Portable Document Format: An Introduction for Programmers

by Kas Thomas

http://www.mactech.com/articles/mactech/Vol.15/15.09/PDFIntro/

Get to know the internals of Adobe's new document interchange standard.

With the growing popularity of the World Wide Web (and the growing complexity of computer-created documents), the need for an extensible, platform-independent standard for document interchange has never been greater. More people need to share more kinds of information than ever before.

But the growing complexity of computer-created documents has led to a kind of free-for-all where data formats are concerned. Bridging the many font technologies, imaging models, data types, and compression standards currently in use (while maintaining a document's "look and feel" across operating systems, output devices, and CPU architectures) would seem to be a fundamentally intractable problem. How can one ever hope to reconcile so many competing "standards," while enforcing consistency of appearance?

Rich Text Format was an early attempt to bring consistency to the digital page. But RTF - conceived in the predawn of ARPANet - was not designed to accommodate non-text data types. Hypertext Markup Language (HTML) addressed that need while introducing the notion of hypertext search. But by abstracting font metrics out of the picture, HTML's creators unwittingly fostered implementation-dependent page appearances - a critical flaw in any system of information display that values consistency.

Adobe PostScript® was the first page description language to tackle the dual problems of consistency and fidelity head-on. The key to its success was the abandonment of old paradigms based on artificial distinctions between text and graphics. In the PostScript world, everything is graphical - especially text.

PostScript embodied a procedural model for graphics, in which typefaces were simply collections of curves. In PostScript, a page consisting of text and graphics was sent to a printer as a series of lineto and arcto commands; the printer would interpret the commands, create a display list, and rasterize the individual graphic elements to recreate the page. Any graphic element that couldn't be described in vector terms - like lineto or arcto - would simply be treated as a bitmap.

Limitations of PostScript®

As a vector-graphics language, PostScript was - and still is, in many ways - without equal. But there are aspects of the language that make it less than ideal as the basis of a universal document-interchange format. For example:

Lack of searchability: Most users of electronic documents expect to be able to search text using keywords or traverse an index or table of contents, then jump quickly to relevant sections. PostScript was not designed to allow hypertext links. Random access to data is, in general, problematic because of the freeform way in which PostScript files are organized.

Font substitution: Fonts are not always present in the file. Unsightly font substitutions occur when needed fonts are not found on the target system.

Poor editability: PostScript files are not easily edited, annotated, or updated. When a PostScript file needs to be changed, it is usually rewritten from scratch.

No support for multimedia data types: PostScript files do not accommodate QuickTime movies, slideshows, sound bites, etc.

No support for restricted access: Security features (such as encryption, passwording, and digital signatures) were not part of PostScript's design specification.

Large file size: Ironically, what was once one of PostScript's strengths (compact representation of complex imagery) has been turned on its head as file size and document complexity have grown hand-in-hand. PostScript files are now often monstrously large.

Slow execution: Large files containing complex graphics can be slow to parse and would lead to unacceptable latency in a viewer program.

Unpredictable errors: Variations in PostScript interpreters and in the quality of PostScript code generated by applications ensure that end users will see errors - errors that are sometimes not handled gracefully. One bad line of code in a large PostScript file can - and often does - render the entire file unusable.

Adobe faced a critical decision in coming up with a new document standard: whether to modify the PostScript language to suit the needs of universal document-sharing (which would mean significantly complicating the language), or come up with an entirely new page description language designed specifically for document interchange. Adobe chose the latter.

PDF Version History

Version 1.0 of the Portable Document Format attended the introduction of Adobe Acrobat (initially called Carousel) in 1993. As originally conceived, PDF was a pure ASCII format; this was quickly changed when Adobe realized that some e-mail transmission systems fail to preserve 7-bit characters and can change line endings, thus corrupting PDF files. PDF is now considered a true 8-bit binary format.

Version 1.1 of PDF accompanied the release of Acrobat 2.0 in March 1996. New features included passwording, device-independent color, the ability to tie related articles into "threads" and an ability to provide links connecting PDF files to each other.

Version 1.2 of PDF came out with Acrobat 3.0 in October 1996. It featured support for interactive page elements (such as radio buttons and checkboxes) and forms, support for mouse events, multimedia types, Unicode, advanced color features (including color-separation spaces, halftone screens, and advanced patterns and spot functions), and image proxying via the Open Prepress Interface (OPI) protocols.

The current version of PDF as this article is written is 1.3 (for Acrobat 4.0), which was released in March 1999. Important features added in this version include support for JavaScript 1.2, digital signatures, image masking and smooth shading, support for right-to-left and left-to-right reading directions, advanced trapping capabilities, and sophisticated web-capture features.

What is PDF?

Portable Document Format is an extensible page-description protocol that implements the native file format of the Adobe Acrobat suite of commercial software products. The goal of the format is to make possible the hardware-independent interchange of high-resolution documents - documents that may contain text, graphics, multimedia elements, and/or custom data types, plus (optionally) links to other files or URLs containing such items. The format supports text search, random access of data, bookmarks, links, annotations, interactive page elements (checkboxes, text-edit fields, etc.), encryption, compression, JavaScript actions, and much more.

The complete 518-page specification for PDF 1.3 is available online at <http://partners.adobe.com/asn/developer/PDFS/TN/PDFSPEC.PDF>. Any developer who wants to support (or even extend) the format is free to do so - it's an open standard, in the same way that TIFF (Tagged Image File Format) is. But as with TIFF, implementing a truly comprehensive PDF-read capability is not something an individual programmer can expect to accomplish unaided, whereas providing a PDF-write capability is fairly straightforward.

PDF implements documents as a hierarchy of tagged objects, organized into trees and/or linked lists. The objects, which can be any of seven basic types (discussed in further detail below), can be purely structural in nature or can encapsulate various types of content, or attributes, or pointers to external resources. There are very few hard and fast rules as to how a document must be structured, because the document's logical structure and physical structure may differ. In broad terms, a PDF file can be thought of as encompassing four types of structure, as shown in Figure 1. At the lowest level, a PDF file consists of objects - names, numbers, arrays, etc. (Most of the object types in PDF have corresponding object types in PostScript.) At a somewhat higher level, there's the file structure of a PDF file, which determines how root-level objects are stored and accessed. On a higher level still is the document structure, which takes into account how the various member objects of all the various hierarchies are organized into pages (and/or sections, chapters, etc.) and how attributes are assigned so as to give the PDF document its particular behavior and appearance when viewed interactively.

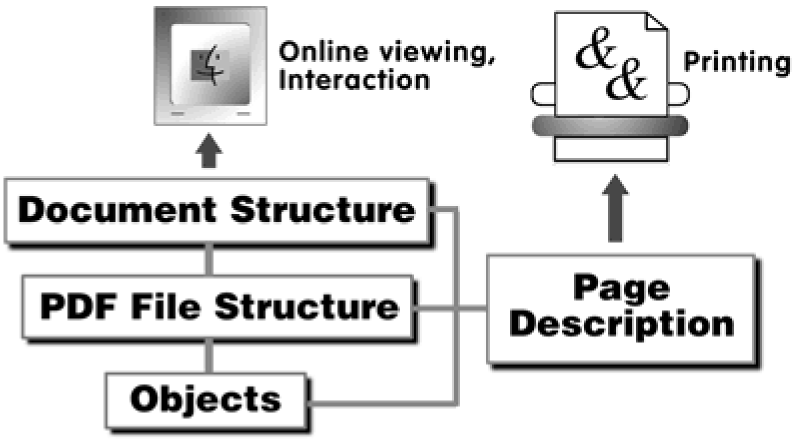

Figure 1. *A PDF page description draws on various levels of content organization, some of which govern the appearance of the printed image, others of which affect the document's behavior in an interactive, online viewing environment.*

Pages are less important as an organizational paradigm than you might imagine. If you think about it, the division of digital content into pages is mostly an arbitrary convention, rooted in the use of sheets of paper. There is no a priori requirement, in the digital world, that a document consist of pages, any more than ice cream has to consist of scoops. Still, most PDF pages will - at some point - be printed out on a laser printer, imagesetter, or platesetter, at some predetermined size. This is where PDF's PostScript heritage comes into play. PDF incorporates 73 page-marking operators of the lineto/stroke/fill variety, 40 of which have direct PostScript counterparts. These operators, occurring in stream objects, govern the appearance of graphical elements on the printed page.

At the page level, then, a PDF document consists of the content objects and page-markup operators needed to render a physical page on an output device.

In a page-description sense, you can think of PDF as a dialect of PostScript. In a document-description sense, it's much more than that, because in the PDF world a document is more than just pages. PDF was created to deal with issues beyond mere printable text and graphics. PDF documents are searchable and annotatable, can be password-protected, may contain multimedia elements (and/or forms), can perform JavaScript actions, and so on.

Differences Between PDF and PostScript®

To the untrained eye, much of PDF looks like PostScript. But there are significant differences, the main one being that whereas PostScript is a true language, PDF is not: PDF lacks the procedures, variables, and control-flow constructs that would otherwise be needed to give it the syntactical power of a bonafide language. In that sense, PDF is really a page-description protocol.

Language features were taken out mainly in order to simplify the parsing of PDF files and reduce the likelihood of serious errors. It would have been hard to guarantee random access to data any other way. A viewer-type program that could extract and display a selected page from a large PostScript file would have no choice but to scan the file from beginning to end in order to find the desired page and all its components. This would, of course, preclude incremental download viewing of the file. But in addition, the time required to find and view a page would depend not only on the complexity of the page but the length of the document - a highly unsatisfactory situation.

Every PDF file has a cross-reference table that can be used to quickly find and access pages and other important resources in the file. The xref table is always stored at the end of the file, so that programs that create PDF files can do so in a single pass, and programs that consume (or read) PDF files can locate needed references quickly. Bottom line: the time needed to access a page in a PDF document is essentially independent of the size of the file.

Incremental updating or user-editing of files is another feature that would have been hard to implement in PostScript. A user working on a massive document shouldn't have to wait for the entire file to be rewritten each time changes to the document are made (as is commonly done with PostScript). PDF allows modifications to be appended to a file, leaving the original data intact. This means changes can be made in a time proportional to the size of the change rather than the size of the file. It also means previous versions of the file are still available, and an infinite-Undo facility is possible.

Further differences between PDF and PostScript include the following:

- PDF files always include sufficient font metrics to ensure viewing fidelity.
- PDF files may contain hypertext links and other objects intended for user interactivity.
- PDF is extensible, yet designed in such a way that viewer programs that only understand earlier versions of the format will not break when they encounter unfamiliar features. (The PDF specification goes into detail on how viewer programs should behave under a variety of non-standard conditions.)

PDF File Structure

A canonical PDF file is organized into four major parts (see Figure 2): a one-line header, a body, a cross-reference table, and a trailer.

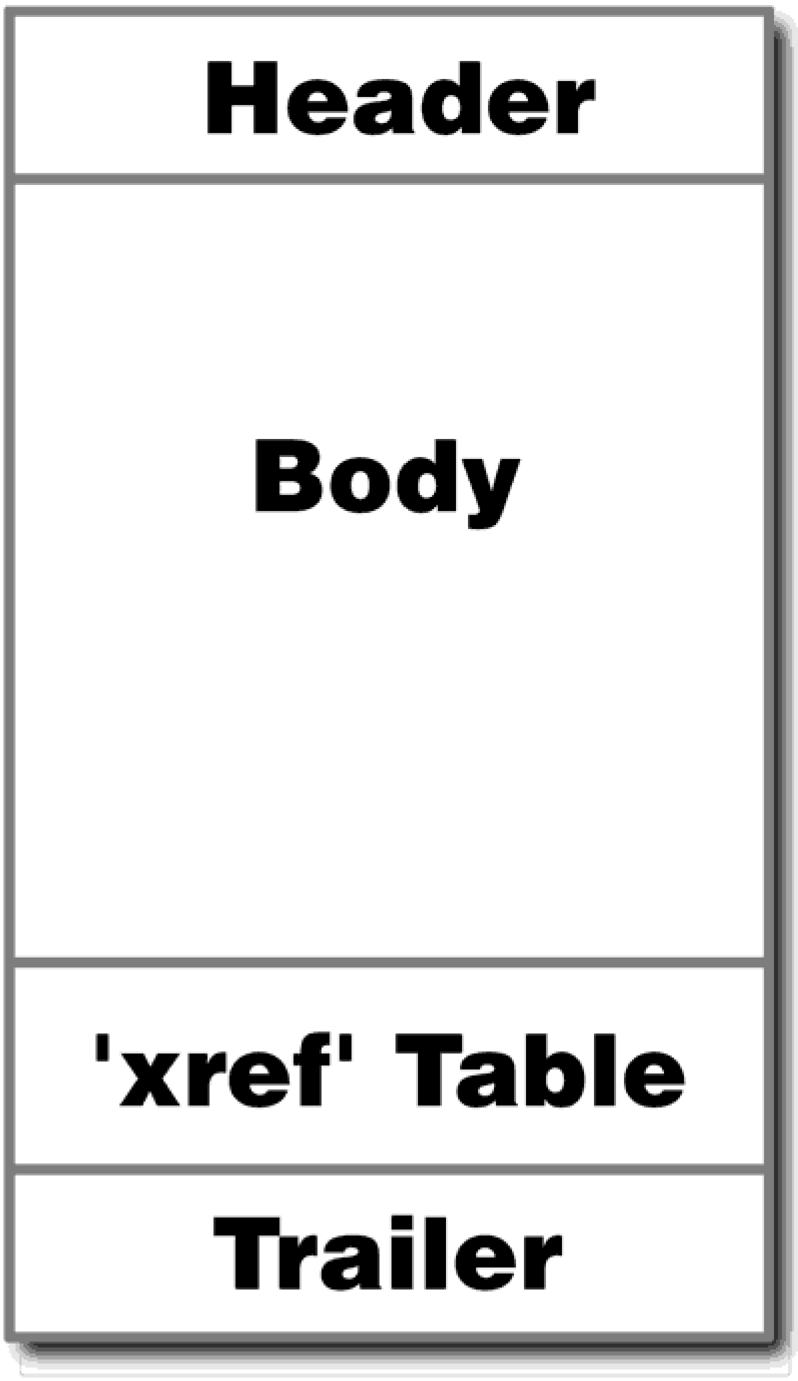

Figure 2. *The structure of a canonical PDF file.*

Header

The first line of the PDF file specifies the version number of the PDF specification to which the document adheres, written as a PostScript-style comment. For example:

```
%PDF-1.3
```

This would indicate that the file conforms to Version 1.3 of the PDF spec. As in PostScript, the % character precedes all comments. Comments may occur anywhere in any file, and all words from the percent sign to the end of the line will be disregarded. (Occurrences of the percent sign within streams or strings are not treated as comments.) By convention, the second line of most PDF files is also a comment, usually containing one or more "high bit" ASCII characters (in the range 0x80 to 0xFF). This signals e-mail clients and other programs that the file contains binary data and should not be treated as 7-bit ASCII text.

Body

The body of a PDF file consists of the objects that comprise the document's contents. These objects would typically include text streams, image data, fonts, annotations, etc. (See the discussion of objects further below.)

The body can also contain numerous types of invisible (non-display) objects that help implement the document's interactivity, security features, or logical structure.

Cross-Reference Table

The cross-reference table contains offsets to all of the objects in the file, so that it is never necessary to scan large portions of a file (or "walk" a linked list) in order to locate needed elements. If no updates have been added to the file, the cross-reference table will be contiguous, consisting of a single section. New sections are added each time the file is modified.

Within any single section of a cross-ref table, there are subsections corresponding to blocks of consecutively numbered objects. The entry for each object is always exactly 20 bytes long, including the line-end character(s). The first ten bytes specify the object's offset, in a ten-digit number; a space separator follows; then a five-digit number giving the object's generation number; then another space; then the letter 'f' or 'n' to indicate whether the object is free or in use; then the end-of-line marker. (There are three legal possibilities for end-of-line. They are, in hex: 0x200A, 0x200D, or 0x0D0A.) It's easier to show the xref in action than to describe it, so here's an example of a cross-reference table containing entries for seven objects, arranged in four subsections:

```
        xref
        0 1
        0000000023 65535 f
        3 1
        0000025324 00000 n
        21 4
        0000025518 00002 n
        0000025632 00000 n
        0000000024 00001 f
        0000000000 00001 f
        36 1
        0000026900 00000 n
(End-of-line characters omitted for clarity.)
```

The first subsection, containing a single object (object zero), is special; its significance will be discussed shortly. The second subsection lists one entry, for object number 3. (The offset to object number 3, from the start of the PDF file to the beginning of the object itself, is 25,324 bytes.) The third subsection lists four objects, the first of which is object number 21. The other objects in this group are numbered consecutively and therefore carry numbers 22, 23, and 24. The fourth subsection has but one object, number 36.

All objects are marked either 'f' for free or 'n' for in use. Better terminology would perhaps have been valid and invalid, or current and obsolete. "Free" essentially means that although the object may still be physically present in the file, it is obsolete and shouldn't be used. "In use," conversely, simply means that the object is valid and usable. (It doesn't mean the object is "checked out" or "busy.") Entries marked 'n' have a byte offset followed by a generation number, whereas entries marked 'f' contain the number of the next free (invalid) object, and the generation number to be used when and if the current object is resurrected.

The first entry in a cross-reference table is always free and has a generation number of 65,535; it sits at the head of a linked list of free objects. The final free object in the table (the tail of the linked list) uses zero as the object number of the next free object.

You can see how this scheme works in the example above. Notice that object zero points to the next free object in the table - namely, object number 23. Since object 23 is free, its table entry doesn't start with a byte offset; instead, it starts with a pointer to the next free object, namely 24. But object 24 happens to be the final free object in the file, so its entry begins with zero.

By convention, an object's generation number is incremented at the time it is freed. That's why objects 23 and 24, above, have generation numbers of 1. Should these objects ever be resurrected, their table entries will go from 'f' to 'n', byte offsets will be used, and the generation number will still be 1. Should the resurrected objects be obsoleted again, they will go back to 'f' status, with a generation number of 2. And so on.

Trailer

The PDF trailer enables an application reading the file to quickly find the cross-reference table and certain special objects. (Applications are expected to read a PDF file from its end.) The last line of a PDF file contains only the end-of-file marker, %%EOF. The two preceding lines contain the keyword startxref and the byte offset from the beginning of the file to the beginning of the word xref in the last cross-reference section in the file. Preceding this is the trailer dictionary; and at the top of the trailer is the word trailer. For example:

```
trailer
<<
/Size 22
/Root 2 0 R
/Info 1 0 R
>>
startxref
24212
%%EOF
```

The byte offset from the start of the file to the start of the word xref at the top of the cross-reference table is, in this instance, 24,212. The trailer dictionary consists of everything between the double angle brackets, << and >>. The mandatory /Size key gives the total number of entries in all sections of the document's xref table. The /Root key (also mandatory) gives the object reference for the document's catalog object, which is a special type of object that contains pointers to the roots of the various object trees that contain the document's content. The /Info key is optional and references a special dictionary that contains information about the document that will appear in the Acrobat viewer's Document Info dialog.

The Incremental Update Mechanism

The trailer, it turns out, plays an important role in the way PDF implements incremental updating. The key concept to understand here is that a PDF file is never overwritten, only added to. That goes for all portions of the PDF file - even the trailer itself, and the end-of-file marker. In other words, a multiply-updated PDF document may contain multiple trailers - and multiple end-of-file markers! (There may be numerous occurrences of %%EOF.) Each time the file is edited, an addendum is written to the tail of the file, consisting of the content objects that have changed, a new xref section, and a new trailer containing all the information that was in the previous trailer, as well as a /Prev key specifying the byte offset (from the beginning of the file) of the previous xref section. The cross-reference info will then be distributed across more than one xref section. To access all of the cross-references, the reader must walk the list of /Prev keys in all the trailers, in reverse order.

Space doesn't permit a detailed exploration of updates here, but you can find several examples in Appendix A of the PDF 1.3 specification (available at <http://partners.adobe.com/asn/developer>).

PDF Data Types

There are seven basic kinds of objects in PDF: Booleans, numbers, names, strings, arrays, dictionaries, and streams. (Technically, there is an eighth type: the null object.) Any object can be labelled so that it can be referenced by other objects. When an object is labelled this way, it is called an indirect object. The principle concept here is, of course, indirection, which can be useful in a variety of circumstances. (More on this in a minute.)

Booleans

In PDF, the keywords true and false represent Boolean objects with values non-null and null. (Note, incidentally, that PDF is case-sensitive: TRUE and True are not the same as true.)

Numbers

PDF supports two types of numbers: integers (32-bit signed) and real (±32,767, with the smallest value being the reciprocal of 65,535). Exponential forms, such as 1.0E4, are not supported.

Names

A name is a sequence of ASCII characters in the range 0x21 through 0x7E (except the characters %, (,), <, >, [,], {, }, /, and #) , preceded by a slash. Any character except null can be represented by its two-digit hex equivalent, preceded by #. The maximum allowable length for a name is 127 bytes. Some examples:

```
/Contents
/Chap6_Section1
/Chap6#5FSection1
/Name#20with#20spaces
/1.5
/.end
```

Strings

In PDF, as in PostScript, a string consists of a series of 8-bit bytes surrounded by parentheses. The maximum supported length is 65,535 bytes. When a string is too long to be written on one line, it can be broken across several lines by using the backslash character (\) at the end of the line to signify continuation. The backslash itself (and the end-of-line carriage return) will not be considered part of the string. For example:

```
( This is a valid string. )
( This is a somewhat longer \
string, split across \
three lines. )
```

Any 8-bit value can be represented either by its octal equivalent (in the form \ddd, where ddd is the octal number), or by its two-digit hex equivalent, surrounded by angle brackets. Thus:

```
(Two + two = four.)
(Two \053 two \075 four.)
(Two <2B> two <3D> four.)
(<54776F202B2074776F203D20> four.)
```

The same escape sequences that apply in PostScript (such as \r for carriage return and \t for tab) also apply in PDF strings.

Arrays

An array is any sequence of PDF objects, not all necessarily the same type, enclosed in square brackets:

```
[ 1 2 3 6.25 ]  % an array of numbers
[ true /Chap9 3.14 (yes) ] % array of misc. objects
```

Dictionaries

A dictionary is a table containing key/value pairs. As in PostScript, a dictionary consists of two left angle brackets, followed by one or more key/value pairs, followed by a pair of right angle brackets:

```
<< /Chapters 29 /Encrypt true /Warn6 (no undo) >>
```

Unlike PostScript, PDF requires that the key always be a Name object, whereas the value can be any kind of object - even another dictionary. The maximum number of entries in any dictionary is 4,095.

Dictionary objects are among the most common objects in a PDF file, since items like pages and fonts are represented through dictionaries. A common idiom is for a /Type key to specify the kind of object represented by the dictionary. (The associated value will typically be a name. For example: /Type /Font.)

Streams

A stream is a sequence of 8-bit bytes bracketed by lines containing the keywords stream and endstream. Any type of content made up of raw binary data is represented by a stream. In some respects, a stream is like a gigantic string object, but whereas strings must be read all at once, in their entirety, streams can be consumed in piecemeal fashion (and usually are, because of their size).

Streams are packaged in a particular way, so they can be located quickly. That is to say, they're represented as indirect objects (see below), which also means the stream will be bracketed by obj and endobj keywords. Within the obj/endobj statement, there must be an attribute dictionary before the stream keyword, giving information about the data that follows. At a bare minimum, the attribute dictionary must contain a /Length key; it may also contain other keys, such as a /Filter key indicating the kind of compression employed. (PDF supports LZW, runlength, CCITT fax, Flate, and DCT compression methods.)

As an example, a small text stream might look like:

```
2 0 obj
<<
/Length 39
>>
stream
BT
/F1 12 Tf
72 712 Td (A short text stream.) Tj
ET
endstream
endobj
```

The top line gives the object number (namely, 2) and generation number (zero). The attribute dictionary contains only a length key, showing the number of bytes from the beginning of the line after stream to the beginning of the line containing endstream. Since the stream consists of displayable text, it is bracketed by the page-markup operators BT and ET, for "begin text" and "end text." The line beginning with /F1 says to find and load Font No. 1 in 12-pt size. The next line begins with 72 712 Td, which means position the text at (x,y) = (72, 712) in user space, which is one inch to the right of the page's left edge and approximately ten inches up from the bottom edge. The text itself is given as a string followed by the display text operator, Tj.

Indirect Objects

An indirect object is a numbered object. The content can be any kind of native PDF object (Boolean, number, name, string, etc.), bracketed between obj and endobj keywords. The endobj keyword exists on its own line, but the obj keyword must occur at the end of the object ID line, which is the first line of the indirect object. The object ID line, in turn, consists of the object number, the generation number, and the keyword obj. For example:

```
9 2 obj % object ID line
39
endobj
```

This indirect object encapsulates a PDF number object, the integer 39. (It could just as easily encapsulate a string, name, or dictionary. But note that indirect objects cannot hold indirect objects. An indirect object can contain only a native, unnumbered PDF object, or direct object.)

The advantage of declaring objects as indirect objects is that they can be catalogued in the document xref table and reused by any number of pages, dictionaries, etc., in the document. The fact that every indirect object has an entry in the xref table means indirect objects can be accessed very quickly.

To reference an indirect object from an array or dictionary, one simply uses a three-component indirect reference consisting of the object number, its generation number, and the letter R. For example, consider the following rewrite of our small text stream from above:

```
2 0 obj
<<
/Length 9 2 R
>>
stream
BT
/F1 12 Tf
72 712 Td (A short text stream.) Tj
ET
endstream
endobj
9 2 obj
39
endobj
```

Here, we have two indirect objects in a row, object 2 (a text stream) and object 9 (an integer). The /Length field of the stream's attribute dictionary now has the value 9 2 R. This is a reference to object 9, which is an integer containing the length of the text stream (i.e., 39 bytes). The text length can now be obtained by lookup, in other words. Think what this means: It means the authoring application can create a text-stream object on the fly, without knowing how long it's going to be - then write the length after the stream, in a separate object, when the stream's length is known. Features like this make it possible for applications that write PDF files to create complex documents in a single pass - an important capability.

The Catalog Tree

The catalog is a dictionary comprising the root node of a PDF document. The catalog contains entries, typically, for /Pages (the root of the document's page tree), /Outlines (the root of the outline tree, if any), and information on how the document should appear when first opened. For example:

```
1 0 obj
<<
/Type /Catalog
/Pages 2 0 R
/Outlines 3 0 R
/PageMode /UseOutlines
>>
endobj
```

The only required member of the catalog is a reference to the document's pages tree, but if the document uses outlines, threads, page-label dictionaries (to designate numbering methods and/or map visible page numbers to logical pages), or private structure trees, references to the roots of these objects will occur in the catalog as well.

The Pages Tree

The pages of a document are accessed through a structure known as the pages tree. The nodes of the pages tree are dictionaries containing references to all of the imageable pages in the document (or to other nodes). Acrobat Distiller constructs balanced trees to hold page info, so as to minimize lookup times. But it isn't necessary to implement the pages tree as a balanced tree, or even as a tree at all: it can be a single node that references all of the page objects in the file.

The leaves in a pages tree are the page objects themselves. The nodes are dictionaries with four required entries: a /Type entry (the value of which is always /Pages); a /Count (giving the number of pages under this node in the tree, including subnodes below this node); a /Kids entry (which is an array containing the object numbers of all available pages); and a /Parent entry (a backpointer to the node's immediate ancestor). The top-level node has no parent.

The following example shows how a pages tree node is formatted:

```
2 0 obj
<<
/Type /Pages
/Kids [6 0 R 10 0 R 18 0 R]
/Count 3
>>
endobj
```

In this case, the node points to three leaves: objects 6, 10, and 18. All leaves (and the node itself) are indirect objects, of course, so they can be referenced by other objects.

Page objects are dictionaries with a type entry of /Page that describe the various objects and attributes that make up a viewable page. Typically, additional entries include /Parent, /MediaBox, /Resources, and /Content, although there can be many others (see Section 6.3.1 of the PDF specification). The page's content will usually be a stream or an array of streams, pointed to by the /Content tag.

For example:

```
8 0 obj
<<
/Type /Page
/Parent 4 0 R
/MediaBox [0 0 612 792]
/Resources <<
        /Font << /F3 7 0 R /F5 9 0 R /F7 11 0 R >>
        /ProcSet [/PDF] >>
/Thumb 12 0 R
/Contents 14 0 R
/Annots [23 0 R 24 0 R]
>>
endobj
```

This page's contents are in object 14. The page's MediaBox (or native page size) is 8.5 by 11 inches; in user-space coords, 612 by 792. There is a thumbnail sketch of the page at object 12; annotations are available in objects 23 and 24. For resources, the page uses fonts 3, 5, and 7 and the /PDF ProcSet, which is a set of PostScript procedure definitions that implement the PDF page description operators in PostScript (so the page can be output on a PostScript device).

A Sample PDF File

Listing 1 shows what a small PDF file looks like. The example shown consists of a two-page document in which the first page contains the words "This is 12-point Times. This sentence will appear near the top of page one." The second page of the document contains the text: "This is 24-point Times, appearing at the middle of page two."

Listing 1: TwoPage PDFfile.pdf

```
The following lines are an ASCII dump of a sample PDF file
consisting of two pages, each page having a small amount of text.
End-of-line characters (0x0D) are not shown.
%PDF-1.1
%íì¦"
1 0 obj
<<
/CreationDate (D:19990628091919)
/Producer (Acrobat Distiller 3.01 for Power Macintosh)
/Author (kas)
/Title (TwoPage PDFfile.pdf)
/Creator (created with MS Word)
>>
endobj
3 0 obj
<<
/Length 168
>>
stream
BT
/F4 1 Tf
12 0 0 12 50.64 731.52 Tm
0 0 0 rg
BX /GS2 gs EX
0 Tc
0 Tw
[(This is 12-point )10(T)41(imes. )18(This sentence will appear near
the top of page one.)]TJ
ET
endstream
endobj
4 0 obj
<<
/ProcSet [/PDF /Text ]
/Font <<
/F4 5 0 R
>>
/ExtGState <<
/GS2 6 0 R
>>
>>
endobj
9 0 obj
<<
/Length 163
>>
stream
BT
/F4 1 Tf
24 0 0 24 47.28 390.48 Tm
0 0 0 rg
BX /GS1 gs EX
0 Tc
0 Tw
[(This is 24-point )20(T)36(imes, appearing at the middle of)]TJ
0 -1.2 TD
(page two.)Tj
```

```
ET
endstream
endobj
10 0 obj
<<
/ProcSet [/PDF /Text ]
/Font <<
/F4 5 0 R
>>
/ExtGState <<
/GS1 11 0 R
>>
>>
endobj
11 0 obj
<<
/Type /ExtGState
/SA false
/OP false
/HT /Default
>>
endobj
6 0 obj
<<
/Type /ExtGState
/SA false
/OP true
/HT /Default
>>
endobj
5 0 obj
<<
/Type /Font
/Subtype /Type1
/Name /F4
/BaseFont /Times-Roman
>>
endobj
2 0 obj
<<
/Type /Page
/Parent 7 0 R
/Resources 4 0 R
/Contents 3 0 R
>>
endobj
8 0 obj
<<
/Type /Page
/Parent 7 0 R
/Resources 10 0 R
/Contents 9 0 R
>>
endobj
7 0 obj
<<
/Type /Pages
/Kids [2 0 R 8 0 R]
/Count 2
/MediaBox [0 0 612 792]
>>
```

```
endobj
12 0 obj
<<
/Type /Catalog
/Pages 7 0 R
>>
endobj
xref
0 13
0000000000 65535 f
0000000016 00000 n
0000002390 00000 n
0000000200 00000 n
0000000419 00000 n
0000001088 00000 n
0000001018 00000 n
0000002551 00000 n
0000002470 00000 n
0000000513 00000 n
0000000727 00000 n
0000000946 00000 n
0000001017 00000 n
trailer
<<
/Size 13
/Root 12 0 R
/Info 1 0 R
>>
startxref
1055
%%EOF
```

The very first line of the file shows that the file is backwards-compatible with version 1.1 of the PDF spec. The second line contains characters in the range 128-255, to signal that the file is binary in nature, although this example contains just uncompressed text.

The trailer shows that the file contains 13 numbered objects, of which the root object is number 12. (The beginning of the xref table occurs at a byte offset of 1,055 from the start of the file.) If we look at the root object, it has a /Type key of /Catalog and contains a reference to the document's /Pages tree root object - object 7. Object 7, in turn, is a /Pages node, with a count of 2 pages beneath it; the /Kids array points to two page objects (objects 2 and 8). There is also a /MediaBox entry giving the native page size for the document, in user space coords (72 units to the inch). The page size is 8.5 by 11 inches.

Object 2 (the first page object) refers us to object 3 for /Contents and object 4 for /Resources. Object 4 shows our resources as consisting of two /ProcSets, a typeface (Font 4, in object 5), and an extended graphics state object in object 6. (We didn't talk about this kind of object. The /ExtGState is a special kind of dictionary that lets you specify certain types of printing behaviors, such as underprint and overprint modes, miter limit, etc. See Chapter 7 of the PDF spec.)

Object 3, which contains the contents of page one of our document, is worth commenting on since it shows how text streams are used in PDF. The object looks like:

```
3 0 obj
<<
/Length 168
>>
stream
BT
/F4 1 Tf
12 0 0 12 50.64 731.52 Tm
```

```
0 0 0 rg
BX /GS2 gs EX
0 Tc
0 Tw
[(This is 12-point )10(T)41(imes. )
        18(This sentence will appear near
        the top of page one.)]TJ
ET
endstream
endobj
```

The stream object (which is 168 bytes long) is bracketed by BT and ET operators, for Begin Text and End Text. The Tf command selects our font and its size in user-space units, which is given as 1. "But aren't we using 12-point type?" you may be wondering. Yes, we are. That's specified in the next line, ending in Tm (which is the set-text-matrix operator). For space reasons, we won't say much about coordinate system transformations and matrices here, but if you're familiar with the use of matrices in PostScript, the same rules apply in PDF. A transform matrix is given by an array of six numbers, the first and fourth of which determine scaling in x and y, respectively. We see in our text matrix, the scaling factor is 12. That means we will use 12-point type. The last two numbers in the matrix (50.64 and 731.52) specify a translation, in user-space units. The effect of the translation is to put our text approximately 10.1 inches high on the page, with a left margin of 0.7 inch.

The line ending with rg sets our ink color to an RGB value of 0 0 0, or black. The BX operator says that we are beginning a section that allows undefined operators. In this section, we apply the gs operator (which sets parameters in the extended graphics state), using /GS2 as our EGS specifications. The EX operator ends the section allowing undefined operators. In essence, we're saying "Any reading application that understands what's in this special section can execute the instructions contained there, but if you don't understand the instructions, just go on." The reason this section has to be handled this way is that extended graphics state instructions often contain device-dependent instructions. The lack of generality means we should bracket those instructions with BX/EX.

The Tc and Tw operators are for setting character spacing and word spacing, respectively.

Finally, we come to the text that will be displayed on our page. Oddly enough, it's specified in an array of text snippets interspersed with integers, such as:

```
(This is 12-point )10(T)41(imes. )
```

The number 10 represents a kerning value, in thousandths of an em. (An em is a typographical unit of measurement equal to the size of the font.) This number is subtracted from the 'x' coordinate of the letter(s) that follow, displacing the text to the left. The capital 'T' is displaced 10 units to the left, while "imes. " is displaced 41 units. The TJ at the end of the array is the operator for "show text, allowing individual character spacing."

Finally, ET closes off the text block, and endstream closes off the stream.

Some of the more commonly used page-marking operators in PDF are shown in Table 1.

Tools for Further Exploration

Obviously, in an article of this size it is not possible to summarize the full specification for PDF 1.3. We've barely been able to hit the high points. Hopefully, in a future article, we can concentrate more heavily on the PDF imaging model, which is the archetype for Apple's coming QuickDraw replacement, Quartz.

b	closepath, fill,and stroke path.
B	fill and stroke path.
b*	closepath, eofill,and stroke path.
B*	eofill and stroke path.

BI	begin image.
BMC	begin marked content.
BT	begin text object.
BX	begin section allowing undefined operators.
c	curveto.
cm	concat. Concatenates the matrix to the current transform.
cs	setcolorspace for fill.
CS	setcolorspace for stroke.
d	setdash.
Do	execute the named XObject.
DP	mark a place in the content stream, with a dictionary.
EI	end image.
EMC	end marked content.
ET	end text object.
EX	end section that allows undefined operators.
f	fill path.
f*	eofill Even/odd fill path.
g	setgray (fill).
G	setgray (stroke).
gs	set parameters in the extended graphics state.
h	closepath.
i	setflat.
ID	begin image data.
j	setlinejoin.
J	setlinecap.
k	setcmykcolor (fill).
K	setcmykcolor (stroke).
l	lineto.
m	moveto.
M	setmiterlimit.
n	end path without fill or stroke.
q	save graphics state.
Q	restore graphics state.
re	rectangle.
rg	setrgbcolor (fill).
RG	setrgbcolor (stroke).
s	closepath and stroke path.
S	stroke path.
sc	setcolor (fill).
SC	setcolor (stroke).
sh	shfill (shaded fill).
Tc	set character spacing.
Td	move text current point.
TD	move text current point and set leading.
Tf	set font name and size.
Tj	show text.
TJ	show text, allowing individual character positioning.
TL	set leading.
Tm	set text matrix.
Tr	set text rendering mode.
Ts	set super/subscripting text rise.
Tw	set word spacing.
Tz	set horizontal scaling.
T*	move to start of next line.
v	curveto.
w	setlinewidth.
W	clip.
y	curveto.

TABLE 1: PDF Page Markup Operators
(Note: Equivalent PostScript operators are in boldface.)

In the meantime, you can learn a great deal more about Adobe's Portable Document Format simply by opening .pdf files with a text editor and studying their contents. To create specimen .pdf files of your own, simply output PostScript to disk (using Microsoft Word, Adobe InDesign, PageMaker 6.5, or any other program that can output PostScript files) and run your .ps file(s) through Adobe Distiller, which is a PostScript-to-PDF converter program (part of the Acrobat suite). The advantage of using it is that with Distiller, you can exercise fine control over various PDF settings involving compression, output resolution, font embedding, and so forth. (Turning off all compression can be handy when you want to be able to read text streams in your test files.)

For the ultimate in PDF "learning tools," you can join the Adobe Developer Network ($195/yr) and request the CD-ROM containing all Acrobat development tools and docfiles. This is a huge collection of online resources (including the voluminous PDF 1.3 specification itself, plus SDKs for Acrobat plug-in development) which you won't want to pass up if you're serious about PDF. Details are at <http://partners.adobe.com/asn/developer>.

In the meantime, start paying attention to PDF. It's the Next Big Thing where prepress workflow, web publishing, and document interchange are concerned - and the PDF graphics model is coming to a Mac near you, sooner than you think.

Kas Thomas (tbo@earthlink.net) has been programming in C and assembly on the Mac since before Desert Storm and has a somewhat dusty shareware plug-ins page at http://users.aol.com/Callisto3D. This is his tenth article for MacTech.