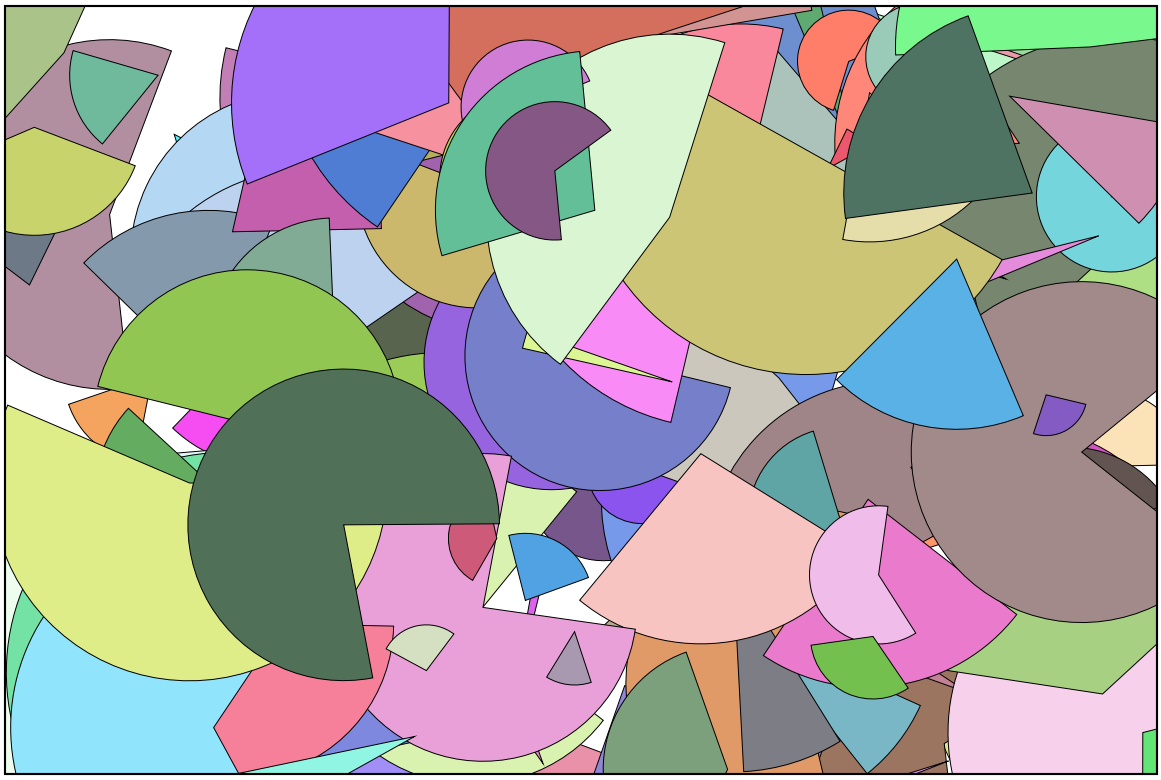
# *ClibPDF Reference Manual*

ANSI C Library for Direct PDF Generation



**FastIO Systems**

http://www.fastio.com/

clibpdf@fastio.com

ClibPDF Library Reference Manual


FastIO Systems
http://www.fastio.com/
clibpdf@fastio.com



All figures including the cover of this manual have been generated
entirely by programs that make use of the ClibPDF library. No
drawing or illustration program has been used to create included
figrues. In some cases, the figure PDF files have been converted
into the EPS format using Tailor by FirstClass NV for importing
them into FrameMaker. This manual has been set by using FrameMaker.

Version 2.00 September 14, 1999.

# Contents

# ClibPDF Library Reference Manual

**[Manual version 2.00-release; September 14, 1999]**


## Introduction

**ClibPDF**™ is a library of C functions for generating PDF files directly. A simple C-language program may be linked with the library to produce a stand-alone executable that can directly generate PDF files without help of any other application. Contrast this with the typical methods of creating PDF files. They generally involve multi-step "work-flows" that require the use of GUI-based applications, tools to merge and customize multiple files, and producing the PDF output via PDFWriter print driver. Alternatively, PDF documents may first be generated in the form of PostScript, which are then processed by Acrobat Distiller (or a equivalent program such as Ghostscript), to convert PostScript into PDF. Such a multi-stage processing is tedious, time consuming, and resource intensive because Acrobat Distiller or Ghostscript is a large application that includes a full PostScript interpreter. Perl or shell script, or AppleScript may be able to automate some of these complex processes, but such a solution is cumbersome and inherently suffers from performance and reliability problems on heavily loaded systems. By directly producing PDF files in a one-step process, a simple ANSI-C program with a small memory foot-print will be able to achieve the same result much more quickly.

Since ClibPDF is written in ANSI-C with minimal platform dependencies and does not depend on other proprietary libraries and packages, it will compile and work on just about any system in popular use. The library is intended primarily for two major applications: **1)** dynamic PDF-based Web page generation, and **2)** platform-independent plotting to be built into programs for custom plotting needs. Because of freely available Adobe Acrobat Reader and

other PDF viewers, PDF now boasts greater ease of use and installation of viewer applications than PostScript.

***1. Dynamic Web-page generation in PDF***

A small, light-weight CGI (Common Gateway Interface) program may be written using the ClibPDF library. Such a CGI program may process user input from Web browsers, combine real-time data available on the server side, and create Web pages dynamically as needed. Such CGI program executables may be as small as 100 kbytes (of course depending on platforms), and therefore will generally load and execute much faster than other more complex schemes that rely on multiple programs, including PS-to-PDF distillers or PERL scripts. In addition, PDF-based graphs and plots can produce much more attractive printed output than anything based on pre-rasterized bitmap images such as GIF/JPEG images typically used for presenting graphs and plots on Web pages. Also, zoomed viewing does not trigger any server access. If a high-quality printed output and tightly-controlled document format are important for dynamic Web pages, a ClibPDF-based CGI program may fill such a need quite nicely.

***2. Platform-independent report generation and plotting for custom or legacy applications***

If a cross-platform or custom application requires high-quality graph plotting for report generation purposes, the use of ClibPDF will greatly simplify the task of coding such reporting functions -- there is no need to use platform-specific graphics API. Instead, a simple C-language program may produce graphs, plots and annotations as a PDF file, and invoke Acrobat Reader or another PDF viewer application on the output file. Adobe Acrobat Reader (free software) supports zoomed viewing without “jaggies” and high-quality printing to almost any printer. If applied to custom applications, for example, in industrial, scientific, and medical test and measurement equipment, reports with attractive plots may be generated directly without transferring data to intermediate applications such as spread-sheet or general purpose analysis tools such as Matlab. For routine analyses, potentially involving hundreds and thousands of reports in the same format, this one-step approach becomes highly advantageous.

Using ClibPDF is also an ideal way to enhance legacy command-line or terminal based programs with attractive graphics output, without converting them into fully GUI-based applications. While such a scheme may be less desirable than fully native GUI support for each platform, the trade-off decision is yours and ClibPDF may provide an option suitable for your situation. In particular, X-Window graphics model is completely deficient in supporting screen viewing and printing with a single code base (and sadly DPS is dead). If your application does not require extensive mouse clicking within the graphics area, the use of ClibPDF instead will allow unified support for both high-quality viewing and printing with no extra effort. Therefore, from now on, such crude printing hacks as screen-dumps should not be tolerated.

## Copyright and Distribution



## Background

Conceptually, ClibPDF is patterned after the public-domain Cgraph library for generating PostScript files (http://totoro.berkeley.edu/software/). There aren't too many ways of producing an API of this type, but ClibPDF borrows the notion of plot domains from the Cgraph (PostScript) library. However, ClibPDF has been written completely from the ground up as a fresh design and implementation with many more useful features.

As an added note, ClibPDF is unrelated to PDFlib from Thomas Merz Consulting and Publishing (http://www.pdflib.com/). It has a completely independent code base, and no code from PDFlib has been included or even been consulted during the development of ClibPDF. These two C libraries for PDF generation address partially overlapping but different needs.

PDFlib presents a general tool that provides a C-language interface to PDF at a level very close to individual basic operators. ClibPDF can do probably everything PDFlib does and more (e.g., aligned text and text boxes), and it is especially strong on graph plotting applications and provides specialized objects for linear and logarithmic plot domains, axes (linear and log), and data point markers, in addition to interfacing at the basic PDF operators. ClibPDF also can produce PDF data completely in memory with PDF size (byte count) which may directly be used for "Content-Length:" in Web server applications.

## Thread Safety

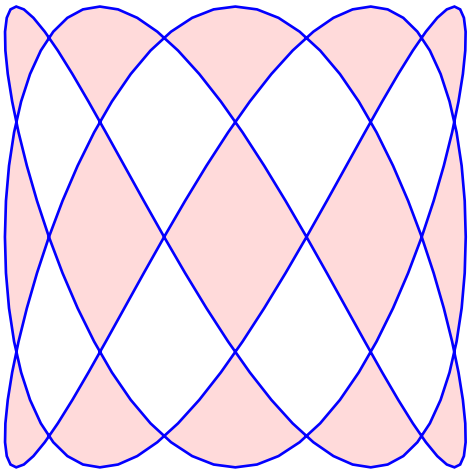
ClibPDF is now ***thread-safe*** for use in multi-threaded applications. A given platform must offer a thread-safe LIBC library. Some time related functions such as localtime() and mktime() have been known to have problems. If your application does not use time domains or axes, this is probably not a problem. Also, a few functions near the beginning of cpdfUtil.c may require a mutex lock, which is platform dependent. There has been an API change, i.e., most

functions now take an additional argument, **CPDFdoc *pdf**, for a pointer to a PDF context struct.

## Overview by a Simple Example: Minimal.c

```
/* Minimal.c -- A simple drawing and one-line text example. */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cpdflib.h"

int main(int argc, char *argv[])
{
CPDFdoc *pdf;
int i;
float x, y, angle;
float radius2= 1.2, xorig = 2.5, yorig = 2.5;  /* in inches */

      /* == [1] Initialization == */
      pdf = cpdf_open(0, NULL);
      cpdf_enableCompression(pdf, YES);  /* use Flate/Zlib compression */
      cpdf_init(pdf);
      cpdf_pageInit(pdf, 1, PORTRAIT, LETTER, LETTER); /* page size */

      /* == [2] Simple graphics drawing example == */
      for(i=0; i<=200; i++) {
             angle = PI*(float)i/100.0;     /* 0 .. 2pi */
             x = radius2 * cos(3.0*angle) + xorig;
             y = radius2 * sin(5.0*angle) + yorig;
             if(i) cpdf_lineto(pdf, x, y);
             else  cpdf_moveto(pdf, x, y);  /* first point */
      }
      cpdf_closepath(pdf);
      cpdf_setrgbcolorFill(pdf, 1.0, 0.7, 0.7); /* pink as fill color */
      cpdf_setrgbcolorStroke(pdf, 0.0, 0.0, 1.0); /* blue stroke color */
      cpdf_eofillAndStroke(pdf);           /* even-odd fill and stroke*/

      /* == [3] Text example == */
      cpdf_setgrayFill(pdf, 0.0);          /* Black */
      cpdf_beginText(pdf, 0);
      cpdf_setFont(pdf, "Times-Italic", "MacRomanEncoding", 16.0);
      cpdf_text(pdf, 1.6, 1.0, 0.0, "x=cos(3t), y=sin(5t)");
      cpdf_endText(pdf);

      /* == [4] Use PDF output */
      cpdf_finalizeAll(pdf);        /* Generate PDF in memory */
      cpdf_savePDFmemoryStreamToFile(pdf, "minimal.pdf");
      cpdf_launchPreview(pdf);

      /* == [5] Clean up == */
      cpdf_close(pdf);              /* shut down */
      return(0);
}
```

Briefly, the program execution proceeds as follows. Step [1] creates a PDF document context via cpdf_open(), a pointer to which is saved used in subsequent calls. It also sets up in-memory output buffers and the page size of the document. Compression of the output is also enabled. In step [2], the “for” loop creates a complex path based on trigonometric functions, paints about half of areas in pink, and strokes the path in blue. You might note that the unit of the coordinate system is in inches. This is the coordinate system of the “default domain” (see below). Of course, the default domain may easily be setup to use other units such as centimeters or millimeters. The “raw” coordinate system defined in points (1/72 inches) is always accessible via a separate set of all drawing functions [see cpdf_raw*() functions]. In step [3], a line of annotation text is drawn in the specified font “Times-Italics” at 16 point. In step [4], the PDF output is finalized and used. In this case, it is saved to a file named “minimal.pdf.” At this point, it is also a simple matter to use the output in any way one desires, e.g, sending out the result via a TCP socket to a remote process. And finally, step [5] closes the ClibPDF system, and additionally in this case, invokes a PDF viewer application on the generated PDF file. The output from the above C language program is shown in Fig. 1.



*x=cos(3t), y=sin(5t)*

**Fig. 1:** Output of Minimal.c example. The path shown in this figure may be considered as a drawing made by a pen attached to a 2-dimensional pendulum. The pendulum swings 5 cycles along the Y direction while it oscillates 3 times along the X dimension. With cpdf_eofillAndStroke(), the path is first filled using the even-odd fill rule, which paints only every other neighboring areas. The path is then stroked using another color.

If you really want to see a "Hello World" example, please delete 11 lines of step [2] in Minimal.c above. And then change the string in cpdf_text() to the desired one.

# Plot Domain: Concept and Examples

ClibPDF is distinguished from other similar libraries by the notion of a "plot domain" useful for plotting scientific or financial data without worrying about scaling data to points or inches, or logarithmic coordinate transformations.

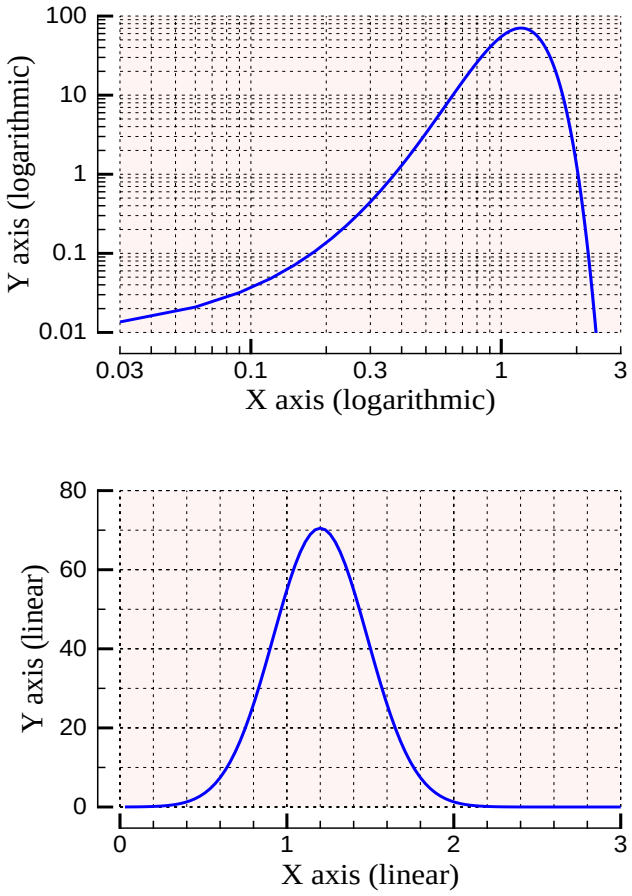



**Fig. 2**: Examples of plot domains presenting the same data in log-log (top), and linear (bottom) coordinate systems. Data for a Gaussian (normal distribution) curve are generated and saved in arrays x[i], y[i] (i=0 .. 99). The two domains are then created and the data are plotted using the following code for each domain:

```
for(i=0; i<100; i++) {
   if(i==0)  cpdf_moveto(pdf, x[i], y[i]);
   else      cpdf_lineto(pdf, x[i], y[i]);
}
cpdf_stroke(pdf);
```

Without any scaling or transformation, the data are displayed correctly for each domain. Therefore, plot domains eliminate potential coding errors and will help keep your source code readable.

The notion of plot domains may most easily be understood by an example. Fig. 2 shows the output of an example program (included as DomainDemo.c) that plots a Gaussian (normal distribution) curve in two ways; one in a log-log coordinate system (top), and the other in a linear-linear system (bottom). Once appropriate domains are created, one can draw curves and place data markers using the original data values directly without any transformation.

Axes may be attached to domains to indicate scaling and values of the data displayed. Given minimum and maximum data values, domains and axes can automatically pick "nice" values for mesh, numbering, and tick marks. You can, of course, over-ride these values to suit your needs and preferences.

# ClibPDF Library Functions

### 1. Minimum Required Call Structure

Any C-language programs that use ClibPDF library functions must have the following structure at a minimum. Additional calls to other functions may be added to modify features and options, and to perform actual drawing.

```
#include "cpdflib.h"
pdf = cpdf_open(0, NULL);
cpdf_init(pdf);
cpdf_pageInit(pdf, 1, PORTRAIT, LETTER, LETTER);
    /* --- Your drawing code goes here. --- */
cpdf_finalizeAll(pdf);
    /* --- PDF output is used here. --- */
cpdf_close(pdf);
```

### 2. Setup, Initialization, Paging, Clean-up Functions

CPDFdoc *__cpdf_open__(int mode, CPDFdocLimits *docLimits);

(REQUIRED)

This must be the first function call to in a ClibPDF program. It signifies the start of ClibPDF processing for a document. The argument “mode” is currently not used, but must be 0. New argument "**docLimits**" may be NULL, if your PDF document is relatively small and the default limits are acceptable. If you wish to increase the limits, pass a pointer to an appropriately initialized CPDFdocLimits struct below as docLimits. The value of -1 may be used if a given parameter should remain at the default value.

The function returns a pointer to a **CPDFdoc** struct, which should be saved and used in most (but not all) ClibPDF API functions as the first argument.

```
typedef struct {
    int nMaxPages;          /* default: 100 pages */
    int nMaxFonts;          /* default 180 */
    int nMaxImages;         /* default 100 */
    int nMaxAnnots;         /* default 100 */
    int nMaxObjects;        /* default 5000 (number of items in "xref" */
} CPDFdocLimits;
```

**EXAMPLE:**

```
CPDFdocLimits dL = {200, -1, -1, -1, 10000};    /* 200 pages */
CPDFdoc *pdf;
   pdf = open(0, &dL);
```

void **cpdf_init**(CPDFdoc *pdf);

(REQUIRED)
This function initializes the PDF document (*pdf as returned by cpdf_open()) as either a memory stream buffer (default) or a file stream if the output filename has been set by cpdf_setOutputFilename() before calling cpdf_init().

void **cpdf_pageInit**(CPDFdoc *pdf, int pagenum, int rot, const char *mediaboxstr, const char *cropboxstr);

(REQUIRED)
This function must also be called, after cpdf_init() and before any functions that draw on a page are called.
The argument "***pagenum***" specifies the page number that you wish to start writing to. For single page PDF files, it should be 1. For multi-page PDF files, you can start with any page (1 or larger). You may also write to different pages in an interleaved manner once each page is initialized. This is accomplished by switching pages by cpdf_setCurrentPage().
"***Rot***" should take on either PORTRAIT or LANDSCAPE as defined in the header file cpdflib.h.
"***Mediaboxstr***" and "***cropboxstr***" should each be a string with 4 numbers separated by spaces, e.g., "0 0 612 792" for the US letter size page. The following predefined strings and more are specified in cpdflib.h for convenience. Any other arbitrary sizes may be specified.

| | |
|---|---|
| LETTER | "0 0 612 792" |
| LEGAL | "0 0 612 1008" |
| A4 | "0 0 595 842" |
| B5 | "0 0 499 708" |
| C5 | "0 0 459 649" |

SEE ALSO: **cpdf_setCurrntPage**(), **cpdf_finalizePage**()

void **cpdf_finalizeAll**(CPDFdoc *pdf);

(REQUIRED)
This function finalizes the buffered PDF stream or a file stream and closes it, so it could be sent to desired destinations. No addition can be made to the PDF file once this function is called. If you forget to call this function, the output file may be empty.

SEE ALSO: **cpdf_savePDFmemoryStreamToFile**()
**cpdf_getBufferForPDF**(), **cpdf_finalizePage**()

void **cpdf_close**(CPDFdoc *pdf);

(REQUIRED)
This function closes and releases all memory and file resources allocated during the use of the library. Memory leaks will result if you do not call this function in a long-running program. It will release only those memory blocks used internally by the library. Objects that you create by calling library functions, such as plot domains and axes, must be freed by you before calling cpdf_close(). In addition, if you malloc your own memory in your code, you must free them as well.

int **cpdf_setCurrentPage**(CPDFdoc *pdf, int page);

(OPTIONAL)
Set current page to another previously initialized page. With this function, you can draw into different pages in an interleaved manner. Switching pages will reset the current domain to the default domain for that page.

SEE ALSO: **cpdf_pageInit**()

void **cpdf_setGlobalDocumentLimits**(int maxPages, int maxFonts, int maxImages, int maxAnnots,
int maxObjects);

(OPTIONAL)
This function globally sets limits on the size of a document such as the maximum number of pages, fonts, images, and annotations. **maxObjects** is the maximum number of objects that appear in the xref (cross-reference) table in a PDF file. The default values are set in cpdflib.h. This function, if used, must be called *before* **cpdf_open**(). Values set by this function will apply to new documents created by all threads in multi-threaded applications.

void **cpdf_finalizePage**(CPDFdoc *pdf, int page);

(OPTIONAL)
This function finalizes **page** (as originally passed to **cpdf_pageInit()**) by closing the content memory stream (or page content file if memory strem is not used) for the page. The purpose of this function is to close and finalize a completed page so that the memory stream for the page may be compressed immeadiately. This will conserve memory usage when generating a PDF file with a large number of pages. If you are generating PDF files with only a few pages, this function call does not make much difference and is not needed. Compression will take place immediately within this call only if content memory streams are used, but not when files are used (pages will be compressed in cpdf_finalizeAll()). If files are used for page content, this function simply

closes the FILE pointer associated with the page, reducing the number of simultaneously open files (often there is a per-process limit on this).

SEE ALSO: **cpdf_useContentMemStream**(), **cpdf_pageInit**()

void **cpdf_enableCompression**(CPDFdoc *pdf, int compressON);

(OPTIONAL)

This function, with the argument YES (defined in cpdflib.h), will enable compression of the Content stream(s) (actual drawing code) using ZLIB/Flate compression. This function should be called before cpdf_init(). By default, the compression is turned off, therefore, you must call this function explicitly to enable it.

There is no support for other compression schemes. Specifically, LZW compression is not used to avoid legal complications with Unisys. Note that ZLIB/Flate compression is a feature of PDF version 1.2 or later. This means that the receiver of the compressed PDF file must use Adobe Acrobat Reader (or Exchange) version 3.0 or later to view the file. Adobe Acrobat Reader is freely available for downloading. Therefore, this should not present a serious problem.

void **cpdf_setDefaultDomainUnit**(CPDFdoc *pdf, float defunit);

(OPTIONAL)

This function set the unit of the default domain that is automatically setup when a page is initialized. The argument “defunit” should be the number of points (1/72 inches) per desired length unit. The function should be called before cpdf_pageInit(). If this function is not called, the default domain is set up using the value of 72.0, where (x, y) are defined in inches. The point-based coordinate system, where (x, y) are defined in points, is always available via separate set of functions prefixed by “cpdf_raw” regardless of the default domain unit.

EXAMPLE:

```
pdf = cpdf_open(0, NULL);
cpdf_enableCompression(pdf, YES);
cpdf_init(pdf);
cpdf_setDefaultDomainUnit(pdf, POINTSPERCM);
        /* POINTSPERCM =28.3464566929 */
cpdf_pageInit(pdf, 1, A4, A4);
```

void **cpdf_setOutputFilename**(CPDFdoc *pdf, const char *file);

(OPTIONAL)
Sets the filename for PDF output. This function may be called at the beginning between cpdf_open(), and cpdf_init(). Doing so will disable the use of a memory stream for the PDF output, and instead opens a file stream to the designated file. For a large file, this saves some memory if the output is sent only to a file or stdout. A special filename, "-", causes the output to be sent to the standard output (stdout). It is not necessary to use this function, as the PDF output may be saved by cpdf_savePDFmemoryStreamToFile().

SEE ALSO: **cpdf_savePDFmemoryStreamToFile**()

void **cpdf_useContentMemStream**(CPDFdoc *pdf, int flag);

(OPTIONAL)
If called with zero as the argument, it will **not** use memory stream for storing the page content. However, this is not recommended for multi-threaded applications. (You may have to put a mutex lock in _cpdf_inc_docID() in cpdfUtil.c.) This may be used if you wish to generate many complex pages on a system with small memory. Temporary files are used for storing content of each page if the memory stream for content is turned off. By default, it will use memory streams. Therefore, it is not necessary to use this function in most cases. If you call this, it should be called before the first cpdf_pageInit() is called, and should not be changed thereafter.

void **cpdf_setPageDuration**(CPDFdoc *pdf, float seconds);

(OPTIONAL)
This function sets the duration (in seconds) the current page is displayed for creating PDF-based slide shows that automatically advance pages. The user can advance the page before the duration has elapsed.

int **cpdf_setPageTransition**(CPDFdoc *pdf, int type, float duration, float direction, int HV, int IO);

(OPTIONAL)
This function sets the type and parameters of page transition effects for the current page. The transition effects applies when going to that page when viewed. Normally (by default), PDF viewers attempt to display a new page instantly. When a transition is specified, transition to the target page takes place over time.

"***Type***" may be one of:

```
#define     TRANS_NONE      0
#define     TRANS_SPLIT     1
#define     TRANS_BLINDS    2
#define     TRANS_BOX       3
#define     TRANS_WIPE      4
#define     TRANS_DISSOLVE  5
#define     TRANS_GLITTER   6
```

"***Duration***" specifies the number of seconds during which the transition is performed. The rest of the arguments only apply to some of the transition types. When not applicable for the specified transition type, the values of arguments are not used. However, you must always pass place-holder arguments in this case.

"***Direction***" specifies the direction of motion of transition boundaries in degrees (0=rightward, 90=upward, 180=leftward, 270=downward). This parameter is applicable only to Wipe and Glitter transition types.

"***HV***" specifies the orientation of the transition borders (1=Horizontal, 0=Vertical). This parameter is applicable only to Split, and Blinds transition types.

"***IO***" specifies the in-out direction of transition (1=IN, 0=OUT). It is applicable only to Split and Box transition types.

### 3. Using PDF Output

char ***cpdf_getBufferForPDF**(CPDFdoc *pdf, int *length);

(OPTIONAL)
This function returns a pointer to the PDF output buffer memory that contains the generated output. The integer argument “length” passed by reference will contain the length of the output in bytes. Note that standard library functions fputs(), puts() should not be used to output the PDF buffer content. An incomplete file will result because PDF files are binary and may contain zeros.

EXAMPLE:
```
cpdf_finalizeAll(pdf);
buf = cpdf_getBufferForPDF(pdf, &length);
printf("Content-Type: application/pdf%c", 10);    /* correct MIME type */
printf("Content-Length: %d%c%c", length, 10, 10);   /* size of PDF */
fwrite((void *)buf, 1, (size_t)length, stdout);       /* Send PDF now */
```

int **cpdf_savePDFmemoryStreamToFile**(CPDFdoc *pdf, const char *file);

(OPTIONAL)
This function saves the content of PDF memory buffer to the specified file. If the file output (including one to stdout) is the only use of the generated PDF, it is possible to set up the PDF stream to go directly to a file without using the memory buffer (thereby saving a little memory). If this is desired, set the output filename at the beginning by calling cpdf_setOutputFilename() before calling cpdf_init(). Then, a PDF memory stream will not be created.

int **cpdf_openPDFfileInViewer**(CPDFdoc *pdf, const char *pdffilepath);
int **cpdf_launchPreview**(CPDFdoc *pdf);

(OPTIONAL)
This launches the default PDF preview application on the file specified as "pdffilepath." If pdffilepath == NULL, it will open the file just created. **cpdf_launchPreview**() is equivalent to **cpdf_openPDFfileInViewer**(NULL). This function is highly platform dependent, and the ClibPDF library as shipped may not contain appropriate code to launch a given viewer for a given platform. Please customize this function in cpdfPreview.c to suit your need and platform, and send in the modifications to us along with the detailed information on the platform. Please include the following information:
*Modified cpdfPreview.c and any other modified files or diffs.
*OS name and version.
*Compiler name and version.
*PDF viewer name and version.
*Your name and e-mail address for acknowledgment.

### 4. Basic Drawing (Path Construction) Functions

One of the key concepts of the imaging model of PDF (and PostScript) is a ***path*** which, in itself, is an invisible contour without any markings on the page. Paths must be acted upon by path-painting operators to produce markings. A path may be stroked with a certain color and width, producing an actual curve on the page. A path may also be filled with a color. It may also be used to define a clipping path. Functions in this section are used to construct paths that are subsequently used to various effects.

Function names for the most part follow those of equivalent PostScript operators, prefixed by “cpdf_” or “cpdf_raw.” Most functions in this section come in two flavors. One type, prefixed by “cpdf_raw,” takes the coordinate values and length in points (1/72 inches). The other type prefixed by “cpdf_” takes the values specified in the current plot domain coordinate system, which by default, is set to an inch-based coordinate system with the origin at the bottom left corner of the page.

void **cpdf_arc**(CPDFdoc *pdf, float x, float y, float r, float sangle, float eangle, int moveto0);
void **cpdf_rawArc**(CPDFdoc *pdf, float x, float y, float r, float sangle, float eangle, int moveto0);

These functions draw an arc centered at (x, y) with radius “r”, starting at angle “sangle” and ending at “eangle.” Angles are specified in degrees. If the last argument “moveto0” is non-zero, e.g., 1, it will perform an initial moveto to the starting point of the arc. If there has not been a pending path construction prior to this call, then “moveto0” should be non-zero. If (moveto0 == 0), it assumes that there is already current point, therefore, a line segment is added from the existing current point to the beginning of the arc. That is, a lineto is performed to the starting point of the arc.

*[In PostScript, the arc operator automatically detects the presence or absence of the current point. But, in PDF, this is not possible, because there is no "arc" operator in PDF and PDF is not a programming language. Therefore, they are drawn using Bezier curves (see cpdf_curveto()) and the programmer has to supply an argument that indicates whether there is a current point or not. The arc function provided here performs some computation in C to obtain coordinates of Bezier control points. Arcs spanning more than 90 degrees are drawn by connecting multiple smaller arcs, each spanning no more than 90 degrees. For more details about this issue, see descriptions of the “arc” operator in the PostScript Language Reference Manual (3-rd ed.), Adobe Systems, Inc. ]*

"Eangle" may be smaller than "sangle." If (eangle > sangle), the arc will be drawn counter-clock-wise. If (eangle < sangle), it is drawn clock-wise. The

direction of the path does not matter for stroking the path, but it makes a difference for the "fill" operator invoked by cpdf_fill() when there are multiple paths. [Look up the term "non-zero winding number rule" in relation to the fill operator.]

SEE ALSO: **cpdf_fill**(), Example C source code in file: Arc.c

EXAMPLE: A 30-degree pie shape may be drawn by:

```
cpdf_moveto(pdf, 0.0, 0.0);
cpdf_arc(pdf, 0.0, 0.0, 2.0, 0.0, 30.0, 0);   /* lineto to start of arc */
cpdf_closepath(pdf);
cpdf_stroke(pdf);
```

void **cpdf_circle**(CPDFdoc *pdf, float x, float y, float r);
void **cpdf_rawCircle**(CPDFdoc *pdf, float x, float y, float r);

These functions draw a circular path centered at (x, y) with radius "r" in the counter-clock-wise direction. If you need a circle drawn in the clock-wise direction, please use "cpdf_arc(pdf, x, y, r, 360.0, 0.0); cpdf_closepath(pdf);." This function performs a move to angle 0 (right edge) of the circle. Current point will also be at the same location after the call.

Note: Specifying radius "r" in cpdf_circle() can be problematic if one or both of the axes of the current domain is logarithmic, or if X and Y scales differ even in a linear domain. This also applies to cpdf_arc(). Therefore, do not use this function for data point marking. Use cpdf_marker() instead for drawing data point markers.

void **cpdf_closepath**(CPDFdoc *pdf);

This closes a path by connecting the first and the last point in the path currently being constructed. Call to this function is often needed to avoid a notch in a stroked path, and to make "line join" work correctly in joining the first and the last points.

void **cpdf_curveto**(CPDFdoc *pdf, float x1, float y1, float x2, float y2, float x3, float y3);
void **cpdf_rawCurveto**(CPDFdoc *pdf, float x1, float y1, float x2, float y2, float x3, float y3);

This function adds a Bezier cubic curve segment to the path starting at the current point as (x0, y0), using two points (x1, y1) and (x2, y2) as control

points, and terminating at point (x3, y3). The new current point will be (x3, y3). If there is no current point, an error will result.

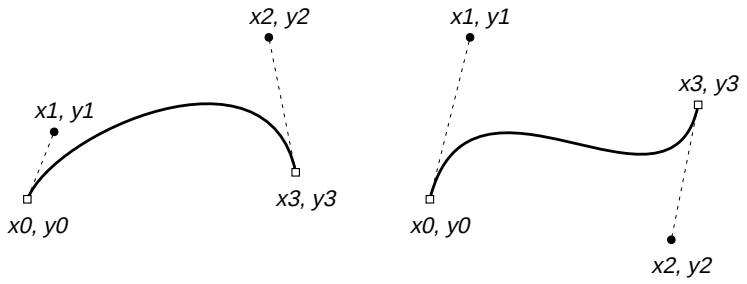



**Fig. 3:** Two Bezier curves are shown. Curveto operator adds a curved path segment from the current point (x0, y0) to (x3, y3), using two points (x1, y1) and (x2, y2) as "control points." The curve emerges and terminates tangent to the dashed lines at (x0, y0) and (x3, y3). (The dashed lines and control points are not visible on the page.) The longer the dashed lines, the longer the curve remains close to the dashed control lines. See the example **beziertest.c** in examples/bezier.

void **cpdf_lineto**(CPDFdoc *pdf, float x, float y);
void **cpdf_rawLineto**(CPDFdoc *pdf, float x, float y);

These functions add a line segment to the path, starting at the current point and ending at point (x, y).

void **cpdf_moveto**(CPDFdoc *pdf, float x, float y);
void **cpdf_rawMoveto**(CPDFdoc *pdf, float x, float y);

These functions move the current point to the location specified by (x, y).

void **cpdf_newpath**(CPDFdoc *pdf);

Clears the current path. Current point becomes undefined.

void **cpdf_rlineto**(CPDFdoc *pdf, float x, float y);
void **cpdf_rawRlineto**(CPDFdoc *pdf, float x, float y);

These are identical to cpdf_lineto() and cpdf_rawLineto(), except that (x, y) specify offset *relative* to the current point, not the absolute position.

void **cpdf_rmoveto**(CPDFdoc *pdf, float x, float y);
void **cpdf_rawRmoveto**(CPDFdoc *pdf, float x, float y);

These move the current point by the offset specified by (x, y).

void **cpdf_rect**(CPDFdoc *pdf, float x, float y, float width, float height);
void **cpdf_rawRect**(CPDFdoc *pdf, float x, float y, float width, float height);

These functions draws a rectangle of size (width, height) with one corner at (x, y).

void **cpdf_rectRotated**(CPDFdoc *pdf, float x, float y, float width, float height, float angle);
void **cpdf_rawRectRotated**(CPDFdoc *pdf, float x, float y, float width, float height, float angle);

These functions draws a rectangle of size (width, height) with one corner at (x, y) as above, but with an additional orientation argument, **angle**, specified in degrees.

### 5. Path Painting Operators

Paths constructed by functions in the previous section are invisible, i.e., constructing a path does not produce any marking on the page. They must be stroked or filled.

void **cpdf_clip**(CPDFdoc *pdf);
void **cpdf_eoclip**(CPDFdoc *pdf);

These functions installs the current paths as the boundary for clipping subsequent drawing. cpdf_eoclip() uses the “even-odd” rule for defining the “inside” that shows through the clipping window. The use of these clip operators may require some care, because clip and eoclip operators do not consume the current path. Also note that there is no practical way of removing a clipping path, except for by “grestore”-ing a graphical state before clipping is imposed. Therefore, a typical usage should look like:

```
cpdf_gsave(pdf);
cpdf_newpath(pdf);
cpdf_rect(pdf, x, y, width, height);
cpdf_clip(pdf);
cpdf_newpath(pdf);
/* Perform drawing restricted to within the clipped area. */
cpdf_grestore(pdf);
```

SEE ALSO: **cpdf_clipDomain**(), and example: DomainDemo.c

void **cpdf_fill**(CPDFdoc *pdf);
void **cpdf_eofill**(CPDFdoc *pdf);

These functions use the current path as the boundary for color filling. cpdf_fill() uses the “non-zero winding number” rule, whereas cpdf_eofill() uses the “even-odd” rule for defining “inside” that is painted.

SEE ALSO: Example files Arcs.c and Minimal.c

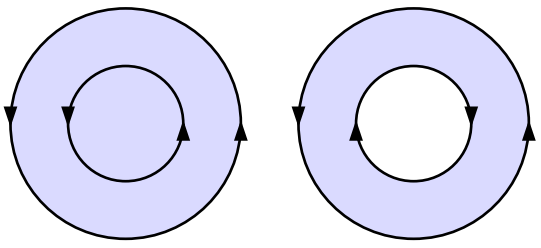


**Fig. 4**: The *non-zero winding number rule* is used by the "fill" operator in the PDF (and PostScript) imaging model to determine the "inside" that is to be filled. The inside of the inner circle on the right is considered outside because its path winds in the opposite direction. The "eofill" operator uses a different rule in which the even-odd-ness of the winding number is used.

void **cpdf_fillAndStroke**(CPDFdoc *pdf);
void **cpdf_eofillAndStroke**(CPDFdoc *pdf);

These functions are used to first fill the inside with the current fill color, and then stroking the path with the current stroke color. PDF's graphics state maintains separate colors for fill and stroke operations, thus these combined operators are made available. [PostScript graphics state maintains only one color at a time.]

void **cpdf_stroke**(CPDFdoc *pdf);

This function strokes the current paths by the current stroke color and current linewidth.

### 6. Graphics State Operators

void **cpdf_gsave**(CPDFdoc *pdf);
void **cpdf_grestore**(CPDFdoc *pdf);

These two functions are used to save and restore current graphics state.

void **cpdf_setdash**(CPDFdoc *pdf, const char *dashspec);
void **cpdf_nodash**(CPDFdoc *pdf);

cpdf_setdash() sets the current dash pattern according to the specification passed in a string. cpdf_nodash() resets the dash pattern back to none, i.e., solid line.

The "***dashspec***" should look like "[6 6] 0" and "[6 2 1 2] 0" where the numbers are in points (1/72 inches). The number following the [ ] determine the phase of the dash pattern.

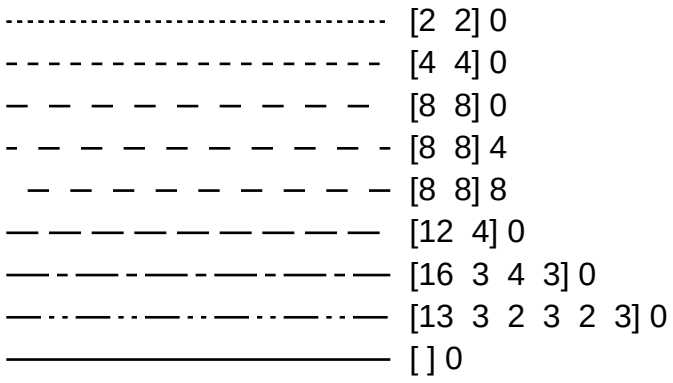



**Fig. 5:** Examples of dash patterns. Numbers in the [ ] are used alternately and repeatedly to define lengths of ON and OFF segments in points (1/72 inches). The last number indicates an offset into the dash array from which the dash pattern starts. Empty [ ] resets the dash pattern to solid line. See **dashtest.c** in examples/dash directory for source code.

void **cpdf_concat**(CPDFdoc *pdf, float a, float b, float c, float d, float e, float f);
void **cpdf_rawConcat**(CPDFdoc *pdf, float a, float b, float c, float d, float e, float f);

This function concatenates a transformation matrix passed to the current transformation matrix (CTM). The last two arguments “e”, and “f” are defined in the raw point-based coordinate system for cpdf_rawConcat(). "E" and "f" are specified in the unit of the current domain for cpdf_concat(). For further descriptions, see the PDF Reference Manual (v1.3, page 33).

Note: the 4 operators in this section: "concat", "rotate", "scale", and "translate" inherently changes the CTM of the raw (point-based) coordinate system. If you try to maintain a state, such as a point-to-domain-unit conversion factor, at the level of plot domain API, the use of these functions may make matters quite confusing. In particular, don’t expect these functions to work intuitively if the domain uses a logarithmic dimension.

void **cpdf_rotate**(CPDFdoc *pdf, float angle);

This rotates the coordinate system by the angle given in degrees (positive is clock wise). The rotation is centered at the current origin of the raw coordinate system.

void **cpdf_scale**(CPDFdoc *pdf, float sx, float sy);

This function scales the coordinate system by scaling factors supplied for X and Y dimensions.

void **cpdf_translate**(CPDFdoc *pdf, float xt, float yt);
void **cpdf_rawTranslate**(CPDFdoc *pdf, float xt, float yt);

This function shifts the origin of the coordinate system by the (xt, yt) specified. The arguments are specified in points for cpdf_rawTranslate(). For cpdf_translate(), they are specified in the unit of the current plot domain.

void **cpdf_setflat**(CPDFdoc *pdf, int flatness);/* flatness = 0 .. 100 */

void **cpdf_setlinecap**(CPDFdoc *pdf, int linecap);/* linecap = 0(butt end), 1(round), 2(projecting square) */

This functions sets the mode that determines what happens at line ends. The default is =0 (butt end).

void **cpdf_setlinejoin**(CPDFdoc *pdf, int linejoin); /* linejoin = 0(miter), 1(round), 2(bevel) */
void **cpdf_setmiterlimit**(CPDFdoc *pdf, float miterlimit);

These two functions determine what happens at corners where two line segments meet, and determine how pointed the corners may become when they form a highly acute angle.

void **cpdf_setlinewidth**(CPDFdoc *pdf, float width);

This function sets the current linewidth to the value specified in points (1/72 inches).

### 7. Color and Gray Functions

Functions here set current color for fill and strokes. In PostScript, the graphics state only maintains one color specification for both fill and strokes. In PDF, fill and stroke colors are separately maintained. In all functions in this section, arguments will take values in the range [0 .. 1].

void **cpdf_setgray**(CPDFdoc *pdf, float gray); /* set both fill and stroke grays */
void **cpdf_setrgbcolor**(CPDFdoc *pdf, float r, float g, float b);/* set both fill and stroke colors */

These functions set both stroke and fill colors to the gray or color values specified.

void **cpdf_setgrayFill**(CPDFdoc *pdf, float gray);
void **cpdf_setrgbcolorFill**(CPDFdoc *pdf, float r, float g, float b);
void **cpdf_setcmykcolorFill**(CPDFdoc *pdf, float c, float m, float y, float k);

These functions set the fill gray or color to the values specified.

void **cpdf_setgrayStroke**(CPDFdoc *pdf, float gray);
void **cpdf_setrgbcolorStroke**(CPDFdoc *pdf, float r, float g, float b);
void **cpdf_setcmykcolorStroke**(CPDFdoc *pdf, float c, float m, float y, float k);

These functions set the stroke gray or color to the values specified.

## 8. Text Functions

A typical usage of text involves the following sequence of functions described in this section. First, a text block is initiated by cpdf_beginText(). Then, a desired font is set by cpdf_setFont(). After this, any number of text lines (at any orientation) may be drawn as long as the text lines share the same font attribute. Finally, the text mode is closed by calling cpdf_endText().

```
cpdf_beginText(pdf, 0);
cpdf_setFont(pdf, "Times-Italic", "MacRomanEncoding", 16.0);
cpdf_text(pdf, 1.6, 2.0, 0.0, "Test of arcs and circles");
cpdf_text(pdf, 1.6, 7.0, 0.0, "Color filled pie shapes");
cpdf_text(pdf, 4.7, 5.0, 0.0, "Non-zero winding rule for fill");
cpdf_endText(pdf);
```

void **cpdf_beginText**(CPDFdoc *pdf, int clipmode);
void **cpdf_endText**(CPDFdoc *pdf, );

(BOTH REQUIRED)
Any explicit use of text must be bracketed by these two functions. It must begin with cpdf_beginText(), and end with cpdf_endText().
In normal uses of text, “clipmode” should be zero. Using a non-zero value for “clipmode” causes the function to perform “gsave” before entering the text mode, and performs “grestore” at the end of corresponding cpdf_endText() processing.

int **cpdf_setFont**(CPDFdoc *pdf, const char *basefontname, const char *encodename, float size);

(REQUIRED)
Sets the current font specified by “basefontname” with character encoding specified by “encodename.” For applications that require fast PDF generation and small PDF size, it is recommended that “basefontname” be one of 41 fonts built into ClibPDF. The following list shows these fonts (See the output PDF file generated by examples/fontlist.c for font samples):

***PDF Base 14 fonts:***

| | |
|---|---|
| "Helvetica" | "Helvetica-Bold" |
| "Helvetica-Oblique" | "Helvetica-BoldOblique" |
| "Times-Roman" | "Times-Bold" |
| "Times-Italic" | "Times-BoldItalic" |
| "Courier" | "Courier-Bold" |
| "Courier-Oblique" | "Courier-BoldOblique" |
| "Symbol" | "ZapfDingbats" |

***Additional 25 PostScript Type 1 fonts:***

| | |
|---|---|
| "AvantGarde-Book" | "AvantGarde-BookOblique" |
| "AvantGarde-Demi" | "AvantGarde-DemiOblique" |
| "Bookman-Demi" | "Bookman-DemiItalic" |
| "Bookman-Light" | "Bookman-LightItalic" |
| "Helvetica-Narrow" | "Helvetica-Narrow-Oblique" |
| "Helvetica-Narrow-Bold" | "Helvetica-Narrow-BoldOblique" |
| "NewCenturySchlbk-Roman" | "NewCenturySchlbk-Italic" |
| "NewCenturySchlbk-Bold" | "NewCenturySchlbk-BoldItalic" |
| "Palatino-Roman" | "Palatino-Italic" |
| "Palatino-Bold" | "Palatino-BoldItalic" |
| "Helvetica-Condensed" | "Helvetica-Condensed-Bold" |
| "Helvetica-Condensed-Oblique" | "Helvetica-Condensed-BoldObl" |
| "ZapfChancery-MediumItalic" | |

***Two additional fonts we configured by manipulating metrics/flag:***

| | |
|---|---|
| "CPDF-Monospace" | (Helvetica-like monospace font) |
| "CPDF-SmallCap" | (SmallCap font derived from Times-Roman) |

Unless the computer has these PostScript fonts installed in such a way Acrobat Reader/Exhange can access them, they are approximated by Multiple-Master fonts for the screen display, and printing to non-PostScript printers. When printing to PostScript printers with these fonts built-in, the rendered fonts take on the true appearance of the fonts specified.

In addition to these built-in fonts, ClibPDF now supports font embedding for PostScript Type 1 fonts using PFM/PFB files. AFM/PFA files must be converted to PFM/PFB format before they can be used with ClibPDF. Font subsetting is not supported. Because font embedding requires search and I/O for font resources and increases the size of PDF file by about 30-50 kbytes per font, attempts should be made to minimize its use for applications that require speed and small file size such as dynamic PDF generation via Web servers. See also: **cpdf_setFontDirectories**() and **cpdf_setFontMapFile**() for setting up external font access for font embedding.

“Encodename” must be one of “MacRomanEncoding“, “MacExpertEncoding“, “WinAnsiEncoding”, and "NULL" in which case the font’s built-in encoding is used. Font “size” must be given in points.
[*Bug: In the current release, the encoding must be the same for a given basefontname. Using more than one encodings for the same base font may cause a crash.*]

void **cpdf_setFontDirectories**(CPDFdoc *pdf, const char *pfmdir, const char *pfbdir);
void **cpdf_setFontMapFile**(CPDFdoc *pdf, const char *mapfile);

These two functions set up the mechanism for locating external font files for font embedding. ClibPDF looks for font files in three places as follows (in the order they appear):

1. Current directory.
2. Directories specified by **cpdf_setFontDirectories**().
3. In locations specified by a font map file specified by **cpdf_setFontMapFile**().

The first funciton, **cpdf_setFontDirectories**(), sets directories in which PFM and PFB files are located. "Pfmdir" is the directory where PFM files exist, and "pfbdir" is that for PFB files. Do **not** end these directory path with the directory separator character such as '/' for Unix, and '\' for Windows.
Font files in these directories must have exactly the same name (including case) as the "basefontname" as specified in cpdf_setFont() function. That is, if you use: cpdf_setFont("Tekton", ..), then, files "Tekton.pfm" and "Tekton.pfb" must exist. This also applies to fonts in the current directory. No enviornment variables for these directories are built into ClibPDF. You may easily implement these yourself if needed.

Fonts that are specified by the font map file may be called by aliases or short names, or need not be located in a single directory. There are examples of font map file in source/fontmap.lst and fontmap-gs.lst.

***Font map file format***

Blank lines and lines starting with '#' and '%' are comments.
Font directory paths specified by **$Font_Directories** will be prepended to each font entry within (), UNLESS the entry starts with strings noted by **$Abs_Path_Prefix**. These two must appear BEFORE all font specification entries. Do **not** include the last directory separate such as '\' or '/' in the **$Font_Directories** specification. These two $* specs must occur together. Having only one of them is not allowed. However, specifying no $* at all is allowed, in which case the spec in () is used as is. Upto 10 items of each less than 8 chars are allowed for **$Abs_Path_Prefix**. In this way, most entries may be specified by filenames alone when they are located in a common font directory. Yet, non-standard font locations are also allowed by specifying absolute paths explicitly.

```
# --- Example for Windows: These two lines must come BEFORE actual font mapping spec. --
# $Font_Directories      (\Library\Resources\Fonts\pfm)(\Library\Resources\Fonts\pfb)
# $Abs_Path_Prefix       (\)        (c:\)        (e:\)
# --- Example for Unix:
```

```
# $Font_Directories      (/usr/local/lib/gs/pfm)(/usr/local/lib/gs/pfb)
$Font_Directories        (.)                    (.)
$Abs_Path_Prefix         (/)

# Font Name is the name passed to cpdf_setFont().  Only if you use fontmap file,
# names for cpdf_setFont() may differ from the actual filenames (basename).
# Font Name does NOT have to be the real name of the font as defined within
# PFM/PFB files.  For example, the real name of "AA-URWGothicL-Book" is "URWGothicL-Book"

# Font Name                  PFM File              PFB File
AA-URWGothicL-Book         (a010013l.pfm)         (a010013l.pfb)
URWBookmanL-DemiBold       (b018015l.pfm)         (b018015l.pfb)
```

### *8.1 Convenience Text Functions*

***The following 8 functions provide convenient and intuitive access to most text drawing needs using standard font characteristics. If you need fine tuning of text drawing beyond the capabilities of these functions, lower-level functions may be used. These additional functions are described further below.***

void **cpdf_text**(CPDFdoc *pdf, float x, float y, float orientation, const char *textstr);
void **cpdf_rawText**(CPDFdoc *pdf, float x, float y, float orientation, const char *textstr);

These two functions draw a text line “textstr” using the current font starting at location (x, y) at “orientation” (in degrees). The lower-left corner of the text string is placed at (x, y). cpdf_text() specifies the (x, y) according to the current plot domain. cpdf_rawText() accepts (x, y) defined in points (1/72 inches). For a given font, these functions may be called multiple times within a text block. Note that these and all other text functions will escape special characters ’**(**’, ’**)**’, and ’\’ for PDF strings automatically (i.e., prefix ’\’ to each of these). Therefore, there is no need to perform this preprocessing on your part.

void **cpdf_textAligned**(CPDFdoc *pdf, float x, float y, float orientation, int alignmode, const char *textstr);
void **cpdf_rawTextAligned**(CPDFdoc *pdf, float x, float y, float orientation, int alignmode,
const char *textstr);

These two functions are identical to those above, except that they provide additional text positioning capability. Text lines may be left- or right-aligned, or centered. The argument "**alignmode**" controls how the string is positioned with

respect to (x, y). Fig. 6 shows the list of positioning modes that should be passed as "alignmode", as defined in cpdflib.h.

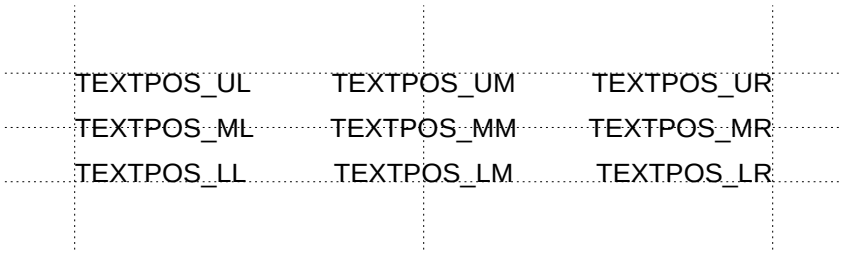



**Fig. 6:** Text alignment modes are illustrated. Intersections of the dashed lines indicate the positions specified by (x, y). For example, passing the mode TEXTPOS_UR as ***alignmode*** positions the string such that the upper-right corner of the string is located at (x, y). See **textalign.c** in examples/text directory.

char ***cpdf_rawTextBox**(CPDFdoc *pdf, float xl, float yl, float width, float height, float angle,
float linespace, CPDFtboxAttr *tbattr, char *text);
char ***cpdf_textBox**(CPDFdoc *pdf, float xl, float yl, float width, float height, float angle,
float linespace, CPDFtboxAttr *tbattr, char *text);

These two functions finally implement one of the top requested features: columnar formatting of text. The function allows simple **text-to-pdf** capability with left-, right-, centering-, and justification. Four float arguments **(xl, yl, width, height)** specify a text box into which text is formatted. **(xl, yl)** is the bottom-left coordinate of the box and **(width, height)** are the size parameters of the box. **Angle** specifies the orientation of the text box. A text box is rotated about **(xl, yl)**. **Linespace** (given in points, 1/72 inches, when it is positive) sets the spacing between lines (distance between baseline of one line to that of the next). When linespace is ***negative***, its absolute value is interpreted as a multiplying factor for the current font size, whereby **fabs(linespace)*fontsize** determines the actual line spacing. The content of text buffer, char ***text**, will be modified by this call. **NOTE:** text data should end with "**\n\n**" for the text to be interpreted as a paragraph. Otherwise, the last line may not be correctly handled.

The argument, **tbattr**, is a pointer to a textbox attribute structure, and has the format shown below. You may pass a **NULL** as **tbattr**, in which case defalut values are used. Note that this structure is likely to have additional members in the future.

```
typedef struct {
    int  alignmode;          /* one of the modes above (default= TBOX_LEFT) */
    int  NLmode;             /* if non-zero, NL is is a line break, if 0 reformatted (default= 0) */
    float paragraphSpacing;  /* extra space between paragraphs (default= 0.0) */
    int  noMark;             /* if non-zero, text is not drawn (used to find out if text fits) */
} CPDFtboxAttr;
```

For passing as "**alignmode**" to cpdf_textBox() and cpdf_rawTextBox():

```
#define     TBOX_LEFT       0
#define     TBOX_CENTER     1
#define     TBOX_RIGHT      2
#define     TBOX_JUSTIFY    3
```

**Alignmode** should be one of the above defines (from cpdflib.h).
**NLmode = 0** will reformat text into the columnar width ignoreing single newline ('\n') characters in the text. Consecutive newline chars of 2 or more will be honored, and terminates a paragraph. In this mode, some non-zero **paragraphSpace** would be useful to give some separation between paragraph blocks. When positive, it specifies the spacing in the unit of points (1/72 inches). When it is negative, its absolute value multiplied by the current font size determines the actual spacing.
**NLmode = 1** will perform simple text-to-PDF formatting honoring newline ('\n') chars literally as line breaks. In this mode, **paragraphSpacing** probably should be 0 because it simply adds to **linespace** (see above) given as a main argument
**noMark**, when non-zero, will cause the text to be not painted actually. This is used when trying to find out if a given amount of text will fit into the box specified. (See cpdf_textBoxFit()).

**RETURN VALUE:**
The return value is **NULL** if all the text fits into the text box defined. If the text is too long to be contained in the textbox, a **(char *) pointer to the remainder** of the text is returned. This points to somewhere in the middle of the original text string. This return value may be used as the new text string for another textbox to continue text placement into another column or to one on the next page. (See the TextBox example in examples/textbox/textbox.c.)

**Bugs and Potential Problems with TextBox:**

This function is not likely to work for CJK (Chinese, Japanese, and Korean) language text, which may not contain any space character.

TABs don't work. (This, we will probably add at some point.)

This function modifies the portion of the text data that have been used (have already been put into a text box). If you need to retain the original text, make a copy and use that with these functions.

I don't think we will extend these text box functions much further. We know things like underline, bold, italics, changing fonts, HTML support, RTF support, etc. would be nice, but those will require a complete rewrite of cpdfTextBox.c.

**EXAMPLE:**

```
char *textbuf = "Some very long text string\nfrom somewhere.....\n\n";
char *currtext;
CPDFtboxAttr tboxatr;
float angle=0.0, linespace = 12.0;
float yl=72.0, w=225, h=648.0;

   tboxatr.alignmode = TBOX_JUSTIFY;
   tboxatr.NLmode = 0;          /* reformat */
   tboxatr.paragraphSpacing = 12.0;
   tboxatr.noMark = 0;
   currtext = textbuf;          /* point to head of text */
   cpdf_beginText(pdf, 0);
   cpdf_setFont(pdf, "Times-Roman", "MacRomanEncoding", 12.0);
   currtext = cpdf_rawTextBox(pdf, 72.0, yl, w, h, angle,
               linespace, &tboxatr, currtext);          /* left column */
   currtext = cpdf_rawTextBox(pdf, 315.0, yl, w, h, angle,
               linespace, &tboxatr, currtext);          /* right column */
   cpdf_endText(pdf);
```

float **cpdf_rawTextBoxFit**(CPDFdoc *pdf, float xl, float yl, float width, float height, float angle,
float fontsize, float fsdecrement, float linespace,
CPDFtboxAttr *tbattr, char *text);
float **cpdf_textBoxFit**(CPDFdoc *pdf, float xl, float yl, float width, float height, float angle,
float fontsize, float fsdecrement, float linespace,
CPDFtboxAttr *tbattr, char *text);

These two functions are similar to cpdf_rawTextBox() and cpdf_textBox(), respectively, except that they will forcibly fit the text into the specified box by reducing the font size (and optionally line spacing) if necessary. Therefore, there will be no overflow of text. Two new parameters, **fontsize** and **fsdecrement**, are explained below. All other parameters are the same as for the standard text box.

**fontsize** sets the requested font size. If text data fits into the box at this font size, it will be used. The font size will not be increased to take up the extra space in the box.

**fsdecrement** specifies the amount of font size decrement which determines the next smaller font size to be tried for fitting. It is specified in points (1/72 inches). Either 1.0 or 0.5 should work well. (Don’t use 0.0. Infinit loop!)

***Negative*** values for **linespace** and **tbattr->paragraphSpacing** are useful for these functions for scaling the line spaceing with the font size (see above). If positive values are used for these, only the font size is progressively reduced until the text fits into the box.

The content of text buffer, char ***text**, will be modified by this call. **NOTE:** text data should end with "**\n\n**" for it to be interpreted as a paragraph. Otherwise, the last line may not be correctly handled or the text will be shrunk excessively.

**RETURN VALUE:**
These two functions return the font size (in points) that is actually used.

All limitations and possible problems for the standard text box also apply to these functions as well.

**EXAMPLE:**
See **examples/textboxfit/textboxfit.c.**

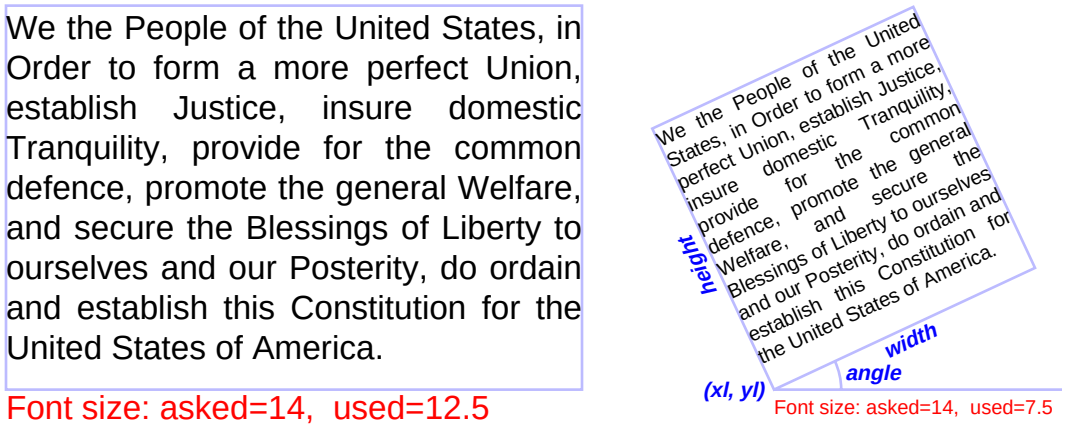



**Fig. 7:** The same text data are fit into text boxes of different sizes using **cpdf_textBoxFit()**. The font size was initially set to 14 point, but scaled down to fit the entire text into respective boxes. Text data for use with text boxes should be terminated by "**\n\n**".

## *8.2 Low-Level Text Functions*

***The following functions allow additional fine tuning of text positioning and rendering. In order to use these functions, you must be familiar with the text imaging model of PDF.***

### 8.2.1 Auto Line Spacing Functions

Functions in this section utilize "***current point***" and "***leading***" (i.e., line spacing in lay terminology) setting of the text graphics state to draw multiple text lines easily. Once the initial current point and leading are set, multiple lines of text may be printed without specifying (x, y) location for additional lines. Typically, the following sequence of calls is used:

```
cpdf_beginText(pdf, 0);
cpdf_setFont(pdf, "Times-Roman", "MacRomanEncoding", 12.0);
cpdf_setTextLeading(pdf, 22.0);       /* 22 point leading */
cpdf_setTextPosition(pdf, 1.0, 7.0);  /* (1, 7) in default domain */
cpdf_textShow(pdf, "Line 1 is drawn here.");
cpdf_textCRLFshow(pdf, "Line 2 is drawn 22 point below line 1.");
cpdf_textCRLFshow(pdf, "Line 3 is drawn 22 point below line 2.");
cpdf_endText(pdf);
```

void **cpdf_setTextLeading**(CPDFdoc *pdf, float leading);

This function sets the current "leading" or line spacing in points (1/72 inches) in the text state. The default value of leading is zero. Once the leading and text current point are set, multiple text lines may be printed repeatedly using cpdf_CRLFshow().

void **cpdf_setTextPosition**(CPDFdoc *pdf, float x, float y);
void **cpdf_rawSetTextPosition**(CPDFdoc *pdf, float x, float y);

These two functions set the text current point and line start position to (x, y) in the current domain or raw coordinate systems, respectively. Internally, these functions sets a new text transformation matrix. If you wish to rotate or skew text, perform these transformations after a call to one of these functions.

SEE ALSO: **cpdf_rotateText**(), **cpdf_skewText**().

void **cpdf_textShow**(CPDFdoc *pdf, const char *txtstr);

This function prints the text at the text current point. The lower-left corner of the text is placed at the current point. The new current point moves to the end of the text string.

void **cpdf_textCRLFshow**(CPDFdoc *pdf, const char *txtstr);

This function first moves the text current point to the beginning of the new line according to the current leading (line spacing), and then draws the text.

void **cpdf_textCRLF**(CPDFdoc *pdf);

This function moves the text current point to the beginning of the new line.

### 8.2.2 Text Coordinate System Transformation Functions

The functions described in this section provides the most general methods of setting the text state by way of text matrix. Text current point, rotation, and skewing may be set.

void **cpdf_setTextMatrix**(CPDFdoc *pdf, float a, float b, float c, float d, float x, float y);
void **cpdf_concatTextMatrix**(CPDFdoc *pdf, float a, float b, float c, float d, float x, float y);

Unlike the current transformation matrix (CTM) for the general graphics state (for which a new matrix is always concatenated to the existing one), PDF text matrix may only be set to a new value at the operator level. The first function, cpdf_setTextMatrix() provides the ability to set the text matrix to a new one. To allow sequential concatenated transformations, ClibPDF also emulates a concatenation behavior by the second function, cpdf_concatTextMatrix(). The text CTM is maintained within the library and updated. The new concatenated text matrix is then set.

void **cpdf_rotateText**(CPDFdoc *pdf, float degrees);

This function sets the orientation of text by concatenating a rotation matrix to the current text matrix (maintained within ClibPDF). It internally uses cpdf_concatTextMatrix() as described above. To draw rotated text, first the text current point should be positioned at the desired position by using cpdf_setTextPosition(). Then, use this function to rotate the text orientation.

SEE ALSO: **cpdf_setTextPosition**(), **cpdf_concatTextMatrix**()

void **cpdf_skewText**(CPDFdoc *pdf, float alpha, float beta);

This function skews the text. Alpha is the angle (in degrees) of text baseline skew from the horizontal. Beta is the angle (in degrees) of skew from the vertical axis. As with text rotation, skew transformation is concatenated to the

existing text matrix (maintained within ClibPDF). It internally uses cpdf_concatTextMatrix() as described above.

SEE ALSO: **cpdf_setTextPosition**(), **cpdf_concatTextMatrix**()

### 8.2.3 Horizontal Spacing and Scaling

void **cpdf_setCharacterSpacing**(CPDFdoc *pdf, float spacing);

This function sets the additional space (in points) that should be inserted between characters.

void **cpdf_setHorizontalScaling**(CPDFdoc *pdf, float scale);

This function sets the horizontal scaling factor in percentage. This essentially expands or compresses the horizontal dimension of the string. The default value for this parameter is 100 (%).

void **cpdf_setWordSpacing**(CPDFdoc *pdf, float spacing);

This functions sets the additional space (in points) that should be inserted between words, i.e., for every space character found in the text string.

### 8.2.4 Text Rendering

void **cpdf_setTextRenderingMode**(CPDFdoc *pdf, int mode);

This function sets the mode that determines how the character outline is used. By default, the character outline is used for filling operation by which inside of the outline path is painted solidly with the current fill color. This may be changed by calling this function. The following modes are available:

```
#define    TEXT_FILL              0
#define    TEXT_STROKE            1
#define    TEXT_FILL_STROKE       2
#define    TEXT_INVISIBLE         3
#define    TEXT_FILL_CLIP         4
#define    TEXT_STROKE_CLIP       5
#define    TEXT_FILL_STROKE_CLIP  6
#define    TEXT_CLIP              7
```

TEXT_FILL is the default mode.

8.2.5 Text Rise (Superscript and Subscript)

void **cpdf_setTextRise**(CPDFdoc *pdf, float rise);

This function sets the "text rise", the amount of text base line offset in the vertical dimension in points (1/72 inches). A positive rise moves the baseline upward, causing subsequent text to become superscript. Similarly, a negative rise may be used to produce subscript text.

### *8.3 Other Text Related Functions*

float **cpdf_stringWidth**(CPDFdoc *pdf, const unsigned char *string);

Returns the width of the current font for the string “string” in points (1/72 inches). Font must have been set by cpdf_setFont() for this function to work.

float **cpdf_capHeight**(CPDFdoc *pdf);

Returns the height of most capital letters in points (1/72 inches).

## 9. Plot Domain Functions

A plot domain is a key feature of ClibPDF library. A plot domain gives you a relatively high-level API for producing graphs and plots, eliminating the drudgery of converting native units of data to points that define the default PDF coordinate system. By creating a domain whose coordinate system matches your data, you can work with the original data set directly. Note that most of plot domain-related functions do **not** require the new PDF document context argument, **CPDFdoc *pdf**. The only function here that needs it is **cpdf_setPlotDomain**().

Here is a simple example that shows how to use a plot domain. The first call creates a semi-logarithmic plot domain of 4.5 x 3 inches ("#define inch 72.0" in cpdflib.h), with the lower-left corner positioned at (1.6, 5.5) inches. Note that cpdf_createPlotDomain() takes arguments in the raw point-based coordinate system. The X value range [0.1 .. 100], and Y value range [0 .. 80] are mapped to the domain. The X range is scaled logarithmically, while the Y range is linearly mapped. The second call sets the newly created "myDomain" as the current plot domain. The function returns the old current domain, so we save it for later restore. The next call fills the domain with a light pink color. Then, we draw a mesh or grid pattern to produce an appearance a graph paper.

The next four lines that include a "for" loop produce a scatter plot in the new domain. A circle (black border, yellow inside) is placed at each of (x[i], y[i]). The arrays x[], and y[] contain values in the above ranges, and no scaling calculation is needed.

The last two functions restore the oldDomain, and frees myDomain.

```
-------------------------------------------------------------------------------------------
int i;
CPDFplotDomain *myDomain, *oldDomain;

myDomain = cpdf_createPlotDomain( 1.6*inch, 5.5*inch, 4.5*inch, 3.0*inch,
      0.1, 100.0, 0.0, 80.0, LOGARITHMIC, LINEAR, 0);
oldDomain = cpdf_setPlotDomain(pdf, myDomain);  /* save the old one */
cpdf_fillDomainWithRGBcolor(myDomain, 1.0, 0.9, 0.9);  /* light pink */
cpdf_drawMeshForDomain(myDomain);

cpdf_setgrayStroke(pdf, 0.0);
cpdf_setrgbcolorFill(pdf, 1.0, 1.0, 0.0);   /* yellow inside */
for(i=0; i < Npoints; i++)
   cpdf_marker(pdf, x[i], y[i], 0, 12.0);

cpdf_setPlotDomain(pdf, oldDomain);    /* restore previous plot domain */
cpdf_freePlotDomain(myDomain);  /* deallocate the plot domain */
-------------------------------------------------------------------------------------------
```

void **cpdf_clipDomain**(CPDFplotDomain *aDomain);

This function clips subsequent drawing to within the plot domain “aDomain.” In order to remove clip path, you should first call cpdf_gsave() before calling this function, then draw within a clipping area. Finally, when clipping is not needed any longer, remove the clipping path by calling cpdf_grestore(). It is useful for restricting plots to within the defined domain.

EXAMPLE: (from DomainDemo.c)

```
myDomain = cpdf_createPlotDomain(
        1.5*inch, 6.0*inch, 4.5*inch, 3.0*inch,
        xmin, xmax, ymin, ymax,
        LOGARITHMIC, LOGARITHMIC, 0);
oldDomain = cpdf_setPlotDomain(pdf, myDomain);
cpdf_gsave(pdf);                /* to later undo clipping */
cpdf_clipDomain(myDomain);
plot_Curve(pdf);                /* Plot it */
cpdf_grestore(pdf);             /* remove clipping */
```

CPDFplotDomain ***cpdf_createPlotDomain**(float x, float y, float w, float h,
float xL, float xH, float yL, float yH,
int xtype, int ytype, int reserved);

(REQUIRED)
This function creates a new plot domain of size (w, h) points with its bottom-left corner located at (x, y) in points (1/72 inches). The X value range [xL .. xH] is scaled to fit the width dimension of the domain. The Y value range [yL .. yH] is mapped to the height dimension of the domain. Currently, linear or logarithmic domains are supported for X and Y dimensions. “Xtype” and “ytype” shold be one of LINEAR or LOGARITHMIC (defined in cpdflib.h). The last argument “reserved” is reserved, and should be set to zero. The function returns a structure “CPDFplotDomain.”

CPDFplotDomain ***cpdf_createTimePlotDomain**(float x, float y, float w, float h,
struct tm *xTL, struct tm *xTH, float yL, float yH,
int xtype, int ytype, int reserved);

(REQUIRED)
Date/time domain needs a special treatment in order to allow specification of the time value in natural calender date/time using the standard structure “tm” as

defined in “man 3 ctime.” Otherwise, the function is the same as cpdf_createPlotDomain(). Only the X dimension is allowed to be date/time. The X range is specified by [xTL .. xTH] where xTL and xTH are pointers to structure tm ad defined below in <time.h>. “Xtype” must be TIME. “Ytype” may be either LINEAR or LOGARITHMIC.

```
struct tm {
    int tm_sec;     /* seconds after the minute (0-59) */
    int tm_min;     /* minutes after the hour (0-59) */
    int tm_hour;    /* hours since midnight (0-23) */
    int tm_mday;    /* day of the month (1-31) */
    int tm_mon;     /* months since January (0-11) */
    int tm_year;    /* years since 1900 */
    int tm_wday;    /* days since Sunday (0-6) */
    int tm_yday;    /* days since Jan. 1 (0-365) */
    int tm_isdst;   /* flag; daylight savings time in effect */
    long tm_gmtoff; /* offset from GMT in seconds */
    char *tm_zone; /* abbreviation of timezone name */
};
```

Note: **xTL->tm_isdst** and **xTH->tm_isdst** must be initialized in addition to the first 6 elements of struct tm. In the U.S., it should probably be set to -1. Strange things will happen if **tm_isdst** is not initialized.

SEE ALSO: **cpdf_createTimeAxis**(), **mktime**(), and "man 3 ctime"

void **cpdf_freePlotDomain**(CPDFplotDomain *aDomain);

(REQUIRED)
This function must be called when a plot domain is no longer needed. You must explicitly free all plot domains you create. Otherwise, memory leaks will result in a long running application.

CPDFplotDomain ***cpdf_setPlotDomain**(CPDFdoc *pdf, CPDFplotDomain *aDomain);

(REQUIRED)
This will set the domain passed as the argument as the current plot domain. It returns the old plot domain, so that you can save it, and later restore it.

You may create multiple plot domains simultaneously and switch domains using this function, drawing objects in different domains in an interleaved manner. Do not switch domains during a path construction, however. Domain switching should take place at natural breaks in drawing operations, such as immediately after a stroke operation and before a new path construction begins.

void **cpdf_fillDomainWithGray**(CPDFplotDomain *aDomain, float gray);
void **cpdf_fillDomainWithRGBcolor**(CPDFplotDomain *aDomain, float r, float g, float b);

This function paints the domain with a given gray [0 .. 1] or a RGB color. This makes the domain opaque. If you do not use this function, the domain is transparent, and marks painted before underneath will show through.

void **cpdf_setMeshColor**(CPDFplotDomain *aDomain,
float meshMajorR, float meshMajorG, float meshMajorB,
float meshMinorR, float meshMinorG, float meshMinorB);

This function sets the color used for grid/mesh pattern for the plot domain.

void **cpdf_drawMeshForDomain**(CPDFplotDomain *aDomain);

Draws the grid/mesh pattern that gives the appearance of graph paper for the plot domain.

void **cpdf_setLinearMeshParams**(CPDFplotDomain *aDomain, int xy,
float mesh1ValMajor, float intervalMajor,
float mesh1ValMinor, float intervalMinor);

(OPTIONAL)
Overrides the default mesh parameters. **Mesh1ValMajor** and **mesh1ValMinor** set the values of the first major and minor mesh lines for one of the dimensions of the domain. If (xy == 0), the function sets the horizontal dimension parameters, while if( xy != 0) it sets the parameters for the vertical dimension. The arguments, **intervalMajor** and **intervalMinor** specify the intervals of major and minor mesh lines. Default values are set when a domain is created. This function is necessary only when you wish to override the default values.

SEE ALSO: **cpdf_setLinearAxisParams**()

void **cpdf_suggestMinMaxForLinearDomain**(float vmin, float vmax, float *recmin, float *recmax);

(UTILITY)
Given minimum and maximum values of the data as (**vmin**, **vmax**), it will return suggested min and max values for the domain and axes in (**recmin**, **recmax**). The values returned are the same as those from the next function below. This function returns only a subset of information provided by **cpdf_suggestLinearDomainParams**().

void **cpdf_suggestLinearDomainParams**(float vmin, float vmax, float *recmin, float *recmax,
float *tic1ValMajor, float *intervalMajor,
float *tic1ValMinor, float *intervalMinor);

(UTILITY)
This function computes suggested values to be set via the preceding function **cpdf_setLinearMeshParams**(). Given minimum and maximum values of the data as (**vmin**, **vmax**), it will return suggested min and max values for the domain and axes in (**recmin**, **recmax**), the values for the first major and minor mesh line positions, and mesh line intervals. These values are also appropriate for setting linear axis parameters.

SEE ALSO: **cpdf_setLinearAxisParams**()

void **cpdf_suggestTimeDomainParams**(struct tm *xTL, struct tm *xTH, struct tm *recTL, struct tm *recTH);

(UTILITY)
Given (xTL, xTH) for starting and ending values, respectively, for time domain and axis in the data, this function suggests "good" values for use with cpdf_createTimeAxis() and cpdf_createTimePlotDomain(), and returns them in (recTL, recTH) as new starting and ending values. All structs "tm" must be allocated by the calling function. See weather/weather.c for usage examples.

SEE ALSO:
cpdf_createTimeAxis(), cpdf_createTimePlotDomain(), tm_to_NumDays().

float **x_Domain2Points**(CPDFdoc *pdf, float x);
float **y_Domain2Points**(CPDFdoc *pdf, float y);

(UTILITY)
Once a domain is set as the current domain, these functions may be used to perform conversion of a given point in the domain (x, y) into raw coordinate values in points (1/72 inches). Separate functions are provided for X and Y dimensions. These functions work correctly for LINEAR and LOGARITHMIC domains, but not for a TIME domain. For TIME domain, use **tm_to_NumDays**() described in the "**17. Miscellaneous Functions**" section to first convert a time difference to a float value which, in turn, may be passed to x_Domain2Points().

SEE ALSO: **tm_to_NumDays**()

## 10. Axis Functions

Axis is an object that is attached to a plot domain, and clarifies the meaning of each dimension of the domain. Usually, horizontal and vertical axes are attached a plot domain to indicate the scaling and values of data plotted in the domain. (However, it is possible to plot data in a domain without using any axis.) An axis object in ClibPDF allows flexible tick marks, numbers at selected ticks, and an axis label which is automatically centered along the axis. Two main types of axes are supported: LINEAR and LOGARITHMIC. In addition, a TIME axis may be created via a separate function: cpdf_createTimeAxis(). The following example illustrates a typical case of attaching a linear horizontal axis to a plot domain (from example: DomainDemo.c in function do_LinearLinear()). None of the axis-related functions uses the new PDF document context, CPDFdoc *pdf, as axes are always attached to a plot domain, and the plot domain knows which PDF document it belongs to.

```
---------------------------------------------------------------------------------------------
CPDFplotDomain *myDomain, *oldDomain;
CPDFaxis *xAxis, *yAxis;
        myDomain = cpdf_createPlotDomain(
                1.5*inch, 1.5*inch, 4.5*inch, 3.0*inch,
                0.0, xmax, 0.0, ymaxL, LINEAR, LINEAR, 0);
        oldDomain = cpdf_setPlotDomain(pdf, myDomain);/*  for later restore */
        cpdf_drawMeshForDomain(myDomain);

        /* X-Axis */
        xAxis = cpdf_createAxis( 0.0, 4.5*inch, LINEAR, 0.0, xmax);
        cpdf_attachAxisToDomain(xAxis, myDomain, 0.0, -0.2*inch);
        cpdf_setAxisNumberFormat(xAxis, "%g", "Helvetica", 16.0);
        cpdf_setAxisLabel(xAxis, "X axis (linear)",
                "Times-Roman", "MacRomanEncoding", 20.0);
        cpdf_drawAxis(xAxis);
        cpdf_freeAxis(xAxis);
---------------------------------------------------------------------------------------------
```

Here, a linear-linear domain (myDomain) is first created, set as the current plot domain, and a mesh pattern is drawn. Then, a linear axis is created using the same length as the domain's horizontal dimension (4.5 inches), and the same limit values [0 .. xmax]. It is then attached to "myDomain" with a small vertical offset so that the axis (tick marks) will lie slightly below the plot domain. The offset may be zero, in which case, the axis will be drawn exactly on the bottom edge of the domain. Using a positive value as the offset, the axis may be drawn anywhere inside the domain itself if desired (e.g., 4-quadrant plotting). Number format, and the label attributes are then set. Finally, the axis is drawn by cpdf_drawAxis(), and then freed. A similar sequence of functions may be used to draw a vertical axis as well. For the vertical axis, the first argument (angle of axis) should be 90.0.

CPDFaxis ***cpdf_createAxis**(float angle, float axislength, int typeflag, float valL, float valH);

(REQUIRED)
This function creates an axis of either LINEAR or LOGARITHMIC type, specified as “typeflag.” Standard horizontal or vertical axis may be created by specifying the “angle” as either 0.0 or 90.0 degrees, respectively. “Axislength” is the length of axis in points (1/72 inches). The range of values for the axis is specified by "valL" and "valH."

CPDFaxis ***cpdf_createTimeAxis**(float angle, float axislength, int typeflag, struct tm *vTL, struct tm *vTH);

(REQUIRED)
This function is similar to cpdf_createAxis(), except that it is specifically for creating a time axis using calendar time. "Typeflag" therefore must always be TIME (as defined in cpdflib.h). The range of values for the axis is specified by *vTL and *vTH, which are pointers to a "struct tm" (see below, <time.h>, or "man 3 ctime"). The function **cpdf_suggestTimeDomainParams**() should be used to pick good starting and ending values for the time axis.

Note: **vTL->tm_isdst** and **vTH->tm_isdst** must be initialized in addition to the first 6 elements of struct tm. In the U.S., it should probably be set to -1. Strange things will happen if **tm_isdst** is not initialized.

```
struct tm {
    int tm_sec;     /* seconds after the minute (0-59) */
    int tm_min;     /* minutes after the hour (0-59) */
    int tm_hour;    /* hours since midnight (0-23) */
    int tm_mday;    /* day of the month (1-31) */
    int tm_mon;     /* months since January (0-11) */
    int tm_year;    /* years since 1900 */
    int tm_wday;    /* days since Sunday (0-6) */
    int tm_yday;    /* days since Jan. 1 (0-365) */
    int tm_isdst;   /* flag; daylight savings time in effect */
    long tm_gmtoff; /* offset from GMT in seconds */
    char *tm_zone; /* abbreviation of timezone name */
};
```

SEE ALSO: **cpdf_createTimePlotDomain**(), **mktime**() [man 3 ctime]

EXAMPLE: See Fig. 8 for examples of time/date axes generated by timeaxis.c.

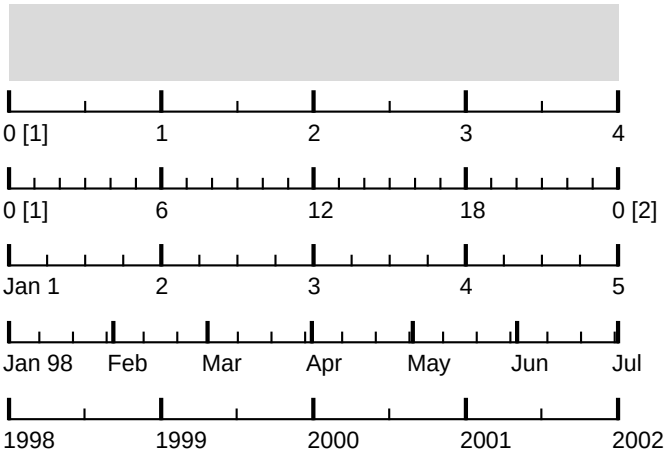



**Fig. 8:** Examples of time/date axes are illustrated. In all 5 axes, the axis starts at Jan. 1, 1998 00:00. From top to bottom, the axis spans 4 hours, 24 hours, 5 days, 6 months, and 4 years, respectively. The time axis tries to display tick marks and numbers sensibly without any effort on your part. In the top two axes, the number in [ ] is day of the month. Carry to the higher order item, e.g., new month when displaying days, causes the display of the new item. As indicated by the bottom axis, the time axis handles time after year 2000 correctly. See **timeaxis.c** in the examples directory for the source code.

void **cpdf_attachAxisToDomain**(CPDFaxis *anAx, CPDFplotDomain *domain, float x, float y);

(REQUIRED)

Once an axis is created, it must be attached to a plot domain. A pointer to a CPDFaxis structure, *anAx, returned from cpdf_createAxis() or cpdf_createTimeAxis(), should be passed as the first argument. The second pointer should be a pointer to a domain to which the axis is to be attached. The last two arguments, (x, y) specify (in points, 1/72 inches) an offset from the bottom-left corner of the domain to the starting point of the axis. With these, the axis may be positioned anywhere; on the edges if (x, y) = (0.0, 0.0), slightly below or to the left of the domain, in the middle of the domain, at the top, or on the right-hand side of the domain. It is also possible to attach more than one axes for each dimension of the domain.

It is possible to re-attach a single axis to another domain after it has been used (drawn by cpdf_drawAxis()) for one domain, if these domains have identical dimensions appropriate for the axis. Re-attaching an axis to the second domain automatically removes it from the first.

void **cpdf_drawAxis**(CPDFaxis *anAx);

(REQUIRED)

Draws an axis passed as the argument.

void **cpdf_freeAxis**(CPDFaxis *anAx);

(REQUIRED)
Eventually, all axes must be freed after they are no longer needed by calling cpdf_freeAxis(). Otherwise, a memory leak will result in a persistent application. It is *not* necessary to detach an axis from the previously attached domain.

void **cpdf_setAxisLineParams**(CPDFaxis *anAx, float axLineWidth, float ticLenMaj, float ticLenMin,
float tickWidMaj, float tickWidMin);

(OPTIONAL)
This function allows customization of major and minor tick lengths, and axis line width. All arguments should be specified in points (1/72 inches).

void **cpdf_setTicNumEnable**(CPDFaxis *anAx, int ticEnableMaj, int ticEnableMin, int numEnable);

(OPTIONAL)
Sets flags that controls whether major and minor tick marks and numbers on the axis are ON or OFF. Specify a non-zero value to enable, and zero to disable.

void **cpdf_setAxisTicNumLabelPosition**(CPDFaxis *anAx, int ticPos, int numPos,
int horizNum, int horizLabel);

(OPTIONAL)
Sets the position of tick marks, numbers with respect to the axis line.
For **ticPos**, and **numPos**:

0: clock-wise position (e.g., below a horizontal axis)
1: centered on the axis line (only for tick marks)
2: counter-clock-wise position (e.g. above a horizontal axis or to the left of a vertical axis)

Non-zero **horizNum** and **horizLabel** cause numbers and axis labels to be displayed horizontally regardless of the axis orientation (even at oblique angles), respectively. If they are zero, numbers and labels rotate with the axis. That is, for a vertical axis (90 degrees), numbers and the label are displayed vertically in the bottom-to-top direction.

void **cpdf_setAxisNumberFormat**(CPDFaxis *anAx, char *format, char *fontName, float fontSize);

(OPTIONAL)
Sets the font and size (in points) of the numbers plotted along the axis.

EXAMPLE:
cpdf_setAxisNumberFormat(yAxis, "%2.f", "Helvetica", 16.0);

void **cpdf_setTimeAxisNumberFormat**(CPDFaxis *anAx, int useMonName, int use2DigYear,
char *fontName, float fontSize);

(OPTIONAL)
Sets the font and the font size (in points) of the numbers (and month names) displayed along the axis. In addition, **useMonName** allows you to choose between a month name display (Jan, Feb, Mar, ...) and numerical month display (1 .. 12). A non-zero value specifies the month name display. The flag, **use2DigYear**, when non-zero, causes displays of year in the 2-digit form (98, 99, 00, 01, ...). If it is zero, full 4 digit year is displayed. Default setting is: useMonName = 1, use2DigYear = 1.

void **cpdf_setAxisLabel**(CPDFaxis *anAx, char *labelstring, char *fontName, char *encoding, float fontSize);

(OPTIONAL)
Sets the axis label, its font and font encoding, and font size (in points).
If this function is not called, no label is displayed for the axis.

void **cpdf_setLinearAxisParams**(CPDFaxis *anAx, float tic1ValMajor, float intervalMajor,
float tic1ValMinor, float intervalMinor);

(OPTIONAL)
Sets the value that corresponds to the first major and minor tick mark in the unit of that dimension, and the major and minor tick intervals. Axis tries to set these values automatically, but this function allows you to override them.

SEE ALSO: **cpdf_setLinearMeshParams**(), **cpdf_setLinearMeshParams**(),
**cpdf_suggestLinearDomainParams**()

void **cpdf_setLogAxisTickSelector**(CPDFaxis *anAx, int ticselect);

(OPTIONAL)
By default, ticks marks on logarithmic axis are displayed for values with exact one-significant digit numbers (e.g., 0.8, 0.9, 1, 2, 3, ... 9, 10, 20, ...). This may be changed by this function. For example, to display tick marks only for each decade (0.1, 1, 10, 100, ...), do:

```
cpdf_setLogAxisTickSelector(anAxis, LOGAXSEL_1);
```

LOXAXSEL_1 is defined in <cpdflib.h> as 0x0002, which corresponds to bit 1 of **ticselect**. Non-zero bit, specified according to Table 1, enables corresponding tick mark. Thus, the default setting is LOGAXSEL_123456789, 0x03FE.

**TABLE 1.** Tick mark and number enable flags for logarithmic axis

| Bit (LSB) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tick / Number | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | - |

void **cpdf_setLogAxisNumberSelector**(CPDFaxis *anAx, int numselect);

(OPTIONAL)
By default, numbers on logarithmic axis are displayed for each decade (i.e., 0.1, 1, 10, 100, ..). This may be changed by this function. For displaying 1's and 3's, do:

```
cpdf_setLogAxisNumberSelector(anAxis, LOGAXSEL_13);
```

As with logarithmic tick marks, determine the value of numselect according to Table 1 above.

## 11. Data Marker Functions

void **cpdf_marker**(CPDFdoc *pdf, float x, float y, int markertype, float size);
void **cpdf_pointer**(CPDFdoc *pdf, float x, float y, int direction, float size);
void **cpdf_rawMarker**(CPDFdoc *pdf, float x, float y, int markertype, float size);
void **cpdf_rawPointer**(CPDFdoc *pdf, float x, float y, int direction, float size);

These four functions provide symbols that indicate data points. The functions prefixed by "cpdf_" accepts (**x, y**) in the current plot domain coordinate system, whereas those prefixed by "cpdf_raw" takes (x, y) in raw point (1/72 inches) coordinate system. "***Size***" should be specified in points. These markers and pointers have separate stroke and fill colors. Therefore, each type may become an open symbol if different stroke and fill colors are used, and become a filled symbol if the two colors are the same.

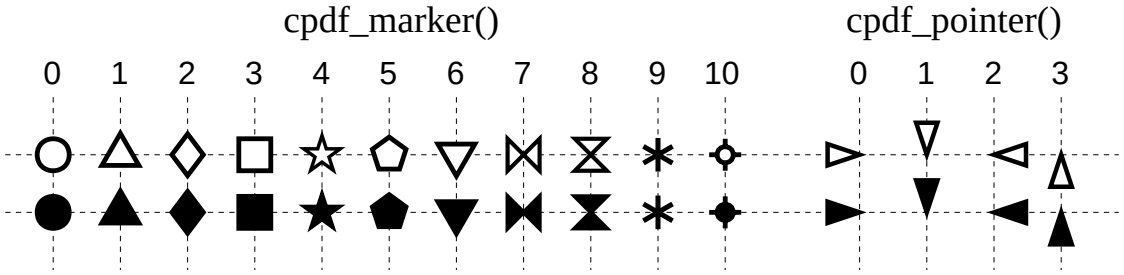



**Fig. 9**: Markers and pointers available in ClibPDF. Most markers may be open or filled by defining fill and stroke colors appropriately. Markers and pointers shown are 12 points in size. See **MarkerTest.c** in the examples/marker directory for the source code.

void **cpdf_errorbar**(CPDFdoc *pdf, float x, float y1, float y2, float capsize);
void **cpdf_rawErrorbar**(CPDFdoc *pdf, float x, float y1, float y2, float capsize);

Draws an error bar at the X position **x**, extending between two Y positions **y1**, and **y2**. **Capsize** is the length (in points) of the horizontal bar at both ends of the error bar. To set the line width of error bars, use cpdf_setlinewidth() prior to calling this function.

void **cpdf_highLowClose**(CPDFdoc *pdf, float x, float vhigh, float vlow, float vclose, float ticklen);
void **cpdf_rawHighLowClose**(CPDFdoc *pdf, float x, float vhigh, float vlow, float vclose, float ticklen);

Draws a high, low, and close value marker typically used for displaying daily stock prices. **X** specifies the position of the vertical high-low bar. **Vhigh**, **vlow**,

and **vclose** are high, low and close values, respectively. **Ticklen** specifies the length of the close value indicator in points (1/72 inches). To set the line width of this marker, use cpdf_setlinewidth() prior to calling this function.

## 12. Image Functions

int **cpdf_importImage**(CPDFdoc *pdf, const char *imagefile, int type, float x, float y, float angle,
float *width, float *height, float *xscale, float *yscale, int gsave);
int **cpdf_rawImportImage**(CPDFdoc *pdf, const char *imagefile, int type, float x, float y, float angle,
float *width, float *height, float *xscale, float *yscale, int gsave);

Imports an image from an external file "imagefile" with scaling and optional rotation. "**Type**" flag specifies the image format as defined in <cpdflib.h>. It should be one of **JPEG_IMG**, **TIFF_IMG**, and **CPDF_IMG** for image formats JPEG, TIFF, and PDFIMG, respectively. The standard ClibPDF available publicly for download supports only the JPEG format (Baseline JPEG only, progressive JPEG files are not supported). **Premium ClibPDF** package (available at additional cost to licensees) contains additinal support for TIFF and PDFIMG image formats. TIFF is a popular image format that is supported by almost any application that handle image data. The PDFIMG format is a custom image format designed by FastIO Systems that is optimized for fast importing of image data into PDF streams. The PDFIMG format eliminates image decoding, re-encoding overhead for applications that imports the same set of images repeatedly (such as form image in Web form application). A utility command program is included in the Premium ClibPDF package for converting TIFF files into PDFIMG format. JPEG is a "lossy" image format that achieves high degree of image compression mainly for natual color images in exchange for some degree of image degradation. It is not suitable for representing scanned text, forms, and line drawings. Please contact FastIO Systems (fastio@fastio.com) for more details on the Premium ClibPDF package.

These two functions are identical except for the domain in which the coordinate system for (x, y) is defined. With cpdf_importImage(), (x, y) are defined in the current domain coordinate system. However, width and height should still be defined in points. With cpdf_rawImportImage(), all of (x, y, width, height) are defined in points in the default coordinate system. "Angle" specifies the rotation angle in degrees. The image is rotated with the center of rotation at (x, y).

When you import an image file, you may not know the image dimensions (in pixels). For each X or Y dimension, specify either size (in points) or scaling factor and set the unspecified one to zero.

The unspecified variable will be set upon return (that's why these parameters are passed by address). If both are set, width and height will take precedence over scaling factors. For example, if the image file "/tmp/myimage.jpg" has the dimension of 150x100 pixels, a call:

```
width = 0.0;
```

```
height = 200.0;
xscale = 0.0;
yscale = 0.0;
cpdf_rawImportImage(pdf, "/tmp/myimage.jpg", JPEG_IMG,
        100.0, 100.0, 0.0, &width, &height, &xscale, &yscale, 1);
```

will return with width = 300, and xscale = yscale = 2.0, while the aspect ratio is preserved. If the ratio of width/height does not match the aspect ratio of the original image, the aspect ratio will not be preserved. If none of the image size parameters is specified (i.e., all of them are set to 0), the pixel size of the image is used by counting each pixel as one point (1/72 inch).

***Important Note:***
When the last argument "***gsave***" is non-zero, the call internally brackets image import with a cpdf_gsave() and cpdf_grestore() pair. For most applications, it should be non-zero, e.g., =1. If the last argument is zero, the function does not perform gsave/grestore internally to allow flexibility having an arbitrary path clip the image or to draw image inside text (See Example 14.2, page 415 in the PDF Reference Manual version 1.3).

Only a single copy of the data for each unique image is included in a PDF file. Therefore, if an image is used multiple times in a document, the image data are shared among all instances of the same image, even if they are displayed with different sizes and orientations. The uniqueness of the image is determined by string comparisons of paths to the image file. Therefore, do not use a single path repeatedly as a scratch pad for importing different images on the fly.

int **cpdf_placeInLineImage**(CPDFdoc *pdf, const void *imagedata, int length,
    float x, float y, float angle, float width, float height,
    int pixwidth, int pixheight, int bitspercomp, int CSorMask, int gsave);
int **cpdf_rawPlaceInLineImage**(CPDFdoc *pdf, const void *imagedata, int length,
    float x, float y, float angle, float width, float height,
    int pixwidth, int pixheight, int bitspercomp, int CSorMask, int gsave);

These two functions place the image data (previously computed or loaded from file) into the current content stream for the page. In-line images differ from those that are imported by **cpdf_importImage**() in that each instance of an in-line image will take up additional space in the PDF file; there is no data sharing even if the same data are used repeatedly on the same or different pages. They should be used to include only small images (less than several kbytes). (***x, y***) specify the lower-left corner position of the image in the current domain coordinate system for the former, and in raw points for the latter. (***width, height***) should always be given in points. "***imagedata***" is a pointer to the image data buffer, and "***length***" the data length in bytes. ***Pixwidth***, and ***pixheight*** specify the size of the image data in the number of pixels in the width

and height dimensions, respectively. "***Bitspercomp***" is the number of bits per color component(s). "***Gsave***" should normally be 1, and has the same meaning as that described for file import function **cpdf_importImage**() above. "***CSorMask***" should be one of the following:

```
#define     IMAGE_MASK      0
#define     CS_GRAY         1
#define     CS_RGB          2
#define     CS_CMYK         3
```

### 13. Annotation, Web-Link, and Hyper-Link Functions

NOTE: The API for these annotation functions has changed from ClibPDF version 1.02 to 1.10. Please note the last argument **CPDFannotAttrib *attr**, has been added to give more control over details of annotions such as color and border style. To maintain the previous behavior and minimize necessary changes, please use **NULL** as its value. The CPDFannotAttrib is defined as a struct below which you must allocate and assign values to its members:

```
typedef struct {
    int flags;          /* see page 83 of the PDF Reference Manual (v1.3) */
    char *border_array; /* [horiz_corner_radius vert_corner_radius width dash_array] */
    char *BS;           /* content of BS (border style dict). This overrides border_array if not NULL */
    float r;            /* RGB color components */
    float g;
    float b;
} CPDFannotAttrib;
```

"**Flags**" may be one or more of the following OR'ed together (see example below):

| | |
|---|---|
| AF_INVISIBLE | (specifies appearance wrt. annotation handler)(LSB) |
| AF_HIDDEN | (if bit set, annotation is hidden) |
| AF_PRINT | (if set, annotation will be printed) |
| AF_NOZOOM | (if set, annotation will not zoom when page is zoomed) |
| AF_NOROTATE | (if set, annotation does not rotate with page) |
| AF_NOVIEW | (if set, annotation is used only for printing) |
| AF_READONLY | (if set, user cannot interact with annotation) |

For example, the CPDFannotAttrib struct may be used as follows:

```
CPDFannotAttrib attrib;
attrib.flags = AF_NOZOOM | AF_NOROTATE | AF_READONLY;
attrib.border_array = "[0 0 1 [4 2]]";
attrib.BS = NULL;
attrib.r = 1.0;
attrib.g = 1.0;
attrib.b = 0.0;
cpdf_setAnnotation(pdf, 0.2, 0.2, 3.0, 1.0, "Title", annotstr, &attrib);
```

void **cpdf_setAnnotation**(CPDFdoc *pdf, float xll, float yll, float xur, float yur, const char *title,
const char *annotstr, CPDFannotAttrib *attr);
void **cpdf_rawSetAnnotation**(CPDFdoc *pdf, float xll, float yll, float xur, float yur, const char *title,
const char *annotstr, CPDFannotAttrib *attr);

These functions insert a text annotation into the current page with the lower-left corner location at (xll, yll), and the upper-right corner's location at (xur, yur). Strings **title**, and **annotstr** sets the title and content of the annotation. The "raw" version accepts the coordinate values in the point-based coordinate system, while the former interprets them according to the current domain.

int **cpdf_includeTextFileAsAnnotation**(CPDFdoc *pdf, float xll, float yll, float xur, float yur,
const char *title, const char *filename, CPDFannotAttrib *attr);
int **cpdf_rawIncludeTextFileAsAnnotation**(CPDFdoc *pdf, float xll, float yll, float xur, float yur,
const char *title, const char *filename, CPDFannotAttrib *attr);

These functions place a text annotation by loading its content from a file (*const char *filename*) instead of a string. Ohterwise, the arguments are identical to those above.

void **cpdf_setActionURL**(CPDFdoc *pdf, float xll, float yll, float xur, float yur,
const char *linkspec, CPDFannotAttrib *attr);
void **cpdf_rawSetActionURL**(CPDFdoc *pdf, float xll, float yll, float xur, float yur,
const char *linkspec, CPDFannotAttrib *attr);

These functions places an active hyper-link area for the rectangle with the lower-left corner location at (xll, yll), and the upper-right corner location at (xur, yur). The "***linkspec***" should be a string specifying a standard URL (Uniform Resource Locator) used in World Wide Web address specification (e.g., "http://www.adobe.com/acrobat/" or "mailto:someone@somedomain.com"). The PDF Reference Manual calls this URI (I for Identifier), but here, we follow the prevailing convention. The default Web browser or e-mail application will be used to access the URL. See **examples/cover/cover.c** for usage examples.

void **cpdf_setLinkGoToPage**(CPDFdoc *pdf, float xll, float yll, float xur, float yur, int page,
const char *fitmode, CPDFannotAttrib *attr);
void **cpdf_rawSetLinkGoToPage**(CPDFdoc *pdf, float xll, float yll, float xur, float yur, int page,
const char *fitmode, CPDFannotAttrib *attr);

These functions places an active hyper-link area for the rectangle with the lower-left corner location at (xll, yll), and the upper-right corner location at (xur,

yur). The hyper-link will cause jump to another page within the same PDF document. "**Page**" should be the integer used in cpdf_pageInit(). "**Fitmode**" is a string specifying how target page should be fit to the view, as specified in Table 7.1 on page 184 of the PDF Reference Manual (version 1.3). This string should contain the part after the page, e.g., after "[3 0 R", and before the closing array bracket "]." For example, it may be "/FitR 10 20 300 50" which will fit the rectangle (left, bottom, right, top) = (10, 20, 300, 50) in the window. The following fitmodes are available:

```
/XYZ left top zoom          (zoom=1.0 is normal size)
/XYZ null null null         (retains current "fitmode")
/Fit
/FitH top
/FitV left
/FitR left bottom right top
/FitB                       (fit bounding box to window) PDF-1.1
/FitBH top                  (fit width of bbox) PDF-1.1
/FitBV left                 (fit height of bbox) PDF-1.1
```

void **cpdf_setLinkAction**(CPDFdoc *pdf, float xll, float yll, float xur, float yur,
const char *action_dict, CPDFannotAttrib *attr);
void **cpdf_rawSetLinkAction**(CPDFdoc *pdf, float xll, float yll, float xur, float yur,
const char *action_dict, CPDFannotAttrib *attr);

This function specifies an "**Action**" type link annotation as specified on page 90 (Table 6.15) of the PDF Reference Manual (version 1.3). Do not use action subtype "GoTo" because it requires referring target page by PDF object number. For that purpse, use **cpdf_setLinkGoToPage**() as defined above.

Some examples of actions are described in "16. Outlines (book marks)" on page 51 of this manual. In general, the action dictionaries may be used for both hyperlink annotations and outlines. For further details of actions, you must consult action specifications as documented in the PDF Reference Manual (v1.3, page 107). One limitation of this function is that it does not allow actions that require additional objects that are specified by indirect reference, or an "action_dict" string longer than 8 kbytes. In other words, "action_dict" must be self-contained, short, and cannot contain binary data, e.g., sound or movie streams.

## 14. Memory Stream Functions

A memory stream allows you to write to memory in a manner similar to the way you write to a file stream using fprintf(), fwrite(), fputs(), etc. The memory buffer is expanded as needed automatically, so that you do not have to worry about buffer over-runs (which result in trashed memory, segmentation fault, nasty bugs and security holes). At any time, you can obtain the entire buffer content and the length of the content. Memory stream functions are not directly related to PDF generation, but ClibPDF extensively uses it internally. The API for the memory stream functions is exposed because its functionality may be useful in general. None of the memory stream functions use the PDF document context argument, CPDFdoc *pdf.

EXAMPLE:

```
char *string1 ="0123456789 This is a test of memory stream 0123456789";
void main(void) {
char sbuf[256];
int i;
CPDFmemStream *memStream = { NULL };
char *memBuffer;
int memLen, bufSize;
  memStream = cpdf_openMemoryStream();
  cpdf_memPuts("Hello World!\n");
  for(i=0; i<5000; i++) {
       sprintf(sbuf, "%05d - %s\n", i, string1);
       cpdf_writeMemoryStream(memStream, sbuf, strlen(sbuf));
  }
  cpdf_getMemoryBuffer(memStream, &memBuffer, &memLen, &bufSize);
  fprintf(stderr,"memLen=%d,  bufSize=%d\n", memLen, bufSize);
  /* You can use the content of the buffer now. */
  cpdf_saveMemoryStreamToFile(memStream, "memtest.txt");
  cpdf_closeMemoryStream(memStream);
}
```

CPDFmemStream ***cpdf_openMemoryStream**(void);

Opens a memory stream, allocates an initial buffer of a small size, and returns the pointer to the stream. The stream must eventually be closed and freed by calling **cpdf_closeMemoryStream**().

void **cpdf_closeMemoryStream**(CPDFmemStream *memStream);

Closes the memory stream passed as the argument and frees the buffer memory and the associated resources.

int **cpdf_writeMemoryStream**(CPDFmemStream *memStream, char *data, int len);

Add the data pointed to by "*data" of length "len" bytes to the memory stream.

void **cpdf_getMemoryBuffer**(CPDFmemStream *memStream, char **streambuf, int *len, int *maxlen);

For a given memory stream, it returns the pointer to the buffer, and its current byte count of the content. Upon return, "maxlen" will be set to the current capacity of the buffer that has so far been malloc'ed (realloc'ed). For uses as strings, the buffer content is always null terminated. The terminating null character is not counted in the byte count "len."

Note: Do NOT use **fputs**() or **puts**() to output the buffer if binary data have been written to the memory stream. Incomplete output will result, and any zero valued byte will terminate output. Also note that ****streambuf*** may change from one write to another as the buffer is re-allocated to accommodate more data. Do not assume that the buffer will always be at the same address.

int **cpdf_saveMemoryStreamToFile**(CPDFmemStream *stream, const char *name);

This function writes the current content of the buffer to the file specified. It writes the buffer content in **binary** mode without any end-of-line character translation. If a given platform requires this translation for text files (MacOS8/ Windows), do not use this function. Instead, write the buffer obtained by **cpdf_getMemoryBuffer**() to a text-mode FILE stream. PDF files are binary, therefore, this is not a concern in the context of ClibPDF.

int **cpdf_memPutc**(int ch, CPDFmemStream *memStream);

Writes a single character to the memory stream, as in fputc().

int **cpdf_memPuts**(char *str, CPDFmemStream *memStream);

Writes a string to the memory stream, as in fputs().

void **cpdf_clearMemoryStream**(CPDFmemStream *aMstrm);

Clears the buffer content (stores '\0' at the head of the buffer) and resets the current buffer content length ("len" in cpdf_getMemoryBuffer()) to zero. It keeps the currently allocated buffer as is (i.e., it does not shrink its size).

## 15. Document Attribute Functions

void **cpdf_setCreator**(CPDFdoc *pdf, cont char *pname);
void **cpdf_setTitle**(CPDFdoc *pdf, const char *pname);
void **cpdf_setSubject**(CPDFdoc *pdf, const char *pname);
void **cpdf_setKeywords**(CPDFdoc *pdf, const char *pname);

These functions set strings to be contained in the Info object of the PDF document. The first 3 functions will truncate strings at 62 chars, while key words string may be up to 120 chars.

void **cpdf_setViewerPreferences**(CPDFdoc *pdf, CPDFviewerPrefs *vP);

This function sets the initial appearance of a PDF document when it is first opened. For example, it is possible to open the document with the outline view already visible. The second argument, *vP, is a pointer to a struct:

```
typedef struct {
   int pageMode;          /* This really belongs directly to Catalog obj, but here for convenience */
   int hideToolbar;       /* when YES, tool bar in viewer will be hidden */
   int hideMenubar;       /* when YES, menu bar in viewer will be hidden */
   int hideWindowUI;      /* when YES, GUI elements in doc window will be hidden */
   int fitWindow;         /* when YES, resize the doc window to the page size */
   int centerWindow;      /* when YES, move the doc window to the screen's center */
   int pageLayout;        /* Specifies 1-page display or two-page columnar displays */
   int nonFSPageMode;     /* Specifies pageMode coming out of the full-screen display mode */
} CPDFviewerPrefs;
```

For **pageMode** and **nonFSPageMode**

```
   PM_NONE          0     /* default - no outline or thumbnails */
   PM_OUTLINES      1     /* open with outlines visible */
   PM_THUMBS        2     /* open with thumbnails visible */
   PM_FULLSCREEN    3     /* open in full screen mode */
```

**nonFSpmode** specifies the mode that is assumed when exiting the full-screen mode. Obviously, PM_FULLSCREEN is not allowed for nonFSpmode.

For **pageLayout**

```
   PL_SINGLE        0     /* default - one page at a time */
   PL_1COLUMN       1     /* display pages in one column */
   PL_2LCOLUMN      2     /* 2-columns, odd pages left */
   PL_2RCOLUMN      3     /* 2-column display, odd pages right */
```

All other parameters should be either non-zero, or zero (YES or NO).

These are somewhat obscure, but are documented on page 69, Table 6.2 of the Adobe PDF Reference Manual version 1.3.

Typically, if you just wish to display the outline (book mark) view to be visible upon file opening, use:

**cpdf_setViewerPreferences**();

*after* **cpdf_open**() and *before* **cpdf_finalizeAll**().

## 16. Outine (Book Mark) functions

ClibPDF supports multi-level outline (book marks) with flexible control of the attributes of outline entries, including outline entries that causes actions. One limitation of the current version is that outline entries must be added in the top-to-bottom order, (i.e., in the order they will appear in the outline view). This version does not support insertion of an outline entry in the middle of an outline tree, or deletion of entries. See outline/outline.c for an example of 3-level outline tree. The number of outline levels and the number of outline entries are essentially unlimited.

CPDFoutlineEntry ***cpdf_addOutlineEntry**(CPDFdoc *pdf, CPDFoutlineEntry *afterThis, int sublevel,
int open, int page, const char *title, int mode,
float p1, float p2, float p3, float p4);

"**afterThis**" is a pointer to another outline entry below which the new entry will be added. A pointer to an entry returned by a previous call to this function should be used. For the very first outline entry for a document, use **afterThis** = NULL. The function returns a pointer to the newly created outline entry.
If "**sublevel**" is non-zero (OL_SUBENT), the new entry will be added as a sub-entry under afterThis. If it is zero (OL_SAME), the new entry will be added at the same level as "afterThis."
If "**open**" is non-zero (OL_OPEN), subentries under this entry will be open and visible. If it is zero (OL_CLOSED), subentries will NOT be visible upon opening the document (which you can manually open).
"**page**" is the destination page number for this outline entry as specified in cpdf_pageInit().
"**title**" is a string for the outline. It will automatically be escaped of special PDF chars.
"**mode**" should be one of (see page 184, Table 7.1 of the PDF Reference v1.3):

| | | |
|---|---|---|
| DEST_NULL | 0 | |
| DEST_XYZ | 1 | /XYZ left top zoom (zoom=1.0 is normal size) |
| DEST_FIT | 2 | /Fit |
| DEST_FITH | 3 | /FitH top |
| DEST_FITV | 4 | /FitV left |
| DEST_FITR | 5 | /FitR left bottom right top |
| DEST_FITB | 6 | /FitB (fit bounding box to window) PDF-1.1 |
| DEST_FITBH | 7 | /FitBH top(fit width of bbox) PDF-1.1 |
| DEST_FITBV | 8 | /FitBV left(fit height of bbox) PDF-1.1 |

"**p1 .. p4**" are parameters for various "modes" above. Not all parameters are used. For example, if only one parameter is used (e.g., DEST_FITH), assign a value to **p1** only, and pass dummy float variables (values don't matter) for those parameters unused for a given "mode." The coordinate values such as left, top, bottom, and right should be specified in points (1/72 inches).

CPDFoutlineEntry ***cpdf_addOutlineAction**(CPDFdoc *pdf, CPDFoutlineEntry *afterThis, int sublevel,
int open, const char *action_dict, const char *title);

This function allows an insertion of outline entries that triggers a variety of actions, instead of causing GoTo page action offered by **cpdf_addOutlineEntry()** above. With this function, it is possible to specify actions that loads another document from a local file system or via the Web. Since action specification is quite complex, you must format the content (whatever that goes inside << >>) of an action dictionary yourself, and pass it to this function as "**action_dict**." Otherwise, all function arguments are the same as those for cpdf_addOutlineEntry() above. You must consult action specifications as documented in the PDF Reference Manual (v1.3, page 107). One limitation of this function is that it does not allow actions that require additional objects that are specified by indirect reference, or "action_dict" strings longer than 8 kbytes. In other words, "action_dict" must be selfcontained, short, and cannot contain binary data, e.g., sound or movie streams.
Also note that this limitation also means that "GoTo" action type should not be used, because it references the destination page by PDF object number (indirect reference). For that purpose, use the standard **cpdf_addOutlineEntry**() function.

***Examples of "action_dict" (contents of << >> ):***

Each one is a single string. Key words are case sensitive - do not make mistakes in capitalization.

```
"/Type /Action /S /GoToR /D [122 /FitR -4 399 199 533]
/F (//cdrom/1998/Q1/profit.pdf) /NewWindow true"
```

Note: With "**GoToR /D**" specification, page must be an actual page number, not PDF object number. The above example loads a new PDF file .../profit.pdf in a new window, and opens page 123 of the file (the first page is 0).

```
"/Type /Action /S /URI
/URI (http://partners.adobe.com/asn/developer/PDFS/TN/PDFSPEC.PDF)"
```

This opens the specified PDF file via the default web browser.

## 17. Miscellaneous Functions

int **cpdf_comments**(CPDFdoc *pdf, const char *comments);

This function inserts a string pointed to by *comments into the Content stream of the page description. Each line should start with a '%' character that indicates a comment line in PDF and PostScript.

void **rotate_xyCoordinate**(float x, float y, float angle, float *xrot, float *yrot);

Rotate (x, y) by angle in degrees, and return output in (*xrot, *yrot). (*Xrot, *yrot) may point to the same input variables x and y, but in that case, x, y must be plain float variables, and NOT expressions.

void **cpdf_setPDFLevel**(CPDFdoc *pdf, int major, int minor);

This function allows you to set the PDF version as indicated by the first line of any PDF file. Note that, currently, only versions 1.0, 1.1, and 1.2 are acceptable. Indicating 1.2 may cause the file to become un-readable by older PDF viewers. For example, Acrobat Reader/Exchange 3.0 or later is needed to read PDF-1.2.

float **tm_to_NumDays**(struct tm *fromDate, struct tm *toDate);

Computes the number of days between two dates "fromDate" and "toDate." This function is used internally in ClibPDF for implementing TIME domain and axis. However, it may be useful in your part of code. Struct **tm** is documented in "man 3 ctime" and in this manual (see cpdf_createTimeAxis()).

SEE ALSO: **cpdf_createTimePlotDomain**(), **cpdf_createTimeAxis**()

int **cpdf_setMonthNames**(CPDFdoc *pdf, char *mnArray[]);

Sets an array of non-English month names for use with the time axis. See source/testpdf.c for examples. The array specified as the argument may take a form such as:

```
char *monthNamesFrench[] = {
    "Jan", "F\216v", "Mar", "Avr", "Mai", "Jun",
    "Jul", "Ao\236", "Sep", "Oct", "Nov", "D\216c"
};
```

and should be specified using *MacRomanEncoding* as specified in "Appendix C: Predefined Font Encodings," *PDF Reference Manual version 1.*3, page 447.

char ***cpdf_convertUpathToOS**(char *pathbuf, char *upath);

Converts Unix style path to the platform's paths, according to DIR_SEPARATOR as defined in <config.h>. This function may be used for increasing cross-platform portability, but is not necessary if this is not a concern.

char ***cpdf_escapeSpecialChars**(const char *instr);

Three characters, '(', ')' and '\' are special characters in PDF strings that must generally be escaped by preceding them with a '\'. This function, given an input string "instr", returns a new string with the special PDF characters escaped. It returns a new string in malloc()'ed memory. It is the programmer's responsibility to free it once the escaped string is no longer needed. Memory leak will result if this is not followed. Most text functions and annotation functions already use this function internally. Therefore, it is not necessary to use this function on your part in most cases.

long **getFileSize**(const char *file);

Returns the size (number of bytes) of a given file.

char ***cpdf_platform**(void)

This function returns the platform string defined in config.h for each platform.

char ***cpdf_version**(void)

This function return the version string defined in version.h.

int **cpdf_setDocumentID**(CPDFdoc *pdf, int documentID);

This function sets a unique (unique within a process) ID number, documentID (positive integer), so it may be used to distinguish multiple documents being created simultaneously within a single process. However, if you do not activate the use of temporary page content files (possibly to conserve RAM use), it is not necessary to use this function. If this function is used, it should be called after cpdf_open(), but before cpdf_init().

## 18. Error Handling

CPDFerrorHandler **cpdf_setErrorHandler**(CPDFdoc *pdf, CPDFerrorHandler handler)

This function installs a per-PDF document custom error hander of the form below. Without this call, the default handling is as shown below. You may modify the behavior to use GUI-based alert panels or you may prevent the program from exiting entirely when level < 0.

```
void myErrorHandler(CPDFdoc *pdf, int level, const char* module, const char* fmt, va_list ap)
{
   if(module != NULL)
         fprintf(stderr, "%s: ", module);
   vfprintf(stderr, fmt, ap);
   fprintf(stderr, ".\n");
   if(level < 0)
        exit(level);
}
```

CPDFerrorHandler **cpdf_setGlobalErrorHandler**(CPDFerrorHandler handler)

This function installs a global custom error hander of the form below. There is only one instance of global error handler per process. Without this call, the default handling is as shown below. You may modify the behavior to use GUI-based alert panels or you may prevent the program from exiting entirely when level < 0.

```
void myGlobalErrorHandler( int level, const char* module, const char* fmt, va_list ap)
{
   if(module != NULL)
         fprintf(stderr, "%s: ", module);
   vfprintf(stderr, fmt, ap);
   fprintf(stderr, ".\n");
   if(level < 0)
        exit(level);
}
```

# Notes on Example Programs

**A. Library Test Program** *[source/testpdf.c]*
This program performs relatively extensive test of key functions included in ClibPDF library. It is not a best example, as test items have been added without much thought as the library was being developed. However, it is a good test of the library when porting to a new environment. Under a Unix environment, "make test" will produce the test program executable without the use of the library.

**B. Arc and Circles** *[examples/arc/Arcs.c]*
Draws arcs of various angles, and color-filled pie shapes.

**C. Bezier Curves** *[examples/bezier/beziertest.c]*
Source code for Fig. 3.
It shows the Bezier curves, and their control points.

**D. Manual Cover** *[examples/cover/cover.c]*
This program generates the cover of this manual. Draws 100+ pie and pacman shapes of random parameters (size, color, position, angles). It is a good example of the use of a clip path. It also demonstrates the use of cpdf_textAligned(), Web- and e-mail hyper text links, and an in-line image. Page 2 of the PDF file is produced by reading in an ascii text file and showing the content as is, using automatic new-line functions of PDF/ClibPDF.

**E. Dash Pattern** *[examples/dash/dashtest.c]*
Source code for Fig. 5.
Demonstrates how one specifies dash patterns for stroking paths.

**F. Plot Domain** *[examples/domain/DomainDemo.c]*
Source code for Fig. 2.
Demonstrates the notion of plot domains and how they ease plotting of numerical data. A Gaussian curve stored in arrays are plotted on both linear-linear, and log-log domains. It also demonstrates how to use clipping to restrict plots within a specified plot domain.

**G. Fill** *[examples/filltest/filltest.c]*
Source code for Fig. 4.
Tests arcs and their drawing directions, "fill" and "eofill" operators.

**H. Data Markers** *[examples/marker/MarkerTest.c]*
Source code for Fig. 9
Draws all marker symbols available for marking data points. It also shows pointers (arrow heads). By setting fill and stroke colors appropriately, most of these markers may become either filled or open symbols.

**I. Minimal Example** *[examples/minimal/Minimal.c]*
Source code for Fig. 1. A simplest graphics and text example program.

**J. Aligned Text** *[examples/test/textalign.c]*
Source code for Fig. 6.
Shows the definition of text alignment modes (left, right, top, and bottom alignment, and centering modes for cpdf_textAligned() function.

**K. Time Axis** *[examples/timeaxis/timeaxis.c]*
Source code for Fig. 8.
Demonstrates the use of time/date axes. Time axes will try to use appropriate tick marks and numbering automatically for axis spanning any duration.

**L. Weather Data Report** *[examples/weather/weather.c]*
Plots weather data (hourly temperature in C or F degrees, and relative humidity) from an ascii data file. This demonstrates a realistic use of time plot domain and axis, and parsing of a simple text data file. The data file included is for the Oakland Airport, California, down-loaded hourly from the National Weather Service (NOAA) web site (http://iwin.nws.noaa.gov/iwin/ca/hourly.html) via a script and a cron job. The data file also includes dew point, wind direction and speed, and atmospheric pressure.

**M. Outline (Book Marks)** *[examples/outline/outline.c]*
This example generates a 200-page file, and adds an outline tree of 3 levels (chapter, section, and subsection). It also demonstrates a use of cpdf_setDocumentLimits() to increase the default page number and other limits.

**N. PDF Clock** *[examples/pdfclock/pdfclock.c]*
This example shows how to use ClibPDF in a dynamic PDF generation for Web serving. It draws analog clock showing the current time.

**O. Font List** *[examples/fontlist/fontlist.c]*
Creates a list of all 41 fonts built into ClibPDF.

**P. Text Box/TEXT2PDF** *[examples/textbox/textbox.c]*
This program gives a simple example for the usage of cpdf_rawTextBox(). It takes a text file and format into a 2-column justified PDF file. By setting attributes, this example program may serve as a starting point for a very capable TEXT2PDF program.

**Q. TextBox with Fitting** *[examples/textboxfit/textboxfit.c]*
Source code for Fig. 7.
This program demostrates the use of cpdf_textBoxFit() function. It also shows a case of rotated text box.

**R. Bar Code**

*[examples/barcode/barcode.c]*

This program tests a bar code font (Type-1) loaded and embedded from an external file. Requires C39 fonts from Azalea Software (www.azalea.com) or equivalent (not included).

# Programming Tips and Performance Tuning

## 1. Reducing Memory Requirement

In multi-page PDF document creation, ClibPDF allows interleaved generation of all pages: You do **not** have to start with page 1 and create subsequent pages only after the preceding page is completed. You may start at any page (do it even in reverse order), and jump back and forth among pages. This is the default capability of ClibPDF. However, this ability can place a demand for a large amount of memory, because ClibPDF keeps page content in memory and a page content stream cannot be compressed as long as you still need to write to it. In most usage, at least the majority of pages do get created sequentially. Multiple factors affect the memory use, and this section tries to outline ways to reduce memory usage depending on applications and requirements. First, here's a standard way of using ClibPDF in Web CGI applications.

**Tips Example 1.1: (no memory conservation attempt)**

```
CPDFdoc *pdf;
   pdf = cpdf_open(0, NULL);
   cpdf_enableCompression(pdf, YES);        /* use Flate/Zlib compression */
   cpdf_init(pdf);
   for(page=1; page < NPAGE; page++) {
      cpdf_pageInit(pdf, page, PORTRAIT, LETTER, LETTER);  /* create page */
      /* --- a lot drawing for this page here --- */
   }
   cpdf_finalizeAll(pdf);
   bufPDF = cpdf_getBufferForPDF(pdf, &length);
   printf("Content-Type: application/pdf%c", 10);            /* correct MIME type */
   printf("Content-Length: %d%c%c", length, 10, 10);   /* size of PDF */
   fwrite((void *)bufPDF, 1, (size_t)length, stdout);       /* Send PDF now */
   cpdf_close(pdf);
```

***a. Pages are created sequentially. How can I save memory in that case?***

Calling **cpdf_finalizePage()** as soon as a given page is completed can reduce the size of that page by up to 50%.

**Tips Example 1.2: (pages closed and compressed)**

```
CPDFdoc *pdf;
   pdf = cpdf_open(0, NULL);
   cpdf_enableCompression(pdf, YES);        /* use Flate/Zlib compression */
   cpdf_init(pdf);
   for(page=1; page < NPAGE; page++) {
```

```
    cpdf_pageInit(pdf, page, PORTRAIT, LETTER, LETTER);  /* create page */
    /* --- a lot drawing for this page here --- */
    cpdf_finalizePage(pdf, page);          /* This closes the page and compresses the content */
  }
  cpdf_finalizeAll(pdf);
  bufPDF = cpdf_getBufferForPDF(pdf, &length);
  printf("Content-Type: application/pdf%c", 10);           /* correct MIME type */
  printf("Content-Length: %d%c%c", length, 10, 10);   /* size of PDF */
  fwrite((void *)bufPDF, 1, (size_t)length, stdout);       /* Send PDF now */
  cpdf_close(pdf);
```

***b. Output is to a file. I don't have to keep the PDF file in memory. Pages are also sequentially generated.***

If you set an output filename for PDF at the beginning before cpdf_init(), a memory stream for retaining the PDF will not be created. This saves memory of the PDF file size near the end of the document creation process. Note however, that you will not be able to use cpdf_savePDFmemoryStreamToFile() or cpdf_getBufferForPDF(), because these functions rely on the PDF content being in a memory stream buffer.

**Tips Example 1.3: (Output to a file. Pages closed and compressed.)**

```
CPDFdoc *pdf;
  pdf = cpdf_open(0, NULL);
  cpdf_enableCompression(pdf, YES);       /* use Flate/Zlib compression */
  cpdf_setOutputFilename(pdf, "MyPDFfile.pdf");
  cpdf_init(pdf);
  for(page=1; page < NPAGE; page++) {
    cpdf_pageInit(pdf, page, PORTRAIT, LETTER, LETTER);  /* create page */
    /* --- a lot drawing for this page here --- */
    cpdf_finalizePage(pdf, page);          /* This closes the page and compresses the content */
  }
  cpdf_finalizeAll(pdf);
  cpdf_close(pdf);
```

***c. I want to use temporary files for keeping contents of each page to further reduce memory usage.***

This is not recommended, particularly if you intend to use ClibPDF in multi-threaded applications. There are issues for creating unique filenames for each page for multiple PDF document contexts. However, this is possible using cpdf_useContentMemStream(pdf, NO);

[end of doc/1999-09-14; doc should end with an even numbered page]

# Index

## A



## B



## C

## D



## E



## F



## G



## H



## I



## J



## L

## M



## N



## O



## P



## R



## S



## T



## U



## W

# X



# Y



# Z