PDFlib GmbH

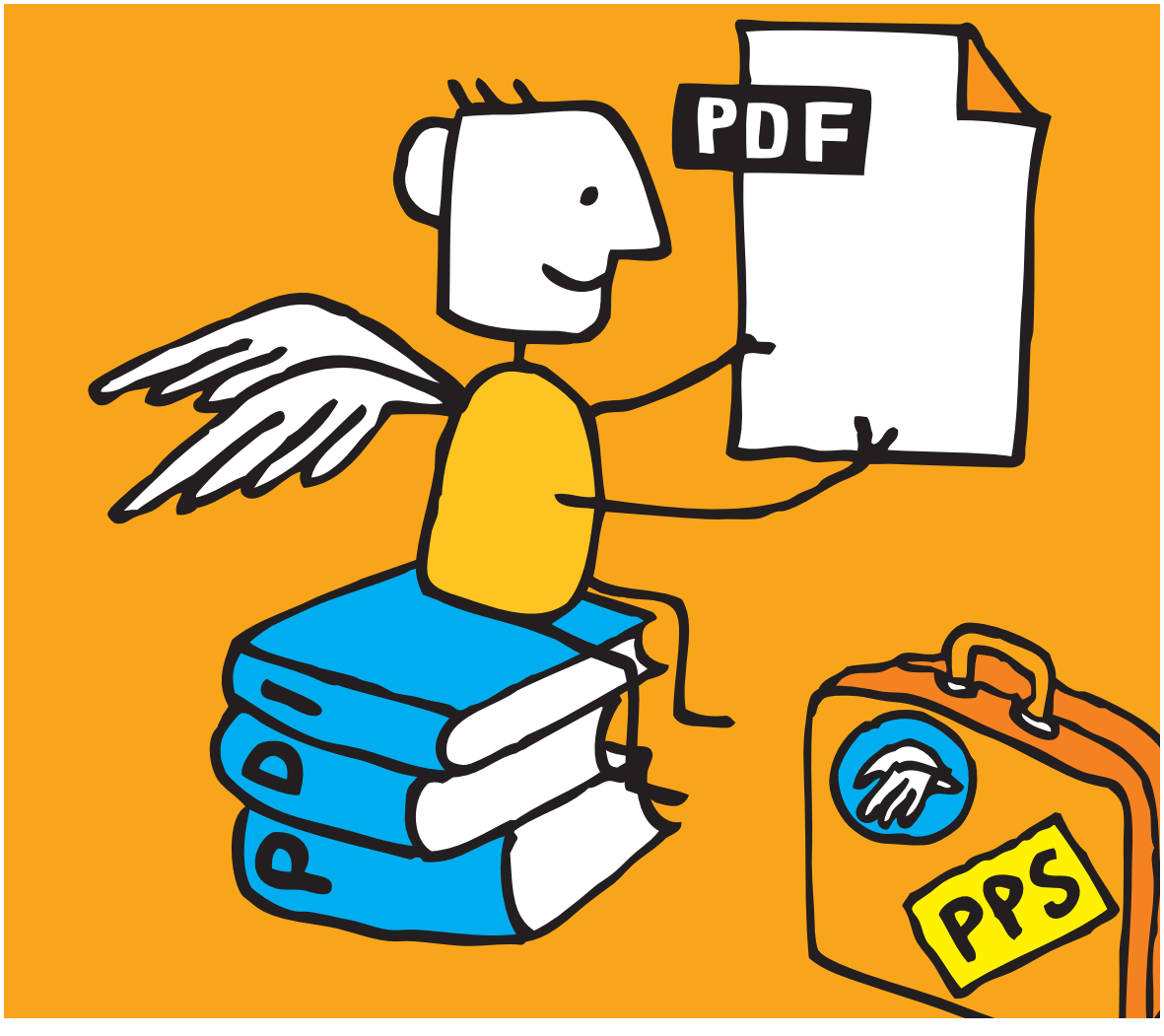
PDFlib, PDFlib+PDI, PPS

A library for generating PDF on the fly

PDFlib 9.0.1

API Reference

For use with C, C++, Cobol, COM, Java, .NET, Objective-C,
Perl, PHP, Python, REALbasic/Xojo, RPG, Ruby



PDFlib GmbH
Franziska-Bilek-Weg 9, 80339 München, Germany
www.pdflib.com
phone +49 • 89 • 452 33 84-0
fax +49 • 89 • 452 33 84-99

If you have questions check the PDFlib mailing list and archive at tech.groups.yahoo.com/group/pdflib

Licensing contact: sales@pdflib.com
Support for commercial PDFlib licensees: support@pdflib.com (please include your license number)

Contents

1 Programming Concepts

1.1 Option Lists

Option lists are a powerful yet easy method for controlling API function calls. Instead of requiring a multitude of function parameters, many API methods support option lists, or *optlists* for short. These are strings which can contain an arbitrary number of options. Option lists support various data types and composite data like lists. In most language bindings optlists can easily be constructed by concatenating the required keywords and values.

Bindings C language binding: you may want to use the *sprintf( )* function for constructing optlists.

.NET language binding: C# programmers should keep in mind that the *AppendFormat( )* StringBuilder method uses the *{* and *}* braces to represent format items which will be replaced by the string representation of arguments. On the other hand, the *Append( )* method does not impose any special meaning on the brace characters. Since the option list syntax makes use of the brace characters, care must be taken in selecting the *AppendFormat( )* or *Append( )* method appropriately.

1.1.1 Syntax

Formal option list syntax definition. Option lists must be constructed according to following rules:

- All elements (keys and values) in an option list must be separated by one or more of the following separator characters: space, tab, carriage return, newline, equal sign '='.
- An outermost pair of enclosing braces is not part of the element. The sequence *{ }* designates an empty element.
- Separators within the outermost pair of braces no longer split elements, but are part of the element. Therefore, an element which contains separators must be enclosed with braces.
- If an element contains brace characters these must be protected with a preceding backslash character.
- If an element contains a sequence of one or more backslash characters in front of a brace, each backslash in the sequence must be protected with another backslash character.
- Option lists must not contain binary zero values.

An option may have a list value according to its documentation in this PDFlib Reference. List values contain one or more elements (which may themselves be lists). They are separated according to the rules above, with the only difference that the equal sign is no longer treated as a separator.

Note *Option names (i.e. the key) never contain hyphen characters. Keep this in mind since the tables with option descriptions may sometimes contain long option names which are hyphenated. The hyphen must be omitted when supplying the option in an option list.*

Simple option lists. In many cases option lists will contain one or more key/value pairs. Keys and values, as well as multiple key/value pairs must be separated by one or

more whitespace characters (space, tab, carriage return, newline). Alternatively, keys can be separated from values by an equal sign '=':

```
key=value
key = value
key value
key1 = value1  key2 = value2
```

To increase readability we recommend to use equal signs between key and value and whitespace between adjacent key/value pairs.

Since option lists will be evaluated from left to right an option can be supplied multiply within the same list. In this case the last occurrence will overwrite earlier ones. In the following example the first option assignment will be overridden by the second, and *key* will have the value *value2* after processing the option list:

```
key=value1 key=value2
```

List values. Lists contain one or more separated values, which may be simple values or list values in turn. Lists are bracketed with *{* and *}* braces, and the values in the list must be separated by whitespace characters. Examples:

```
dasharray={11 22 33}                        (list containing three numbers)
position={ center bottom }                  (list containing two keywords)
```

A list may also contain nested lists. In this case the lists must also be separated from each other by whitespace. While a separator must be inserted between adjacent *}* and *{* characters, it can be omitted between braces of the same kind:

```
polylinelist={{10 20 30 40} {50 60 70 80}}    (list containing two lists)
```

If the list contains exactly one list the braces for the nested list must not be omitted:

```
polylinelist={{10 20 30 40}}                  (list containing one nested list)
```

Nested option lists and list values. Some options accept the type *option list* or *list of option lists*. Options of type *option list* contain one or more subordinate options. Options of type *list of option lists* contain one or more nested option lists. When dealing with nested option lists it is important to specify the proper number of enclosing braces. Several examples are listed below.

The value of the option *metadata* is an option list which itself contains a single option *filename:*

```
metadata={filename=info.xmp}
```

The value of the option *fill* is a list of option lists containing a single option list:

```
fill={{ area=table fillcolor={rgb 1 0 0} }}
```

The value of the option *fill* is a list of option lists containing two option lists:

```
fill={{ area=rowodd fillcolor={rgb 0 1 0} } { area=roweven fillcolor={rgb 1 0 0} }}
```

List containing one option list with a value that includes spaces:

```
attachments={{filename={foo bar.xml} }}
```

List containing three strings:

```
itemnamelist = { {Isaac Newton} {James Clark Maxwell} {Albert Einstein} }
```

List containing two keywords:

```
position={left bottom}
```

List containing different types (float and keyword):

```
position={10 bottom}
```

List containing one rectangle:

```
boxes={{10 20 30 40}}
```

List containing two polylines with percentages:

```
polygons = {{10 20 40 60 90 120}} {12 87 34 98 34% 67% 34% 7%}}
```

Common traps and pitfalls. This paragraph lists some common errors regarding option list syntax.

Braces are not separators; the following is wrong:

```
key1 {value1}key2 {value2}                    WRONG!
```

This will trigger the error message *Unknown option 'value2'*. Similarly, the following are wrong since the separators are missing:

```
key{value}                                    WRONG!
key={{value1}{value2}}                        WRONG!
```

Braces must be balanced; the following is wrong (see below for unquoted string syntax):

```
key={open brace {}                            WRONG!
```

This will trigger the error message *Braces aren't balanced in option list 'key={open brace {}'*. A single brace as part of a string must be preceded by an additional backslash character:

```
key={closing brace \} and open brace \{}      CORRECT!
```

A backslash at the end of a string value must be preceded by another backslash if it is followed by a closing brace character:

```
key={\value\}                                 WRONG!
key={\value\\}                                CORRECT!
```

Unquoted string values in option lists. In the following situations conflicts between the characters in an option value and optlist syntax characters may arise:

- Passwords may contain unbalanced braces, backslashes and other special characters
- Japanese SJIS filenames in option lists (reasonable only in non-Unicode-capable language bindings)
- Supplying JavaScript code in options is problematic due to the use of *{* and *}* braces

In order to provide a simple mechanism for supplying arbitrary text or binary data which does not interfere with option list syntax elements, unquoted option values can be supplied along with a length specifier in the following syntax variants:

```
key[n]=value
key[n]={value}
```

The decimal number *n* represents the following:

- in Unicode-capable language bindings: the number of UTF-16 code units
- in non-Unicode aware language bindings: the number of bytes comprising the string

The braces around the string value are optional, but strongly recommended. They are required for strings starting with a space or other separator character. Braces, separators and backslashes within the string value are taken literally without any special interpretation.

Example for specifying a 7-character password containing space and brace characters. The whole string is surrounded by braces which are not part of the option value:

```
password[7]={ ab}c d}
```

Note that if an option value in a nested option list is provided with a length count, the enclosing option list must also supply a length count, e.g.

```
fitannotation[34]={contents[19]={this is a brace '}'}}
```

1.1.2 Simple Data Types

String. Strings are plain ASCII strings (or EBCDIC strings on EBCDIC platforms) which are generally used for non-localizable keywords. Strings containing whitespace or *'='* characters must be bracketed with *{* and *}*:

```
password={ secret string }                (string value contains three blanks)
contents={length=3mm}                     (string value containing one equal sign)
```

The characters *{* and *}* must be preceded by an additional `\` character if they are supposed to be part of the string:

```
password={weird\}string}                  (string value contains a right brace)
```

A backslash in front of the closing brace of an element must be preceded by a backslash character:

```
filename={C:\path\name\\}                 (string ends with a single backslash)
```

An empty string can be constructed with a pair or braces:

```
{}
```

Content strings, hypertext strings and name strings: these can hold Unicode content in various formats. Single bytes can be expressed by an escape sequence if the option *escapesequence* is set. For details on these string types and encoding choices for string options see the *PDFlib Tutorial*.

Non-Unicode capable language bindings: if an option list starts with a [EBCDIC-]UTF-8 BOM, each content, hypertext or name string of the option list is interpreted as a [EBCDIC-]UTF-8 string.

Unichar. A Unichar is a single Unicode value where several syntax variants are supported: decimal values ≥ 10 (e.g. 173), hexadecimal values prefixed with *x, X, ox, oX*, or U+ *(xAD, oxAD, U+ooAD)*, numerical references, character references, and glyph name references but without the '&' and ';' decoration *(shy, #xAD, #173)*. Alternatively, literal characters can be supplied. Examples:

```
replacementchar=?                (literal)
replacementchar=63               (decimal)
replacementchar=x3F              (hexadecimal)
replacementchar=0x3F             (hexadecimal)
replacementchar=U+003F           (Unicode notation)
replacementchar=euro             (HTML character reference)
replacementchar=.question        (standard glyph name reference)
replacementchar=.marina          (font-specific glyph name reference)
```

Single characters which happen to be a number are treated literally, not as decimal Unicode values:

```
replacementchar=3                (U+0033 THREE, not U+0003!)
```

Unichars must be in the hexadecimal range *0-0x10FFFF* (decimal *0-1114111)*. However, some options are restricted to the range 0-0xFFFF (0-65535). This is noted in the respective option description.

Unicode range. A Unicode range identifies a contiguous range of Unicode characters via start and end characters of the range. The start and end values of a Unicode range must be separated by a minus sign '-' without any spaces, e.g.

```
forcechars={U+03AC-U+03CE}
```

Boolean. Booleans have the values *true* or *false*; if the value of a Boolean option is omitted, the value *true* is assumed. As a shorthand notation *noname* can be used instead of *name=false*:

```
embedding           (equivalent to embedding=true)
noembedding         (equivalent to embedding=false)
```

Keyword. An option of type keyword can hold one of a predefined list of fixed keywords. Example:

```
blendmode=overlay
```

For some options the value hold either a number or a keyword.

Number. Option list support several numerical types.
Integer types can hold decimal and hexadecimal integers. Positive integers starting with x, X, ox, or oX specify hexadecimal values:

```
-12345
0
0xFF
```

Floats can hold decimal floating point or integer numbers; period and comma can be used as decimal separators for floating point values. Exponential notation is also supported. The following values are all equivalent:

```
size = -123.45
size = -123,45
size = -1.2345E2
size = -1.2345e+2
```

Percentages are numbers with a % character directly after the numerical value. Some options allow negative percentages:

```
leading=120%
topoffset=-20.5%
```

Handle. Handles identify various types of objects, e.g. fonts, images, ICC profiles or actions. Technically these are integer values which have been returned earlier by an API function. For example, an image handle is returned by *PDF_load_image()*. Handles must always be treated as opaque types; they must never be modified or created by the application directly (as opposed to using a handle returned by an API function). Handles must always be valid for the respective type of object. For example, an option which expects an image handle must not be supplied with a graphics handle, although both handles are integer types.

1.1.3 Fontsize and Action Data Types

Fontsize. A fontsize can be defined in several ways which allow the size of text to be specified in absolute values, relative to some external entity, or relative to some font property. In general the fontsize must be different from 0 unless the option description mentions otherwise.

In the most common case a fontsize contains a single float value which specifies refers to units in the user coordinate system:

```
fontsize=12
```

The second variant contains a percentage, where the basis of the percentage depends on the context (e.g. the width of the fitbox for *PDF_fit_textline()*:

```
fontsize=8%
```

In the third variant, the fontsize is specified as an option list which must contain a keyword and a number. The keyword describes the desired font metric according to Table 1.1, and the number contains the desired size. PDFlib will calculate the proper fontsize so that the selected text metric matches the supplied value:

```
fontsize={capheight 5}
```

Action list. An action list specifies one or more actions. Each entry in the list consists of an event keyword (trigger) and a list of action handles which must have been created