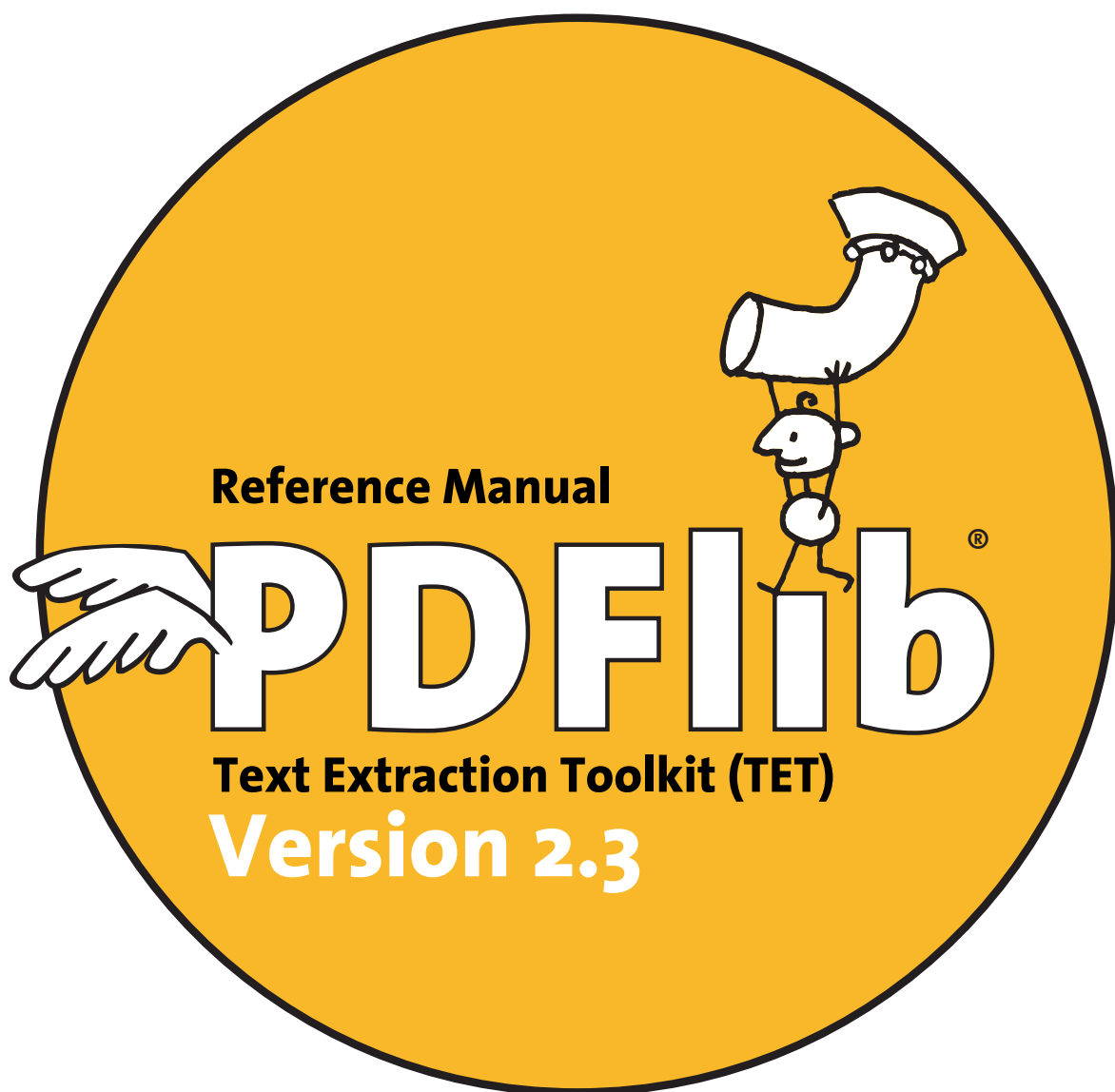


PDFlib GmbH München, Germany

www.pdflib.com



Copyright © 2002-2008 PDFlib GmbH. All rights reserved.
Protected by European patents.
Patents pending in the U.S. and other countries.

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
Technical support: support@pdflib.com (please include your license number)

This publication and the information herein is furnished as is, is subject to change without notice, and should not be construed as a commitment by PDFlib GmbH. PDFlib GmbH assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes and noninfringement of third party rights.

PDFlib and the PDFlib logo are registered trademarks of PDFlib GmbH. PDFlib licensees are granted the right to use the PDFlib name and logo in their product documentation. However, this is not required.

Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Inc. AIX, IBM, OS/390, WebSphere, iSeries, and zSeries are trademarks of International Business Machines Corporation. ActiveX, Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation. Apple, Macintosh and TrueType are trademarks of Apple Computer, Inc. Unicode and the Unicode logo are trademarks of Unicode, Inc. Unix is a trademark of The Open Group. Java and Solaris are a trademark of Sun Microsystems, Inc. Other company product and service names may be trademarks or service marks of others.

*The PDFlib Text Extraction Toolkit contains modified parts of the following third-party software:
Zlib compression library, Copyright © 1995-2002 Jean-loup Gailly and Mark Adler
Cryptographic software written by Eric Young, Copyright © 1995-1998 Eric Young (ey@cryptsoft.com)
Cryptographic software, Copyright © 1998-2002 The OpenSSL Project (www.openssl.org)*

The PDFlib Text Extraction Toolkit contains the RSA Security, Inc. MD5 message digest algorithm.



Contents

o First Steps with TET 5

- o.1 Installing the Software 5
- o.2 Applying the TET License Key 6

1 Introduction 9

- 1.1 TET Application Scenarios 9
- 1.2 TET Features 9
- 1.3 TET Command-Line Tool or TET Library? 11
- 1.4 The TET Plugin for Adobe Acrobat 11

2 TET Command-Line Tool and XML 13

- 2.1 Command-Line Options 13
- 2.2 XML Output 16

3 TET Library Language Bindings 19

- 3.1 Exception Handling 19
- 3.2 C Binding 20
- 3.3 C++ Binding 22
- 3.4 COM Binding 23
- 3.5 Java Binding 24
- 3.6 .NET Binding 25
- 3.7 Perl Binding 26
- 3.8 PHP Binding 27
- 3.9 RPG Binding 29

4 Text Extraction Details 31

- 4.1 Characters and Glyphs 31
- 4.2 Page and Text Geometry 33
- 4.3 Support for Chinese, Japanese, and Korean Text 36
- 4.4 Unicode Mapping 37
- 4.5 Advanced Unicode Mapping Controls 39
- 4.6 Content Analysis 44
- 4.7 Resource Configuration and File Searching 47
- 4.8 Recommendations for common Scenarios 50

5 The pCOS Interface 53

- 5.1 Simple pCOS Examples 53
- 5.2 Handling Basic PDF Data Types 55

5.3 Composite Data Structures and IDs 56

5.4 Path Syntax 57

5.5 Pseudo Objects 59

5.6 Encrypted PDF Documents 65

6 TET Library API Reference 67

6.1 Option Lists 67

6.2 General Functions 69

6.3 Exception Handling 71

6.4 Document Functions 73

6.5 Page Functions 77

6.6 Text and Metrics Retrieval Functions 80

6.7 Option Handling 83

6.8 pCOS Functions 85

A TET Library Quick Reference 89

B Revision History 90

Index 91

o First Steps with TET

o.1 Installing the Software

TET is delivered as an MSI installer package for Windows systems, and as a compressed archive for all other supported operating systems. All TET packages contain the TET command-line tool and the TET library/component, plus support files, documentation, and examples. After installing or unpacking TET the following steps are recommended:

- ▶ Users of the TET command-line tool can use the executable right away. The available options are discussed in Section 2.1, »Command-Line Options«, page 13, and are also displayed when you execute the TET command-line tool without any options.
- ▶ Users of the TET library/component should read one of the sections in Chapter 3, »TET Library Language Bindings«, page 19, corresponding to their preferred development environment, and review the installed examples. On Windows, the TET programming examples are accessible via the Start menu.

If you obtained a commercial TET license you must enter your TET license key according to Section o.2, »Applying the TET License Key«, page 6.

CJK configuration. In order to extract Chinese, Japanese, or Korean (CJK) text TET generally requires the corresponding CMap files for mapping CJK encodings to Unicode. The CMap files are contained in all TET packages, and will be installed in the *resource/cmap* directory within the TET installation directory. On Windows systems simply choose the full installation option when installing TET. The CMap files will be found automatically via the registry.

On other systems you must manually configure the CMap files:

- ▶ For the TET command-line tool this can be achieved by supplying the name of the directory holding the CMap files with the *--searchpath* option.
- ▶ For the TET library/component you can set the *searchpath* at runtime:

```
TET_set_option(tet, "searchpath=/path/to/resource/cmap");
```

As an alternative method for configuring access to the CJK CMap files you can set the *TETRESOURCEFILE* environment variable to point to a UPR configuration file which contains a suitable *searchpath* definition.

Restrictions of the evaluation version. The TET command-line tool and library can be used as fully functional evaluation versions even without a commercial license. Unlicensed versions support all features, but will only process PDF documents with up to 10 pages and 1 MB size. Evaluation versions of TET must not be used for production purposes, but only for evaluating the product. Using TET for production purposes requires a valid TET license.

o.2 Applying the TET License Key

Using TET for production purposes requires a valid TET license key. Once you purchased a TET license you must apply your license key in order to allow processing of arbitrarily large documents. There are several methods for applying the license key; choose one of the methods detailed below.

Note TET license keys are platform-dependent, and can only be used on the platform for which they have been purchased.

Entering the license key in the Windows installer. Windows users can enter the license key when they install TET using the supplied installer. This is the recommended method on Windows. If you do not have write access to the registry or cannot use the installer, refer to one of the alternate methods below instead.

Setting the license key with a TET library call. If you use the TET library, add a line to your script or program which sets the license key at runtime:

- In COM/VBScript:

```
oTET.set_option "license=...your license key..."
```

- In C:

```
TET_set_option(tet, "license=...your license key...");
```

- In C++, .NET/C#, Java, and PHP 5 with object-oriented interface:

```
tet.set_option("license=...your license key...");
```

- In PHP 4 and PHP 5 with function-based interface:

```
TET_set_option(tet, "license=...your license key...");
```

- In RPG:

```
d licensekey      s          20
d licenseval      s          50
c                  eval      licenseopt='license=... your license key ...'+x'00'
c                  callp      TET_set_option(TET:licenseopt:0)
```

The *license* option must be set immediately after instantiating the TET object, i.e., after calling *TET_new()* (in C, PHP 4) or creating a TET object (in C++, COM, .NET, Java, and PHP 5).

Entering the license key in a license file. Set an environment (shell) variable which points to a license file before TET functions are called. If you are using the TET library you can alternatively set the path to the license file by setting the *licensefile* parameter with the *TET_set_option()* function. The license file must be a text file according to the sample below; you can use the license file template *licensekeys.txt* which is contained in all TET distributions. Lines beginning with a '#' characters contain comments, and will be ignored; the second line contains version information for the license file itself:

```
# Licensing information for PDFlib GmbH products
PDFlib license file 1.0
TET 2.3 ...your license key...
```

The details of setting environment variables vary across systems, but a typical statement for a Unix shell looks as follows:

```
export PDFLIBLICENSEFILE="/path/to/licensekeys.txt"
```

On IBM eServer iSeries the license file can be specified as follows:

```
ADDENVVAR ENVVAR(PDFLIBLICENSEFILE) VALUE(<... path ...>) LEVEL(*SYS)
```

This command can be specified in the startup program *QSTRUP* and will work for all PDFlib GmbH products.

1 Introduction

1.1 TET Application Scenarios

The PDFlib Text Extraction Toolkit (TET) is targeted at extracting the text contents of PDF documents, but can also be used to retrieve other information from PDF. TET interprets text-related data structures in the PDF, such as fonts, encodings, and text strings, and uses it to convert the text to Unicode. TET does not apply any OCR (optical character recognition), and is therefore much faster than typical OCR software. TET can be used for the following tasks:

- ▶ searching the text contents of PDF
- ▶ creating a list of all words contained in a PDF (concordance)
- ▶ implementing a search engine for processing large numbers of PDF files
- ▶ extracting text from PDF to store, translate, or otherwise re-purpose it
- ▶ writing software to convert the text contents of PDF to other formats
- ▶ processing PDFs based on their contents
- ▶ comparing the text contents of multiple PDF documents

1.2 TET Features

TET consists of the following main functional blocks:

- ▶ Unicode mapping: the text found on a page will be normalized to Unicode, regardless of the original font and encoding.
- ▶ Geometry: precise position, glyph width, and font information is made available for the text on a page.
- ▶ Content analysis: Advanced content processing algorithms enhance the results of the low-level text extraction by determining word breaks, removing redundant text (e.g. duplicate text created by shadow effects), and rearranging the text into proper reading order.
- ▶ pCOS: The PDFlib Comprehensive Object Syntax interface retrieves arbitrary objects from a PDF which are not related to the text of a page (e.g. meta data, hypertext).
- ▶ Using the TET command-line tool the information retrieved from a PDF document can be presented in XML format for processing with other tools.

TET has been designed for standalone use, and does not require any third-party software. It is robust and suitable for multi-threaded server use. The core library is written in highly optimized C code for maximum performance and minimum overhead. Several language bindings are available for use with all major programming languages.

Supported PDF input. TET has been tested against thousands of PDF test files from various sources. It accepts all relevant flavors of PDF:

- ▶ PDF versions 1.0-1.7 (corresponding to Acrobat 1-8);
- ▶ all compression filters;
- ▶ all font and encoding combinations: base 14 fonts, TrueType, PostScript, OpenType, single- and multi-byte CID fonts;
- ▶ documents encrypted with 40- or 128-bit keys (only if content extraction is allowed by the document's permission settings, or the master password is supplied);

Unicode support. TET includes considerable amounts of algorithms and data to achieve reliable Unicode mappings for all text. Although text in PDF documents is not usually encoded in Unicode, TET will normalize the text from a PDF document to Unicode:

- ▶ TET converts all text contents to Unicode. In C the text will be returned in UTF-8 or UTF-16 format; in other language bindings as native Unicode strings.
- ▶ Ligatures and other multi-character glyphs will be decomposed into a sequence of their constituent Unicode characters.
- ▶ Vendor-specific Unicode values (Corporate Use Subarea, CUS) are identified, and will be mapped to characters with precisely defined meanings if possible.
- ▶ Glyphs which are lacking Unicode mapping information are identified as such, and will be mapped to a configurable replacement character.
- ▶ UTF-16 surrogate pairs for characters outside the Basic Multilingual Plane (BMP) are properly interpreted and maintained. Surrogate pairs and UTF-32 values can be retrieved in all language bindings.

Some PDF documents do not contain enough information for reliable Unicode mapping. In order to successfully extract the text nevertheless TET offers various configuration options which can be used to supply auxiliary information for proper Unicode mappings. In order to facilitate writing the required mapping tables we make available PDFlib FontReporter, a free plugin for Adobe Acrobat. This plugin can be used for analyzing fonts, encodings, and glyphs in PDF.

CJK support. TET includes full support for extracting Chinese, Japanese, and Korean text:

- ▶ All predefined CJK CMaps (encodings) are recognized; CJK text will be converted to Unicode. CMap files are shipped with the TET distribution.
- ▶ Both horizontal and vertical writing modes are supported.
- ▶ CJK font names will be normalized to Unicode.

pCOS interface for simple access to PDF objects. TET includes pCOS (*PDFlib Comprehensive Object System*) for retrieving arbitrary PDF objects. With pCOS you can retrieve PDF metadata, hypertext (e.g. bookmark text, contents of form fields), or any other information from a PDF document with a simple query interface.

Geometry. TET provides precise metrics for the text, such as the position on the page, glyph widths, and text direction. Specific areas on the page can be excluded or included in the text extraction process, e.g. to ignore headers and footers or margins.

Word detection and content analysis. TET can be used to retrieve low-level glyph information, but also includes advanced algorithms for high-level content analysis:

- ▶ Detect word boundaries to retrieve words instead of characters.
- ▶ Recombine the parts of hyphenated words (dehyphenation).
- ▶ Remove duplicate instances of text, e.g. shadow and fake bold text.
- ▶ Recombine paragraphs into reading order.
- ▶ Reorder text which is scattered over the page.
- ▶ Reconstruct lines of text.

What is text? While TET deals with a large class of PDF documents, not all visible text can successfully be extracted. The text must be encoded using PDF's text and encoding facilities (i.e., it must be based on a font). Although the following flavors of text may be visible on the page they cannot be extracted with TET:

- ▶ Rasterized (pixel image) text, e.g. scanned pages
- ▶ Vectorized text

Note that metadata and text in hypertext elements (such as bookmarks, form fields, notes, or annotations) can be retrieved with the pCOS interface. On the other hand, TET may extract some text which is *not* visible on the page. This may happen in the following situations:

- ▶ Text using PDF's *invisible* attribute (however, there is an option to exclude this kind of text from the text retrieval process)
- ▶ Text which is obscured or clipped by some other element on the page, e.g. an image.
- ▶ PDF layers are currently ignored; TET will retrieve the text from all layers regardless of their visibility.

1.3 TET Command-Line Tool or TET Library?

TET is available as a programming library (component) for various development environments, and as a command-line tool for batch operations. Both offer almost the same features, but are suitable for different deployment tasks. Here are some guidelines for choosing among both TET flavors:

- ▶ The TET programming library can be used for integration into your desktop or server application. Examples for using the TET library with all supported language bindings are included in the TET package.
- ▶ The TET command-line tool is suited for batch processing PDF documents. It doesn't require any programming, but offers command-line options which can be used to integrate it into complex workflows. As an additional feature the TET command-line tool offers XML output which includes all information retrieved by TET in easily accessible XML format.

1.4 The TET Plugin for Adobe Acrobat

In addition to the TET command-line tool and TET library, the TET Plugin offers another packaging of the core TET product. It can be installed in Adobe Acrobat and allows interactive use of TET with any PDF document that is currently open in Acrobat. Using the TET plugin you can access TET's functionality and experiment with TET options. The TET plugin can freely be downloaded from www.pdflib.com

2 TET Command-Line Tool and XML

2.1 Command-Line Options

The TET command-line tool allows you to extract text from one or more PDF documents without the need for any programming. In addition, it can be used to save the results of text extraction as XML, and as a frontend to the pCOS interface. The TET program can be controlled via a number of command-line options. The program will insert space characters (U+0020) after each word, U+000A after each line, and U+000C after each page. It is called as follows for one or more input PDF files:

```
tet [<options>] <filename>...
```

The TET command-line tool is built on top of the TET library. You can supply library options using the `--docopt`, `--teto`, and `--pageopt` options according to the option list tables in Chapter 6, »TET Library API Reference«, page 67. Table 2.1 lists all TET command-line options (this list will also be displayed if you run the TET program without any options).

Note In order to extract CJK text you must configure access to the CMap files which are shipped with TET according to Section 0.1, »Installing the Software«, page 5.

Table 2.1 TET command-line options

long option	short option	parameters	function
<code>--articles</code>	<code>-a</code>		Process page contents according to article threads
<code>--docopt</code>		<option list>	Additional option list for <code>TET_open_document()</code> (see Table 6.2, page 73)
<code>--firstpage</code>	<code>-f</code>	integer	The number of the page where text extraction will start. The keyword <code>last</code> can be used to specify the last page. Default: 1
<code>--format</code>		utf8 utf16	Specifies the format for text output (default: utf8): utf8 UTF-8 with BOM (byte order mark) utf16 UTF-16 in native byte ordering with BOM
<code>--help</code> (or no option)	<code>-?</code>		Display help with a summary of available options
<code>--inmemory</code>			Load the input file(s) into memory and process it from there. This can result in a significant performance gain on some systems at the expense of memory usage.
<code>--lastpage</code>	<code>-l</code>	integer	The number of the page where text extraction will finish. The keyword <code>last</code> can be used to specify the last page. Default: <code>last</code>
<code>--outfile</code>	<code>-o</code>	<filename>	Output file name. The file name <code>»-«</code> can be used to designate standard output. Default: name of the input file, with <code>.pdf</code> or <code>.PDF</code> replaced with <code>.txt</code> (for text output) or <code>.xml</code> (for XML output).
<code>--pageopt</code>		<option list>	Additional option list for <code>TET_open_page()</code> (see Table 6.4, page 77); the option granularity will always be set to page.
<code>--password</code>	<code>-p</code>	<password>	User or master password for encrypted documents

Table 2.1 TET command-line options

long option	short option	parameters	function
<code>--searchpath¹</code>	<code>-s</code>	<code><path>...</code>	Name of one or more directories where files (e.g. CMaps) will be searched. Default: installation-specific
<code>--targetdir</code>	<code>-t</code>	<code><dirname></code>	Output directory name; the directory must exist. Default: .
<code>--teto</code>		<code><option list></code>	Additional option list for <code>TET_set_option()</code> (see Table 6.7, page 83). The option output format will be ignored (use <code>--format</code> instead).
<code>--verbose</code>	<code>-v</code>	0, 1, 2, 3	verbosity level (default: 1): 0 no output at all 1 emit only errors 2 emit errors and file names 3 detailed reporting
<code>--version</code>	<code>-V</code>		Print the TET version number.
<code>--xml</code>	<code>-x</code>	glyph word word2 line zone page	Create glyph-, word-, line-, zone-, or page-based XML output containing the text and metrics information retrieved by TET. The <code>--format</code> option will be ignored (see Section 2.2, »XML Output«, page 16). The word2 mode is similar to word mode, but includes details for all the characters in a word.

1. This option can be supplied more than once.

Constructing TET command lines. The following rules must be observed for constructing TET command lines:

- ▶ Input files will be searched in all directories specified as *searchpath*.
- ▶ Short forms are available for some options, and can be mixed with long options.
- ▶ Long options can be abbreviated provided the abbreviation is unique (e.g. `--tet` instead of `--teto`)
- ▶ Depending on the encryption status of the input file, a user or master password may be required for successfully extracting text. It must be supplied with the `--password` option. TET will check whether this password is sufficient for text extraction, and will generate an error if it isn't.

TET checks the full command line before processing any file. If an error is encountered in the options anywhere on the command line, no files will be processed at all.

File names. File names which contain blank characters require some special handling when used with command-line tools like TET. In order to process a file name with blank characters you should enclose the complete file name with double quote " characters. Wildcards can be used according to standard practice. For example, **.pdf* denotes all files in a given directory which have a *.pdf* file name suffix. Note that on some systems case is significant, while on others it isn't (i.e., **.pdf* may be different from **.PDF*). Also note that on Windows systems wildcards do not work for file names containing blank characters.

Exit codes. The TET command-line tool returns with an exit code which can be used to check whether or not the requested operations could be successfully carried out:

- ▶ Exit code 0: all command-line options could be successfully and fully processed.
- ▶ Exit code 1: one or more file processing errors occurred, but processing continued.

- Exit code 2: some error was found in the command-line options. Processing stopped at the particular bad option, and no input file has been processed.

2.2 XML Output

With the `--xml` option the TET command-line tool will create XML output which represents the information retrieved by TET. Depending on the parameter supplied for this option the output granularity will be glyph, word, word2, line, zone, or page.

What's included in the XML output? XML output created by TET will be encoded in UTF-8 (on zSeries with USS or MVS: EBCDIC-UTF-8), and includes the following information:

- ▶ A *Creation* element showing the date and operating system platform for the TET execution, plus the version number of TET
- ▶ A *Document* element with general information including PDF file name and size, PDF version number, metadata (a *Metadata* element with XMP if present, or a *DocInfo* element with document info fields otherwise)
- ▶ A *Page* element for each page of the PDF document, containing page size attributes and the page contents
- ▶ For each page of the PDF document, a *Structure* element with the actual text and coordinates according to the chosen granularity. For glyph and word granularity a *Font* element will be written with font information. In glyph granularity a *Glyph* element will contain the position and width of the corresponding glyph.

The XML output also includes relevant document- and page-related options which were supplied to TET. A DTD (Document Type Definition) describing the TET XML output in detail can be found at the following Web location:

www.pdflib.com/XML/tet2/tet-1.0.dtd

The following sample demonstrates TET XML output in *word* mode:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by the PDFlib Text Extraction Toolkit TET (www.pdflib.com) -->
<!DOCTYPE TET SYSTEM "http://www.pdflib.com/XML/tet2/tet-1.0.dtd">
<TET>
  <Creation Platform="Linux" TET_version="2.3" Date="D:20070110120810+01'00'" />
  <Document name="hello.pdf" numofpages="1" size="1495" linearized="false"
    pdf_version="1.5">
    <DocInfo>
      <Title> Hello, world (C)! </Title>
      <Author> Thomas Merz </Author>
      <Creator> hello.c </Creator>
      <CreationDate> D:20070110120810+01'00' </CreationDate>
      <Producer> PDFlib Personalization Server 6.0.2p5 (Linux) </Producer>
    </DocInfo>
  </Document>
  <Page number="1" height="842" width="595">
    <Options> granularity=word </Options>
    <Structure>
      <Font name="Helvetica-Bold" size="24" />
      <Word box="[50.00 700.00 108.68 724.00]">
        <Text> Hello </Text>
      </Word>
      <Word box="[108.68 700.00 115.35 724.00]">
        <Text> , </Text>
      </Word>
      <Word box="[122.02 700.00 186.03 724.00]">
```



```

        <Text> world </Text>
    </Word>
    <Word box="[186.03 700.00 194.02 724.00]">
        <Text> ! </Text>
    </Word>
    <Word box="[50.00 676.00 57.99 700.00]">
        <Text> ( </Text>
    </Word>
    <Word box="[57.99 676.00 111.37 700.00]">
        <Text> says </Text>
    </Word>
    <Word box="[118.04 676.00 135.37 700.00]">
        <Text> C </Text>
    </Word>
    <Word box="[135.37 676.00 143.36 700.00]">
        <Text> ) </Text>
    </Word>
</Structure>
</Page>
</TET>

```

The following sample shows XML output for the same file, but in *glyph* mode:

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by the PDFlib Text Extraction Toolkit TET (www.pdflib.com) -->
<!DOCTYPE TET SYSTEM "http://www.pdflib.com/XML/tet2/tet-1.0.dtd"
<TET>
    <Creation Platform="Linux" TET_version="2.3" Date="D:20070110120810+01'00'" />
    <Document name="hello.pdf" numofpages="1" size="1495" linearized="false" pdf_
version="1.5">
        <DocInfo>
            <Title> Hello, world (C)! </Title>
            <Author> Thomas Merz </Author>
            <Creator> hello.c </Creator>
            <CreationDate> D:20070110120810+01'00' </CreationDate>
            <Producer> PDFlib Personalization Server 6.0.2p5 (Linux) </Producer>
        </DocInfo>
    </Document>
    <Page number="1" height="842" width="595" granularity="glyph">
        <Options> granularity=glyph </Options>
        <Structure>
            <Font name="Helvetica-Bold" size="24" />
            <Glyph text="H" x="50.00" y="700.00" width="17.33" />
            <Glyph text="e" x="67.33" y="700.00" width="13.34" />
            <Glyph text="l" x="80.67" y="700.00" width="6.67" />
            <Glyph text="l" x="87.34" y="700.00" width="6.67" />
            <Glyph text="o" x="94.02" y="700.00" width="14.66" />
            <Glyph text="," x="108.68" y="700.00" width="6.67" />
            ...
            <Glyph text="s" x="98.02" y="676.00" width="13.34" />
            <Glyph text=" " x="111.37" y="676.00" width="6.67" />
            <Glyph text="C" x="118.04" y="676.00" width="17.33" />
            <Glyph text=")" x="135.37" y="676.00" width="7.99" />
        </Structure>
    </Page>
</TET>

```


3 TET Library Language Bindings

This chapter discusses specifics for the language bindings which are supplied for TET. The TET distribution contains full sample code for a generic text extraction application in all supported language bindings.

3.1 Exception Handling

Errors of a certain kind are called exceptions in many languages for good reasons – they are mere exceptions, and are not expected to occur very often during the lifetime of a program. The general strategy is to use conventional error reporting mechanisms (read: special error return codes) for function calls which may go wrong often times, and use a special exception mechanism for those rare occasions which don't justify cluttering the code with conditionals. This is exactly the path that TET goes: Some operations can be expected to go wrong rather frequently, for example:

- ▶ Trying to open a PDF document for which one doesn't have the proper password
- ▶ Trying to open a PDF document with a wrong file name
- ▶ Trying to open a PDF document with damaged file structure.

TET signals such errors by returning a value of `-1` as documented in the API reference. Other events may be considered harmful, but will occur rather infrequently, e.g.

- ▶ running out of virtual memory;
- ▶ supplying wrong function parameters (e.g. an invalid document handle);
- ▶ supplying malformed option lists;
- ▶ a required resource (e.g. a CMap file for CJK text extract) cannot be found.

When TET detects such a situation, an exception will be thrown instead of passing a special error return value to the caller. In languages which support native exceptions throwing the exception will be done using the standard means supplied by the language or environment. For the C language binding TET supplies a custom exception handling mechanism which must be used by clients (see Section 3.2, »C Binding«, page 20).

It is important to understand that processing a document must be stopped when an exception occurred. The only methods which can safely be called after an exception are `TET_delete()`, `TET_get_apiname()`, `TET_get_errnum()`, and `TET_get_errmsg()`. Calling any other method after an exception may lead to unexpected results. The exception will contain the following information:

- ▶ A unique error number;
- ▶ The name of the API function which caused the exception;
- ▶ A descriptive text containing details of the problem;

Querying the reason of a failed function call. Some TET function calls, e.g. `TET_open_document()` or `TET_open_page()`, can fail without throwing an exception (they will return `-1` in case of an error). In this situation the functions `TET_get_errnum()`, `TET_get_errmsg()`, and `TET_get_apiname()` can be called immediately after a failed function call in order to retrieve details about the nature of the problem.

3.2 C Binding

Exception handling. The TET API provides a mechanism for acting upon exceptions thrown by the library in order to compensate for the lack of native exception handling in the C language. Using the `TET_TRY()` and `TET_CATCH()` macros client code can be set up such that a dedicated piece of code is invoked for error handling and cleanup when an exception occurs. These macros set up two code sections: the try clause with code which may throw an exception, and the catch clause with code which acts upon an exception. If any of the API functions called in the try block throws an exception, program execution will continue at the first statement of the catch block immediately. The following rules must be obeyed in TET client code:

- ▶ `TET_TRY()` and `TET_CATCH()` must always be paired.
- ▶ `TET_new()` will never throw an exception; since a try block can only be started with a valid TET object handle, `TET_new()` must be called outside of any try block.
- ▶ `TET_delete()` will never throw an exception, and therefore can safely be called outside of any try block. It can also be called in a catch clause.
- ▶ Special care must be taken about variables that are used in both the try and catch blocks. Since the compiler doesn't know about the transfer of control from one block to the other, it might produce inappropriate code (e.g., register variable optimizations) in this situation.

Fortunately, there is a simple rule to avoid this kind of problem: Variables used in both the try and catch blocks must be declared *volatile*. Using the *volatile* keyword signals to the compiler that it must not apply dangerous optimizations to the variable.

- ▶ If a try block is left (e.g., with a return statement, thus bypassing the invocation of the corresponding `TET_CATCH()`), the `TET_EXIT_TRY()` macro must be called before the return statement to inform the exception machinery.
- ▶ As in all TET language bindings document processing must stop when an exception was thrown.

The following code fragment demonstrates these rules with the typical idiom for dealing with TET exceptions in client code (a full sample can be found in the TET package):

```
volatile int pageno;
...
if ((tet = TET_new()) == (TET *) 0)
{
    printf("out of memory\n");
    return(2);
}
TET_TRY(tet)
{
    for (pageno = 1; pageno <= n_pages; ++pageno)
    {
        /* process page */

        if (/* error happened */)
        {
            TET_EXIT_TRY(tet);
            return -1;
        }
    }
    /* statements that directly or indirectly call API functions */
}
```

```

TET_CATCH(tet)
{
    printf("Error %d in %s() on page %d: %s\n",
        TET_get_errnum(tet), TET_get_apiname(tet), pageno, TET_get_errmsg(tet));
}
TET_delete(tet);

```

Unicode handling for name strings. The C language does not natively support Unicode. Some string parameters for API functions may be declared as *name strings*. These are handled depending on the *length* parameter and the existence of a BOM at the beginning of the string. In C, if the *length* parameter is different from 0 the string will be interpreted as UTF-16. If the *length* parameter is 0 the string will be interpreted as UTF-8 if it starts with a UTF-8 BOM, or as EBCDIC UTF-8 if it starts with an EBCDIC UTF-8 BOM, or as *host* encoding if no BOM is found (or *ebcdic* on all EBCDIC-based platforms).

Unicode handling for option lists. Strings within option lists require special attention since they cannot be expressed as Unicode strings in UTF-16 format, but only as byte arrays. For this reason UTF-8 is used for Unicode options. By looking for a BOM at the beginning of an option TET decides how to interpret it. The BOM will be used to determine the format of the string. More precisely, interpreting a string option works as follows:

- ▶ If the option starts with a UTF-8 BOM (`\xEF\xBB\xBF`) it will be interpreted as UTF-8.
- ▶ If the option starts with an EBCDIC UTF-8 BOM (`\x57\x8B\xAB`) it will be interpreted as EBCDIC UTF-8.
- ▶ If no BOM is found, the string will be treated as *winansi* (or *ebcdic* on EBCDIC-based platforms).

Note The `TET_utf16_to_utf8()` utility function can be used to create UTF-8 strings from UTF-16 strings, which is useful for creating option lists with Unicode values.

3.3 C++ Binding

In addition to the *tetlib.h* C header file, an object-oriented wrapper for C++ is supplied for TET clients. It requires the *tet.hpp* header file, which in turn includes *tetlib.h*. The corresponding *tet.cpp* module must be linked against the application in addition to the generic TET C library.

Using the C++ object wrapper replaces the functional approach with API functions and *TET_* prefixes in all TET function names with a more object-oriented approach: a *TET* object offers methods, but the method names no longer have the *TET_* prefix.

The TET C++ binding will package Unicode text in standard C++ strings in UTF-16 format. Clients must be prepared to process such strings appropriately.

3.4 COM Binding

Installing the TET COM edition. TET can be deployed in all environments that support COM components. Installing TET is an easy and straight-forward process. Please note the following:

- ▶ If you install on an NTFS partition all TET users must have read permission for the installation directory, and execute permission for ...\\TET 2.3\\COM\\bin\\tet_com.dll.
- ▶ The installer must have write permission for the system registry. Administrator or Power Users group privileges will usually be sufficient.

Exception Handling. Exception handling for the TET COM component is done according to COM conventions: when a TET exception occurs, a COM exception will be raised and furnished with a clear-text description of the error. In addition the memory allocated by the TET object is released. The COM exception can be caught and handled in the TET client in whichever way the client environment supports for handling COM errors.

Using the TET COM Edition with .NET. As an alternative to the TET.NET edition (see Section 3.6, ».NET Binding«, page 25) the COM edition of TET can also be used with .NET. First, you must create a .NET assembly from the TET COM edition using the *tlbimp.exe* utility:

```
tlbimp tet_com.dll /namespace:tet_com /out:Interop.tet_com.dll
```

You can use this assembly within your .NET application. If you add a reference to *tet_com.dll* from within Visual Studio .NET an assembly will be created automatically. The following code fragment shows how to use the TET COM edition with C#:

```
using TET_com;
...
static TET_com.ITET tet;
...
tet = New TET();
...
```

All other code works as with the .NET edition of TET.

3.5 Java Binding

Installing the TET Java edition. TET is organized as a Java package with the name *com.pdflib.TET*. This package relies on a native JNI library; both pieces must be configured appropriately.

In order to make the JNI library available the following platform-dependent steps must be performed:

- ▶ On Unix systems the library *libtet_java.so* (on Mac OS X: *libtet_java.jnilib*) must be placed in one of the default locations for shared libraries, or in an appropriately configured directory.
- ▶ On Windows the library *pdf_tet.dll* must be placed in the Windows system directory, or a directory which is listed in the PATH environment variable.

The TET Java package is contained in the *tet.jar* file and contains a single class called *tet*. In order to supply this package to your application, you must add *tet.jar* to your *CLASSPATH* environment variable, add the option *-classpath tet.jar* in your calls to the Java compiler, or perform equivalent steps in your Java IDE. In the JDK you can configure the Java VM to search for native libraries in a given directory by setting the *java.library.path* property to the name of the directory, e.g.

```
java -Djava.library.path=. extractor
```

You can check the value of this property as follows:

```
System.out.println(System.getProperty("java.library.path"));
```

Exception handling. The TET language binding for Java will throw native Java exceptions of the class *TETException*. TET client code must use standard Java exception syntax:

```
TET tet = null;

try {
    ...TET method invocations...
} catch (TETException e) {
    System.err.print("TET exception occurred:\n");
    System.err.print "[" + e.get_errnum() + "] " + e.get_apiname() + ": " +
        e.get_errmsg() + "\n");
} catch (Exception e) {
    System.err.println(e.getMessage());
} finally {
    if (tet != null) {
        tet.delete();
    }
} /* delete the TET object */
```

Since TET declares appropriate *throws* clauses, client code must either catch all possible exceptions or declare those itself.

3.6 .NET Binding

The Microsoft .NET architecture supports a variety of programming languages based on the Common Language Runtime (CLR) which provides a common environment for executing programs.

The .NET edition of TET supports all relevant .NET concepts. In technical terms, the TET.NET edition is a C++ class (with a managed wrapper for the unmanaged TET core library) which runs under control of the .NET framework. It is packaged as a static assembly with a strong name. The TET assembly (*tet_dotnet.dll*) contains the actual library plus meta information.

Note TET.NET requires the .NET framework 1.1 or above. It does not work with framework 1.0. If you must work with framework 1.0 we recommend using the TET COM component as a .NET assembly as detailed in Section 3.4, »COM Binding«, page 23.

TET.NET can be deployed in all environments that support the .NET Framework 1.1 or above. The TET distribution package contains code samples for various .NET languages.

The TET MSI installer will install the TET assembly plus auxiliary data files, documentation and samples on the machine interactively. The installer will also register TET so that it can easily be referenced on the .NET tab in the *Add Reference* dialog box of Visual Studio .NET.

Exception handling. TET.NET supports .NET exceptions, and will throw an exception with a detailed error message when a runtime problem occurs. The client is responsible for catching such an exception and properly reacting on it. Otherwise the .NET framework will catch the exception and usually terminate the application.

In order to convey exception-related information TET defines its own exception class *TET_dotnet.TETException* with the members *get_errnum*, *get_errmsg*, and *get_api-name*.

3.7 Perl Binding

Installing the TET Edition for Perl. TET is implemented as a C library which can dynamically be attached to Perl. This requires Perl to be built with support for loading extensions at runtime. The name of the PLOP Perl extension is *tetlib_pl*.

Installing the TET Edition for Perl. The Perl extension mechanism loads shared libraries at runtime through the DynaLoader module. The Perl executable must have been compiled with support for shared libraries (this is true for the majority of Perl configurations).

For the TET binding to work, the Perl interpreter must access the TET Perl wrapper and the module file *tetlib_pl.pm*. In addition to the platform-specific methods described below you can add a directory to Perl's *@INC* module search path using the *-I* command line option:

```
perl -I/path/to/tet extractor.pl
```

Unix. Perl will search both *tetlib_pl.so* (on Mac OS X: *tetlib_pl.dylib*) and *tetlib_pl.pm* in the current directory, or the directory printed by the following Perl command:

```
perl -e 'use Config; print $Config{sitearchexp};'
```

Perl will also search the subdirectory *auto/tetlib_pl*. Typical output of the above command looks like

```
/usr/lib/perl5/site_perl/5.8/i686-linux
```

Windows. PDFlib supports the ActiveState port of Perl 5 to Windows, also known as ActivePerl. Both *tetlib_pl.dll* and *tetlib_pl.pm* will be searched in the current directory, or the directory printed by the following Perl command:

```
perl -e "use Config; print $Config{sitearchexp};"
```

Typical output of the above command looks like

```
C:\Program Files\Perl5.8\site\lib
```

Exception Handling in Perl. When a TET exception occurs, a Perl exception is thrown. It can be caught and acted upon using an *eval* sequence:

```
eval {  
    ...some TET instructions...  
};  
die "Exception caught: $@" if $@;
```

3.8 PHP Binding

Installing the TET Edition for PHP. TET is implemented as a C library which can dynamically be attached to PHP. TET supports several versions of PHP. Depending on the version of PHP you use you must choose the appropriate TET library from the unpacked TET archive.

Detailed information about the various flavors and options for using TET with PHP, including the question of whether or not to use a loadable TET module for PHP, can be found in the *PDFlib-in-PHP-HowTo* document which is available on the PDFlib web site. Although it is mainly targeted at using PDFlib with PHP the discussion applies equally to using TET with PHP.

You must configure PHP so that it knows about the external TET library. You have two choices:

- ▶ Add one of the following lines in *php.ini*:

```
extension=libtet_php.dll      ; for Windows
extension=libtet_php.so       ; for Unix
extension=libtet_php.sl       ; for HP-UX
extension=libtet_php.dylib    ; for Mac OS X
```

PHP will search the library in the directory specified in the *extension_dir* variable in *php.ini* on Unix, and additionally in the standard system directories on Windows. You can test which version of the PHP TET binding you have installed with the following one-line PHP script:

```
<?phpinfo()?>
```

This will display a long info page about your current PHP configuration. On this page check the section titled *tet*. If this section contains the phrase

```
PDFlib TET Support          enabled
```

(plus the TET version number) you have successfully installed TET for PHP.

- ▶ Alternatively, you can load TET at runtime with one of the following lines at the start of your script:

```
dl("libtet_php.dll");        # for Windows
dl("libtet_php.so");         # for Unix
dl("libtet_php.sl");         # for HP-UX
dl("libtet_php.dylib");      # for Mac OS X
```

PHP 5 features. TET takes advantage of the following features in PHP 5:

- ▶ New object model: the TET functions are encapsulated within a TET object.
- ▶ Exceptions: TET exceptions will be propagated as PHP 5 exceptions, and can be caught with the usual try/catch technique. New-style exception handling can be used with both the new object-oriented approach and the old API functions. See below for more details on exception handling.

File name handling in PHP. Unqualified file names (without any path component) and relative file names are handled differently in Unix and Windows versions of PHP:

- ▶ PHP on Unix systems will find files without any path component in the directory where the script is located.

- PHP on Windows will find files without any path component only in the directory where the PHP DLL is located.

Error handling in PHP 4. When a TET exception occurs, a PHP exception is thrown. Since PHP 4 does not support structured exception handling there is no way to catch exceptions and act appropriately. Do not disable PHP warnings when using TET, or you will run into serious trouble.

Exception handling in PHP 5. Since PHP 5 supports structured exception handling, TET exceptions will be propagated as PHP exceptions. You can use the standard *try/catch* technique to deal with TET exceptions:

```
try {  
  
    ...some TET instructions...  
  
} catch (TETException $e) {  
    print "TET exception occurred:\n";  
    print "[" . $e->get_errnum() . "] " . $e->get_apiname() . ": "  
        $e->get_errmsg() . "\n";  
}  
catch (Exception $e) {  
    print $e;  
}
```

Note that you can use PHP 5-style exception handling regardless of whether you work with the old function-based TET interface, or the new object-oriented one.

3.9 RPG Binding

Installing the TET Edition for RPG. TET provides a */copy* module that defines all prototypes and some useful constants needed to compile ILE-RPG programs with embedded TET functions.

Since all functions provided by TET are implemented in the C language, you have to add *x'oo'* at the end of each string value passed to a TET function. All strings returned from TET will have this terminating *x'oo'* as well.

Using TET functions from RPG requires the compiled TET service program. To include the TET definitions at compile time you have to specify the name in the *D* specs of your ILE-RPG program:

```
d/copy QRPGLSRC,TETLIB
```

If the TET source file library is not on top of your library list you have to specify the library as well:

```
d/copy tetsrclib/QRPGLSRC,TETLIB
```

Before you start compiling your ILE-RPG program you have to create a binding directory that includes the TETLIB service program shipped with TET. The following example assumes that you want to create a binding directory called TETLIB in the library TETLIB:

```
CRTBNDDIR BNDDIR(TETLIB/TETLIB) TEXT('TETlib Binding Directory')
```

After creating the binding directory you need to add the TETLIB service program to your binding directory. The following example assumes that you want to add the service program TETLIB in the library TETLIB to the binding directory created earlier.

```
ADDBNDDIRE BNDDIR(TETLIB/TETLIB) OBJ((TETLIB/TETLIB *SRVPGM))
```

Now you can compile your program using the *CRTBNDRPG* command (or option 14 in PDM):

```
CRTBNDRPG PGM(TETLIB/EXTRACTOR) SRCFILE(TETLIB/QRPGLSRC) SRCMBR(*PGM) DFTACTGRP(*NO) BNDDIR(TETLIB/TETLIB)
```

Exception Handling in RPG. TET clients written in ILE-RPG can use a limited form of TET's try/catch mechanism as follows:

```
c          eval      rtn=tet_try(tet)
c          if        TET_open_document(tet:in_filename:0:optlist)=-1
c          or tet_catch(tet)=1
c      *
c          callp     TET_delete(tet)
c          eval      error='Couldn''t open input file '+
c                  %trim(out_filename)
c          exsr      exit
c          endif
```


4 Text Extraction Details

4.1 Characters and Glyphs

When dealing with text it is important to clearly distinguish the following concepts:

- ▶ *Characters* are the smallest units which convey information in a language. Common examples are the letters in the Latin alphabet, Chinese ideographs, and Japanese syllables. Characters have a meaning: they are semantic entities.
- ▶ *Glyphs* are different graphical variants which represent one or more particular characters. Glyphs have an appearance: they are representational entities.

There is no one-to-one relationship between characters and glyphs. For example, a ligature is a single glyph which is represented by two or more separate characters. On the other hand, a specific glyph may be used to represent different characters depending on the context (some characters look identical, see Figure 4.1).

Composite characters and sequences. Some glyphs map to a sequence of multiple characters. For example, some ligatures will be mapped to multiple characters according to their constituent characters. However, composite characters (such as the Roman numeral in Figure 4.1) may or may not be split, subject to information in the font and PDF. See Table 4.1, page 37, for a list of characters which will be post-processed by TET.

If appropriate, TET will split composite characters into a sequence of constituent characters. The corresponding sequence will be part of the text returned by `TET_get_text()`. For each character details of the underlying glyph can be obtained via `TET_get_char_info()`, including the information whether the character is the start or continuation of a sequence. Position information will only be returned for the first character of a sequence. Subsequent characters of a sequence will not have any associated position or width information, but must be processed in combination with the first character.

Characters

Glyphs

U+0067 LATIN SMALL LETTER G



U+0066 LATIN SMALL LETTER F +
U+0069 LATIN SMALL LETTER I



U+2126 OHM SIGN or
U+03A9 GREEK CAPITAL LETTER OMEGA



U+2167 ROMAN NUMERAL EIGHT or
U+0056 V U+0049 I U+0049 I U+0049 I



Fig. 4.1
Relationship of glyphs
and characters

Characters without any corresponding glyph. Although every glyph on the page will be mapped to one or more corresponding Unicode characters, not all characters delivered by TET actually correspond to a glyph. Characters which correspond to a glyph are called real characters, others are called artificial characters. There are several classes of artificial characters which will be delivered although a directly corresponding glyph is not available:

- ▶ A composite character (see above) will map to a sequence of multiple Unicode characters. While the first character in the sequence corresponds to the actual glyph, the remaining characters do not correspond to any glyph.
- ▶ Separator characters inserted via the *lineseparator/wordseparator/zoneseparator* options are artefacts without any corresponding glyph.
- ▶ While the leading value of a surrogate pair will be associated with a glyph, the trailing value will be treated as not having a corresponding glyph on the page (see Section 4.4, »Unicode Mapping«, page 37, section »Characters outside the BMP and surrogate handling«).

Text filtering. There are several situations where TET will modify the actual character values found on the page in order to make the results more useful. Most of these steps can be controlled via options. The following list gives an overview of all operations which may modify the text:

- ▶ Dehyphenation will remove hyphen characters and combine the parts of a hyphenated word. This can be disabled with the *dehyphenate* suboption of the *contentanalysis* option for *TET_open_page()*.
- ▶ Redundant text which creates only visual artifacts such as shadow effects or artificial bold text will be removed. This can be disabled with the *shadowdetect* suboption of the *contentanalysis* option for *TET_open_page()*.
- ▶ Very small or very large text can be ignored. The limits can be controlled with the *fontsize* option of *TET_open_page()*.
- ▶ Unicode post-processing will replace certain Unicode characters with more familiar ones. For example, Latin ligatures will be replaced with their constituent characters, and fullwidth ASCII variants in CJK fonts will be replaced with the corresponding non-fullwidth characters. For details see Table 4.1, page 37.
- ▶ Invisible text (text with *textrendering=3*) will be extracted by default, but this can be changed with the *ignoreinvisibletext* option of *TET_open_page()*.
- ▶ Glyphs which cannot be mapped to Unicode will be replaced with the Unicode character defined in the *unknownchar* option of *TET_open_document()*. See section »Un-mappable glyphs«, page 38.

4.2 Page and Text Geometry

Coordinate system. TET represents all page and text metrics in the default coordinate system of PDF. However, the origin of the coordinate system (which could be located outside the page) will be adjusted to the lower left corner of the visible page. More precisely, the origin will be located in the lower left corner of the *CropBox* if it is present, or the *MediaBox* otherwise. Page rotation will be applied if the page has a *Rotate* key. The coordinate system uses the DTP point as unit:

$$1 \text{ pt} = 1 \text{ inch} / 72 = 25.4 \text{ mm} / 72 = 0.3528 \text{ mm}$$

The first coordinate increases to the right, the second coordinate increases upwards. All coordinates expected or returned by TET are interpreted in this coordinate system, regardless of their representation in the underlying PDF document. See Section 5.1, »Simple pCOS Examples«, page 53 to see how to determine the size of a PDF page.

Acrobat 5 or above (full version only, not the free Reader) has a helpful facility for measuring distances on a PDF page. Simply choose *Window, Info* to display a measurement palette which uses points as units. Note that the coordinates displayed refer to an origin in the top left corner of the page, and not to an origin in the lower left corner as used in TET.

Glyph metrics. Using `TET_get_char_info()` you can retrieve font and metrics information for the characters which are returned for a particular glyph. The following values are available for each character in the output (see Figure 4.2 and Table 6.6, page 82):

- ▶ The *uv* value contains the UTF-32 Unicode value of the current character, i.e. the character for which details are retrieved. This field will always contain UTF-32, even in language bindings that can deal only with UTF-16 strings in their native Unicode strings. Accessing the *uv* field allows applications to deal with characters outside the BMP without having to interpret surrogate pairs. Since surrogate pairs will be reported as two separate characters, the *uv* field of the leading surrogate value will contain the actual Unicode value (larger than U+FFFF). The *uv* field of the trailing surrogate value will be treated as an artificial character, and will have an *uv* value of 0.
- ▶ The *type* field specifies how the character was created. There are two groups: real and artificial characters. The group of real characters comprises normal characters (i.e. the complete result of a single glyph) and characters which start a multi-character sequence that corresponds to a single glyph (e.g. the first character of a ligature). The group of artificial characters comprises the continuation of a multi-character sequence (e.g. the second character of a ligature), the trailing value of a surrogate pair, and inserted separator characters. For artificial characters the position (*x*, *y*) will specify the endpoint of the most recent real character, the *width* will be 0, and all other fields except *uv* will be those of the most recent real character. The endpoint is the point (*x*, *y*) plus the *width* added in direction *alpha* (in horizontal writing mode) or plus the *fontsize* in direction -90° (in vertical writing mode).
- ▶ The *unknown* field will usually be false (in C and C++: 0), but has a value of true (in C and C++: 1) if the original glyph could not be mapped to Unicode and has therefore been replaced with the character specified in the *unknownchar* option. Using this field you can distinguish real document content from replaced characters if you specified a common character as *unknownchar*, such as a question mark or space.

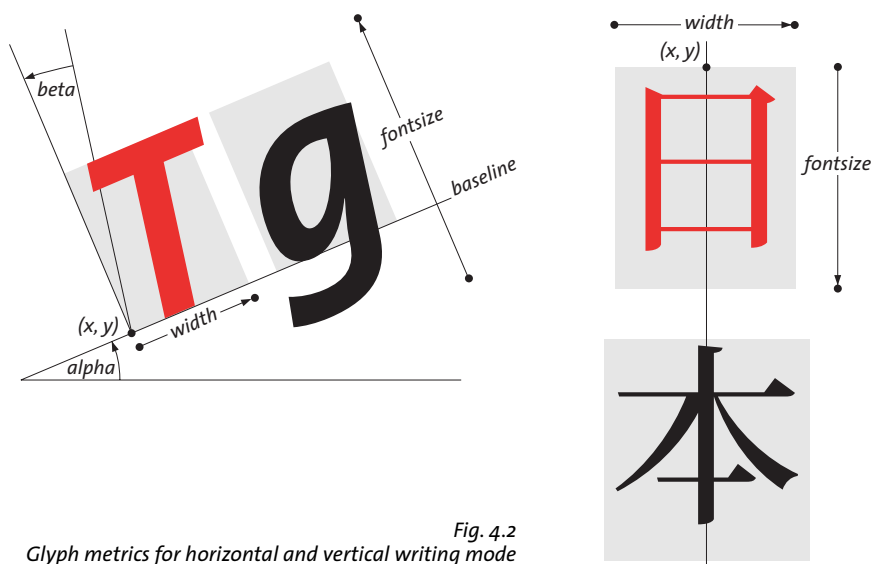


Fig. 4.2
Glyph metrics for horizontal and vertical writing mode

- ▶ The (x, y) fields specify the position of the glyph's reference point, which is the lower left corner of the glyph rectangle in horizontal writing mode, and the top center in vertical writing mode (see Section 4.3, «Support for Chinese, Japanese, and Korean Text», page 36 for details on vertical writing mode). For artificial characters, which do not correspond to any glyph on the page, the point (x, y) specifies the end point of the most recent real character.
- ▶ The *width* field specifies the width of a glyph according to the corresponding font metrics and text output parameters, such as character spacing and horizontal scaling. Since these parameters control the position of the next glyph, the distance between the reference points of two adjacent glyphs may be different from *width*. The *width* may be zero for non-spacing characters. On the other hand, the outline may actually be wider than the glyph's *width* value, e.g. for slanted text. The *width* will be 0 for artificial characters.
- ▶ The angle *alpha* provides the direction of inline text progression, specified as the deviation from the standard direction. The standard direction is 0° for horizontal writing mode, and -90° for vertical writing mode (see below for more details on vertical writing mode). Therefore, the angle *alpha* will be 0° for standard horizontal text as well as for standard vertical text.
- ▶ The angle *beta* specifies any skewing which has been applied to the text, e.g. for slanted (italicized) text. The angle will be measured against the perpendicular of *alpha*. It will be 0° for standard upright text (for both horizontal and vertical writing mode). If the absolute value of *beta* is greater than 90° the text will be mirrored at the baseline.
- ▶ The *fontid* field contains the pCOS ID of the font used for the glyph. It can be used to retrieve detailed font information, such as the font name, embedding status, writing mode (horizontal/vertical), etc. Section 5.1, «Simple pCOS Examples», page 53 shows sample code for retrieving font details.
- ▶ The *fontsize* field specifies the size of the text in points. It will be normalized, and therefore always be positive.

- The *textrendering* field specifies the kind of rendering for a glyph, e.g. stroked, filled, or invisible. It will reflect the numerical text rendering mode as defined for PDF page descriptions (see Table 6.6, page 82). Invisible text will be extracted by default, but this can be changed with the *ignoreinvisibletext* option of *TET_open_page()*.

End points of glyphs and words. Using the start point coordinates *x*, *y* and the *width* and *alpha* values returned by *TET_get_char_info()* you can determine the end point of a glyph in horizontal writing mode as follows:

```
xend = x + width * cos(alpha)
yend = y + width * sin(alpha)
```

In the common case of horizontally oriented text (i.e. *alpha*=0) this reduces to

```
xend = x + width
yend = y
```

For CJK text with vertical writing mode the end point calculation must be performed as follows:

```
xend = x
yend = y - fontsize
```

In order to calculate the end position of a word (e.g. for highlighting) determine the end position of the last character in the word.

Area of text extraction. By default, TET will extract all text from the visible page area. Using the *clippingarea* option of *TET_open_page()* (see Table 6.4, page 77) you can change this to any of the PDF page box entries (e.g. TrimBox). With the keyword *unlimited* all text regardless of any page boxes can be extracted.

The area of text extraction can be specified in more detail by providing an arbitrary number of rectangular areas in the *includebox* and *excludebox* options of *TET_open_page()*. This is useful for extracting partial page content (e.g. selected columns), or for excluding irrelevant parts (e.g. margins, headers and footers). The final clipping area is constructed by determining the union of all rectangles specified in the *includebox* option, and subtracting the union of all rectangles specified in the *excludebox* option. A character is considered inside the clipping area if its reference point is inside the clipping area. This means that a character could be considered inside the clipping area even if parts of it extend beyond the clipping area.

4.3 Support for Chinese, Japanese, and Korean Text

CJK support. TET supports Chinese, Japanese, and Korean (CJK) text, and will convert horizontal and vertical CJK text in arbitrary encodings (CMaps) to Unicode. TET supports all of Adobe's CJK character collections, which cover all PDF CMaps used in PDF versions up to and including 1.6:

- ▶ Simplified Chinese: *Adobe-GB1-4*
- ▶ Traditional Chinese: *Adobe-CNS1-4*
- ▶ Japanese: *Adobe-Japan1-6*
- ▶ Korean: *Adobe-Korea1-2*

The PDF CMaps in turn cover all of the CJK character encodings which are in use today, such as Shift-JIS, EUC, Big-5, KSC, and many others.

Note In order to extract CJK text you must configure access to the CMap files which are shipped with TET according to Section 0.1, »Installing the Software«, page 5.

Several groups of CJK characters will be modified (see Table 4.1, page 37, for details):

- ▶ Fullwidth ASCII variants and fullwidth symbol variants will be mapped to the corresponding halfwidth characters.
- ▶ CJK compatibility forms (prerotated glyphs for vertical text) and small form variants will be mapped to the corresponding normal variants.

CJK font names which are encoded with locale-specific encodings (e.g. Japanese font names encoded in Shift-JIS) will also be normalized to Unicode. The wordfinder will treat all ideographic CJK characters as individual words, while Katakana characters will not be treated as word boundaries (a sequence of Katakana will be treated as a single word).

CJK text with vertical writing mode. TET supports both horizontal and vertical writing modes, and performs all metrics calculations as appropriate for the respective writing mode. Keep the following in mind when dealing with text in vertical writing mode:

- ▶ The glyph reference point in vertical writing mode is at the top center of the glyph box. The text position will advance downwards as determined by the font size and character spacing, regardless of the glyph width (see Figure 4.2).
- ▶ The angle *alpha* will be 0° for standard vertical text. In other words, fonts with vertical writing mode and *alpha*=0° will progress downwards, i.e. in direction -90°.
- ▶ Because of the differences noted above client code must take the writing mode into account by using the pCOS code shown in Section 5.1, »Simple pCOS Examples«, page 53 for determining the writing mode of a font. Note that not all text which appears vertically actually uses a font with vertical writing mode.
- ▶ Prerotated glyphs for Latin characters and punctuation will be mapped to the corresponding unrotated Unicode character (see Table 4.1).

4.4 Unicode Mapping

TET is completely based on the Unicode standard which is only concerned about characters, but not about glyphs. While text in PDF can be represented with a variety of font and encoding schemes, TET will abstract from glyphs and normalize all text to Unicode characters, regardless of the original text representation in the PDF. Converting the information found in the PDF to the corresponding Unicode values is called *Unicode mapping*, and is crucial for understanding the semantics of the text (as opposed to rendering a visual representation of the text on screen or paper). In order to provide proper Unicode mapping TET consults various data structures which are found in the PDF document, embedded or external font files, as well as builtin and user-supplied tables. In addition, it applies several methods to determine the Unicode mapping for non-standard glyph names.

However, despite all efforts there are still PDF documents where some text cannot be mapped to Unicode. In order to deal with these cases TET offers a number of configuration features which can be used to control Unicode mapping for problematic PDF files. These features are discussed in Section 4.5, »Advanced Unicode Mapping Controls«, page 39.

Post-processing for certain Unicode values. In some cases the Unicode values which are determined as a result of font and encoding processing will be modified by a post-processing step, e.g. to split ligatures. Table 4.1 lists all Unicode values which are affected by post-processing.

Table 4.1 Post-processing for various Unicode values

UTF-16 values	Processing
U+0000-U+001F, U+007F-U+009F	Control characters will be removed. ¹ Mapping to U+0000 may be useful for eliminating unwanted characters.
U+0020, U+00A0, U+3000, U+2000-U+200B	Space characters will be mapped to U+0020.
U+D800 - U+DBFF (high) U+DC00 - U+DFFF (low) surrogates	Leading (high) surrogates and trailing (low) surrogates will be maintained in the UTF-16 output, and the corresponding UTF-32 value will be available in the uv field (see »Characters outside the BMP and surrogate handling«, page 38).
U+E000-U+F8FF (Private Use Area, PUA)	PUA characters will be kept or replaced according to the <i>keepua</i> option (see Section »Unmappable glyphs«).
U+F600-U+F8FF (Adobe CUS)	Will be mapped to the corresponding characters outside the CUS.
U+FB00-U+FB17	Latin and Armenian ligatures will be decomposed into their constituent characters. ²
U+FF01-U+FF5E, U+FF60-U+FF6F	Fullwidth ASCII and symbol variants will be mapped to the corresponding non-fullwidth characters. ³
U+FE30-U+FE6F	CJK compatibility forms (prerotated glyphs for vertical text) and small form variants (U+FE30-U+FE6F) will be mapped to the corresponding normal variants
undefined Unicode values	no modification

1. Characters inserted via the *wordseparator*, *lineseparator*, and *zoneseparator* options are not subject to this removal.
2. Ligatures in the Arabic and Hebrew presentation forms will not be decomposed.
3. The following characters will be left unchanged: halfwidth CJK punctuation (U+FF61-U+FF64), Katakana variants (U+FF65-U+FF9F), Hangul variants (U+FFA0-U+FFDC), and symbol variants (U+FFE8-U+FFEE).

Characters outside the BMP and surrogate handling. Characters outside Unicode's Basic Multilingual Plane (BMP), i.e. those with Unicode values above 0xFFFF, cannot be expressed as a single UTF-16 value, but require a pair of UTF-16 values called a surrogate pair. Examples of characters outside the BMP include certain mathematical and musical symbols at U+1DXXX as well as thousands of CJK extension characters starting at U+20000.

TET interprets and maintains surrogates, and allows access to the corresponding UTF-32 value even in programming languages where native Unicode strings support only UTF-16. Leading (high) surrogates and trailing (low) surrogates will be maintained. The string returned by `TET_get_text()` will return two UTF-16 values where the leading value will be treated as a real character which carries the glyph properties (size, font, etc.). The trailing value will be treated as an artificial character without any corresponding glyph.

The `uv` field returned by `TET_get_char_info()` for the leading surrogate value will contain the corresponding UTF-32 value. This allows direct access to the UTF-32 value of a non-BMP character even if you are working in an UTF-16 environment without any support for UTF-32. The `uv` field for the trailing surrogate value will be 0.

Unmappable glyphs. There are several reasons why text in a PDF cannot reliably be mapped to Unicode. If the document does not contain a ToUnicode CMap with Unicode mapping information for the codes used on the page, Unicode values can be missing for various reasons:

- ▶ Type 1 fonts may contain unknown glyph names, and TrueType, OpenType, or CID fonts may be addressed with glyph ids without any Unicode values in the font or PDF. By default, TET will assign the Unicode Replacement Character U+FFFD to these characters. You can select a different replacement character (e.g. the space character U+0020) for unmappable glyphs with the `unknownchar` option of `TET_open_document()`.

However, if the `keepua` option of `TET_open_document()` is `true`, unknown characters will be mapped to increasing values in the Private Use Area (PUA), starting at U+F200. The same glyph name used in different fonts will end up with the same PUA value, while TrueType or OpenType glyph ids from different fonts will have different PUA values assigned.

- ▶ If the font or PDF provides Unicode values, these may be contained in the Private Use Area (PUA). Since PUA characters are generally not very useful, TET will replace them with `unknownchar` (by default: U+FFFD). However, if the `keepua` option of `TET_open_document()` is `true`, PUA values will be returned without any modification. This may be useful if you can deal with PUA values, e.g. for a specific font, or for all fonts from a specific font vendor.

Since not all glyphs in a document may have proper Unicode values (e.g. custom symbols), TET may have to map some glyphs to `unknownchar`. Your code should be prepared for this character. If you don't care about Unicode mapping problems you can simply ignore it, or use the `unknownchar` option of `TET_open_document()` to set a different character as a replacement for unmappable glyphs (e.g. the `space` character).

In order to check for unmappable glyphs you can use the `unknown` field returned by `TET_get_char_info()`.

4.5 Advanced Unicode Mapping Controls

TET implements many workarounds in order to process PDF documents which actually don't contain Unicode values so that it can successfully extract the text nevertheless. However, there are still documents where the text cannot be extracted since not enough information is available in the PDF and relevant font data structures. TET contains various configuration features which can be used to supply additional Unicode mapping information. These features are detailed in this section.

Summary of Unicode mapping controls. Using the *glyphmapping* option of *TET_open_document()* (see Section 6.4, »Document Functions«, page 73) you can control Unicode mapping for glyphs in several ways. The following list gives an overview of available methods (which can be combined). These controls can be applied on a per-font basis or globally for all fonts in a document:

- ▶ The suboption *forceencoding* can be used to completely override all occurrences of the predefined PDF encodings *WinAnsiEncoding* or *MacRomanEncoding*.
- ▶ The suboptions *codelist* and *tounicodecmap* can be used to supply Unicode values in a simple text format (a *codelist* resource).
- ▶ The suboption *glyphlist* can be used to supply Unicode values for non-standard glyph names.
- ▶ The suboption *glyphrule* can be used to define a rule which will be used to derive Unicode values from numerical glyph names in an algorithmic way. Several rules are already built into TET. The option *encodinghint* can be used to control the internal rules.
- ▶ In addition to dozens of predefined encodings, custom encodings can be defined for use with the *encodinghint* option or the *encoding* suboption of the *glyphrule* option.
- ▶ External fonts can be configured to provide Unicode mapping information if the PDF does not provide enough information and the font is not embedded in the PDF.

Analyzing PDF documents with the PDFlib FontReporter plugin¹. In order to obtain the information required to create appropriate Unicode mapping tables you must analyze the problematic PDF documents.

PDFlib GmbH provides a free companion product to TET which assists in this situation: PDFlib FontReporter is an Adobe Acrobat plugin for easily collecting font, encoding, and glyph information. The plugin creates detailed font reports containing the actual glyphs along with the following information:

- ▶ The corresponding code: the first hex digit is given in the left-most column, the second hex digit is given in the top row. For CID fonts the offset printed in the header must be added to obtain the code corresponding to the glyph.
- ▶ The glyph name if present.
- ▶ The Unicode value(s) corresponding to the glyph (if Acrobat can determine them).

These pieces of information play an important role for TET's glyph mapping controls. Figure 4.3 shows two pages from a sample font report. Font reports created with the FontReporter plugin can be used to analyze PDF fonts and create mapping tables for successfully extracting the text with TET. It is highly recommended to take a look at the corresponding font report if you want to write Unicode mapping tables or glyph name heuristics to control text extraction with TET.

1. The PDFlib FontReporter plugin is available for free download at www.pdflib.com/products/fontreporter

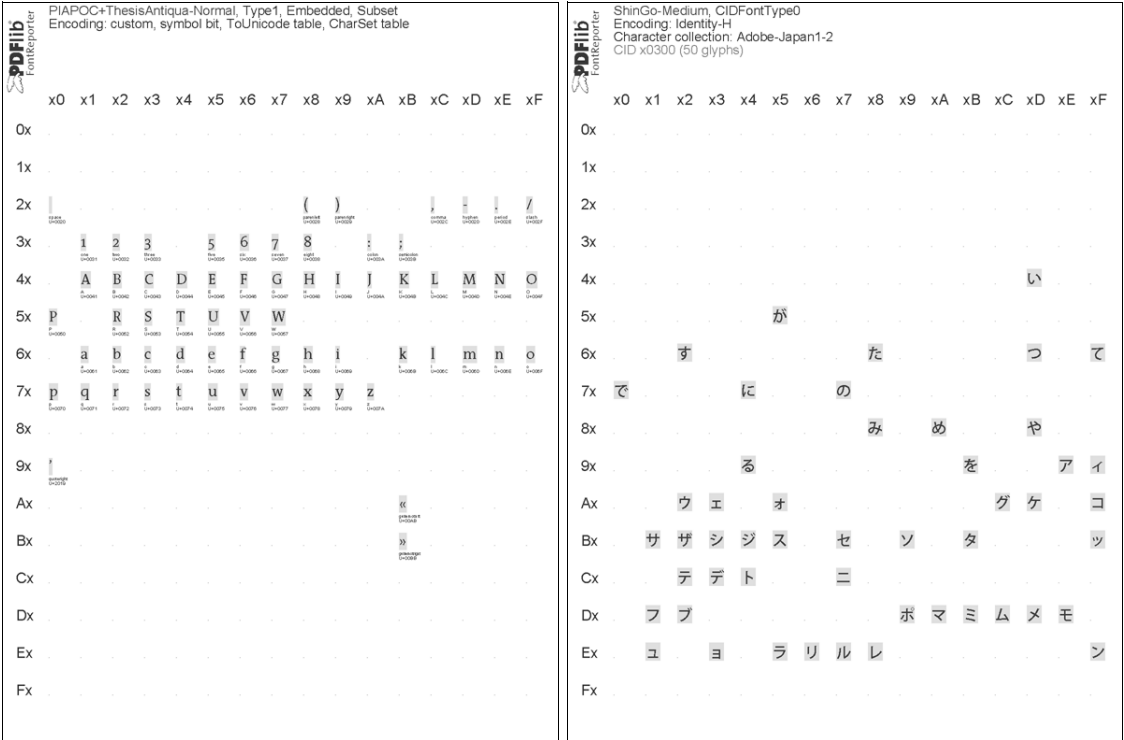


Fig. 4.3
Sample font reports created with the PDFlib FontReporter plugin for Adobe Acrobat

Precedence rules. TET will apply the glyph mapping controls in the following order:

- ▶ Codelist and ToUnicode CMap resources will be consulted first.
- ▶ If the font has an internal ToUnicode CMap it will be considered next.
- ▶ For glyph names TET will apply an external or internal glyph name mapping rule if one is available which matches the font and glyph name.
- ▶ Lastly, a user-supplied glyph list will be applied.

Code list resources for all font types. Code lists are similar to glyph lists except that they specify Unicode values for individual codes instead of glyph names. Although multiple fonts from the same foundry may use identical code assignments, codes (also called glyph ids) are generally font-specific. As a consequence, separate code lists will be required for individual fonts. A code list is a text file where each line describes a Unicode mapping for a single code according to the following rules:

- ▶ Text after a percent sign '%' will be ignored; this can be used for comments.
- ▶ The first column contains the glyph code in decimal or hexadecimal notation. This must be a value in the range 0-255 for simple fonts, and in the range 0-65535 for CID fonts.
- ▶ The remainder of the line contains up to 7 Unicode values for the code. The values can be supplied in decimal notation or (with the prefix x or 0x) in hexadecimal notation.



Fig. 4.4
The font report for a logotype font shows that the font contains wrong Unicode mappings.
A custom code list can correct such mappings.

By convention, code lists use the file name suffix `.cl`. Code lists can be configured with the `codelist` resource. If no code list resource has been specified explicitly, TET will search for a file named `<mycodelist>.gl` (where `<mycodelist>` is the resource name) in the `search-path` hierarchy (see Section 4.7, »Resource Configuration and File Searching«, page 47 for details). In other words: if the resource name and the file name (without the `.cl` suffix) are identical you don't have to configure the resource since TET will implicitly do the equivalent of the following call (where `name` is an arbitrary resource name):

```
TET_set_option(tet, "codelist {name name.cl}");
```

The following sample demonstrates the use of code lists. Consider the mismapped logotype glyphs in Figure 4.4 where a single glyph of the font actually represents multiple characters, and all characters together create the company logotype. However, the glyphs are wrongly mapped to the characters *a*, *b*, *c*, *d*, and *e*. In order to fix this you could create the following code list:

% Unicode mappings for codes in the GlobeLogosOne font

```
x61      x0054 x0068 x0065 x0020      % The
x62      x0042 x006F                      % Bo
x63      x0073 x0074 x006F x006E x0020 % ston
x64      x0047 x006C x006F              % Glo
x65      x0062 x0065                    % be
```

Then supply the codelist with the following option to `TET_open_document()` (assuming the code list is available in a file called `GlobeLogosOne.cl` and can be found via the search path):

```
glyphmapping {{fontname=GlobeLogosOne codelist=GlobeLogosOne}}
```

ToUnicode CMap resources for all font types. PDF supports a data structure called ToUnicode CMap which can be used to provide Unicode values for the glyphs of a font. If this data structure is present in a PDF file TET will use it. Alternatively, a ToUnicode CMap can be supplied in an external file. This is useful when a ToUnicode CMap in the PDF is incomplete, contains wrong entries, or is missing. A ToUnicode CMap will take precedence over a code list. However, code lists use an easier format the ToUnicode CMaps so they are the preferred format.

By convention, CMaps don't use any file name suffix. ToUnicode CMaps can be configured with the `cmap` resource (see Section 4.7, »Resource Configuration and File Searching«, page 47). The contents of a `cmap` resource must adhere to the standard CMap syntax.¹ In order to apply a ToUnicode CMap to all fonts in the *Warnock* family use the following option to `TET_open_document()`:

1. See partners.adobe.com/public/developer/en/acrobat/5411.ToUnicode.pdf

```
glyphmapping {{fontname=Warnock* tounicodecmap=warnock}}
```

Glyph list resources for simple fonts. Glyph lists (short for: glyph name lists) can be used to provide custom Unicode values for non-standard glyph names, or override the existing values for standard glyph names. A glyph list is a text file where each line describes a Unicode mapping for a single glyph name according to the following rules:

- ▶ Text after a percent sign '%' will be ignored; this can be used for comments.
- ▶ The first column contains the glyph name. Any glyph name used in a font can be used (i.e. even the Unicode values of standard glyph names can be overridden). In order to use the percent sign as part of a glyph name the sequence \% must be used (since the percent sign serves as the comment introducer).
- ▶ The remainder of the line contains up to 7 Unicode values for the glyph name. The values can be supplied in decimal notation or (with the prefix x or ox) in hexadecimal notation.
- ▶ Unprintable characters in glyph names can be inserted by using escape sequences for text files (see Section 4.7, »Resource Configuration and File Searching«, page 47)

By convention, glyph lists use the file name suffix *.gl*. Glyph lists can be configured with the *glyphlist* resource. If no glyph list resource has been specified explicitly, TET will search for a file named <myglyphlist>.gl (where <myglyphlist> is the resource name) in the *searchpath* hierarchy (see Section 4.7, »Resource Configuration and File Searching«, page 47, for details). In other words: if the resource name and the file name (without the *.gl* suffix) are identical you don't have to configure the resource since TET will implicitly do the equivalent of the following call (where *name* is an arbitrary resource name):

```
TET_set_option(tet, "glyphlist {name name.gl}");
```

Due to the precedence rules for glyph mapping glyph lists will not be consulted if the font contains a ToUnicode CMap. The following sample demonstrates the use of glyph lists:

% Unicode values for glyph names used in TeX documents

```
precedesequal 0x227C
similarequal  0x2243
negationslash 0x2044
union         0x222A
prime         0x2032
```

In order to apply a glyph list to all font names starting with *CMSY* use the following option for *TET_open_document()*:

```
glyphmapping {{fontname=CMSY* glyphlist=tarski}}
```

Rules for interpreting numerical glyph names in simple fonts. Sometimes PDF documents contain glyphs with names which are not taken from some predefined list, but are generated algorithmically. This can be a »feature« of the application generating the PDF, or may be caused by a printer driver which converts fonts to another format: sometimes the original glyph names get lost in the process, and are replaced with schematic names such as *Goo*, *Go1*, *Go2*, etc. TET contains builtin glyph name rules for processing numerical glyph names created by various common applications and drivers. Since the same glyph names may be created for different encodings you can provide the

encodinghint option to *TET_open_document()* in order to specify the target encoding for schematic glyph names encountered in the document. For example, if you know that the document contains Russian text, but the text cannot successfully be extracted for lack of information in the PDF, you can supply the option *encodinghint=cp1250* to specify a Cyrillic codepage.

In addition to the builtin rules for interpreting numerical glyph names you can define custom rules with the *fontname* and *glyphrule* suboptions of the *glyphmapping* option of *TET_open_document()*. You must supply the following pieces of information:

- ▶ The full or abbreviated name of the font to which the rule will be applied (*fontname* option)
- ▶ A prefix for the glyph names, i.e. the characters before the numerical part (*prefix* suboption)
- ▶ The base (decimal or hexadecimal) in which the numbers will be interpreted (*base* suboption)
- ▶ The encoding in which to interpret the resulting numerical codes (*encoding* suboption)

For example, if you determined (e.g. using PDFlib FontReporter) that the glyphs in the fonts *T1*, *T2*, *T3*, etc. are named *co0*, *co1*, *co2*, ..., *cFF* where each glyph name corresponds to the WinAnsi character at the respective hexadecimal position (*00*, ..., *FF*) use the following option for *TET_open_document()*:

```
glyphmapping {{fontname=T* glyphrule={prefix=c base=hex encoding=winansi} }}
```

External font files and system fonts. If a PDF does not contain sufficient information for Unicode mapping and the font is not embedded, you can configure additional font data which TET will use to derive Unicode mappings. Font data may come from a TrueType or OpenType font file on disk, which can be configured with the *fontoutline* resource category. As an alternative on Mac and Windows systems, TET can access fonts which are installed on the host operating system. Access to these host fonts can be disabled with the *usehostfonts* option in *TET_open_document()*.

In order to configure a disk file for the *WarnockPro* font use the following call:

```
TET_set_option(tet, "fontoutline {WarnockPro=WarnockPro.otf}");
```

See Section 4.7, »Resource Configuration and File Searching«, page 47 for more details on configuring external font files.

4.6 Content Analysis

PDF documents provide the semantics (Unicode mapping) of individual text characters as well as their position on the page. However, they generally do not convey information about words, lines, columns or other high-level text units. The fragments comprising text on a page may contain individual characters, syllables, words, lines, or an arbitrary mixture thereof, without any explicit marks designating the start or end of a word, line, or column.

To make matters worse, the ordering of text fragments on the page may be different from the logical (reading) order. There are no rules for the order in which portions of text are placed on the page. For example, a page containing two columns of text could be produced by creating the first line in the left column, followed by the first line of the right column, the second line of the left column, the second line of the right column etc. However, logical order requires all text in the left column to be processed before the text in the right column is processed. Extracting text from such documents by simply replaying the instructions on the PDF page generally provides undesirable results since the logical structure of the text is lost.

TET's content analysis engine analyzes the contents, position, and relationship of text fragments in order to achieve the following goals:

- ▶ create words from characters, and insert separator characters between words if desired
- ▶ remove redundant text, such as duplicates which are only present to create a shadow effect
- ▶ recombine the parts of hyphenated words which span more than one line
- ▶ identify text columns (zones)
- ▶ sort text fragments within a zone, as well as zones within a page

These operations will be discussed in more detail below, as well as options which provide some control over content processing.

Text granularity. The *granularity* option of `TET_open_page()` specifies the amount of text that will be returned by a single call to `TET_get_text()`:

- ▶ With *granularity=glyph* each fragment contains the result of mapping one glyph, which may be more than one character (e.g. for ligatures). In this mode content analysis will be disabled. TET will return the original text fragments on the page in their original order. Although this is the fastest mode, it is only useful if the TET client intends to do sophisticated post-processing (or is only interested in the text position, but not in its logical structure) since the text may be scattered all over the page.
- ▶ With *granularity=word* the wordfinder algorithm will group characters into logical words. Each fragment contains a word; punctuation characters (comma, colon, question mark, quotes, etc.) will be returned as separate fragments as well.
- ▶ With *granularity=line* the words identified by the wordfinder will be grouped into lines. If dehyphenation is enabled (which is the default) the parts of hyphenated words at the end of a line will be combined, and the full dehyphenated word will be part of the line.
- ▶ With *granularity=zone* all words contained in a rectangular area called zone will be returned. A zone can be considered a single text column or similar unit.
- ▶ With *granularity=page* all words on the page will be returned in a single fragment.

Separator characters will be inserted between multiple words, lines, or zones if the chosen granularity is larger than the respective unit. For example, with *granularity=word* there's no need to insert separator characters since each call to *TET_get_text()* will return exactly one word.

The separator characters can be specified with the *wordseparator*, *lineseparator*, and *zoneseparator* suboptions of the *contentanalysis* option in *TET_open_page()* (use U+0000 to disable a separator), for example:

```
contentanalysis={zoneseparator=U+000C}
```

By default, all content processing operations will be disabled for *granularity=glyph*, and enabled for all other granularity settings. However, more fine-grain control is possible via separate options (see below).

Word boundary detection. The wordfinder, which is enabled for all granularity modes except *glyph*, creates logical words from multiple glyphs which may be scattered all over the page in no particular order. Word boundaries are identified by two criteria:

- ▶ A sophisticated algorithm analyzes the geometric relationship among glyphs to find character groups which together form a word. The algorithm takes into account a variety of properties and special cases in order to accurately identify words even in complicated layouts and for arbitrary text ordering on the page.
- ▶ Some characters, such as space and punctuation characters (e.g. colon, comma, full stop, parentheses) will be considered a word boundary, regardless of their width and position. Note that ideographic CJK characters will be considered word boundaries, while Katakana characters will not be treated as word boundaries. If the *punctuationbreaks* option in *TET_open_page()* is set to *false*, the wordfinder will no longer treat punctuation characters as word boundaries:

```
contentanalysis={punctuationbreaks=false}
```

Ignoring punctuation characters for word boundary detection can, for example, be useful for maintaining Web URLs where period and slash characters are usually considered part of a word (see Figure 4.5).

Note Currently there is no dedicated support for right-to-left scripts and bidirectional text. Although Unicode values and glyph metrics can be retrieved, the wordfinder does not apply any special handling for right-to-left text.

Dehyphenation. Hyphenated words at the end of a line are usually not desired for applications which process the extracted text on a logical level. TET will therefore dehyphenate, or recombine the parts of a hyphenated word. More precisely, if a word at the end of a line ends with a hyphen character and the first word on the next line starts with a lowercase character, the hyphen will be removed and the first part of the word



Fig. 4.5
The default setting *punctuationbreaks=true* will separate the parts of URLs (top), while *punctuationbreaks=false* will keep the parts together (bottom).

will be combined with the part on the next line, provided there is at least one more line in the same zone. The parts of a hyphenated word will not be modified, only the hyphen will be removed. Special handling for hyphenated words can be disabled with the *dehyphenate* suboption for the *contentanalysis* option of *TET_open_page()*.

Note Hyphenated words at the end of a zone will not be identified, and consequently there won't be any dehyphenation (i.e. the hyphen will remain part of the text).

Shadow and fake bold text removal. PDF documents sometimes include redundant text which does not contribute to the semantics of a page, but creates certain visual effects only. Shadow text effects are usually achieved by placing two or more copies of the actual text on top of each other, where a small displacement is applied. Applying opaque coloring to each layer of text provides a visual appearance where the majority of the text in lower layers is obscured, while the visible portions create a shadow effect.

Similarly, word processing applications sometimes support a feature for creating artificial bold text. In order to create bold text appearance even if a bold font is not available, the text is placed repeatedly on the page in the same color. Using a very small displacement the appearance of bold text is simulated.

Shadow simulation, artificial bold text, and similar visual artifacts create severe problems when reusing the extracted text since redundant text contents which contribute only to the visual appearance will be processed although the text does not contribute to the page contents.

If the wordfinder is enabled, TET will identify and remove such redundant visual artifacts by default. This process can be disabled with the *shadowdetect* suboption for the *contentanalysis* option of *TET_open_page()*.

Zones and reading order. Zones can be thought of as text columns, although they may sometimes cover other areas on the page, such as headers and footers, marginal notes, or pagination artifacts. Conceptually, a zone is an »island of text«, consisting of text lines which are placed close to each other, and surrounded by white space which separates it from other zones. Technically, zones are a combination of logically connected rectangular strips holding at most one line of text. A zone may contain multiple paragraphs.

TET will arrange the zones identified on a page so that their ordering reflects the logical (reading) order of the text. This process may not work perfectly for complex layouts.

4.7 Resource Configuration and File Searching

UPR files and resource categories. In some situations TET needs access to resources such as encoding definitions or glyph name mapping tables. In order to make resource handling platform-independent and customizable, a configuration file can be supplied for describing the available resources along with the names of their corresponding disk files. In addition to a static configuration file, dynamic configuration can be accomplished at runtime by adding resources with `TET_set_option()`. For the configuration file a simple text format called *Unix PostScript Resource* (UPR) is used. The UPR file format as used by TET will be described below. TET supports the resource categories listed in Table 4.2.

Table 4.2 Resource categories (all file names must be specified in UTF-8)

category	format	explanation
<i>cmap</i>	<i>key=value</i>	Resource name and file name of a CMap
<i>codelist</i>	<i>key=value</i>	Resource name and file name of a code list
<i>encoding</i>	<i>key=value</i>	Resource name and file name of an encoding
<i>glyphlist</i>	<i>key=value</i>	Resource name and file name of a glyph list
<i>hostfont</i>	<i>key=value</i>	Name of a host font resource (key is the PDF font name; value is the UTF-8 encoded host font name) to be used for an unembedded font
<i>fontoutline</i>	<i>key=value</i>	Font and file name of a TrueType or OpenType font to be used for an unembedded font
<i>searchpath</i>	<i>value</i>	Relative or absolute path name of directories containing data files

The UPR file format. UPR files are text files with a very simple structure that can easily be written in a text editor or generated automatically. To start with, let’s take a look at some syntactical issues:

- ▶ Lines can have a maximum of 255 characters.
- ▶ A backslash ‘\’ escapes newline characters. This may be used to extend lines.
- ▶ An isolated period character ‘.’ serves as a section terminator.
- ▶ Comment lines may be introduced with a percent ‘%’ character, and terminated by the end of the line.
- ▶ Whitespace is ignored everywhere except in resource names and file names.

UPR files consist of the following components:

- ▶ A magic line for identifying the file. It has the following form:

PS-Resources-1.0

- ▶ A section listing all resource categories described in the file. Each line describes one resource category. The list is terminated by a line with a single period character.
- ▶ A section for each of the resource categories listed at the beginning of the file. Each section starts with a line showing the resource category, followed by an arbitrary number of lines describing available resources. The list is terminated by a line with a single period character. Each resource data line contains the name of the resource (equal signs have to be quoted). If the resource requires a file name, this name has to be added after an equal sign. The *searchpath* (see below) will be applied when TET searches for files listed in resource entries.

Sample UPR file. The following listing gives an example of a UPR configuration file:

```
PS-Resources-1.0
searchpath
glyphlist
codelist
encoding
.
searchpath
/usr/local/lib/cmaps
/users/kurt/myfonts
.
glyphlist
myglyphlist=/usr/lib/sample.gl
.
codelist
mycodelist=/usr/lib/sample.cl
.
encoding
myencoding=sample.enc
.
```

File searching and the *searchpath* resource category. In addition to relative or absolute path names you can supply file names without any path specification to TET. The *searchpath* resource category can be used to specify a list of path names for directories containing the required data files. When TET must open a file it will first use the file name exactly as supplied, and try to open the file. If this attempt fails, TET will try to open the file in the directories specified in the *searchpath* resource category one after another until it succeeds. Multiple *searchpath* entries can be accumulated, and will be searched in reverse order (paths set at a later point in time will be searched before earlier ones). In order to disable the search you can use a fully specified path name in the TET functions.

On Windows TET will initialize the *searchpath* resource category with a value read from the following registry key:

```
HKLM\SOFTWARE\PDFlib\TET\2.3\searchpath
```

This registry entry may contain a list of path names separated by a semicolon ';' character. The Windows installer will initialize the *searchpath* registry entry with the following directory names (or similar if you installed TET in a custom directory):

```
C:\Program Files\PDFlib\TET 2.3\resource
C:\Program Files\PDFlib\TET 2.3\resource\cmap
```

On IBM iSeries the *searchpath* resource category will be initialized with the following values:

```
/tet/2.3/resource
/tet/2.3/resource/cmap
```

On MVS the *searchpath* feature is not supported.

Searching for the UPR resource file. If resource files are to be used you can specify them via calls to *TET_set_option()* (see below) or in a UPR resource file. TET reads this file automatically when the first resource is requested. The detailed process is as follows:

- ▶ If the environment variable *TETRESOURCEFILE* is defined TET takes its value as the name of the UPR file to be read. If this file cannot be read an exception will be thrown.
- ▶ If the environment variable *TETRESOURCEFILE* is not defined TET tries to open a file with the following name:

```
upr (on MVS; a dataset is expected)
/tet/2.3/tet.upr (on IBM eServer iSeries)
tet.upr (Windows, Unix, and all other systems)
```

If this file cannot be read no exception will be thrown.

- ▶ On Windows TET will additionally try to read the following registry entry:

```
HKLM\SOFTWARE\PDFlib\TET\2.3\resourcefile
```

The value of this key (which will be created with the value *<installdir>/tet.upr* by the TET installer, but can also be created by other means) will be taken as the name of the resource file to be used. If this file cannot be read an exception will be thrown.

- ▶ The client can force TET to read a resource file at runtime by explicitly setting the *resourcefile* option:

```
TET_set_option(tet, "resourcefile=/path/to/tet.upr");
```

This call can be repeated arbitrarily often; the resource entries will be accumulated.

Configuring resources at runtime. In addition to using a UPR file for the configuration, it is also possible to directly configure individual resources at runtime via *TET_set_option()*. This function takes a resource category name and pairs of corresponding resource names and values as it would appear in the respective section of this category in a UPR resource file, for example:

```
TET_set_option(tet, "glyphlist={myglyphnames=/usr/local/glyphnames.gl}");
```

Multiple resource names can be configured in a single option list for a resource category option (but the same resource category option cannot be repeated in a single call to *TET_set_option()*). Alternatively, multiple calls can be used to accumulate resource settings.

Escape sequences for text files. Special character sequences can be used to include unprintable characters in text files. All sequences start with a backslash '**' character:

- ▶ *\x* introduces a sequence of two hexadecimal digits (*0-9, A-F, a-f*), e.g. *\x0D*
- ▶ *\nnn* denotes a sequence of three octal digits (*0-7*), e.g. *\015*. The sequence *\ooo* will be ignored.
- ▶ The sequence ** denotes a single backslash.
- ▶ A backslash at the end of a line will cancel the end-of-line character.

Escape sequences are supported in all text files except UPR configuration files and CMap files.

4.8 Recommendations for common Scenarios

TET offers a variety of options which you can use to control various aspects of the text extraction process. In this section we provide some recommendations for typical TET application scenarios. Please refer to Chapter 6, »TET Library API Reference«, page 67 for details on the functions and options mentioned below.

Optimizing performance. In certain situations, particularly for search engines, text extraction speed may be crucial, and may play a more important role than optimal output. The default settings of TET have been selected to achieve the best possible output, but can be adjusted to speed up processing. Some tips for choosing options in `TET_open_page()` to maximize text extraction throughput:

- ▶ `contentanalysis={merge=0}`

This will disable the expensive strip and zone merging step, and reduces processing times for typical files to ca. 60% compared to default settings. However, documents where the contents are scattered across the pages in arbitrary order may result in some text which is not extracted in logical order.

- ▶ `contentanalysis={dehyphenate=false}`

This will disable the combination of the parts of hyphenated words. If dehyphenation is not required this option can slightly reduce processing times.

- ▶ `contentanalysis={shadowdetect=false}`

This will disable detection of redundant shadow and fake bold text, which can also reduce processing times.

Multiple suboptions of the `contentanalysis` option must be combined into a single list, for example:

```
contentanalysis={merge=0 shadowdetect=false}
```

Words vs. line layout vs. reflowable text. Different applications will prefer different kinds of output (hyphenated words will always be dehyphenated with these settings):

- ▶ Individual words (ignore layout): a search engine may not be interested in any layout-related aspects, but only the words comprising the text. In this situation use `granularity=word` in `TET_open_page()` to retrieve one word per call to `TET_get_text()`.
- ▶ Keep line layout: use `granularity=page` in `TET_open_page()` for extracting the full text contents of a page in a single call to `TET_get_text()`. Text lines will be separated with a linefeed character to retain the existing line structure.
- ▶ Reflowable text: in order to avoid line breaks and facilitate reflowing of the extracted text use `contentanalysis={lineseparator=U+0020}` and `granularity=page` in `TET_open_page()`. The full page contents can be fetched with a single call to `TET_get_text()`. Zones will be separated with a linefeed character, and a space character will be inserted between the lines in a zone.

Writing a search engine or indexer. Indexers are usually not interested in the position of text on the page (unless they provide search term highlighting). In many cases they will tolerate errors which occur in Unicode mapping, and process whatever text contents they can get. Recommendations:

- ▶ Use *granularity=word* in *TET_open_page()*.
- ▶ If the application knows how to process punctuation characters you can keep them with the adjacent text by setting the option *contentanalysis={punctuationbreaks=false}* in *TET_open_page()*.

Geometry. The geometry features may be useful for some applications:

- ▶ The *TET_get_char_info()* interface is only required if you need the position of text on the page, the respective font name, or other details. If you are not interested in text coordinates calling *TET_get_text()* will be sufficient.
- ▶ If you have advance information about the layout of pages you can use the *include-box* and/or *excludebox* options in *TET_open_page()* to get rid of headers, footers, or similar items which are not part of the main text.

Which parts of a document? The text contained in a PDF document may be part of various data structures:

- ▶ The actual page contents can be extracted with *TET_get_text()*.
- ▶ Document info fields, bookmark text, form field contents, XMP metadata, and other hypertext elements can be retrieved with *TET_pcos_get_string()* and *TET_pcos_get_stream()* (see Section 5.1, »Simple pCOS Examples«, page 53).
- ▶ PDF documents may contain file attachments which are themselves PDF documents. In order to extract the text of PDF file attachments you must first fetch the attachments with *TET_pcos_get_stream()*, and then feed the attachment to *TET_open_document_mem()* (request sample code from PDFlib GmbH support for details).
- ▶ TET can not extract text from raster images or vectorized text.

Unknown characters. TET may be unable to determine the appropriate Unicode mapping for one or more characters, and represent it with the Unicode replacement character U+FFFD. If your application is not concerned about unmappable characters you can simply discard all occurrences of this character. Applications which require more fine-grain results could take the corresponding font into account, and use it to decide on processing of unmappable characters.

Legal documents. When dealing with legal documents there is usually a zero tolerance for wrong Unicode mappings since they might alter the content or interpretation of a document. In many cases the text position is not required, and the text must be extracted word by word. Recommendations:

- ▶ Use the *granularity=word* option in *TET_open_page()*.
- ▶ Use the *password=xxx* option in *TET_open_document()* if you must process documents which require a password for opening, or if content extraction is not allowed in the permission settings.
- ▶ For absolute text fidelity: stop processing as soon as the *unknown* field in the character info structure returned by *TET_get_char_info()* is 1, or if the Unicode replacement character U+FFFD is part of the string returned by *TET_get_text()*. Do not set the *unknownchar* option to any common character since you may be unable to distinguish it from correctly mapped characters without checking the *unknown* field. If

you know that certain characters are not critical, do not skip them, but provide a mapping table which maps them to appropriate values.

- ▶ Also to ensure text fidelity you may want to set the *ignoreinvisibletext* option in *TET_open_page()* to *true* so that only text is extracted which is actually visible on the page.

Processing documents with PDFlib+PDI. When using PDFlib+PDI to process PDF documents on a per-page basis you can integrate TET for controlling the splitting or merging process. For example, you could split a PDF document based on the contents of a page. If you have control over the creation process you can insert separator pages with suitable processing instructions in the text. An example for splitting a PDF document with PDFlib+PDI based on the page contents can be found in the TET distribution.

Legacy PDF documents with missing Unicode values. In some situations PDF documents created by legacy applications must be processed where the PDF may not contain enough information for proper Unicode mapping. Using the default settings TET may be unable to extract some or all of the text contents. Recommendations:

- ▶ Start by extracting the text with default settings, and analyze the results. Identify the fonts which do not provide enough information for proper Unicode mapping.
- ▶ Write custom encoding tables and glyph name lists to fix problematic fonts. Use the PDFlib FontReporter plugin for analyzing the fonts and preparing Unicode mapping tables.
- ▶ Configure the custom mapping tables and extract the text again, using a larger number of documents. If there are still unmappable glyphs or fonts adjust the mapping tables as appropriate.
- ▶ If you have a large number of documents with unmappable fonts PDFlib GmbH may be able to assist you in creating the required mapping tables.

Convert PDF documents to another format. If you want to import the page contents of PDF documents into your application, while retaining as much information as possible you'll need precise character metrics. Recommendations:

- ▶ Use *TET_get_char_info()* to retrieve precise character metrics and font names. Even if you use the *uv* field to retrieve the Unicode values of individual characters, you must also call *TET_get_text()* since it fills the *char_info* structure.
- ▶ Use *granularity=glyph* or *word* in *TET_open_page()*, depending on what is better suited to your application.

Corporate fonts with custom-encoded logos. In many cases corporate fonts containing custom logos have missing or wrong Unicode mapping information for the logos. If you have a large number of PDF documents containing such fonts it is recommended to create a custom mapping table with proper Unicode values.

Start by creating a font report (see »Analyzing PDF documents with the PDFlib FontReporter plugin«, page 39) for a PDF containing the font, and locate mismapped glyphs in the font report. Depending on the font type you can use any of the available configuration tables to provide the missing Unicode mappings. See »Code list resources for all font types«, page 40, for a detailed example of a code list for a logotype font.

5 The pCOS Interface

The pCOS (*PDFlib Comprehensive Object Syntax*) interface provides a simple and elegant facility for retrieving arbitrary information from all sections of a PDF document which do not describe page contents, such as page dimensions, metadata, interactive elements, etc. pCOS users are assumed to have some basic knowledge of internal PDF structures and dictionary keys, but do not have to deal with PDF syntax and parsing details.

We strongly recommend that pCOS users obtain a copy of the *PDF Reference*, which is available as follows:

Adobe Systems Incorporated: PDF Reference, Fifth Edition: Version 1.6. Downloadable PDF from partners.adobe.com/public/developer/pdf/index_reference.html

5.1 Simple pCOS Examples

Assuming a valid PDF document handle is available, the pCOS functions *TET_pcos_get_number()*, *TET_pcos_get_string()*, and *TET_pcos_get_stream()* can be used to retrieve information from a PDF using the pCOS path syntax. Table 5.1 lists some common pCOS paths and their meaning.

Table 5.1 pCOS paths for commonly used PDF objects

pCOS path	type	explanation
<i>length:pages</i>	<i>number</i>	<i>number of pages in the document</i>
<i>/Info/Title</i>	<i>string</i>	<i>document info field Title</i>
<i>/Root/Metadata</i>	<i>stream</i>	<i>XMP stream with the document's metadata</i>
<i>fonts[...]/name</i>	<i>string</i>	<i>name of a font; the number of entries can be retrieved with length:fonts</i>
<i>fonts[...]/vertical</i>	<i>boolean</i>	<i>check a font for vertical writing mode</i>
<i>fonts[...]/embedded</i>	<i>boolean</i>	<i>embedding status of a font</i>
<i>pages[...]/width</i>	<i>number</i>	<i>width of the visible area of the page</i>

Number of pages. The total number of pages in a document can be queried as follows:

```
pagecount = p.pcos_get_number(doc, "length:pages");
```

Document info fields. Document information fields can be retrieved with the following code sequence:

```
objtype = p.pcos_get_string(doc, "type:/Info/Title");
if (objtype.equals("string"))
{
    /* Document info key found */
    title = p.pcos_get_string(doc, "/Info/Title");
}
```

Page size. Although the *MediaBox*, *CropBox*, and *Rotate* entries of a page can directly be obtained via pCOS, they must be evaluated in combination in order to find the actual size of a page. Determining the page size is much easier with the *width* and *height* keys

of the *pages* pseudo object. The following code retrieves the width and height of page 3 (note that indices for the *pages* pseudo object start at 0):

```
pagenum = 2
width = p.pcos_get_number(doc, "pages[" + pagenum + "]/width");
height = p.pcos_get_number(doc, "pages[" + pagenum + "]/height");
```

Listing all fonts in a document. The following sequence creates a list of all fonts in a document along with their embedding status:

```
fontcount = p.pcos_get_number(doc, "length:fonts");

for (i=0; i < fontcount; i++)
{
    fontname = p.pcos_get_string(doc, "fonts[" + i + "]/name");
    embedded = p.pcos_get_number(doc, "fonts[" + i + "]/embedded");
}
```

Writing mode. Using pCOS and the *fontid* value provided in the *char_info* structure you can easily check whether a font uses vertical writing mode:

```
if (p.pcos_get_number(doc, "fonts[" + ci->fontid + "]/vertical"))
{
    /* font uses vertical writing mode */
}
```

Encryption status. You can query the *pcosmode* pseudo object to determine the pCOS mode for the document:

```
if (p.pcos_get_number(doc, "pcosmode") == 2)
{
    /* full pCOS mode */
}
```

Text extraction status. By default, content extraction is possible with TET if the document can successfully be opened. However, with *infomode=true* this is not necessarily true. Depending on the *nocopy* permission setting, content extraction may or may not be allowed in restricted pCOS mode (content extraction is always allowed in full pCOS mode). The following expression can be used to check whether extraction is allowed:

```
if ((int) p.pcos_get_number(doc, "pcosmode") == 2 ||
    ((int) p.pcos_get_number(doc, "pcosmode") == 1 &&
     (int) p.pcos_get_number(doc, "encrypt/nocopy") == 0))
{
    /* text extraction allowed */
}
```

XMP meta data. A stream containing XMP meta data can be retrieved with the following code sequence:

```
objtype = p.pcos_get_number(doc, "type:/Root/Metadata");
if (objtype.equals("stream"))
{
    /* XMP meta data found */
    metadata = p.pcos_get_stream(doc, "", "/Root/Metadata");
}
```

5.2 Handling Basic PDF Data Types

pCOS offers the three functions *TET_pcos_get_number()*, *TET_pcos_get_string()*, and *TET_pcos_get_stream()*. These can be used to retrieve all basic data types which may appear in PDF documents.

Numbers. Objects of type *integer* and *real* can be queried with *TET_pcos_get_number()*. pCOS doesn't make any distinction between integer and floating point numbers.

Names and strings. Objects of type *name* and *string* can be queried with *TET_pcos_get_string()*. Name objects in PDF may contain non-ASCII characters and the # syntax (decoration) to include certain special characters. pCOS deals with PDF names as follows:

- ▶ Name objects will be undecorated (i.e. the # syntax will be resolved) before they are returned.
- ▶ Name objects will be returned as Unicode strings in most language bindings. However, in the C and C++ language bindings they will be returned as UTF-8.

Since the majority of strings in PDF are text strings *TET_pcos_get_string()* will treat them as such. However, in rare situations strings in PDF are used to carry binary information. In this case strings should be retrieved with the function *TET_pcos_get_stream()* which preserves binary strings and does not modify the contents in any way.

Booleans. Objects of type *boolean* can be queried with *TET_pcos_get_number()* and will be returned as 1 (true) or 0 (false). *TET_pcos_get_string()* can also be used to query boolean objects; in this case they will be returned as one of the strings *true* and *false*.

Streams. Objects of type *stream* can be queried with *TET_pcos_get_stream()*. Depending on the pCOS data type (*stream* or *fstream*) the contents will be compressed or uncompressed. Using the *keepfilter* option of *TET_pcos_get_string()* the client can retrieve compressed data even for type *stream*.

Stream data in PDF may be preprocessed with one or more filters. The list of filters present at the stream can be queried from the stream dictionary; for images this information is much easier accessible in the image's *filterinfo* dictionary. If a stream's filter chain contains only supported filters its type will be *stream*. When retrieving the contents of a *stream* object, *TET_pcos_get_stream()* will remove all filters and return the resulting unfiltered data.

Note pCOS does not support the following stream filters: CCITTFax, JBIG2, and JPX.

If there is at least one unsupported filter in a stream's filter chain, the object type will be reported as *fstream* (filtered stream). When retrieving the contents of an *fstream* object, *TET_pcos_get_stream()* will remove the supported filters at the beginning of a filter chain, but will keep the remaining unsupported filters and return the stream data with the remaining unsupported filters still applied. The list of applied filters can be queried from the stream dictionary, and the filtered stream contents can be retrieved with *TET_pcos_get_stream()*. Note that the names of supported filters will not be removed when querying the names of the stream's filters, so the client should ignore the names of supported filters.

5.3 Composite Data Structures and IDs

Objects with one of the basic data types can be arranged in two kinds of composite data structures: arrays and dictionaries. pCOS does not offer specific functions for retrieving composite objects. Instead, the objects which are contained in a dictionary or array can be addressed and retrieved individually.

Arrays. Arrays are one-dimensional collections of any number of objects, where each object may have arbitrary type.

The contents of an array can be enumerated by querying the number N of elements it contains (using the *length* prefix in front of the array's path, see Table 5.2), and then iterating over all elements from index 0 to $N-1$.

Dictionaries. Dictionaries (also called associative arrays) contain an arbitrary number of object pairs. The first object in each pair has the type *name* and is called the key. The second object is called the value, and may have an arbitrary type except *null*.

The contents of a dictionary can be enumerated by querying the number N of elements it contains (using the *length* prefix in front of the dictionary's path, see Table 5.2), and then iterating over all elements from index 0 to $N-1$. Enumerating dictionaries will provide all dictionary keys in the order in which they are stored in the PDF using the *.key* suffix at the end of the dictionary's path. Similarly, the corresponding values can be enumerated with the *.val* suffix. Inherited values (see below) and pseudo objects will not be visible when enumerating dictionary keys, and will not be included in the *length* count.

Some page-related dictionary entries in PDF can be inherited across a tree-like data structure, which makes it difficult to retrieve them. For example the *MediaBox* for a page is not guaranteed to be contained in the page dictionary, but may be inherited from an arbitrarily complex page tree. pCOS eliminates this problem by transparently inserting all inherited keys and values into the final dictionary. In other words, pCOS users can assume that all inheritable entries are available directly in a dictionary, and don't have to search all relevant parent entries in the tree. This merging of inherited entries is only available when accessing the pages tree via the *pages[]* pseudo object; accessing the */Pages* tree, the *objects[]* pseudo object, or enumerating the keys via *pages[][]* will return the actual entries which are present in the respective dictionary, without any inheritance applied.

pCOS IDs for dictionaries and arrays. Unlike PDF object IDs, pCOS IDs are guaranteed to provide a unique identifier for an element addressed via a pCOS path (since arrays and dictionaries can be nested an object can have the same PDF object ID as its parent array or dictionary). pCOS IDs can be retrieved with the *pcosid* prefix in front of the dictionary's or array's path (see Table 5.2).

The pCOS ID can therefore be used as a shortcut for repeatedly accessing elements without the need for explicit path addressing. For example, this will improve performance when looping over all elements of a large array. Use the *objects[]* pseudo object to retrieve the contents of an element identified by a particular ID.

5.4 Path Syntax

The backbone of the pCOS interface is a simple path syntax for addressing and retrieving any object contained in a PDF document. In addition to the object data itself pCOS can retrieve information about an object, e.g. its type or length. Depending on the object's type (which itself can be queried) one of the functions *TET_pcos_get_number()*, *TET_pcos_get_string()*, and *TET_pcos_get_stream()* can be used to obtain the value of an object. The general syntax for pCOS paths is as follows:

```
[<prefix>:][pseudoname[<index>]]/<name>[<index>]/<name>[<index>] ... [.key|.val]
```

The meaning of the various path components is as follows:

- ▶ The optional *prefix* can attain the values listed in Table 5.2.
- ▶ The optional *pseudo object name* may contain one of the values described in Section 5.5, »Pseudo Objects«, page 59.
- ▶ The *name* components are dictionary keys found in the document. Multiple names are separated with a / character. An empty path, i.e. a single / denotes the document's Trailer dictionary. Each name must be a dictionary key present in the preceding dictionary. Full paths describe the chain of dictionary keys from the initial dictionary (which may be the Trailer or a pseudo object) to the target object.
- ▶ Paths or path components specifying an array or dictionary can have a numerical index which must be specified in decimal format between brackets. Nested arrays or dictionaries can be addressed with multiple index entries. The first entry in an array or dictionary has index 0.
- ▶ Paths or path components specifying a dictionary can have an index qualifier plus one of the suffixes *.key* or *.val*. This can be used to retrieve a particular dictionary key or the corresponding value of the indexed dictionary entry, respectively. If a path for a dictionary has an index qualifier it must be followed by one of these suffixes.

When a path component contains any of the characters */*, *[*, *]*, or *#*, these must be expressed by a number sign *#* followed by a two-digit hexadecimal number.

Path prefixes. Prefixes can be used to query various attributes of an object (as opposed to its actual value). Table 5.2 lists all supported prefixes.

The *length* prefix and content enumeration via indices are only applicable to plain PDF objects and pseudo objects of type *array*, but not any other pseudo objects. The *pcosid* prefix cannot be applied to pseudo objects. The *type* prefix is supported for all pseudo objects.

Table 5.2 pCOS path prefixes

prefix	explanation
length	(Number) Length of an object, which depends on the object's type: array Number of elements in the array dict Number of key/value pairs in the dictionary stream Number of key/value pairs in the stream dict (not the stream length; use the Length key to determine the length of stream data in bytes) fstream Same as stream other 0
pcosid	(Number) Unique pCOS ID for an object of type dictionary or array. If the path describes an object which doesn't exist in the PDF the result will be -1. This can be used to check for the existence of an object, and at the same time obtaining an ID if it exists.
type	(String or number) Type of the object as number or string: 0, null Null object or object not present (use to check existence of an object) 1, boolean Boolean object 2, number Integer or real number 3, name Name object 4, string String object 5, array Array object 6, dict Dictionary object (but not stream) 7, stream Stream object which uses only supported filters 8, fstream Stream object which uses one or more unsupported filters

5.5 Pseudo Objects

Pseudo objects extend the set of pCOS paths by introducing some useful elements which can be used as an abbreviation for information which is present in the PDF, but cannot easily be accessed by reading a single value. The following sections list all supported pseudo objects. Pseudo objects of type *dict* can not be enumerated.

Universal pseudo objects. Universal pseudo objects are always available, regardless of encryption and passwords. This assumes that a valid document handle is available, which may require setting the option *requiredmode* suitably when opening the document. Table 5.3 lists all universal pseudo objects.

Table 5.3 Universal pseudo objects

object name	explanation
encrypt	(Dict) Dictionary with keys describing the encryption status of the document: length (Number) Length of the encryption key in bits algorithm (Number) description (String) Encryption algorithm number or description: -1 Unknown encryption 0 No encryption 1 40-bit RC4 (Acrobat 2-4) 2 128-bit RC4 (Acrobat 5) 3 128-bit RC4 (Acrobat 6) 4 128-bit AES (Acrobat 7) 5 Public key on top of 128-bit RC4 (Acrobat 5) (unsupported) 6 Public key on top of 128-bit AES (Acrobat 7) (unsupported) 7 Adobe Policy Server (Acrobat 7) (unsupported) master (Boolean) True if the PDF requires a master password to change security settings (permissions, user or master password), false otherwise user (Boolean) True if the PDF requires a user password for opening, false otherwise noaccessible, noannots, noassemble, nocopy, noforms, nohiresprint, nomodify, noprint (Boolean) True if the respective access protection is set, false otherwise plainmetadata (Boolean) True if the PDF contains unencrypted meta data, false otherwise
filename	(String) Name of the PDF file.
filesize	(Number) Size of the PDF file in bytes
xinfo	(Boolean) True if and only if security settings were ignored when opening the PDF document; the client must take care of honoring the document author's intentions. For TET the value will be true, and text extraction will be allowed, if all of the following conditions are true: xinfo mode has been enabled (only possible under a special license agreement), the document has a master password but this has not been supplied, the user password (if any) has been supplied, and text extraction is not permitted.
linearized	(Boolean) True if the PDF document is linearized, false otherwise
major minor revision	(Number) Major, minor, or revision number of the library, respectively.

Table 5.3 Universal pseudo objects

object name	explanation						
<i>pcosinterface</i>	(Number) Interface number of the underlying pCOS implementation. This specification describes interface number 3. The following table details which product versions implement various pCOS interface numbers: <table><tr><td>1</td><td>TET 2.0, 2.1</td></tr><tr><td>2</td><td>pCOS 1.0</td></tr><tr><td>3</td><td>PDFlib+PDI 7, PPS 7, TET 2.2, TET 2.3, PLOP 3.0, pCOS 2.0</td></tr></table>	1	TET 2.0, 2.1	2	pCOS 1.0	3	PDFlib+PDI 7, PPS 7, TET 2.2, TET 2.3, PLOP 3.0, pCOS 2.0
1	TET 2.0, 2.1						
2	pCOS 1.0						
3	PDFlib+PDI 7, PPS 7, TET 2.2, TET 2.3, PLOP 3.0, pCOS 2.0						
<i>pcosmode</i>	(Number/string) pCOS mode as number or string:						
<i>pcos-modename</i>	<table><tr><td>0</td><td>minimum</td></tr><tr><td>1</td><td>restricted</td></tr><tr><td>2</td><td>full</td></tr></table>	0	minimum	1	restricted	2	full
0	minimum						
1	restricted						
2	full						
<i>pdfversion</i>	(Number) PDF version number multiplied by 10, e.g. 16 for PDF 1.6						
<i>version</i>	(String) Full library version string in the format <major>.<minor>.<revision>, possibly suffixed with additional qualifiers such as beta, rc, etc.						

Pseudo objects for PDF objects, pages, and interactive elements. Table 5.4 lists pseudo objects which can be used for retrieving object or page information, or serve as short-cuts for various interactive elements.

Table 5.4 Pseudo objects for PDF objects, pages, and interactive elements

object name	explanation
articles	<p>(Array of dicts) Array containing the article thread dictionaries for the document. The array will have length 0 if the document does not contain any article threads. In addition to the standard PDF keys pCOS supports the following pseudo key for dictionaries in the articles array:</p> <p>beads (Array of dicts) Bead directory with the standard PDF keys, plus the following:</p> <p>destpage (Number) Number of the target page (first page is 1)</p>
bookmarks	<p>(Array of dicts) Array containing the bookmark (outlines) dictionaries for the document. In addition to the standard PDF keys pCOS supports the following pseudo keys for dictionaries in the bookmarks array:</p> <p>level (Number) Indentation level in the bookmark hierarchy</p> <p>destpage (Number) Number of the target page (first page is 1) if the bookmark points to a page in the same document, -1 otherwise.</p>
fields	<p>(Array of dicts) Array containing the form fields dictionaries for the document. In addition to the standard PDF keys in the field dictionary and the entries in the associated Widget annotation dictionary pCOS supports the following pseudo keys for dictionaries in the fields array:</p> <p>level (Number) Level in the field hierarchy (determined by ».« as separator)</p> <p>fullname (String) Complete name of the form field. The same naming conventions as in Acrobat 7 will be applied.</p>
names	<p>(Dict) A dictionary where each entry provides simple access to a name tree. The following name trees are supported: AP, AlternatePresentations, Dests, EmbeddedFiles, IDS, JavaScript, Pages, Renditions, Templates, URLs.</p> <p>Each name tree can be accessed by using the name as a key to retrieve the corresponding value, e.g.: names/Dests[0].key retrieves the name of a destination names/Dests[0].val retrieves the corresponding destination dictionary</p> <p>In addition to standard PDF dictionary entries the following pseudo keys for dictionaries in the Dests names tree are supported:</p> <p>destpage (number) Number of the target page (first page is 1) if the destination points to a page in the same document, -1 otherwise.</p> <p>In order to retrieve other name tree entries these must be queried directly via /Root/Names/Dests etc. since they are not present in the name tree pseudo objects.</p>
objects	<p>(Array) Address an element for which a pCOS ID has been retrieved earlier using the pcoid prefix. The ID must be supplied as array index in decimal form; as a result, the PDF object with the supplied ID will be addressed. The length prefix cannot be used with this array.</p>

Table 5.4 Pseudo objects for PDF objects, pages, and interactive elements

object name	explanation
pages	(Array of dicts) Each array element addresses a page of the document. Indexing it with the decimal representation of the page number minus one addresses that page (the first page has index 0). Using the length prefix the number of pages in the document can be determined. A page object addressed this way will incorporate all attributes which are inherited via the /Pages tree. The entries /MediaBox and /Rotate are guaranteed to be present. In addition to standard PDF dictionary entries the following pseudo entries are available for each page: colorspaces, extgstates, fonts, images, patterns, properties, shadings, templates (Arrays of dicts) High-level page resources according to Table 5.5.
annots	(Array of dicts) In addition to the standard PDF keys pCOS supports the following pseudo key for dictionaries in the annots array: destpage (Number; only for Subtype=Link and if a Dest entry is present) Number of the target page (first page is 1)
blocks	(Array of dicts) Shorthand for pages[]/PieceInfo/PDFLib/Private/Blocks[], i.e. the page's block dictionary. In addition to the existing PDF keys pCOS supports the following pseudo key for dictionaries in the blocks array: rect (Rectangle) Similar to Rect, except that it takes into account any relevant CropBox/MediaBox and Rotate entries and normalizes coordinate ordering.
height	(Number) Height of the page. The MediaBox or the CropBox (if present) will be used to determine the height. Rotate entries will also be applied.
isempty	(Boolean) True if the page is empty, and false if the page is not empty
label	(String) The page label of the page (including any prefix which may be present). Labels will be displayed as in Acrobat. If no label is present (or the PageLabel dictionary is malformed), the string will contain the decimal page number. Roman numbers will be created in Acrobat's style (e.g. VI), not in classical style which is different (e.g. XLV). If /Root/PageLabels doesn't exist, the document doesn't contain any page labels.
width	(Number) Width of the page (same rules as for height) The following entries will be inherited: CropBox, MediaBox, Resources, Rotate.
pdfa	(String) PDF/A conformance level of the document (e.g. PDF/A-1a:2005) or none
pdfx	(String) PDF/X conformance level of the document (e.g. PDF/X-1a:2001) or none
tagged	(Boolean) True if the PDF document is tagged, false otherwise

Pseudo objects for simplified resource handling. Resources are a key concept for managing various kinds of data which are required for completely describing the contents of a page. The resource concept in PDF is very powerful and efficient, but complicates access with various technical concepts, such as recursion and resource inheritance. pCOS greatly simplifies resource retrieval and supplies several groups of pseudo objects which can be used to directly query resources. Some of these pseudo resource dictionaries contain entries in addition to the standard PDF keys in order to further simplify resource information retrieval.

pCOS supports two groups of pseudo objects for resource retrieval. Global resource arrays contain all resources in a PDF document, while page resources contain only the resources used by a particular page. The resource entries in the global and page-based resource arrays reflect resources from the user’s point of view. They differ from native PDF resources in several ways:

- ▶ Some entries may be added (e.g. inline images, simple color spaces) or deleted (e.g. the parts of multi-strip images).
- ▶ In addition to the original PDF dictionary keys resource dictionaries may contain some user-friendly keys for auxiliary information (e.g. embedding status of a font, number of components of a color space).

The following list details the two categories using the *images* resource type as an example; the same scheme applies to all resource types listed in Table 5.5:

- ▶ A list of image resources in the document is available in *images*[].
- ▶ A list of image resources on each page is available in *pages*[]/*images*[].

Table 5.5 Pseudo objects for resource retrieval; each pseudo object *P* in this table creates two arrays with high-level resources *P*[] and *pages*[]/*P*[].

object name	explanation
colorspaces	(Array of dicts) Array containing dictionaries for all color spaces on the page or in the document. In addition to the standard PDF keys in color space and ICC profile stream dictionaries the following pseudo keys are supported:
alternateid	(Integer; only for name=Separation and DeviceN) Index of the underlying alternate color space in the colorspaces[] pseudo object.
alternateonly	(Boolean) If true, the colorspace is only used as the alternate color space for (one or more) Separation or DeviceN color spaces, but not directly.
baseid	(Integer; only for name=Indexed) Index of the underlying base color space in the colorspaces[] pseudo object.
colorantname	(Name; only for name=Separation) Name of the colorant
colorantnames	(Array of names; only for name=DeviceN) Names of the colorants
components	(Integer) Number of components of the color space
name	(String) Name of the color space
csarray	(Array; not for name=DeviceGray/RGB/CMYK) Array describing the underlying native color space.
High-level color space resources will include all color spaces which are referenced from any type of object, including the color spaces which do not require any native PDF resources (i.e. DeviceGray, DeviceRGB, and DeviceCMYK).	

Table 5.5 Pseudo objects for resource retrieval; each pseudo object *P* in this table creates two arrays with high-level resources *P*[] and pages[]/*P*[] .

object name	explanation
extgstates	(Array of dicts) Array containing the dictionaries for all extended graphics states (ExtGstates) on the page or in the document
fonts	<p>(Array of dicts) Array containing dictionaries for all fonts on the page or in the document. In addition to the standard PDF keys in font dictionaries, the following pseudo keys are supported:</p> <p>name (String) PDF name of the font without any subset prefix. Non-ASCII CJK font names will be converted to Unicode.</p> <p>embedded (Boolean) Embedding status of the font</p> <p>type (String) Font type</p> <p>vertical (Boolean) true for fonts with vertical writing mode, false otherwise</p>
images	<p>(Array of dicts) Array containing dictionaries for all images on the page or in the document. High-level image resources will include all image XObjects and inline images, while native PDF resources contain only image XObjects.</p> <p>In addition to the standard PDF keys the following pseudo keys are supported:</p> <p>bpc (Integer) The number of bits per component. This entry is usually the same as BitsPerComponent, but unlike this it is guaranteed to be available.</p> <p>colorspaceid (Integer) Index of the image's color space in the colorspace[] pseudo object. This can be used to retrieve detailed color space properties.</p> <p>filterinfo (Dict) Describes the remaining filter for streams with unsupported filters or when retrieving stream data with the keepfilter option set to true. If there is no such filter no filterinfo dictionary will be available. The dictionary contains the following entries:</p> <p>name (Name) Name of the filter</p> <p>supported (Boolean) True if the filter is supported</p> <p>decodeparms (Dict) The DecodeParms dictionary if one is present for the filter</p> <p>maskid (Integer) Index of the image's mask in the images[] pseudo object if the image is masked, otherwise -1</p> <p>maskonly (Boolean) If true, the image is only used as a mask for (one or more) other images, but not directly</p>
patterns	(Array of dicts) Array containing dictionaries for all patterns on the page or in the document
properties	(Array of dicts) Array containing dictionaries for all properties on the page or in the document
shadings	<p>(Array of dicts) Array containing dictionaries for all shadings on the page or in the document. In addition to the standard PDF keys in shading dictionaries the following pseudo key is supported:</p> <p>colorspaceid (Integer) Index of the underlying color space in the colorspace[] pseudo object.</p>
templates	(Array of dicts) Array containing dictionaries for all templates (Form XObjects) on the page or in the document

5.6 Encrypted PDF Documents

pCOS supports encrypted and unencrypted PDF documents as input. However, full object retrieval for encrypted documents requires the appropriate master password to be supplied when opening the document. Depending on the availability of user and master password, encrypted documents can be processed in one of the pCOS modes described below.

Full pCOS mode (mode 0): Encrypted PDFs can be processed without any restriction provided the master password has been supplied upon opening the file. All objects will be returned unencrypted. Unencrypted documents will always be opened in full pCOS mode.

Restricted pCOS mode (mode 1). If the document has been opened without the appropriate master password and does not require a user password (or the user password has been supplied) pCOS operations are subject to the following restriction: The contents of objects with type *string*, *stream*, or *fstream* can not be retrieved with the following exceptions:

- ▶ The objects */Root/Metadata* and */Info/** (document info keys) can be retrieved if *nocopy=false* or *plainmetadata=true*.
- ▶ The objects *bookmarks[...]/Title* and *annots[...]/Contents* (bookmark and annotation contents) can be retrieved if *nocopy=false*, i.e. if text extraction is allowed for the main text on the pages.

Minimum pCOS mode (mode 2). Regardless of the encryption status and the availability of passwords, the universal pCOS pseudo objects listed in Table 5.3 are always available. For example, the *encrypt* pseudo object can be used to query a document's encryption status. Encrypted objects can not be retrieved in minimum pCOS mode.

Table 5.6 lists the resulting pCOS modes for various password combinations. Depending on the document's encryption status and the password supplied when opening the file, PDF object paths may be available in minimum, restricted, or full pCOS mode. Trying to retrieve a pCOS path which is inappropriate for the respective mode will raise an exception.

Table 5.6 Resulting pCOS modes for various password combinations

If you know...	...pCOS will run in...
none of the passwords	restricted pCOS mode if no user password is set, minimum pCOS mode otherwise
only the user password	restricted pCOS mode
the master password	full pCOS mode

6 TET Library API Reference

6.1 Option Lists

Option lists are a powerful yet easy method to control TET operations. Instead of requiring a multitude of function parameters, many API methods support option lists, or optlists for short. Options lists are strings which may contain an arbitrary number of options. 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. Optlists support various data types and composite data like arrays. In most languages optlists can easily be constructed by concatenating the required keywords and values. C programmers may want to use the *sprintf()* function in order to construct optlists.

An optlist is a string containing one or more pairs of the form

```
name value
```

Names and values, as well as multiple name/value pairs can be separated by arbitrary whitespace characters (space, tab, carriage return, newline). The value may consist of a list of multiple values. You can also use an equal sign '=' between name and value:

```
name=value
```

Simple values. Simple values may use any of the following data types:

- ▶ Boolean: *true* or *false*; if the value of a boolean option is omitted, the value *true* is assumed. As a shorthand notation *nofoo* can be used instead of *foo=false* to disable option *foo*.
- ▶ String: these are plain ASCII strings which are generally used for non-localizable keywords. Strings containing whitespace or '=' characters must be bracketed with { and }. An empty string can be constructed with {}. The characters {, }, and \ must be preceded by an additional \ character if they are supposed to be part of the string.
- ▶ Strings and name strings: these can hold Unicode content in various formats; see Section 3.2, »C Binding«, page 20 for C- and C++-specific details regarding name strings.
- ▶ Unichar: these are single Unicode characters, where several syntax variants are supported: decimal values (e.g. 173), hexadecimal values prefixed with x, X, ox, oX, or U+ (xAD, oxAD, U+oAAD), numerical or character references (see below), but without the '&' and ';' decoration (shy, #xAD, #173). Alternatively, literal characters can be supplied. Unichars must be in the range 0-65535 (0-xFFFF).
- ▶ Keyword: one of a predefined list of fixed keywords
- ▶ Float and integer: decimal floating point or integer numbers; point and comma can be used as decimal separators for floating point values. Integer values can start with x, X, ox, or oX to specify hexadecimal values. Some options (this is stated in the respective function description) support percentages by adding a % character directly after the value.
- ▶ Handle: several internal object handles, e.g., document or page handles. Technically these are integer values.

Depending on the type and interpretation of an option additional restrictions may apply. For example, integer or float options may be restricted to a certain range of values;

handles must be valid for the corresponding type of object, etc. Some examples for simple values (the first line shows a password string containing a blank character):

```
TET_open_document(): password {secret string}
TET_open_document(): lineseparator={ CRLF }
```

List values. List values consist of multiple values, which may be simple values or list values in turn. Lists are bracketed with { and }. Example:

```
TET_set_option(): searchpath={/usr/lib/tet d:\tet}
```

Note The backslash \ character requires special handling in many programming languages

Rectangles. A rectangle is a list of four float values specifying the coordinates of the lower left and upper right corners of a rectangle. Rectangle coordinates will be interpreted in the standard or user coordinate system (see »Coordinate system«, page 33). Example:

```
TET_open_page(): includebox = {{0 0 500 100} {0 500 500 600}}
```

Character references in option lists. Some environments require the programmer to write source code in 8-bit encodings. This makes it cumbersome to include isolated Unicode characters in 8-bit encoded text without changing all characters in the text to multi-byte encoding. In order to aid developers in this situation, TET supports character references, a method known from markup languages such as SGML and HTML.

TET supports all numeric character references and character entity references defined in HTML 4.0, but in option lists they must be used without the '&' and ';' decoration. Numeric character references can be supplied in decimal or hexadecimal notation for the character's Unicode value. The following are examples for valid character references along with a description of the resulting character:

```
#173          soft hyphen
#xAD          soft hyphen
shy           soft hyphen
```

In addition to the HTML-style references above TET supports the custom character entity names for control characters (see Table 6.1).

Table 6.1 Custom character entity names for control characters

Unicode character (VB equivalents)	custom entity name	Unicode character (VB equivalents)	custom entity name
U+0020	SP, space	U+00AD	SHY, shy
U+00A0	NBSP, nbsp	U+000B	VT, verttab
		U+2028	LS, linesep
U+0009 (VbTab)	HT, hortab	U+000A (VbLf)	LF, linefeed
		U+000D (VbCr)	CR, return
		U+000D and U+000A (VbCrLf)	CRLF
		U+0085	NEL, newline
		U+2029	PS, parasep
U+002D	HY, hyphen	U+000C (VbFormFeed)	FF, formfeed

6.2 General Functions

<i>Perl PHP</i>	<i>resource TET_new()</i>
<i>C</i>	<i>TET *TET_new(void)</i>
	Create a new TET object.
<i>Returns</i>	A handle to a TET object to be used in subsequent calls. If this function doesn't succeed due to unavailable memory it will return NULL.
<i>Bindings</i>	This function is not available in object-oriented language bindings since it is hidden in the TET constructor.
<i>Java</i>	<i>void delete()</i>
<i>C#</i>	<i>void Dispose()</i>
<i>Perl PHP</i>	<i>resource TET_delete(resource tet)</i>
<i>C</i>	<i>void TET_delete(TET *tet)</i>
	Delete a TET object and release all related internal resources.
<i>Details</i>	Deleting a TET object automatically closes all of its open documents. The TET object must no longer be used in any function after it has been deleted.
<i>Bindings</i>	In object-oriented language bindings this function is generally not required since it is hidden in the TET destructor. However, in Java it is available nevertheless to allow explicit cleanup in addition to automatic garbage collection. In .NET <i>Dispose()</i> should be called at the end of processing to clean up unmanaged resources.
<i>C++</i>	<i>string utf8_to_utf16(string utf8string, string ordering, int *size)</i>
<i>Perl PHP</i>	<i>string TET_utf8_to_utf16(resource tet, string utf8string, string ordering)</i>
<i>C</i>	<i>const char *TET_utf8_to_utf16(TET *tet, const char *utf8string, const char *ordering, int *size)</i>
	Convert a string from UTF-8 format to UTF-16.
	utf8string String to be converted. It must contain a valid UTF-8 sequence (on EBCDIC platforms it must be encoded in EBCDIC). If a Byte Order Mark (BOM) is present, it will be removed.
	ordering Specifies the byte ordering of the result string:
	▶ <i>utf16</i> or an empty string: The converted string will not have a BOM, and will be stored in the platform's native byte order.
	▶ <i>utf16le</i> : The converted string will be formatted in little endian format, and will be prefixed with the LE BOM (\xFF\xFE).
	▶ <i>utf16be</i> : The converted string will be formatted in big endian format, and will be prefixed with the BE BOM (\xFE\xFF).
	size Pointer to a memory location where the length of the returned string (in bytes, but excluding the terminating two null bytes) will be stored.

Returns The converted UTF-16 string. In C it will be terminated by two null bytes. The returned string is valid until the next call to any function other than *TET_utf8_to_utf16()*, or until an exception is thrown. Clients must copy the string if they need it longer.

Bindings This function is not available in Unicode-capable language bindings. The memory used for the converted string will be managed by TET, and must not be freed by the client.

6.3 Exception Handling

C++

const string get_apiname()

C# Java

String get_apiname()

Perl PHP

string TET_get_apiname(resource tet)

VB

Function get_apiname() As String

C

const char *TET_get_apiname(TET *tet)

Get the name of the API function which caused an exception or failed.

Returns

The name of the function which threw an exception, or the name of the most recently called function which failed with an error code. An empty string will be returned if there was no error.

C++

const string get_errmsg()

C# Java

String get_errmsg()

Perl PHP

string TET_get_errmsg(resource tet)

VB

Function get_errmsg() As String

C

const char *TET_get_errmsg(TET *tet)

Get the text of the last thrown exception or the reason for a failed function call.

Returns

Text containing the description of the last exception thrown, or the reason why the most recently called function failed with an error code. An empty string will be returned if there was no error.

C++

int get_errno()

C# Java

int get_errno()

Perl PHP

long TET_get_errno(resource tet)

VB

Function get_errno() As Long

C

int TET_get_errno(TET *tet)

Get the number of the last thrown exception or the reason for a failed function call.

Returns

The number of an exception, or the error code of the most recently called function which failed with an error code. This function will return 0 if there was no error.

C

TET_TRY(tet)

C

TET_CATCH(tet)

C

TET_RETHROW(tet)

C

TET_EXIT_TRY(tet)

Set up an exception handling block; catch or rethrow an exception; or inform the exception machinery that a `TET_TRY()` block will be left without entering the corresponding

TET_CATCH() block. *TET_RETHROW()* can be used to throw an exception again to a higher-level function after catching it.

Details (C language binding only) See Section 3.2, »C Binding«, page 20.

6.4 Document Functions

C++	<code>int open_document(string filename, string optlist)</code>
C# Java	<code>int open_document(String filename, String optlist)</code>
Perl PHP	<code>long TET_open_document(resource tet, string filename, string optlist)</code>
VB	<code>Function open_document(filename As String, optlist As String) As Long</code>
C	<code>int TET_open_document(TET *tet, const char *filename, int len, const char *optlist)</code>

Open a PDF document from file for content extraction.

filename (Name string, but Unicode file names are only supported on Windows) Absolute or relative name of the PDF input file to be processed. The file will be searched in all directories specified in the *searchpath* resource category. On Windows it is OK to use UNC paths or mapped network drives. In PHP Unicode filenames must be UTF-8.

len (Only C language binding) Length of *filename* (in bytes) for UTF-16 strings. If *len* = 0 a null-terminated string must be provided.

optlist An option list specifying document options according to Table 6.2.

Returns -1 on error, or a document handle otherwise. If -1 is returned it is recommended to call `TET_get_errmsg()` to find out more details about the error.

Details Within a single TET object an arbitrary number of documents may be kept open at the same time. However, a single TET object must not be used in multiple threads simultaneously without any locking mechanism for synchronizing the access.

Encryption: if the document is encrypted its user password must be supplied in the *password* option if the permission settings allow content extraction. The document's master password must be supplied if the permission settings do not allow content extraction.

Supported file systems on iSeries: TET has been tested with PC type file systems only. Therefore input and output files should reside in PC type files in the IFS (Integrated File System). The *QSYS.lib* file system for input files has not been tested and is not supported. Since *QSYS.lib* files are mostly used for record-based or database objects, unpredictable behavior may be the result if you use TET with *QSYS.lib* objects. TET file I/O is always stream-based, not record-based.

Table 6.2 Document options for `TET_open_document()`, `TET_open_document_callback()`, and `TET_open_document_mem()`

option	description
copy	(Boolean; Only for <code>TET_open_document_mem()</code> , and only useful for the C and C++ bindings) If true, TET will immediately make an internal copy of the supplied PDF data. Otherwise the client is responsible for keeping the data available until the corresponding call to <code>TET_close_document()</code> . Default: false
encodinghint	(String ¹) The name of an encoding which will be used to determine Unicode mappings for glyph names which cannot be mapped by standard rules, but only by a predefined internal glyph mapping rule. The keyword none can be used to disable all predefined rules. Default: winansi

Table 6.2 Document options for `TET_open_document()`, `TET_open_document_callback()`, and `TET_open_document_mem()`

option	description
glyphmapping	(List of option lists) A list of option lists where each option list describes a glyph mapping method for one or more font/encoding combinations which cannot reliably be mapped with standard methods. The mappings will be used in least-recently-set order. If the last option list contains the fontname wildcard <code>»*</code> , preceding mappings will no longer be used. Each rule consists of an option list according to Table 6.3 (default: predefined internal glyph rules will be applied).
infomode	(Boolean) Control handling of encrypted documents (default: false): <div>false The function call will fail (i.e. return -1) if the document's encryption status, permission settings, and supplied password forbid text extraction. It is guaranteed that the pCOS mode will be full or restricted if the function call succeeds. true The function call will succeed even if text extraction is not allowed. Before attempting to extract text or apply any pCOS operations the client must check the actual pCOS mode and permission settings using the <code>pcosmode</code> and <code>encrypt pseudo objects</code> (see Section 5.6, <code>»Encrypted PDF Documents«</code>, page 65).</div>
keepua	(Boolean) If true, PUA (Private Use Area) values will be returned as such; otherwise they will be mapped to the Unicode replacement character (see option <code>unknownchar</code>). Default: false
inmemory	(Boolean; Only for <code>TET_open_document()</code>) If true, TET will load the complete file into memory and process it from there. This can result in a tremendous performance gain on some systems (especially MVS) at the expense of memory usage. If false, individual parts of the document will be read from disk as needed. Default: false
password	(String; Maximum string length: 32 characters) The user or master password for encrypted documents. If the document's permission settings allow text copying then the user password is sufficient, otherwise the master password must be supplied. Note: vendors of a search engines may want to locate a document without making available the actual text contents to the user. Premium customers may obtain a custom version of TET under a special license agreement which allows text retrieval even without the master password, assuming no user password has been set. See Section 5.6, <code>»Encrypted PDF Documents«</code> , page 65, to find out how to query a document's encryption status, and pCOS operations which can be applied even without knowing the user or master password.
repair	(Keyword) Specifies how to treat damaged PDF documents. Repairing a document takes more time than normal parsing, but may allow processing of certain damaged PDFs. Note that some documents may be damaged beyond repair (default: auto): <div>force Unconditionally try to repair the document, regardless of whether or not it has problems. auto Repair the document only if problems are detected while opening the PDF. none No attempt will be made at repairing the document. If there are problems in the PDF the function call will fail.</div>
unknown-char	(Unichar) The character to be used as a replacement for unknown characters which cannot be mapped to Unicode (see Section 4.4, <code>»Unicode Mapping«</code> , page 37) . Default: U+FFFD (Replacement Character)
usehostfonts	(Boolean) If true, data for fonts which are not embedded, but are required for determining Unicode mappings will be searched on the Mac or Windows host operating system. Default: true

1. See footnote 2 in Table 6.3.

Table 6.3 Suboptions for the glyphmapping option of `TET_open_document()`, `TET_open_document_callback()`, and `TET_open_document_mem()`

option	description
codelist	(String) Name of a codelist resource to be applied to the font. It will have higher priority than an embedded ToUnicode CMap or encoding entry.
fontname	(Name string) Prefix or full name of the font to which the rule will be applied (subset prefixes in the font name must be excluded). Limited wildcards ¹ are supported. Default: *
force-encoding	(List with one or two strings ² , If there are two names, the first must be winansi or macroman) Replace the first encoding with the encoding resource specified by the second name. If only one entry is supplied, the specified encoding will be used to replace all instances of MacRoman, WinAnsi, and MacExpert encoding.
forcettsymbol-encoding	(Keyword or string ²) The name of an encoding which will be used to determine Unicode mappings for embedded pseudo TrueType symbol fonts which are actually text fonts, or one of the following keywords (default: auto): <div>auto If the font's builtin encoding (see below) contains at least one Unicode character in the symbolic range U+FO000-U+FOFF, the encoding specified in the encodinghint option will be used to map the pseudo symbol characters to real text characters. Otherwise encodinghint will not be used, and the characters will be mapped according to the builtin keyword. builtin Use the font's builtin encoding, which results from the Unicode mappings of the glyph names in the font's post table. The well-known TrueType fonts Wingdings* and Webdings* will always be treated as symbol fonts.</div>
glyphlist	(String) Name of a glyphlist resource to be applied
glyphrule	(Option list) Mapping rule for numerical glyph names (in addition to the predefined rules). The option list must contain the following suboptions: <div>prefix (String; may be empty) Prefix of the glyph names to which the rule will be applied. base (Keyword) One of the keywords hex or dec for hexadecimal or decimal representation of codes within a glyph name. encoding (String) Name of an encoding resource which will be used for this rule, or the keyword none to disable the rule.</div>
tounicode-cmap	(String) Name of a ToUnicode CMap resource to be applied to the font; it will have higher priority than an embedded ToUnicode CMap or encoding entry.

1. Limited wildcards: The standalone character »*« denotes all fonts; Using »*« after a prefix (e.g. »MSTT*«) denotes all fonts starting with the specified prefix.

2. The following predefined encoding names can be used without additional configuration: winansi, macroman, macroman_apple, macroman_euro, ebcdic, ebcdic_37, iso8859-X, cpXXXX, and U+XXXX. Custom encodings can be defined as resources.

C++	<code>int open_document_mem(const char *data, long size, string optlist)</code>
C# Java	<code>int open_document_mem(byte[] data, String optlist)</code>
Perl PHP	<code>long TET_open_document_mem(resource tet, string data, string optlist)</code>
VB	<code>Function open_document_mem(data As Variant, optlist As String) As Long</code>
C	<code>int TET_open_document_mem(TET *tet, const void *data, size_t size, const char *optlist)</code>

Open a PDF document from memory for content extraction.

data A reference to the data containing the PDF document. In C and C++ this is a pointer. In Java and C# this is a byte array. In PHP this is a string. In COM this is a variant of type byte.

size (Only for the C and C++ bindings) The length of the data in bytes.

optlist An option list specifying document options according to Table 6.2.

Returns See [TET_open_document\(\)](#).

Details See [TET_open_document\(\)](#).

```
C++ int open_document_callback(void *opaque, size_t filesize,
                             size_t (*readproc)(void *opaque, void *buffer, size_t size),
                             int (*seekproc)(void *opaque, long offset),
                             string optlist)
C   int TET_open_document_callback(TET *tet, void *opaque, size_t filesize,
                             size_t (*readproc)(void *opaque, void *buffer, size_t size),
                             int (*seekproc)(void *opaque, long offset),
                             const char *optlist)
```

Open a PDF document from a custom data source for content extraction.

opaque A pointer to some user data that might be associated with the input PDF document. This pointer will be passed as the first parameter of the callback functions, and can be used in any way. TET will not use the opaque pointer in any other way.

filesize The size of the complete PDF document in bytes.

readproc A C callback function which copies *size* bytes to the memory pointed to by *buffer*. If the end of the document is reached it may copy less data than requested. The function must return the number of bytes copied.

seekproc A C callback function which sets the current read position in the document. *offset* denotes the position from the beginning of the document (0 meaning the first byte). If successful, this function must return 0, otherwise -1.

optlist An option list specifying document options according to Table 6.2.

Returns See [TET_open_document\(\)](#).

Details See [TET_open_document\(\)](#).

Bindings This function is only available in the C and C++ language bindings.

```
C++ void close_document(int doc)
C# Java void close_document(int doc)
Perl PHP TET_close_document(resource tet, long doc)
VB Sub close_document(doc As Long)
C void TET_close_document(TET *tet, int doc)
```

Release a document handle and all internal resources related to that document.

doc A valid document handle obtained with [TET_open_document*\(\)](#).

Details Closing a document automatically closes all of its open pages. All open documents and pages will be closed automatically when [TET_delete\(\)](#) is called. It is good programming practice, however, to close documents explicitly when they are no longer needed. Closed document handles must no longer be used in any function call.

6.5 Page Functions

C++	<code>int open_page(int doc, int pagenumber, string optlist)</code>
C# Java	<code>int open_page(int doc, int pagenumber, String optlist)</code>
Perl PHP	<code>long TET_open_page(resource tet, long pagenumber, string optlist)</code>
VB	<code>Function open_page(doc As Long, pagenumber As Long, optlist As String) As Long</code>
C	<code>int TET_open_page(TET *tet, int doc, int pagenumber, const char *optlist)</code>

Open a page for text extraction.

- doc** A valid document handle obtained with `TET_open_document*`().
- pagenumber** The physical number of the page to be opened. The first page has page number 1. The total number of pages can be retrieved with `TET_pcos_get_number()` and the pCOS path `length:pages`.
- optlist** An option list specifying page options according to Table 6.4.

Returns A handle for the page, or -1 in case of an error.

Details Within a single document an arbitrary number of pages may be kept open at the same time. The same page may be opened multiply with different options. However, options can not be changed while processing a page.

Layer definitions (optional content groups) which may be present on the page are not taken into account: all text on all layers of the page will be extracted, regardless of the visibility of layers.

Table 6.4 Page options for `TET_open_page()`

option	description
clippingarea	(Keyword) Specifies the clipping area (default: cropbox): mediabox Use the MediaBox (which is always present) cropbox Use the CropBox (the area visible in Acrobat) if present, else MediaBox bleedbox Use the BleedBox if present, else use cropbox trimbox Use the TrimBox if present, else use cropbox artbox Use the ArtBox if present, else use cropbox unlimited Consider all text, regardless of its location
content-analysis	(Option list; Not for granularity=glyph) List of suboptions according to Table 6.5 for controlling high-level text processing.
excludebox	(List of rectangles) Exclude the combined area of the specified rectangles from content extraction. Default: empty
fontsize-range	(List of two floats) Two numbers specifying the minimum and maximum font size of text. Text with a size outside of this interval will be ignored. The maximum can be specified with the keyword unlimited, which means that no upper limit will be active. Default: { 0 unlimited }
ignore-invisibletext	(Boolean) If true, text with rendering mode 3 (invisible) will be ignored. Default: false (since invisible text is mainly used for image+text PDFs containing scanned pages and the corresponding OCR text)
includebox	(List of rectangles) Restrict content extraction to the combined area of the specified rectangles. Default: the complete clipping area

Table 6.4 Page options for `TET_open_page()`

option	description
granularity	(Keyword) The granularity of the text fragments returned by <code>TET_get_text()</code> ; all modes except <code>glyph</code> will enable the wordfinder. See »Text granularity«, page 44, for more details (default: <code>word</code>).
glyph	A fragment contains the result of mapping one glyph, but may contain more than one character (e.g. for ligatures).
word	A fragment contains a word as determined by the wordfinder.
line	A fragment contains a line of text, or the closest approximation thereof. Word separators will be inserted between two consecutive words.
zone	A fragment contains a graphical unit of text; depending on the layout this may be a column or other entity. Word and line separators will be inserted between two consecutive words or lines, respectively.
page	A fragment contains the contents of a single page. Word, line, and zone separators will be inserted as appropriate.

Table 6.5 Suboptions for the contentanalysis option of `TET_open_page()`

option	description
dehyphenate	(Boolean) If <code>true</code> , hard hyphens (<code>U+002D</code> and <code>U+2010</code>) and soft hyphens (<code>U+00AD</code>) at the end of a line will be removed, and the text fragments surrounding the hyphen will be combined. Default: <code>true</code>
includebox-order	(Integer) When multiple include boxes have been supplied (see option <code>includebox</code>), this option controls how the order of boxes affects the wordfinder (default: <code>0</code>): 0 Ignore include box ordering when analyzing the page contents. The result will be the same as if all the text outside the include boxes was deleted. This is useful for eliminating unwanted text (e.g. headers and footers) while not affecting the Wordfinder in any way. 1 Take include box ordering into account when creating words and zones, but not for zone ordering. A word will never belong to more than one box. The resulting zones will be sorted in logical order. In case of overlapping boxes the text will belong to the box which is earlier in the list. This is useful for extracting text from preprinted forms, extracting text from tables, or when include boxes overlap for complicated layouts. 2 Consider include box ordering for all operations. The contents of each include box will be treated independently from other boxes, and the resulting text will be concatenated according to the order of the include boxes. This is useful for extracting text from printed forms in a particular ordering, or extracting article columns in a magazine layout in a predefined order. In all cases advance knowledge about the page layout is required in order to specify the include boxes in appropriate order.
lineseparator	(Unichar; Only for granularity=zone and page) Character to be inserted between lines ¹ . Default: <code>U+000A</code>
merge	(Integer) Controls strip and zone merging (default: <code>2</code>): 0 No merging after strip creation. This can significantly increase processing speed, but may create less than optimal output. 1 Simple strip-into-zone merging: strips will be merged into a zone if they overlap this particular zone, but don't overlap strips other than the next one (to avoid zone overlapping for non-shadow cases). 2 Advanced zone merging for out-of-sequence text: in addition to <code>merge=1</code> , multiple overlapping zones will be combined into a single zone, provided the text contents of both zones do not overlap.
shadow-detect	(Boolean) If <code>true</code> , redundant instances of overlapping text fragments which create a shadow or fake bold text will be detected and removed. Default: <code>true</code>

Table 6.5 Suboptions for the contentanalysis option of `TET_open_page()`

option	description
punctuation breaks	(Boolean) If true, punctuation characters which are placed close to a letter will be treated as word boundaries, otherwise they will be included in the adjacent word. Default: true
wordseparator	(Unichar; Only for granularity=line, zone, and page) Character to be inserted between words ¹ . Default: U+0020
zoneseparator	(Unichar; Only for granularity=page) Character to be inserted between zones ¹ . Default: U+000A

1. Use U+0000 to disable the separator.

C++	void close_page(int page)
C# Java	void close_page(int page)
Perl PHP	TET_close_page(resource tet, long page)
VB	Sub close_page(page As Long)
C	void TET_close_page(TET *tet, int page)

Release a page handle and all related resources.

page A valid page handle obtained with `TET_open_page()`.

Details All open pages of the document will be closed automatically when `TET_close_document()` is called. It is good programming practice, however, to close pages explicitly when they are no longer needed. Closed page handles must no longer be used in any function call.

6.6 Text and Metrics Retrieval Functions

C++ `const string get_text(int page)`

C# Java `String get_text(int page)`

Perl PHP `string TET_get_text(resource tet, long page)`

VB `Function get_text(page As Long) As String`

C `const char *TET_get_text(TET *tet, int page, int *len)`

Get the next text fragment from a page's content.

page A valid page handle obtained with `TET_open_page()`.

len (C language binding only) A pointer to a variable which will hold the length of the returned string in UTF-16 values (not bytes!). To determine the number of bytes this value must be multiplied by 2 if `outputformat=utf16`; the string length of the returned null-terminated string must be used if `outputformat=utf8`.

Returns A string containing the next text fragment on the page. The length of the fragment is determined by the `granularity` option of `TET_open_page()`. Even for `granularity=glyph` the string may contain more than one character (see Section 4.1, »Characters and Glyphs«, page 31).

If all text on the page has been retrieved an empty string will be returned (in C: a NULL pointer and `*len=0`). In this case `TET_get_errnum()` should be called to find out whether there is no more text because of an error on the page, or because the end of the page has been reached.

Bindings C language binding: the result will be provided as null-terminated UTF-8 (default) or UTF-16 string according to the `outputformat` option of `TET_set_option()`. On iSeries and zSeries EBCDIC-encoded UTF-8 can also be selected, and is enabled by default. The returned data buffer can be used until the next call to this function.

C++, COM, Java and .NET language bindings: the result will be provided as standard Unicode string in UTF-16 format.

PHP language binding: the result will be provided as UTF-8 string.

RPG language binding: the result will be provided as null-terminated ASCII- or EBCDIC-encoded UTF-8 string, or as a null-terminated UTF-16 string according to the `outputformat` option of `TET_set_option()`.

C++ `const TET_char_info *get_char_info(int page)`

C# Java `int get_char_info(int page)`

Perl PHP `object TET_get_char_info(resource tet, long page)`

VB `Function get_char_info(int page) As Long`

C `const TET_char_info *TET_get_char_info(TET *tet, int page)`

Get detailed information for the next character in the most recent text fragment.

page A valid page handle obtained with `TET_open_page()`.

Returns If no more characters are available for the most recent text fragment returned by `TET_get_text()`, a binding-specific value will be returned. See section *Bindings* below for more details.

Details This function can be called after `TET_get_text()`. It will advance to the next character for the current text fragment associated with the supplied page handle (or return 0 or NULL if there are no more characters), and provide detailed information for this character. There will be *N* successful calls to this function where *N* is the number of UTF-16 characters in the text fragment returned by the most recent call to `TET_get_text()`.

For granularities other than *glyph* this function will advance to the next character of the string returned by the most recent call to `TET_get_text()`. This way it is possible to retrieve character metrics when the wordfinder is active and a text fragment may contain more than one character. In order to retrieve all character details for the current text fragment this function must be called repeatedly until it returns NULL or 0.

The character details in the structure or properties/fields are valid until the next call to `TET_get_char_info()` or `TET_close_page()` with the same page handle (whichever occurs first). Since there is only a single set of character info properties/fields per TET object, clients must retrieve all character info before they call `TET_get_char_info()` again for the same or another page or document.

Bindings C and C++ language bindings: If no more characters are available for the most recent text fragment returned by `TET_get_text()`, a NULL pointer will be returned. Otherwise, a pointer to a `TET_char_info` structure containing information about a single character will be returned. The members of the data structure are detailed in Table 6.6.

COM, Java and .NET language bindings: -1 will be returned if no more characters are available for the most recent text fragment returned by `TET_get_text()`, otherwise 1. Individual character info can be retrieved from the TET properties/public fields according to Table 6.6. All properties/fields will contain a value of -1 (the *unknown* field will contain *false*) if they are accessed although the function returned 0.

Perl language binding: 0 will be returned if no more characters are available for the most recent text fragment returned by `TET_get_text()`, otherwise a hash containing the keys listed in Table 6.6. Individual character info can be retrieved with the keys in this hash.

PHP language binding: 0 will be returned if no more characters are available for the most recent text fragment returned by `TET_get_text()`, otherwise an object containing the fields listed in Table 6.6. Individual character info can be retrieved from the member fields of this object. All fields will contain a value of -1 (the *unknown* field will contain *false*) if they are accessed although the function returned 0. Integer fields in the character info object are implemented as *long* in the PHP language binding.

Table 6.6 Members of the **TET_char_info** structure (C and C++), equivalent public fields (Java, PHP), keys (Perl) or properties (COM and .NET) with their type and meaning. See »Glyph metrics«, page 33, for more details.

property/ field name	explanation
uv	(Integer) UTF-32 Unicode value of the current character. It will be 0 if the corresponding UTF-16 value is the trailing value of a surrogate pair (i.e. if type=11).
type	<p>(Integer) Type of the character. The following types describe real characters which correspond to a glyph on the page. The values of all other properties/fields are determined by the corresponding glyph:</p> <p>0 Normal character which corresponds to exactly one glyph</p> <p>1 Start of a sequence (e.g. ligature)</p> <p>The following types describe artificial characters which do not correspond to a glyph on the page. The x and y fields will specify the most recent real character's endpoint, the width field will be 0, and all other fields except uv will contain the values corresponding to the most recent real character:</p> <p>10 Continuation of a sequence (e.g. ligature)</p> <p>11 Trailing value of a surrogate pair; the leading value has type=0, 1, or 10.</p> <p>12 Inserted word, line, or zone separator</p>
unknown	(Boolean, in C and C++: integer) Usually false (0), but will be true (1) if the original glyph could not be mapped to Unicode and has been replaced with the character specified as unknownchar.
x, y	(Double) Position of the glyph's reference point. The reference point is the lower left corner of the glyph box for horizontal writing mode, and the top center point for vertical writing mode. For artificial characters the x, y coordinates will be those of the end point of the most recent real character.
width	(Double) Width of the corresponding glyph (for both horizontal and vertical writing mode). For artificial characters the width will be 0.
alpha	(Double) Direction of inline text progression in degrees measured counter-clockwise. For horizontal writing mode this is the direction of the text baseline; for vertical writing mode it is the digression from the standard -90° direction. The angle will be in the range -180° < alpha ≤ +180°. For standard horizontal text as well as for standard text in vertical writing mode the angle will be 0°.
beta	(Double) Text slanting angle in degrees (counter-clockwise), relative to the perpendicular of alpha. The angle will be 0° for upright text, and negative for italicized (slanted) text. The angle will be in the range -180° < beta ≤ 180°, but different from ±90°. If abs(beta) > 90° the text is mirrored at the baseline.
fontid	(Integer) Index of the font in the fonts[] pseudo object (see Table 5.5). fontid is never negative.
fontsize	(Double) Size of the font (always positive); the relation of this value to the actual height of glyphs is not fixed, but may vary with the font design. For most fonts the font size is chosen such that it encompasses all ascenders (including accented characters) and descenders.
textrendering	<p>(Integer) Text rendering mode:</p> <p>0 fill text</p> <p>1 stroke text (outline)</p> <p>2 fill and stroke text</p> <p>3 invisible text (often used for OCR results)</p> <p>4 fill text and add it to the clipping path</p> <p>5 stroke text and add it to the clipping path</p> <p>6 fill and stroke text and add it to the clipping path</p> <p>7 add text to the clipping path</p>

6.7 Option Handling

C++
C# Java
Perl PHP
VB
C

void set_option(string optlist)
void set_option(String optlist)
TET_set_option(resource tet, string optlist)
Sub set_option(optlist As String)
void TET_set_option(TET *tet, const char *optlist)

Set one or more global options for TET.

optlist An option list specifying global options according to Table 6.7. If an option is provided more than once the last instance will override all previous ones. In order to supply multiple values for a single option (e.g. *searchpath*) supply all values in a list argument to this option.

Details Multiple calls to this function can be used to accumulate values for those options marked in Table 6.7. For unmarked options the new value will override the old one.

Table 6.7 Global options for **TET_set_option()**

option	description
cmap¹	(List of name strings) A list of string pairs, where each pair contains the name and value of a CMap resource (see Section 4.7, »Resource Configuration and File Searching«, page 47).
codelist¹	(List of name strings) A list of string pairs, where each pair contains the name and value of a codelist resource (see Section 4.7, »Resource Configuration and File Searching«, page 47).
encoding¹	(List of name strings) A list of string pairs, where each pair contains the name and value of an encoding resource (see Section 4.7, »Resource Configuration and File Searching«, page 47).
fontoutline¹	(List of name strings) A list of string pairs, where each pair contains the name and value of a FontOutline resource (see Section 4.7, »Resource Configuration and File Searching«, page 47).
glyphlist¹	(List of name strings) A list of string pairs, where each pair contains the name and value of a glyphlist resource (see Section 4.7, »Resource Configuration and File Searching«, page 47).
license	(String) Set the license key. It must be set before the first call to TET_open_document*() .
licensefile	(String) Set the name of a file containing the license key(s). The license file can be set only once before the first call to TET_open_document*() . Alternatively, the name of the license file can be supplied in an environment variable called PDFLIBLICENSEFILE or (on Windows) via the registry.
output-format	(Keyword; Only for the C and RPG language bindings) Specifies the format of the text returned by TET_get_text() (default on zSeries with USS or MVS: ebcdicutf8; on all other systems: utf8): utf8 Strings will be returned in null-terminated UTF-8 format (on both ASCII- and EBCDIC-based systems). ebcdicutf8 (Only available on EBCDIC-based systems) Strings will be returned in null-terminated EBCDIC-encoded UTF-8 format. Code page 37 will be used on iSeries, code page 01047 on zSeries. utf16 Strings will be returned in UTF-16 format in the machine's native byte ordering (on Intel x86 architectures the native byte order is little-endian, while on Sparc and PowerPC systems it is big-endian).

Table 6.7 Global options for `TET_set_option()`

option	description
resourcefile	<p>(Name string) Relative or absolute file name of the UPR resource file. The resource file will be loaded immediately. Existing resources will be kept; their values will be overridden by new ones if they are set again. Explicit resource options will be evaluated after entries in the resource file.</p> <p>The resource file name can also be supplied in the environment variable <code>TETRESOURCEFILE</code> or with a Windows registry key (see Section 4.7, »Resource Configuration and File Searching«, page 47). Default: <code>tet.upr</code> (on MVS: <code>upr</code>)</p>
searchpath¹	<p>(List of name strings) Relative or absolute path name(s) of a directory containing files to be read. The search path can be set multiply; the entries will be accumulated and used in least-recently-set order (see Section 4.7, »Resource Configuration and File Searching«, page 47). An empty string deletes all existing search path entries. On Windows the search path can also be set via a registry entry. Default: empty</p>

1. Option values can be accumulated with multiple calls.

6.8 pCOS Functions

The full pCOS syntax for retrieving object data from a PDF is supported; see Chapter 5, »The pCOS Interface«, page 53 for a detailed description.

C++

double pcos_get_number(int doc, string path)

C# Java

double pcos_get_number(int doc, String path)

Perl PHP

double TET_pcos_get_number(resource tet, long doc, string path)

VB

Function pcos_get_number(doc as Long, path As String) As Double

C

double TET_pcos_get_number(TET *tet, int doc, const char *path, ...)

Get the value of a pCOS path with type *number* or *boolean*.

doc A valid document handle obtained with *TET_open_document*(.)*.

path A full pCOS path for a numerical or boolean object.

Additional parameters (C language binding only) A variable number of additional parameters can be supplied if the *key* parameter contains corresponding placeholders (%s for strings or %d for integers; use %% for a single percent sign). Using these parameters will save you from explicitly formatting complex paths containing variable numerical or string values. The client is responsible for making sure that the number and type of the placeholders matches the supplied additional parameters.

Returns The numerical value of the object identified by the pCOS path. For Boolean values 1 will be returned if they are *true*, and 0 otherwise.

C++

const string pcos_get_string(int doc, string path)

C# Java

String pcos_get_string(int doc, String path)

Perl PHP

string TET_pcos_get_string(resource tet, long doc, string path)

VB

Function pcos_get_string(doc as Long, path As String) As String

C

const char *TET_pcos_get_string(TET *tet, int doc, const char *path, ...)

Get the value of a pCOS path with type *name*, *string*, or *boolean*.

doc A valid document handle obtained with *TET_open_document*(.)*.

path A full pCOS path for a string, name, or boolean object.

Additional parameters (C language binding only) A variable number of additional parameters can be supplied if the *key* parameter contains corresponding placeholders (%s for strings or %d for integers; use %% for a single percent sign). Using these parameters will save you from explicitly formatting complex paths containing variable numerical or string values. The client is responsible for making sure that the number and type of the placeholders matches the supplied additional parameters.

Returns A string with the value of the object identified by the pCOS path. For Boolean values the strings *true* or *false* will be returned.

Details This function will raise an exception if pCOS does not run in full mode and the type of the object is *string* (see Section 5.6, »Encrypted PDF Documents«, page 65). As an exception, the objects */Info/** (document info keys) can also be retrieved in restricted pCOS

mode if *nocopy=false* or *plainmetadata=true*, and *bookmarks[...]/Title* and *annots[...]/contents* can be retrieved in restricted pCOS mode if *nocopy=false*.

This function assumes that strings retrieved from the PDF document are text strings. String objects which contain binary data should be retrieved with *TET_pcos_get_stream()* instead which does not modify the data in any way.

Bindings C and C++ language bindings: The string will be returned in UTF-8 format.
C binding: The returned string can be used until the next call to this function.

C++	<i>const unsigned char *pcos_get_stream(int doc, int *length, string optlist, string path)</i>
C# Java	<i>final byte[] pcos_get_stream(int doc, String optlist, String path)</i>
Perl PHP	<i>string TET_pcos_get_stream(resource tet, long doc, string path)</i>
VB	<i>Function pcos_get_stream(doc as Long, optlist As String, path As String)</i>
C	<i>const unsigned char *TET_pcos_get_stream(TET *tet, int doc, int *length, const char *optlist, const char *path, ...)</i>

Get the contents of a pCOS path with type *stream*, *fstream*, or *string*.

doc A valid document handle obtained with *TET_open_document*()*.

length (C and C++ language bindings only) A pointer to a variable which will receive the length of the returned stream data in bytes.

optlist An option list specifying stream retrieval options according to Table 6.8.

path A full pCOS path for a stream or string object.

Additional parameters (C language binding only) A variable number of additional parameters can be supplied if the *key* parameter contains corresponding placeholders (%s for strings or %d for integers; use %% for a single percent sign). Using these parameters will save you from explicitly formatting complex paths containing variable numerical or string values. The client is responsible for making sure that the number and type of the placeholders matches the supplied additional parameters.

Returns The unencrypted data contained in the stream or string. The returned data will be empty (in C and C++: NULL) if the stream or string is empty.

If the object has type *stream*, all filters will be removed from the stream contents (i.e. the actual raw data will be returned). If the object has type *fstream* or *string* the data will be delivered exactly as found in the PDF file, with the exception of ASCII85 and ASCII-Hex filters which will be removed.

Details This function will throw an exception if pCOS does not run in full mode (see Section 5.6, »Encrypted PDF Documents«, page 65). As an exception, the object */Root/Metadata* can also be retrieved in restricted pCOS mode if *nocopy=false* or *plainmetadata=true*. An exception will also be thrown if *path* does not point to an object of type *stream*, *fstream*, or *string*.

Despite its name this function can also be used to retrieve objects of type *string*. Unlike *TET_pcos_get_string()*, which treats the object as a text string, this function will not modify the returned data in any way. Binary string data is rarely used in PDF, and cannot be reliably detected automatically. The user is therefore responsible for selecting the appropriate function for retrieving string objects as binary data or text.

Bindings COM: Most client programs will use the Variant type to hold the stream contents.
C and C++ language bindings: The returned data buffer can be used until the next call to this function.

Note *This function can be used to retrieve embedded font data from a PDF. Users are reminded of the fact that fonts are subject to the respective font vendor's license agreement, and must not be reused without the explicit permission of the respective intellectual property owners. Please contact your font vendor to discuss the relevant license agreement.*

Table 6.8 Options for `TET_pcos_get_stream()`

option	description
--------	-------------

(Currently no options are supported)

A TET Library Quick Reference

General Functions

Function prototype	page
<code>TET *TET_new(void)</code>	69
<code>void TET_delete(TET *tet)</code>	69
<code>const char *TET_utf8_to_utf16(TET *tet, const char *utf8string, const char *ordering, int *size)</code>	69

Exception Handling Functions

Function prototype	page
<code>const char *TET_get_apiname(TET *tet)</code>	71
<code>const char *TET_get_errmsg(TET *tet)</code>	71
<code>int TET_get_errnum(TET *tet)</code>	71

Document Functions

Function prototype	page
<code>int TET_open_document(TET *tet, const char *filename, int len, const char *optlist)</code>	73
<code>int TET_open_document_mem(TET *tet, const void *data, size_t size, const char *optlist)</code>	75
<code>int TET_open_document_callback(TET *tet, void *opaque, size_t filesize, size_t (*readproc)(void *opaque, void *buffer, size_t size), int (*seekproc)(void *opaque, long offset), const char *optlist)</code>	76
<code>void TET_close_document(TET *tet, int doc)</code>	76

Page Functions

Function prototype	page
<code>int TET_open_page(TET *tet, int doc, int pagenumber, const char *optlist)</code>	77
<code>void TET_close_page(TET *tet, int page)</code>	79

Text and Metrics Retrieval Functions

Function prototype	page
<code>const char *TET_get_text(TET *tet, int page, int *len)</code>	80
<code>const TET_char_info *TET_get_char_info(TET *tet, int page)</code>	80

Option Handling

Function prototype	page
<code>void TET_set_option(TET *tet, const char *optlist)</code>	83

pCOS Functions

Function prototype	page
<code>double TET_pcos_get_number(TET *tet, int doc, const char *path, ...)</code>	85
<code>const char *TET_pcos_get_string(TET *tet, int doc, const char *path, ...)</code>	85
<code>const unsigned char *TET_pcos_get_stream(TET *tet, int doc, int *length, const char *optlist, const char *path, ...)</code>	86

B Revision History

Revision history of this manual

Date	Changes
January 16, 2008	► Updated the manual for TET 2.3
January 23, 2007	► Minor additions for TET 2.2
December 14, 2005	► Additions and corrections for TET 2.1.0; added descriptions for the PHP and RPG language bindings
June 20, 2005	► Expanded and reorganized the manual for TET 2.0.0
October 14, 2003	► Updated the manual for TET 1.1
November 23, 2002	► Added the description of <code>TET_open_doc_callback()</code> and a code sample for determining the page size for TET 1.0.2
April 4, 2002	► First edition for TET 1

Index

A

- API (Application Programming Interface)
 - reference 67
- area of text extraction 35
- article threads 13
- attachments 51

B

- Byte Order Mark (BOM) 69

C

- C binding 20
- C++ binding 22
- categories of resources 47
- character references 68
- characters 31
- CJK (Chinese, Japanese, Korean) 36
 - compatibility forms 36
 - configuration 5
- codelist 40
- COM binding 23
- command-line tool 13
- composite characters 31
- content analysis 44
- coordinate system 33
- CUS (Corporate Use Subarea) 37

D

- dehyphenation 45
- Dispose() 69
- document and page functions 73
- DTD (Document Type Definition) 16

E

- EBCDIC-based systems 83
- encrypted PDF documents 65
- end points of glyphs and words 35
- evaluation version 5
- exception handling 19
 - in C 20

F

- fake bold removal 46
- file searching 48
- FontReporter plugin 10, 39
- fullwidth variants 36

G

- glyph metrics 33
- glyph rules 42
- glyphlist 42
- glyphs 31
- granularity 44

H

- halfwidth variants 36
- highlighting 35

I

- inch 33
- installing TET 5

J

- Java binding 24

L

- license key 6
- ligatures 31
- list values in option lists 68

M

- millimeters 33

N

- .NET binding 25

O

- optimizing performance 50
- option lists 67

P

- page boxes 35
- page size 53
- pCOS 53
 - API functions 85
 - data types 55
 - encryption 65
 - path syntax 57
 - pseudo objects 59
- PDF Reference Manual 53
- performance optimization 50

Perl binding 26
PHP binding 27
points 33
post-processing for Unicode values 37
prerotated glyphs 36
PUA (Private Use Area) 37

R

reading order 46
rectangles in option lists 68
replacement character 38
resource configuration 47
resourcefile parameter 49
RPG binding 29

S

searchpath 48
sequences 31
shadow removal 46
surrogates 32, 33

T

TET command-line tool 13
TET plugin 11
tet.upr 49
TET_CATCH() 71
TET_close_document() 76
TET_close_page() 79
TET_delete() 69
TET_EXIT_TRY() 20, 71
TET_get_apiname() 71
TET_get_char_info() 80
TET_get_errmsg() 71
TET_get_errnum() 71
TET_get_text() 80
TET_new() 69
TET_open_document() 73

TET_open_document_callback() 76
TET_open_document_mem() 75
TET_open_page() 77
TET_pcos_get_number() 85
TET_pcos_get_stream() 86
TET_pcos_get_string() 85
TET_RETHROW() 71
TET_set_option() 83
TET_TRY() 71
TET_utf8_to_utf16() 69
TETRESOURCEFILE environment variable 49
text filtering 32
ToUnicode CMap 41

U

Unicode mapping 37
units 33
unmappable glyphs 38
UPR file format 47
UTF-32 38
UTF-8 and UTF-16 69

V

vertical writing mode 36

W

word boundary detection 45
wordfinder 45

X

XML output 16

Z

zones 46