libpdf: a Libary for PDF File Generation

Joseph J. Pfeiffer, Jr.
Department of Computer Science
New Mexico State University
pfeiffer@cs.nmsu.edu
NMSU-CS-2002-012

7th November 2002

Abstract

libpdf is a library for the creation of PDF documents from within programs. The library provides support for text, graphics, standard and embedded fonts, images, and interactive form fields. Operators provided by libpdf are at a very similar level of abstraction to those provided by the PDF document model itself. The library is designed to be easily extended.

It is most suitable for the generation of fairly small documents, as (1) it does not provide support for Linearized PDF and (2) it doesn't support the document structuring features of PDF (for instance, it uses a single resource dictionary for an entire document rather than a hierarchy of resource dictionaries). It also does not support several PDF features such as encryption, modifiable documents, or thumbnails. Many of these features could be added easily, however.

Contents

1 Introduction **3**
- 1.1 Capabilities and Limitations . . . 3
- 1.2 Organization of This Document . . . 4

2 User's Manual **5**
- 2.1 An Example: "Hello World" in PDF . . . 5
- 2.2 Data Types and Naming Conventions . . . 6
- 2.3 #include files, libraries, and configuration . . . 6
- 2.4 Document Hierarchy . . . 7
- 2.5 Graphics State . . . 7
- 2.6 Path Operators . . . 10
  - 2.6.1 Path Construction . . . 10
  - 2.6.2 Path Painting . . . 11
  - 2.6.3 Path Clipping . . . 12
  - 2.6.4 Example: Drawing a Path . . . 12
- 2.7 Text Objects . . . 12
  - 2.7.1 Text State Operators (PDF Reference Manual Section 5.2) . . . 13
  - 2.7.2 Text Positioning Operators . . . 13
  - 2.7.3 Text Showing Operators . . . 14
  - 2.7.4 Example: Text . . . 14
- 2.8 Fonts . . . 15
- 2.9 Images . . . 15
  - 2.9.1 Example: Graphics State and Images . . . 16
- 2.10 Forms . . . 16
  - 2.10.1 Example: Forms . . . 17
- 2.11 Output . . . 17

3 Programmer's Manual **18**
- 3.1 Data Types in PDF and libpdf . . . 18
  - 3.1.1 PDF Data Types . . . 18
  - 3.1.2 Direct and Indirect Objects . . . 20
  - 3.1.3 libpdf data types . . . 20
    - 3.1.3.1 Low Level and Mid Level libpdf Types . . . 20
    - 3.1.3.2 Atomic Types . . . 21

3.1.3.3 List Types 22
3.2 Source and Header Files 23
3.3 PDF Object Manipulation in libpdf 23
3.3.1 Constructors 23
3.3.2 Destructors 24
3.3.3 Accessors 24
3.3.4 Mutators 24
3.3.5 Output 25
3.3.5.1 Top-Level Output Queue Processing 25
3.3.5.2 Recursive Object Output 25

Chapter 1

Introduction

libpdf has grown out of the need to provide support for the author's work as director of the Southwestern New Mexico Regional Science and Engineering Fair. It was necessray to develop two applications to assist the administration of the fair: first, a program which would permit us to scan a paper form, identify the form's blanks, fill them in on-line, and print the result; the second to create several barcoded pages and stickers for judges to use in reporting results (we hope to release these when ready, as well).

The first of these programs, in particular, requires PDF to be practical. The author uses Linux while the assistant director uses Windows, so a common document standard supporting form fields was needed. HTML is not suitable for this task, as the form has to be the scanned original with fields added; attempting to accomplish this with the rendering tools available in HTML, particularly in the presence of varying resolutions on display devices, did not seem to be practical. PDF's interactive forms, however, are ideal for the purpose. While the barcode generation and scanning could have been done several ways (for instance, generating TeX code and processing the result), directly generating PDF from the fair database seemed the most direct.

With that in mind, the development of a PDF generation library with the capabilities needed for both applications seemed to be a logical step.

The remainder of this document is organized as follows: the rest of this Introduction further explores the motivation for the development of a new PDF generation library instead of pursuing other available options. Following this will be two reference manuals: one for users of this library, and another for programmers who will be extending it. As, at present, the library provides exactly those capabilities required by the author, it is anticipated that these communities are likely to be the same for some time to come, so providing both manuals in a single document seems reasonable.

1.1 Capabilities and Limitations

libpdf is capable of creating small PDF documents using standard and some TrueType fonts, drawings, grayscale images, and forms with text fields.

It does not support as rich a page hierarchy as permitted by the PDF document

model; rather than a tree of pages with a hierarchy of resource definitions, it uses a single resource dictionary for the entire document and has only a single level of pages. It does not support graphics state dictionaries. It does not support page thumbnails, outlines, or annotations other than text annotations for text form fields. Due to PDF's use of a small number of data structures, and libpdf's echoing of this internally, many of these features could be added with minimal effort using the existing code as a guide. The most troublesome features to add would probably be those involving extensions to the file format itself: incremental additions to PDF files, and linearized PDF.

The code documented here should be considered as a rough draft implementation of the API. It is functional, and is being used by the author. There are undoubtedly many parts of the libary which should be rewritten. In addition, several items needed by the author have been implemented while closely related items have not; for instance, adding color images would be very straightforward.

One final limitation is that the library is designed for final output of documents, not for their manipulation. Consequently, it is in some respects a "write only" library: in general it is only possible to create data structures with the library, and not to manipulate them once they have been created. Only those accessor and mutator functions necessary for document creation have been implemented; in many cases, extending the library by adding these functions would involve linear searches, so a redesign of many of the data structures would be required.

1.2 Organization of This Document

Due to the preliminary nature of this implementation, it seems quite likely that early users of the library will also be contributing code to it. For that reason, it seems most sensible to combine the user's manual with the programmer's manual in a single document. Consequently, Chapter 2 will be the User's Manual, while Chapter 3 will be the Programmer's Manual.

Chapter 2

User's Manual

This section describes the use of libpdf. An attempt will be made to avoid a prior familiarity with PDF's document model; if necessary, the PDF reference manual[1] can be used to supplement this description.

2.1 An Example: "Hello World" in PDF

The following program makes use of libpdf to create a PDF file which displays the string Hello World on a page. It demonstrates a simple case of PDF's document model: a document is a tree rooted at a node of type `PDF_Document`, created with a call to `PDF_Document_new()`. This function takes the width and height of a page (in 1/72" points) as its parameters. Any fonts to be used in the document must also be allocated; for the 14 standard fonts required to be present in any compliant PDF reader, `PDF_Font_new_standard()`is used to do this. Pages are created and appended to the document with `PDF_Page_new()` (this document only has a single page); a page can have one or more "content streams," defining the contents of the page; these are created with `PDF_PageContent_new()`. Content streams may contain text, graphics, or a mixture; this page's lone content stream contains one text object. The text object is created with `PDF_Text_new()`, a font is selected for it with `PDF_Text_set_font()`, it is positioned with `PDF_Text_newline_xy()`, and the actual text is written with `PDF_Text_string()`.

Once the document is defined, it is written to `stdout` with `PDF_Document_write()`.

```
#include <stdio.h>
#include <libpdf/pdf.h>


main() {
    PDF_Document doc;
    PDF_Page page;
    PDF_Font font;
    PDF_Graphics content;
```

```
    PDF_Text text1;
    char outbuf[65536];
    int numwritten;
    /* create a document, paper size is
       US letter standard 8.5" by 11" */
    doc = PDF_Document_new(612.0, 792.0);
    /* standard fonts to be used in document */
    font = PDF_Font_new_standard(doc, PDF_FONT_HELVETICA);
    /* create a page and add it to the document */
    page = PDF_Page_new(doc);
    /* create a content stream and attach it to the page */
    content = PDF_PageContent_new(page);
    /* create a text object, determine its font, set its location, and define the text */
    text = PDF_Text_new(content);
    PDF_Text_set_font(text, font, 12.0);
    PDF_Text_newline_xy(text, 100.0, 500.0);
    PDF_Text_string(text, "Hello World");
    /* put document to standard out */
    numwritten = PDF_Document_write(doc, stdout);
    return 0;
}
```

This example appears in the file `../tests/text.c`. The remainder of this user's manual discusses the definition and use of libpdf.

2.2 Data Types and Naming Conventions

libpdf's user-visible data types are all opaque types; in general, they take the form `PDF_`*`Objtype`*, where *`Objtype`* is a type such as `Document`, `Page`, etc. All libpdf types begin with `PDF_`, and capitalize the first letter of the *`Objtype`*. We will generically refer to objects of all these types as PDF objects.

Functions using or affecting PDF objects are named according to the object type affected by the function as `PDF_`*`Objtype_operation`*, where `PDF` is used for all the functions in order to define a namespace for the library, *`Objtype`* is the object type, and *`operation`* is the operation to be performed. For instance, the function to create a text object is `PDF_Text_new()`.

A number of enumerated types are also used; these all have type names of the same form as other data types, while their members all have names of the form `PDF_`*`OBJTYPE_NAME`*.

Finally, two data types were created from C types with `typedef`: `PDF_CBool` is a boolean type, with members `PDF_CBOOL_FALSE=0=0` and `PDF_CBOOL_TRUE=1=1`, and `PDF_CString` is a `char*`.

2.3 #include files, libraries, and configuration

To use libpdf, your program must

```
#include <libpd/pdf.h>
```

libpdf makes use of the pkg-config[4] tool to configure paths and libraries, under the package name libpdf.

Libpdf requires additional libraries to function. A version of FreeType2[5] new enough to contain the functions `FT_Get_First_Char()` and `FT_Get_Next_Char()`[1] is needed, as are netpbm[3] and zlib[2].

2.4 Document Hierarchy

A PDF document is an object created with

```
PDF_Document PDF_Document_new(float width, float height);
```

This function will create a document dictionary and an empty page hierarchy, and will define the document's media size as *width*×*height*. The units are points (1/72 of an inch); in the example, US standard letter size (8.5 inches by 11 inches, or 612 by 792 points) is used.

Any number of pages can be created and attached to the document with

```
PDF_Page PDF_Page_new(PDF_Document doc);
```

In the case of a document with more than one page, they will be displayed in the same order that they were created.

The actual content of the pages (text and graphics) are represented in PDF by *content streams* containing *graphics states*, any number of which may be created and attached to the pages with

```
PDF_Graphics PDF_PageContent_new(PDF_Page page);
```

Content streams are rendered by viewers (such as Adobe Acrobat) in the order in which they are attached to pages; if multiple streams cover the same area of a page, the later ones will overwrite the earlier.

2.5 Graphics State

PDF uses the term *graphics state* to describe the current values of a variety of parameters relating to the display of graphics and text on a page. This includes things like the current color, the current line width, a general transformation matrix affecting the location and size of objects, and more. A default graphics state is established with each content stream on a page; it isn't necessary to explicitly create a new graphics state if drawing is desired.

PDF maintains a stack of graphic states; it is possible to enter a new graphics state, modify it, draw graphics or text, and then restore the previous graphics state. This capability is encapsulated in libpdf through the following function.

[1] The author is unclear as to which version of the freetype2 library added these functions. Consequently, the build scripts simply check for any version of the library, and for the existence of these functions.

- `PDF_Graphics PDF_Graphics_new(PDF_Graphics old);`

  Create a new graphics state, and push it onto the specified old state. Drawing operations (including text, path, graphics state, and other) operations can be appended to either the new or the old graphics state. Operators are automatically generated to push the new state onto the state stack, and to pop it off.

PDF uses a 3×3 transformation matrix to translate, rotate, or scale objects for display. libpdf provides four functions to modify the matrix; in all cases, the transformation specified is appended to the pre-existing matrix. The default transformation matrix is an identity matrix.

- `PDF_CBool PDF_Graphics_concat_matrix(PDF_Graphics obj, float a, float b, float c, float d, float e, float f);`

  Concatenate an arbitrary matrix to the existing matrix. The values of the entries in the new matrix are $\begin{bmatrix} \texttt{a} & \texttt{b} & 0 \\ \texttt{c} & \texttt{d} & 0 \\ \texttt{e} & \texttt{f} & 1 \end{bmatrix}$

The following three functions are all shortcuts intended to make matrix manipulation easier.

- `PDF_CBool PDF_Graphics_translate(PDF_Graphics obj, float x, float y);`

  Translate succeeding objects by (`x`, `y`).

- `PDF_CBool PDF_Graphics_rotate(PDF_Graphics obj, float theta);`

  Rotate succeeding objects by `theta`.

- `PDF_CBool PDF_Graphics_scale(PDF_Graphics obj, float x, float y);`

  Scale succeeding objects by `x` horizontally and `y` vertically.

The next group of functions affect rendering of succeeding lines and other graphics.

- `PDF_CBool PDF_Graphics_line_width(PDF_Graphics obj, float width);`

  Set the width of succeeding lines

- `PDF_CBool PDF_Graphics_line_cap(PDF_Graphics obj, PDF_CapStyle style);`

  Set the style for line ends. The possible cap styles are

  - `PDF_CAPSTYLE_BUTT`

    End the line even with the line end.

  - `PDF_CAPSTYLE_ROUND`

    Put a rounded cap on the line end.

  - `PDF_CAPSTYLE_PROJECTING`

    Put a projecting square cap on the line end.

More information regarding the definition of the cap styles can be found in the PDF Reference Manual, Table 4.4.

- `PDF_CBool PDF_Graphics_line_join(PDF_Graphics obj, PDF_JoinStyle style);`

  Set the style for polygon vertices. The possible join styles are

  - `PDF_JOIN_MITER`
    Use "pointed" vertices.
  - `PDF_JOIN_ROUND`
    Use "rounded" vertices.
  - `PDF_JOIN_BEVEL`
    Use "flattened" vertices.

  More information regarding the definition of the join styles can be found in the PDF Reference Manual, Table 4.5.

- `PDF_CBool PDF_Graphics_mitre_limit(PDF_Graphics obj, float limit);`

  Limit the extent of mitred vertices. See the PDF Reference Manual, Figure 4.7.

- `PDF_CBool PDF_Graphics_dash(PDF_Graphics stream, float dashArray[], float phase);`

  Define a dashed line. dashArray is a null-terminated array defining the lengths of the black and white segments; phase defines the point in the sequence where line-drawing begins. More information regarding the definition of the dash array can be found in the PDF Reference Manual, Table 4.6.

- `PDF_CBool PDF_Graphics_intent(PDF_Graphics obj, PDF_ColorIntent intent);`

  Give the PDF rendering program a hint regarding color rendering. The possible hints are

  - `PDF_COLORINTENT_ABSOLUTECOLORIMETRIC`
    Reproduce colors with regard only to the light source, and not the output media.
  - `PDF_COLORINTENT_RELATIVECOLORIMETRIC`
    Compensate for output media.
  - `PDF_COLORINTENT_SATURATION`
    Emphasize color saturation.
  - `PDF_COLORINTENT_PERCEPTUAL`
    Attempt to render in a "perceptually pleasing" manner.
    More information about rendering hints can be found in the PDF Reference Manual, Table 4.19.

- `PDF_CBool PDF_Graphics_flatness(PDF_Graphics gs, float flatness);`
  - Define the permissible variation from a perfect curve when rendered as a series of line segments. The parameter is in the range 0 to 100; a smaller flatness gives a higher-quality result at the expense of computation time.
- `Bool PDF_Graphics_append(PDF_Graphics gs, PDF_Object image);`

  Append an arbitrary object to a graphics state.

2.6 Path Operators

PDF uses the term *path* to describe objects to be drawn on the page; this includes lines, polygons, splines, and rectangles. libpdf provides four functions for path construction, one for path display, and one to define a path as a clipping region for other graphics. It should be noted that it isn't sufficient to define a path; it is necessary to define it and then paint it.

2.6.1 Path Construction

PDF begins a graphic state with an empty path, and adds graphic elements to that path. The term *current point* is used to describe the starting point of a new element to be added to the path.

- `PDF_Path PDF_Path_new(PDF_Graphics obj);`

  Create a new, empty path. This function is only needed if more than one path is to exist in a single graphic object; the example in Section 2.6.4 does not use it.
- `PDF_CBool PDF_Path_move(PDF_Path obj, float x, float y);`

  Move the current point to (`x`, `y`) without drawing.
- `PDF_CBool PDF_Path_line(PDF_Path obj, float x, float y);`

  Draw a line from the current point to `(x, y)`, and set `(x, y)` as the new current point.
- `PDF_CBool PDF_Path_curve(PDF_Path obj, float x1, float y1, float x2, float y2, float x3, float y3, PDF_ControlPoints controlpoints);`

  Start or append to a cubic Bezier spline. A Bezier spline segment is defined by four control points; the first control point (P_0) is always the current point, while up to three of the remaining control points are specified by the parameters of this function under the control of the controlpoints parameter:
  - `PDF_CONTROL_EXPLICIT`: the three points specified in the function are used as the control points P_1, P_2, and P_3.
  - `PDF_CONTROL_FIRST`: `x1` and `y1` are disregarded, and the current point is also used as control point P_1.

– `PDF_CONTROL_LAST`: `x2` and `y2` are disregarded, and (`x3`, `y3`) is also used as control point P_2.

This function actually generates any of three different PDF operators, as appropriate. The PDF reference manual describes the effect of the three operators in section 4.4.1.

In all cases, the current point is moved to (`x3`, `y3`).

- `PDF_CBool PDF_Path_rectangle(PDF_Path obj, float x, float y, float width, float height);`

  Create a rectangle at `(x, y)` with the specified width and height.

2.6.2 Path Painting

- `PDF_CBool PDF_Path_paint(PDF_Path obj, PDF_CBool closepath, PDF_CBool stroke, PDF_CBool fill, PDF_FillType filltype);`

  This function paints the path that was created using the path construction operators in Section 2.6.1, under the control of the various parameters:

  – `PDF_CBool closepath`

  Draw a line from the current point to the first point in the path. It is highly recommended (in the PDF reference manual) that any polygon be explicitly closed rather than just having a line drawn from the last point back to the first – PDF uses separate graphics state entries to define the appearance of line ends and polygon vertices; if the polygon is closed it will have a common appearance with the other vertices rather than being rendered as two line ends which happen to lie on the same point.

  – `PDF_CBool stroke`

  Draw the lines and curves making up the path. If this is used in conjunction with the `fill` parameter, the result will be a filled polygon with its edges drawn.

  – `PDF_CBool fill`

  Fill the polygon. There are two algorithms available to determine the interior of the polygon for purposes of filling; these are specified by the `filltype` parameter, which can be either `PDF_FILL_EVENODD` or `PDF_FILL_WINDING`. `filltype` is ignored if `fill` is `PDF_FALSE`. The filltype does not matter for simple paths; its implications for complex paths are described in Section 4.4.2 of the PDF Reference Manual.

  This function actually generates any of the ten different path-terminating operators described in the PDF Reference Manual's section 4.4.2. These operators differ according to how the path is to be rendered; they seemed to map into a programming language as parameters in a general path-termination function, instead.

2.6.3 Path Clipping

A clipping path defines a "window" on the page through which later objects (including paths, text, and images) are viewed: only those parts of the later objects which occupy filled areas of the clipping path are actually displayed. A path defined using the operators in Section 2.6.1 can be made a using the following function.

- `PDF_CBool PDF_Path_clip(PDF_Path obj, PDF_FillType fill);`

  Use the path as a clipping path, defining its filled region under control of `fill` as with `PDF_Path_paint()` (section 2.6.2).

2.6.4 Example: Drawing a Path

The following example is taken from the file `../tests/lines.c`

```
flower = PDF_PageContent_new(page);
PDF_Path_move(flower, XC + RAD, YC);
angle = 0;
do {
    angle += ANGLE;
    if (angle > 2.0*M_PI - FUZZ)
        angle -= 2.0*M_PI;
    PDF_Path_line(flower, XC + RAD*cos(angle), YC + RAD*sin(angle));
} while (angle > FUZZ);
PDF_Path_paint(flower, PDF_CBOOL_TRUE, PDF_CBOOL_TRUE, PDF_CBOOL_FALSE, PDF_FILL_EVENODD);
```

This code creates a new graphic object and attaches it to the content stream; moves to a starting location; draws a series of lines (the result resembles a spyrograph, though with straight lines); and finally draws the result, closing but not filling the polygon.

2.7 Text Objects

A text object can be created and attached to a graphics state with

```
PDF_Text PDF_Text_new(PDF_Graphics graphic);
```

Once a text object has been defined, a number of functions are available for describing the text to be rendered on the page. These functions are divided into text state operators, text positioning operators, and text showing operators; the functions provided by libpdf correspond exactly to the operators available in PDF. In general, these operators are self-explanatory; where further information is required the user is referred to the PDF reference manual.

Many of the operators described in the following subsections use operands in so-called text space units. The precise meaning of a text space unit is complex (and described in PDF reference manual section 5.3.3); the default text space unit is one point (1/72 inch).

All of the operators in the following sections append the new operators to the text object (passed as `obj`); rather than returning a new object (as the functions in the previous sections did), they return a `PDF_CBoo`lean success or failure code `PDF_CBOOL_TRUE` or `PDF_CBOOL_FALSE` according to the success or failure of the operation.

2.7.1 Text State Operators (PDF Reference Manual Section 5.2)

- `PDF_CBool PDF_Text_set_charSpace(PDF_Text obj, float space);`

  Set the inter-character spacing, in text space units. The default value is 0.

- `PDF_CBool PDF_Text_set_wordSpace(PDF_Text obj, float space);`

  Set the inter-word spacing, in text space units. The default value is 0.

- `PDF_CBool PDF_Text_set_scale(PDF_Text obj, float scale);`

  Set the horizontal scale of characters as a percent of their standard width as given in the current font. A horizontal scale of 50 would be half-width; 200 would be double-width, and so forth. The default value is 100.

- `PDF_CBool PDF_Text_set_leading(PDF_Text obj, float leading);`

  Set the text leading, in text space units. This refers to the spacing between lines of text, when a newline is used to advance from one line to the next. The default value is 0.

- `PDF_CBool PDF_Text_set_font(PDF_Text obj, PDF_Font font, float size);`

  Set the current font and text size, in text space units. Font definitions will be covered in Section 2.8.

- `PDF_CBool PDF_Text_set_render(PDF_Text obj, PDF_Text_Render_Mode rendermode);`

  Set the text rendering mode. Available modes are `PDF_TEXT_FILL`, `PDF_TEXT_STROKE`, `PDF_TEXT_FILLSTROKE`, `PDF_TEXT_INVISIBLE`, `PDF_TEXT_FILLCLIP`, `PDF_TEXT_STROKECLIP`, `PDF_TEXT_FILLSTROKECLIP`, and `PDF_TEXT_CLIP`. The default value, which will simply render text as would be expected, is `PDF_TEXT_FILL`. The remainder are described in the PDF Reference Manual, Section 5.2.5.

- `PDF_CBool PDF_Text_set_rise(PDF_Text obj, float rise);`

  Set the text's baseline above (positive values of `rise`) or below (negative values of `rise`) its default location; in other words, create superscripts or subscripts. `rise` is specified in text space units.

2.7.2 Text Positioning Operators

There are four operators which determine where text will appear on a page:

- `PDF_CBool PDF_Text_newline_xy(PDF_Text obj, float xoff, float yoff);`

  Start a new line of text, at a location (`xoff`, `yoff`) from the start of the previous line. `xoff` and `yoff` are in text space units.

- `PDF_CBool PDF_Text_newline_leading(PDF_Text obj, float xoff, float yoff);`

  Start a new line of text, at a location (`xoff`, `yoff`) from the start of the previous line, and set the leading to `-yoff`.

- `PDF_CBool PDF_Text_set_matrix(PDF_Text obj, float a, float b, float c, float d, float e, float f);`

  See PDF Reference Manual section 5.3.3.

- `PDF_CBool PDF_Text_newline(PDF_Text obj);`

  Start a new line of text at an x offset of 0 and a y offset determined by the current text leading parameter (see the description of `PDF_Text_set_leading()` in Section 2.7.1).

2.7.3 Text Showing Operators

Three operators are used for text display.

- `PDF_CBool PDF_Text_string(PDF_Text obj, PDF_String string);`

  Display a text string in the current location, using the current font, size, and render mode.

- `PDF_CBool PDF_Text_newline_string(PDF_Text obj, PDF_String string);`

  Display a line of text, at a location defined by an x offset of 0 and a y offset given by the current text leading.

- `PDF_CBool PDF_Text_space_string(PDF_Text obj, float w, float c, PDF_String string);`

  Display a text string using word spacing `w` and character spacing `c`.

- `PDF_CBool PDF_Text_array(PDF_Text obj, PDF_Array arr);`

2.7.4 Example: Text

An example of displaying text at a fixed location can be extracted from the file `../tests/text.c`.

```
text1 = PDF_Text_new(graphic1);
PDF_Text_set_font(text1, font1, 12.0);
PDF_Text_newline_xy(text1, 100.0, 500.0);
PDF_Text_string(text1, "Hello World");
```

These four lines of code

1. Create a new text object, and attaches it to a content stream.
2. Set the text object to use a previously defined font called `font1`, with a size of 12 points.

3. Start a new line of text at location (100, 500) on the page.
4. Display the string "Hello World" using the specified font and size, at the specified location.

2.8 Fonts

libpdf provides support for the fourteen standard fonts supported by all compliant PDF viewers, and for some TrueType fonts.

- `PDF_Font PDF_Font_new_standard(PDF_Text obj, PDF_FontName name);`

  Obtain one of the fourteen standard PDF font names. These are `PDF_FONT_COURIER`, `PDF_FONT_COURIERBOLD`, `PDF_FONT_COURIEROBLIQUE`, `PDF_FONT_COURIERBOLDOBLIQUE`, `PDF_FONT_HELVETICA`, `PDF_FONT_HELVETICABOLD`, `PDF_FONT_HELVETICAOBLIQUE`, `PDF_FONT_HELVETICABOLDOBLIQUE`, `PDF_FONT_TIMESROMAN`, `PDF_FONT_TIMESBOLD`, `PDF_FONT_TIMESITALIC`, `PDF_FONT_TIMESBOLDITALIC`, `PDF_FONT_SYMBOL`, and `PDF_FONT_ZAPFDINGBATS`.

- `PDF_Font PDF_Font_new_embed(PDF_Document doc, PDF_CString *fontdata, int size);`

  Load a font from the specified buffer, and embed it in the document. The `size` parameter specifies the length of the `fontdata` buffer. At present, this is limited to TrueType fonts. It is necessary to read the font file into a buffer before embedding the font in the document; the intent is that, if the program creating the document has need of the font parameters, it will be able to use the FreeType font library to obtain them without interference from libpdf.

  Note: this only supports fonts encoded using Adobe's StandardEncoding (this is approximately ASCII; see PDF Reference Manual, Appendix D for details), or arbitrary subsets of that encoding. It does not support mapping subsets of 16-bit fonts.

- `PDF_Font PDF_Font_new_embed_file(PDF_Document doc, PDF_String fname);`

  Load a font from the specified file, and embed it in the document. This function simply reads the file and calls `PDF_Font_new_embed()`, so all of that function's capabilities and limitations apply here as well.

2.9 Images

libpdf makes use of the netpbm library[3] for image manipulation. A new greyscale image can be created with

- `PDF_Image PDF_Image_new_pgm(PDF_Document doc, gray** grays, int cols, int rows, int maxval);`

Space for the array to be created is allocated, and the `grays` array copied into it. At present only eight-bit images are supported, but an image with a higher resolution than that will automatically be scaled to fit according to `maxval`; if `maxval` is less than 256 it is copied directly; if it is greater than 255 but less than 65536 it is right-shifted eight bits; if it is greater than 65535 it is right-shifted 24 bits.

The size of the resulting `PDF_Image` is only one point on a side. In order to make practical use of images, it is necessary to set the graphics state (see section 2.5) to position and scale it appropriately.

PDF permits images to appear in multiple places in a document, such as letterheads appearing on each page of a document. In order to support this, `PDF_Image_new_pgm()` does not insert the image when created; rather, the image is inserted into a graphics state later using `PDF_Graphics_append()`.

2.9.1 Example: Graphics State and Images

An example of an image inserted into a document is in `../tests/image.c`

```
imagefile =  fopen("kitten.pgm", "r");
inbuf = pgm_readpgm(imagefile, &cols, &rows, &maxval);
image1 = PDF_Image_new_pgm(doc, inbuf, cols, rows, maxval);
PDF_Graphics_translate(graphic1, (612-cols/2)/2, 792-rows/2);
PDF_Graphics_scale(graphic1, cols/2, rows/2);
PDF_Graphics_append(graphic1, image1);
```

These lines of code

1. Open a PGM file (no, the picture is not my cat!).
2. Use the NetPBM library to read the file contents into a buffer.
3. Create an image from the buffer.
4. Modify the current graphics state to position the image at the top center of the page, scaled to be 1/2 the height and 1/2 the width of the page.
5. Insert the image into a graphic object.

2.10 Forms

Working with forms requires the creation of the form itself, and creation of the form fields at their desired locations. The following functions are used to create a form and text fields.

- `PDF_Form PDF_Form_new(PDF_Document doc, PDF_Font font, float size);`

  Create a form and connect it to the document. All of the form's text fields will use the specified font and font size.

- `PDF_Field PDF_FormFieldText_new(PDF_Form form, PDF_Page page, float x, float y, float wid, float ht);`

  Attach a text field to the form, on the specified page, at the specified location, as a rectangle with the given width and height.

A variety of other field types are also supported by PDF; these have not been implemented in libpdf.

2.10.1 Example: Forms

The following example of attaching a form with two text fields to a document is extracted from `../tests/form.c`

```
/* create a form object and attach it to the document */
form = PDF_Form_new(doc, courier, 12.0);
/* Create the user prompts */
text1 = PDF_Text_new(stream1);
PDF_Text_set_font(text1, helvetica, 12.0);
PDF_Text_newline_xy(text1, 100.0, 150.0);
PDF_Text_string(text1, "First Field:");
PDF_Text_newline_xy(text1, 0, -20.0);
PDF_Text_string(text1, "Second Field:");
```

2.11 Output

The only output function intended for end-users is

- `PDF_Document_write(PDF_Document doc, FILE *ostream);`

  `doc` is a document constructed using the other functions described in this manual, and `ostream` is an output stream.

Chapter 3

Programmer's Manual

libpdf is a new library for the creation of PDF files. Our plan is to create a framework for the entire format, but only implement the code needed for the PDF files we are actually using.

This project will develop a programmer's API for creating Adobe PDF files. The objective is for the programmer to create a data structure which closely models Adobe's document model; it will then be possible to write the document to a file. The programmer will not need to deal with the details of the PDF file format; issues such as the cross-reference area in the format will be generated automatically.

3.1 Data Types in PDF and libpdf

A PDF file is essentially the representation of a data structure in a file. It uses a small number of basic data types to represent a rich set of objects; for instance, a "dictionary" is a basic type, while a "page" is a dictionary with the correct contents to define a page in a document. This structure is mirrored in libpdf. As with PDF, a small number of data types are defined, with obects more closely related to document objects are created by using custom constructors on the basic data types. `typedef`s are used to convey the intent of the document structure to the programmer. So, in libpdf, a `PDF_Queue` is a basic type; a `PDF_Dictionary` is a list which will be output in the correct format to be recognized by a PDF reader as a dictionary, and a `PDF_Page` is a dictionary with the correct contents to be used as a page of the document. This section describes the data types used in PDF and libpdf.

3.1.1 PDF Data Types

PDF defines a relatively small number of data types, which are combined in various ways to create documents (see PDF Reference section 3.2. The reference goes into complete detail regarding valid separators and other necessary syntactic information which is not repeated here). The PDF types can be divided into atomic types (Boolean,

Numeric, and Name objects) and structured types (strings, arrays, dictionaries, and streams).

Atomic types use the representations one would expect: Booleans are represented by `true` and `false` and numbers by integer or floating point numbers (12, 3.5, etc). Names are slightly more complex; a valid name is a slash followed by alphanumeric characters (`/Fred`, `/r1`).

The structured types all use representations which are modelled well as lists.

- A *string* consists of a left parenthesis, a sequence of alphanumeric characters, and a right parenthesis, as in `(The quick brown fox jumped over the lazy dog)`. PDF also has a hexadecimal representation of characters, which is not supported by libpdf.

- An *array* consists of a left bracket, a series of PDF objects separated by whitespace, and a right bracket. An example of an array would be `[/a /b 1 47.3 (a test)]`. The elements of an array can be any PDF type (this array contains two names, two numbers, and a string).

- A *dictionary* consists of a pair of less-than signs, a sequence of ordered pairs of objects, and a pair of greater-than signs. An example dictionary would be

  ```
  <</Type /Font
  /Size 12>>
  ```

  The first element in each pair (the key) is a name; the second (the value) can be any PDF type.

- Finally, a *content stream* has two components: a dictionary describing the content of the stream, followed by the stream contents. The stream contents consist of the word `stream`, the data itself, and the word `endstream`.

  A valid stream might be

  ```
  <</Length 18
  >>
  stream
  Here is some data
  endstream
  ```

  The /Length entry in the stream's dictionary is required for all streams; it gives the number of characters in the stream (not counting the stream and endstream delimiters).

PDF uses dictionaries and content streams with stereotyped contents to represent classes of objects: for instance, a document catalog (the root node of the tree representing a document) is a dictionary with entries of

```
/Type /Catalog
/Pages pageTreeNode
```

(where a pageTreeNode is itself a dictionary), while an embedded TrueType font is a content stream with entries of

```
/Type /Font
/Subtype /TrueType
```

and whose content is the actual font definition.

3.1.2 Direct and Indirect Objects

PDF files also have a notion analogous to a pointer, referred to as an indirect object. An object can be defined using the syntax

```
objno rev obj
object as defined in Section 3.1.1
endobj
```

The object can then be referenced as

```
objno rev R
```

With very few exceptions, any place an object can be used in PDF, an indirect object can be used instead.

3.1.3 libpdf data types

libpdf uses a layered datatype hierarchy which closely mimics the PDF hierarchy: a few low-level types are defined directly; other types are `typedef`ed from these basic types. As an example, a `PDF_Queue` is a very heavily used low-level type permitting the output of formatted lists such as dictionaries and arrays. A `PDF_Dictionary` is `typedef`ed from `PDF_Queue`, and specifies a "`<<`" before, "`\n`" between elements, and "`>>`" after printing the dictionary contents. Functions are defined for inserting and searching for dictionary entries, built on top of the `PDF_Queue` functions. Finally, a `PDF_Document` is a `PDF_Dictionary` with specified dictionary entries.

A user writing a program using libpdf should never make direct use of either `PDF_Queue` or `PDF_Dictionary` functions; all the functionality required to manipulate a `PDF_Document` should be provided in `PDF_Document_`*`action`*`()` functions.

3.1.3.1 Low Level and Mid Level libpdf Types

libpdf's PDF objects are inherited from the opaque `PDF_Object` data type. Internally, `PDF_Object` is `typedef`ed as a pointer to

```
struct PDF_object {
    struct PDF_header header;
};
```

libpdf's other internal data types generally inherit this header, and have additional fields appropriate to the data type. The header is defined as

```
struct PDF_header {
    PDF_ObjType objtype;
    PDF_CBool indirect;
    int objnum;
    int version;
};
```

The fields have the following meaning:

`objtype` is an enumerated type specifying the object type. The possible object types are `PDF_Queue`, `PDF_Dict_Entry`, `PDF_Stream`, `PDF_Literal`, and `PDF_Int`; the value of `objtype` for each of these is the object type name translated to all-caps (as in `PDF_QUEUE` for `PDF_Queue`).

`indirect` specifies whether this is an indirect object.

`objnum` is the object number, as used in Section 3.1.2.

`version` is the object's version number; at present all objects have version number 0.

The remainder of this section will describe data types inherited from `PDF_Object`.

3.1.3.2 Atomic Types

libpdf uses `PDF_Literal` to represent most atomic objects. All of the following functions generate an object of type `PDF_Literal`:

- `PDF_Literal PDF_Literal_new(PDF_CString newval);` Given any null-terminated string, construct a `PDF_Literal` object whose value is that string.
- `PDF_Literal PDF_Bool_new(PDF_CBool newval);` Create a `PDF_Literal` with a value of `true` or `false`.
- `PDF_Literal PDF_Float_new(float newval);` Create a `PDF_Literal` whose value is a floating point number.

A `PDF_Name` is a typedefed `PDF_Literal`. It is the programmer's responsibility to provide the leading "`/`" (as in `/Pages`).

A few objects require counters (for instance, the number of pages in a document). Rather than require the user to maintain this, a second atomic type also exists. A `PDF_Int` maintains its value as an integer (instead of a string, as in `PDF_Literal`), and converts it to a string when it is printed.

3.1.3.3 List Types

The primary list data structure used within libpdf is the PDF queue, a linked list of PDF objects. It is defined as

```
struct PDF_queue {
    struct PDF_header header;
    PDF_CString before;
    PDF_CString between;
    PDF_CString after;
    PDF_Queue_Node first;
    PDF_Queue_Node last;
};
```

This structure is used to represent the contents of arrays, dictionaries, and similar lists. When a queue is created, the `before`, `between`, and `after` strings are defined according to the PDF object being represented by the queue. The following types are defined in terms of `PDF_Queue`:

Type	Before	Between	After
`PDF_Array`	`[`	`\n`	`]`
`PDF_Dictionary`	`<<`	(space)	`>>`
`PDF_Graphics`	`q\n`	(empty)	`Q\n`
`PDF_String`	`(`	(empty)	`)`
`PDF_Text`	`BT\n`	(empty)	`ET\n`

for instance, a `PDF_array` uses a `before` string of "`[`", a `between` string of "`\n`" (a newline), and an `after` string of "`]`".

The `PDF_Queue_Node` structure simply contains pointers to a `struct PDF_object`, and to the next node in the queue.

Dictionary entries require two objects (the key and the value), so they are defined as

```
struct PDF_dict_entry {
    struct PDF_header header;
    PDF_Name key;
    PDF_Object value;
};
```

Content streams are constructed from two queues:

```
struct PDF_stream {
    struct PDF_header header;
    PDF_Dictionary dictionary;
    PDF_Queue content;
};
```

3.2 Source and Header Files

libpdf uses a source and a header file for each type, be it low-, middle-, or high-level. In order to avoid namespace pollution, all header file names take the form `pdf`*`type`*`.h`; source file names take the form *`type`*`.c`. So, for instance, the font manipulation code is in `font.c`, while the header file required for a program to make use of the font code is `pdffont.h`.

3.3 PDF Object Manipulation in libpdf

A guiding principle of libpdf's design is that it should be easy to wrap in other languages, notably object-oriented languages such as C++. With that in mind, the library's functions are divided into well-organized constructors, destructors, accessors, mutators, and output functions; in all reasonable cases, a function's first argument will be the object being manipulated.

3.3.1 Constructors

Constructor names take the form

```
PDF_Type PDF_Type_new(ParentType parent);
```

For example, a page is created and added to a document by calling

```
page = PDF_Page_new(doc);
```

A constructor may require extra parameters, as in

```
PDF_Form PDF_Form_new(PDF_Document doc, PDF_Font font, float size);
```

In some cases there may be several variants of a constructor; in these cases the constructor name is

```
PDF_Type PDF_Type_new_variant(ParentType parent);
```

So, for instance, the three variants for font creation are

```
PDF_Font PDF_Font_new_standard(PDF_Document doc, PDF_FontName name);
PDF_Font PDF_Font_new_embed(PDF_Document doc, PDF_CString fontdata, int fontsize);
PDF_Font PDF_Font_new_embed_file(PDF_Document doc, PDF_CString fname);
```

If an object does not require a parent, that parameter is omitted.

3.3.2 Destructors

The sole destructor is

```
PDF_Object_delete(PDF_Object obj);
```

This destructor calls

```
PDF_Objtype_delete(PDF_Objtype obj);
```

as appropriate for the object being deleted to delete the extra fields which are not part of the header, and then frees the header.

3.3.3 Accessors

The general form of an accessor is

```
type PDF_Objtype_get_field(PDF_Objtype obj);
```

This obtains *field* from *obj*, returning a *type*. As an example,

```
PDF_Dictionary PDF_Stream_get_dictionary(PDF_Stream stream);
```

obtains a stream's dictionary. Only those accessors have been defined which are required for the implementation of libpdf, in accordance with libpdf's intended role as an output library.

3.3.4 Mutators

There are several types of mutator. The simplest takes the form

```
PDF_CBool PDF_Objtype_set_field(PDF_Objtype obj, type value);
```

Which sets `objs`'s field to `value`. Types derived from queues have an append function,

```
PDF_CBool PDF_Objtype_append(PDF_Objtype obj, PDF_Object value);
```

which appends an object to the queue.

Very few mutators have been defined, in accordance with libpdf's intended role as an output library.

3.3.5 Output

The sole output function intended for users is

```
int PDF_Document_write(PDF_Document doc, FILE *ostream);
```

This function writes the document to the output stream `ostream`, returning the number of characters written.

A PDF file consists of a header, a series of objects, a cross reference table, and a trailer. `PDF_Document_write()` performs the following operations:

1. It creates the cross reference table, an output queue to hold indirect objects to be output, and a dictionary for the trailer.
2. It prints the header.
3. It puts the `PDF_Document` in the output queue.
4. Each entry in the output queue is processed and output. This may well result in appending more objects to the queue; these are also processed (in particular, processing the root `PDF_Document` will put other objects in the queue). The processing of objects will be discussed shortly. It should be mentioned that the trailer dictionary mentioned in step 1 is not put in the output queue.
5. The cross reference table is printed.
6. The trailer dictionary is printed.
7. The trailer is printed.

Object output is determined by whether the object is being output from the top-level processing of the output queue or from a recursive call, and whether it is direct or indirect (see Section 3.1.2).

3.3.5.1 Top-Level Output Queue Processing

Only indirect objects are placed in the output queue (the initial `PDF_Document` is an indirect object, and direct objects are always output directly rather than placed in the queue). The object is assigned an object number (if it did not have one previously), and output using the `object...endobj` syntax described in Section 3.1.2).

3.3.5.2 Recursive Object Output

In this case, the object is either output directly, or an indirect reference is output and the object is placed in the output queue if it is not already there. The low-level objects defined in Section 3.1.3 each have an output function, with a name of the form `PDF_`*`type`*`_write()`. This function is used to write the object.

Bibliography

[1] Adobe Systems Incorporated. The pdf reference manual version 1.3. http://www.pdfzone.com/pdfs/PDFSPEC13.PDF, 2000.

[2] Mark Adler Jean-loup Gailly and Greg Roelofs. zlib: A massively spiffy yet delicately unobtrusive compression library (also free, not to mention unencumbered by patents). http://www.gzip.org/zlib/, 2002.

[3] Netpbm - graphics tools and convertors. http://sourceforge.net/projects/netpbm/, 2002.

[4] pkg-config. http://www.freedesktop.org/software/pkgconfig/, 2002.

[5] The FreeType Project. Freetype2. http://www.freetype.org/, 2002.

Index

PDF_Array, 22
PDF_CapStyle, 8
  PDF_CAPSTYLE_BUTT, 8
  PDF_CAPSTYLE_PROJECTING, 8
  PDF_CAPSTYLE_ROUND, 8
PDF_CBool, 6
  PDF_CBOOL_FALSE, 6
  PDF_CBOOL_TRUE, 6
PDF_ColorIntent, 9
  PDF_COLORINTENT _ ABSOLUTECOLORIMETRIC, 9
  PDF_COLORINTENT_ PERCEPTUAL, 9
  PDF_COLORINTENT_ RELATIVECOLORIMETRIC, 9
  PDF_COLORINTENT_ SATURATION, 9
PDF_Control_Points
  PDF_CONTROL_EXPLICIT, 10
  PDF_CONTROL_FIRST, 10
PDF_ControlPoints, 10
  PDF_CONTROL_LAST, 11
PDF_CString, 6
PDF_Dictionary, 22
  PDF_dict_entry, 22
PDF_Document, 7, 15–17
  PDF_Document_new, 7
  PDF_Document_write, 17, 25
PDF_Field, 17
  PDF_FormFieldText_new, 17
PDF_FillType, 11, 12
  PDF_FILL_EVENODD, 11
  PDF_FILL_WINDING, 11
PDF_Font, 13, 15, 16
  PDF_Font_new_embed, 15
  PDF_Font_new_embed_file, 15
  PDF_Font_new_standard, 15
PDF_FontName, 15
  PDF_FONT_COURIER, 15
  PDF_FONT_COURIERBOLD, 15
  PDF_FONT_COURIERBOLDOBLIQUE, 15
  PDF_FONT_COURIEROBLIQUE, 15
  PDF_FONT_HELVETICA, 15
  PDF_FONT_HELVETICABOLD, 15
  PDF_FONT_HELVETICABOLDOBLIQUE, 15
  PDF_FONT_HELVETICAOBLIQUE, 15
  PDF_FONT_SYMBOL, 15
  PDF_FONT_TIMESBOLD, 15
  PDF_FONT_TIMESBOLDITALIC, 15
  PDF_FONT_TIMESITALIC, 15
  PDF_FONT_TIMESROMAN, 15
  PDF_FONT_ZAPFDINGBATS, 15
PDF_Form, 16, 17
  PDF_Form_new, 16
PDF_Graphics, 7–10, 12, 22
  PDF_Graphics, 8
  PDF_Graphics_append, 10, 16
  PDF_Graphics_concat_matrix, 8
  PDF_Graphics_dash, 9
  PDF_Graphics_flatness, 10
  PDF_Graphics_intent, 9
  PDF_Graphics_line_cap, 8
  PDF_Graphics_line_join, 9
  PDF_Graphics_line_width, 8
  PDF_Graphics_mitre_limit, 9
  PDF_Graphics_new, 8
  PDF_Graphics_rotate, 8

PDF_Graphics_scale, 8
PDF_Graphics_translate, 8
PDF_PageContent_new, 7
PDF_header, 21
PDF_Image, 15, 16
PDF_Image_new_pgm, 15, 16
PDF_Int, 21
PDF_JoinStyle, 9
PDF_JOIN_BEVEL, 9
PDF_JOIN_MITER, 9
PDF_JOIN_ROUND, 9
PDF_Literal, 21
PDF_Bool_new, 21
PDF_Float_new, 21
PDF_Literal_new, 21
PDF_Name, 21
PDF_Object, 20
PDF_Object_delete, 24
PDF_object, 20
PDF_Page, 7, 17
PDF_Page_new, 7
PDF_Path, 10–12
PDF_Path_clip, 12
PDF_Path_curve, 10
PDF_Path_line, 10
PDF_Path_move, 10
PDF_Path_new, 10
PDF_Path_paint, 11
PDF_Path_rectangle, 11
PDF_Queue, 20
PDF_queue, 22
PDF_Queue_Node, 22
PDF_stream, 22
PDF_String, 15, 22
PDF_Text, 12–15, 22
PDF_Text
PDF_Text_newline, 14
PDF_Text_array, 14
PDF_Text_new, 12
PDF_Text_newline_leading, 14
PDF_Text_newline_string, 14
PDF_Text_newline_xy, 13
PDF_Text_set_charSpace, 13
PDF_Text_set_font, 13
PDF_Text_set_leading, 13, 14
PDF_Text_set_matrix, 14
PDF_Text_set_render, 13
PDF_Text_set_rise, 13
PDF_Text_set_scale, 13
PDF_Text_set_wordSpace, 13
PDF_Text_space_string, 14
PDF_Text_string, 14
PDF_Text_Render_Mode, 13
PDF_TEXT_CLIP, 13
PDF_TEXT_FILL, 13
PDF_TEXT_FILLCLIP, 13
PDF_TEXT_FILLSTROKE, 13
PDF_TEXT_FILLSTROKECLIP, 13
PDF_TEXT_INVISIBLE, 13
PDF_TEXT_STROKE, 13
PDF_TEXT_STROKECLIP, 13