if there were any drawing operations to follow, for example table boundaries or even text underlining.

That is really the end. Now we have some special stuff which is purely there to help the viewer handle large files efficiently,

```
xref
0 7
0000000000 65535 f
0000000021 00000 n
0000000155 00000 n
0000000252 00000 n
0000000405 00000 n
0000000571 00000 n
0000000649 00000 n
```

After the xref line you get a count of the number of entries followed by a table in a very fixed format. Ignore the ‘zeroth’ row which is some kind of placeholder. What we then have is the exact byte-offset into the file (or vector in the workspace) of each object. For example:

```
1212521pdf
3 0 obj %
```

... leads us directly to object number 3. Note that the objects need not be in order here, it was just convenient for me to make the file this way.

Finally, we end the file with a trailer which a PDF-reader application can use to find its way quickly to the xref and the root note of the data:

```
trailer
<<
  /Size 7
  /Info 1 0 R
  /Root 2 0 R
>>
startxref
772
%%EOF
```

Here is the APL code which formatted it:

```

▽ pdfΔMakeXref;xref;xp;mat
[1]  A Finish off the xref table
[2]  A We are using LF-delimited data here so we need a trailing blank.
[3]  xp←pdfΔCurrSize ◇ xref←'xref',n1,'0 ',(⌈1+pXREF),n1
[4]  A This is the only place where the formatting is very fussy!
[5]  mat←0 65535,[1]XREF,[1.5]0
[6]  mat←1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0\`ZI10,ZI5'␣FMT mat
[7]  mat[;18]←'n' ◇ mat[1;18]←'f'
[8]  pdfΔcat xref,,mat,n1
[9]  pdfΔcat'trailer'
[10] pdfΔcat `1↓pdfΔMakeDict'/Size' '/Info' '/Root',[1.5]
      (⌈1+pXREF)'1 0 R' '2 0 R'
[11] pdfΔcat'startxref',n1,(⌈xp)
▽

```

Each line in the xref must be exactly 20 bytes long (including the newline), hence the very careful use of format and expand.

Working with Outlines

I said at the beginning that this was a very simple example. One of the essential components of a real PDF is a proper contents tree, which is defined as a set of *Outline* objects. First, we should make a small example:

```

A Outline tree example - lists a few functions
fns←(c[2] 'T' ␣n1 3)~'' '
n1Use letter
'aplfn' n1DefineStyle ('font' 'CO,10')
:For fn :in fns
  n1Bookmark fn
  'subhead' n1Place fn
  'aplfn' n1Place ␣vr fn
:End
PG←n1Close

```

This simply includes the function names in the navigation pane of the PDF viewer. Of course we can make a proper tree-structure here too, and we can even control which parts of the tree are opened up when the file is loaded, but for now I will stay with the simple case:

Outline example

<pre> %PDF-1.1 %©Causeway 1 0 obj % document info << /Creator (Causeway NewLeaf) /Producer (Causeway PDF Filter) /CreationDate (D:200110072250) >> endobj 2 0 obj % Root node << /Type /Catalog /Outlines 6 0 R </pre>	<pre> /Pages 19 0 R /PageMode /UseOutlines >> endobj % ===== Resources ===== 3 0 obj % Font << /Type /Font /Subtype /Type1 /Name /F1 /BaseFont /Times-Roman /Encoding /WinAnsiEncoding >> </pre>
--	---

```

endobj

4 0 obj  % Font
<<
  /Type /Font
  /Subtype /Type1
  /Name /F7
  /BaseFont /Helvetica-Bold
  /Encoding /WinAnsiEncoding
>>
endobj

5 0 obj  % Font
<<
  /Type /Font
  /Subtype /Type1
  /Name /F13
  /BaseFont /Courier
  /Encoding /WinAnsiEncoding
>>
endobj

% ===== Content =====

6 0 obj  % Outlines
<<
  /Type /Outlines
  /Count 12
  /First 7 0 R
  /Last 18 0 R
>>
endobj

7 0 obj  % Leaf node
<<
  /Parent 6 0 R
  /Dest [ 20 0 R /XYZ 60 804 0 ]
  /Title (TTest)
  /Next 8 0 R
>>
endobj

8 0 obj  % Leaf node
<<
  /Parent 6 0 R
  /Dest [ 20 0 R /XYZ 60 511.2 0 ]
  /Title (TableDemo)
  /Prev 7 0 R
  /Next 9 0 R
>>
endobj

```

... and so on. The rest of it is very similar to the first example. Looking carefully at object-6, we can see that it is like the page catalogue – it is the root-node of a tree of outlines, each level of which form a linked list. If we examine object-7 in detail:

- It has a *parent*, which is just the outline root node.
- It has a *destination* which is an array giving the target object (as a relative reference, of course) and the exact position of the destination on the page. *XYZ* actually stands for *x,y,zoom* which makes more sense than *(x,y,z)* here. Setting the zoom to 0 simply preserves the current magnification.
- The *title* is what is written into the outline tree

Note that the first node in the set just points to the *next* one, all the middle nodes will point to *prev* and *next*, and the last one just points to *prev*. They all point to *parent*.

This was Just Scratching the Surface

As I said at the start, this is all firmly in the category ‘straightforward but very tedious’. Writing the PDF driver for NewLeaf was not particularly hard, as NewLeaf has already done the tough part, and has worked out where all those words and lines physically go on the paper. I added some extra stuff to do the bookmarking (and hence generate the outlines automatically) and left it at that. There is lots more you can do with this format, like fancy gradient fills and links to external resources like webpages. If you want to make PDFs and are feeling lazy, have a look at NewLeaf and RainPro and see if they fit in to your existing reporting engine. If you are feeling brave, dig out the PDF specification from the Adobe site, and start from where these notes end.

If in doubt, call the number below!

Adrian Smith
 Email: adrian@causeway.co.uk
 Tel: +44 (0) 1653 696760